

AD-A240 565



2

Final Technical Report

for

Languages Beyond Ada and Lisp

DTIC

ELECTR
AUG 30 1991

C

Compiled by:

David A. Fisher
David A. Mundie

91-09299



5 August 1991

Sponsored by:

Defense Advanced Research Projects Agency
Software and Intelligent Systems Technology Office

ARPA Order No. 6487-1; Program Code No. 9E20
Issued by DARPA/CMO under Contract No. MDA972-88-C-0076

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

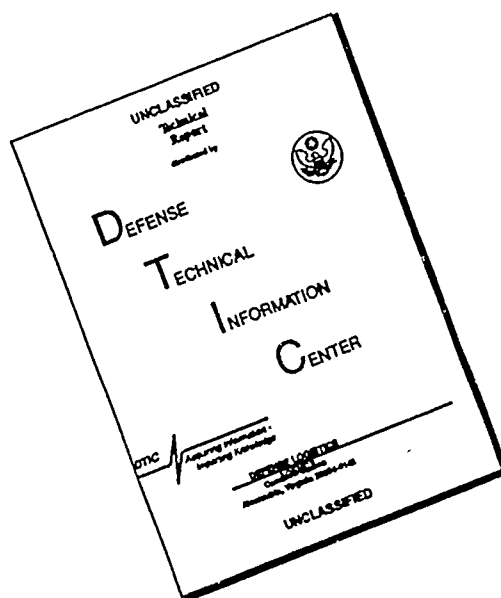
91 8 29 035

APPROVED FOR PUBLIC RELEASE
DISTRIBUTION UNLIMITED



Incremental
SYSTEMS CORPORATION

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST
QUALITY AVAILABLE. THE COPY
FURNISHED TO DTIC CONTAINED
A SIGNIFICANT NUMBER OF
PAGES WHICH DO NOT
REPRODUCE LEGIBLY.

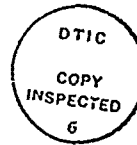
REPORT DOCUMENTATION PAGE

Form Approved

OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE 91.08.05	3. REPORT TYPE AND DATES COVERED Final - 88,09,22 - 91-06.30	
4. TITLE AND SUBTITLE Final Technical Report for Languages Beyond Ada and Lisp		5. FUNDING NUMBERS MDA-972-88-C-0076	
6. AUTHOR(S) David A. Fisher David A. Mundie		ARPA Order No. 6487-1 Program Code 9E20	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Incremental Systems Corporation 319 South Craig Street Pittsburgh, Pennsylvania 15213		8. PERFORMING ORGANIZATION REPORT NUMBER TR-910801	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) DARPA/SISTO 3701 North Fairfax Drive Arlington, Virginia 22203-1714		10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES			
12a. DISTRIBUTION/AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED		12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) This report summarizes the goals, concepts, research development, and conclusions of the "Languages Beyond Ada and Lisp" effort. The effort was born from the frustrations with progress in software engineering and with language design in particular. The goal of the effort, informally called Prism, was to devise ways to extend programming languages to encompass more of the total environment, and to remove the myriad barriers to cooperation among the pieces of software environments. The Prism effort has generated a wide variety of new ideas and approaches to solving traditional problems along the whole spectrum of formal systems. The new methods and approaches taken together have come to be called "informalism". Informalism involves automated reasoning systems that are capable of dealing with and even exploiting incompleteness and inconsistency. Informalism offers a potential for interdisciplinary computational interoperability previously unattainable.			
14. SUBJECT TERMS programming languages, informalism, computational linguistics, formal systems, incompleteness, type systems anaphora, prototype, persistence, epistemology		15. NUMBER OF PAGES 507	
		16. PRICE CODE	
17. SECURITY CLASSIFICATION OF THIS REPORT 010	18. SECURITY CLASSIFICATION OF THIS PAGE 010	19. SECURITY CLASSIFICATION OF ABSTRACT 010	20. LIMITATION OF ABSTRACT 010



Accession	
RTI	<input checked="checked" type="checkbox"/>
DTIC	<input type="checkbox"/>
DTIC	<input type="checkbox"/>
Distribution	
Availability Codes	
Avail and/or	
Dist	Special
A-1	

Distribution Statement:

one copy	DARPA/SISTO Attn: LTC Erik Mettala 3701 North Fairfax Drive Arlington, VA 22203-1714
one copy	DARPA/DASB Attn: Library 3701 North Fairfax Drive Arlington, VA 22203-1714
two copies	Defense Technical Information Center Building 5, Cameron Station Attn: DTIC-FDAC Alexandria, VA 22304-6145

Final Technical Report

for

Languages Beyond Ada and Lisp

1. Task Objectives

The Prism effort was born out of frustration with the current state of the art of software engineering. Little progress has been made in language design in the last thirty years; despite the innovations made along the way, Ada and Lisp are basically very similar languages. The diffusion of software innovations through the software economy is excruciatingly slow, due to the myriad barriers between the pieces of the programming environment (languages, versions, operating systems, &c.) The goal of Prism was to devise ways to extend programming languages to encompass more of the total environment, permitting those barriers to be bridged more easily. (See "Towards Full Spectrum Languages.")

2. Technical Problems

Early in the Prism project we concluded that a major source of the barriers in software engineering is the formalist model underlying the entirety of current language design. Although it was not until near the end of the project that we coined the term "informalism," the major outlines of informalism were clear from early on. In contrast to formalism, informalism is *semantically-based*, in that it assumes that the transformations to be applied to symbols can depend in an essential way upon the interpretation of those symbols; it is *open-ended* in that the meaning of an expression is always open to change; and it assumes that its data are intrinsically *incomplete* and *inconsistent*. The major technical challenge faced by the project was to devise implementation mechanisms for such a system. (See "A Conceptual Overview of Prism" and "Proceedings of the Workshop on Informal Computing.")

3. General Methodology

The methodology pursued by the Prism team consisted of four major techniques: 1. A wide-ranging review of current thinking about the problems being tackled, including attending conferences and talking with consultants. This review was deliberately not limited to computer science research, but covered relevant developments in the philosophy of language, cognitive science, and linguistics. (See "A Bibliography for Prism.") 2. A series of white papers setting forth the principle conclusions of the research effort. 3. A language design effort which incorporated the innovations suggested by the research. 4. A workshop which brought together like-minded members of the community and exposed the Prism conclusions to a broader audience.

4. Technical results

The Prism effort produced three major technical results. First, an epistemic, property-based type system which overcomes many of the limitations of traditional, extensional type systems, and allows the treatment of intensionality, a necessary first step towards raising the level of programming languages. (See "Epistemic Type Systems.") Secondly, a representation mechanism which generalizes all other known representations and permits a hybridization of connectionist-style processing with symbolic-style processing (see "Ideographs.") Finally, a language design which incorporates the property-based type system into a programming language based on current advances in computational linguistics (see "Unnatural Languages," "Reply to I. D. Hill," "Prism 0.5," "Prismatic Samples," and "Prism Primer.")

5. Important findings and conclusions

The Prism project has generated a wide variety of new ideas and approaches to solving traditional problems along the whole spectrum of formal systems. Many of these innovations are still in the formulative stage, but are already beginning to find application. For example, property-based types provide a way to explain other type systems, object-oriented inheritance, and derived types, all in a common framework. The linguistic mechanisms of Prism can be used to provide more expressive and less cumbersome programming languages. The ability to separate abstraction from representation satisfies a key requirement for design reuse, and will likely be applied initially in a specification language. The mechanisms employed in the effort can be refined to provide a common framework that eliminates the duplicative developments in computational linguistics. The methods developed for exploiting incompleteness and managing inconsistency can be used as the foundations for scalable software, for reasoning about the physical world, and for developing proof procedures with lower performance cost for underconstrained systems.

The effort taken as a whole constitutes a vision of and a feasibility study for a new domain of informal systems. Informal systems represent a new approach to real-world problem solving using computers: one that recognizes that complete descriptions of the physical things are impossible, that meaning must be grounded with an interpretative semantics, that the requirement for completeness drastically restricts the applicability of formal methods, and that there can be sound automated reasoning systems that support and exploit partial descriptions, cope with inconsistent specifications, and distinguish between formal models and the objects they represent.

The interdisciplinary nature of interest in informal systems arises from two independent causes. First, informalism derives from shared frustrations over the inherent limitations of formal methods, whether the field be software engineering, linguistics, psychology, or philosophy. Secondly, informalism offers a potential for interdisciplinary computational interoperability previously obtainable only in nonautomatable human reasoning.

6. Significant Hardware Development

None

Bibliography of Papers and Articles Published as a Result of this Effort

Baker, Deborah A., David A. Fisher and Jonathan C. Shultis; "IRIS as an Enabling Mechanism for the Use of Formal Methods", Incremental Systems TR-891101, November 1989, 7 pp.

Balzer, R., F.C. Belz, R. Dewer, D.A. Fisher, R. Gabriel, J. Guttag, P. Hudak and M. Wand; "Draft Report on Requirements for a Common Prototyping System", *SigPlan Notices*, Vol. 24, No. 3, March 1989, pp 93-165.

Baker, Deborah A. and Frank P. Tadman, "Reference Manual for Iris-Ada, Version 1.1", Incremental Systems TR900301, 16 March 1990, 67 pp;
also issued as an Arcadia Document.

Baker, Deborah A., David A. Fisher and Frank P. Tadman; "Management of Small- and Large-Grained Objects in the Iris Framework" (abstract only), Incremental Systems TR900302, 30 March 1990.

Mundie, David A. and David A. Fisher; "Optimized Overload Resolution and Type Matching for Ada", *Proceedings of the Symposium on Environments and Tools for Ada (SETA-1)*, Rodondo Beach, California, April 1990;
also issued as Incremental Systems TR900401, 8 pp.

Baker, Deborah A., David A. Fisher and Frank P. Tadman; "Persistence with Integrity and Efficiency", *Proceedings of the Fifth Annual RADC Knowledge Based Software Assistant Conference*, Syracuse, New York, September 1990;
also issued as Incremental Systems TR900901, 15 pp.

Baker, Deborah A., David A. Fisher, David A. Mundie, Jonathan C. Shultis and Frank P. Tadman; "A Conceptual Overview of Prism", Incremental Systems TR910201, February 22, 1991, 24 pp.

Shultis, Jon; "Epistemic Type Systems", Incremental Systems TR9102xx, February 22, 1991, 23 pp.

Baker, Deborah A., and Frank P. Tadman; "Reference Manual for Iris-Ada, Version 1.2", Incremental Systems TR910401, April 30, 1991, 52 pp;
also issued as Arcadia Document.

Mundie, David; "Natural and Unnatural Languages", *Proceedings of the Workshop on Informal Computing*, Santa Cruz, California, May 29, 1991, pp 13-16.

Fisher, David; "What is Informalism?", *Proceedings of the Workshop on Informal Computing*, Santa Cruz, California, May 30, 1991, pp 83-89.

Standish, Tim; "Informality and the Epistemology of Computer Programming", *Proceedings of the Workshop on Informal Computing*, Santa Cruz, California, May 30, 1991, pp 114-123.

Shultis, Jon; "Ideographs, Epistemic Types, and Interpretive Semantics", *Proceedings of the Workshop on Informal Computing*, Santa Cruz, California, May 31, 1991, pp 166-174.

Proceedings of the Workshop on Informal Computing, Santa Cruz, California, 1991 May 29-31;
edited by David A. Mundie and Jonathan C. Shultis, 190 pp.

Appendices

Goals and Concepts

Towards Full Spectrum Languages:

A New Approach to Software*

A Conceptual Overview of Prism*

Development of Research Ideas

Prism -- Design Notes†

Outline of a Type System for Prism†

Versions, Configurations and Object Deletion†

Untyped l in Prism 0.4†

Descriptive Processes†

Prism Eight Queens Example°

Persistence with Integrity and Efficiency*

Epistemic Type Systems*

Conclusions

Unnatural Languages

Grammar for Prism 0.5

A Prism Primer

Prismatic Samples

Reply to I.D. Hill

Proceedings of the Workshop on Informal Computing

Contributions to Arcadia Consortium and Other DARPA Activities

IRIS as an Enabling Mechanism

for the Use of Formal Methods*

Reference Manual for Iris-Ada*

Experience Using Iris

Management of Small- and Large-Grained Objects

in the Iris Framework°

Draft Report on Requirements

for a Common Prototyping System*#

Optimized Overload Resolution and Type Matching for Ada*

* technical reports and published papers

† working notes

° presentation visuals

◊ abstract

only cover sheet included

Goals and Concepts

for

Languages Beyond Ada and Lisp

Incremental Systems Corporation
319 South Craig Street
Pittsburgh, Pennsylvania

Towards Full Spectrum Languages: A New Approach to Software

Deborah A. Baker, David A. Fisher, David A. Mundie,
Jonathan C. Shultis and Frank P. Tadman

TR871002
October 1987

Incremental Systems Corporation
319 South Craig Street
Pittsburgh, PA 15213
412-621-8888

©Copyright, Incremental Systems Corporation, 1987



Incremental
SYSTEMS CORPORATION

Towards Full Spectrum Languages: A New Approach to Software

Deborah A. Baker
David A. Fisher
David A. Mundie
Jonathan C. Shultis
Frank P. Tadmán

Incremental Systems Corp.
319 S. Craig St.
Pittsburgh, PA 15213 USA

Abstract.

There have been few if any revolutionary advances in practical programming languages over the past 25 years. The key characteristics of languages today are for the most part indistinguishable from those of the early 1960's. Despite apparent advances in language design, compiler construction, development and maintenance tools, software engineering practice, database technology, and hardware reliability and performance, the development and maintenance of reliable and efficient software for large, long-lived, continuously changing applications remains an unachieved goal. This is the problem of programming-in-the-large.

As things stand, language technology is isolated from design technology, which is isolated from environment technology, which is isolated from database technology and so forth. The requirements, design, implementation, analysis, and testing of software are all specified using different tools which cannot communicate and cooperate with each other.

The solution lies not so much in more or improved technology in any of these areas, but in providing an integration framework which can be used to exploit existing and emerging technology. We need mechanisms to give users (i.e., application developers and researchers alike) access to the best of existing technology. In particular, we need an integration mechanism to allow the various component technologies to cooperate and build on each others' strengths, and those of their predecessors.

Languages have always been the most effective means of integration. We therefore propose to extend programming languages to the full breadth of software

concerns. This effort will develop a language which goes far beyond the capabilities of currently available programming languages. It will develop a *full spectrum* language which encompasses not only implementation issues, but the requirements, design, analysis, measurement, and environmental aspects of software development and maintenance. It will develop a language capable of absorbing new software technology dynamically as it becomes available.

Success in the effort will depend on our ability (1) to engineer a usable and easily understood specification mechanism based on abstract types, (2) to identify a small set of efficient and composable primitives adequate to encompass the full intended scope of the language, and (3) to integrate these into a simple and practical full spectrum language.

Success in the effort will mean that the entire life cycle of an application can be managed without leaving the language. The effort has high likelihood of success because the solution is much simpler than may first appear. The number of core concepts in environments, for example, is actually quite small because many of the concepts of extant environments are borrowed from (and therefore duplicative of) the concepts of programming languages. By combining environments technology and implementation languages in a single mechanism, we eliminate the duplication and along with it much of the apparent complexity of current software technology.

1. The Problem.

Over the years, the boundary between the automated and manual portions of the software development task has been continually pushed to higher and higher levels. This has always been done by capturing the best automatic implementation technology of the day in a class of languages and related tools, which then define the boundary. We contend that this process is fundamentally flawed, and is responsible for many of the ills which beset the world of software today. To understand how we reach this conclusion, consider the historical development of programming languages.

When software was implemented in machine language, the mappings from symbolic names to machine addresses were considered specifications. The programmer had to correctly and reliably translate those specifications into a formal implementation described in the machine language of the target computer. It was soon realized that many aspects of the programmer's task of mapping symbolic names of instructions, registers, and machine addresses to their machine representations could be automated with considerable advantages in both programmer productivity and reliability of the resulting implementation. Most importantly, the programmer was freed from concerns of the details of address and

operation code translation to concentrate on the more important aspects of software design.

When software was implemented in assembly languages, the formulae to be implemented were considered specifications. The programmer had to correctly and reliably translate those specifications into a formal implementation described in the assembly language. It was soon realized that many aspects of the programmers' task of translating formulae into sequences of assembly language operations could be automated with considerable advantages in both programmer productivity and reliability of the implementation. Most importantly, the programmer was freed from concerns of the details of the target machine instruction set architecture to concentrate on the more important aspects of software design.

Throughout the 1960's and 1970's we learned how to automate more and more aspects of the data, control, and algorithmic structure of program specifications with corresponding enhancements in the level of programming languages. At one end of the spectrum are a wide variety of very high-level special purpose languages where, by specializing the language and limiting its breadth of application, it has been possible to provide very high levels of automatic translation. At the other end of the spectrum are broad-based languages such as Ada¹ and Common Lisp which provide a variety of abstraction mechanisms which can be used to implement software in a large number of application areas. The continued raising of the level of both special purpose and general purpose languages has permitted more and more of the implementation decisions in applications to be assumed by the language implementation with considerable advantages in both programmer productivity and reliability of the resulting implementations.

Even more recently, logic and transformational programming have come to the fore with the realization that many aspects of the programmer's task of translating specifications into algorithmic processes could be automated with considerable advantages in both programmer productivity and reliability of the implementation. Most importantly, the programmer is freed from concern about the details of the algorithmic processes required in the implementation to concentrate on the more important aspects of software design.

The raising of the level of programming languages has been made possible by first learning how to formally specify more and more aspects of a software design, and then learning how to automate the translation of more and more of those specification mechanisms into operational computer programs. This process has made programmers more productive by freeing them from those aspects of the implementation which can be automatically translated from higher-level

¹Ada is a registered trademark of the U.S. Government, Ada Joint Program Office (AJPO)

specifications, by significantly raising the level of implementation specification required of them, and by allowing the implementation description to capture more of the engineering abstractions used by the designer.

Yet despite all these gains, the process of raising the level of programming languages has been an increasingly slow, evolutionary process that has not led and will not lead to any revolutionary gains in productivity or reliability of software design, implementation and maintenance. Most of the gains evident today had already been accomplished by 1960. The languages of the mid 1980's are for the most part characteristically indistinguishable from languages such as Fortran and Algol-60. The limited gains of the past 25 years have been at great expense. Even the slowest machines today are nearly 100 times faster than the fastest machines of 1960, and yet, in many applications, we are barely able to obtain 10 times the throughput. Languages such as Common Lisp permit us to address problems that were inconceivable 25 years ago, but only with such enormous consumption of machine resources that the language can seldom be used other than for research and prototyping purposes. Ada offers the potential for efficient use of machine resources, but at the expense of very early binding times and a static run-time model which greatly limit its applicability.

Even worse than the inability of this evolutionary process to make further gains of great significance is the fact that *the process itself holds down the level of programming languages by limiting them to formal specification mechanisms and techniques which compiler writers know how to implement (efficiently) at the time of language design.*

The assumption that a language must have a fixed definition leads to programming languages (e.g., Common Lisp and Ada) which remove many important aspects of the design of systems from the formal specification and places them inside the compiler where they cannot be seen, controlled, or modified by the application developer.

Furthermore, the rationale for this method of language design incorrectly assumes that the existence of efficient implementation technology will result in its incorporation in actual compilers. The latter point is most conspicuously illustrated by the Ada community, where it is clear that the technology exists to provide better-quality compilers (in terms of reliability of translation and execution performance) than with any previous operational programming language, and yet the available Ada compilers produce target code which is notoriously inefficient when compared with compilers for most of Ada's predecessor languages.

Other facets of current language technology are conspicuously absent from many widely used languages. For example, despite the long-standing recognition of

the advantages of strong typing, user-defined types, information hiding and other abstraction mechanisms, especially in large, complex, and continuously changing applications, they are almost absent from Common Lisp. Much greater steps backward are evident in C where no pretension of an abstraction facility is made. Instead the world of computation is reduced to Fortran-level descriptions of primitive machine operations acting on integers and sequences of bytes.

The functional, rule-based and object-oriented programming language paradigms have not provided the answer, either. Certain information, such as inheritance rules in Smalltalk, or the resolution algorithm in Prolog, are fixed, inaccessible, or only indirectly accessible.

Even the wide spectrum languages span a fixed range, from a fixed formal specification language to the details of implementation. Moreover, they do so with a fixed set of mechanisms; for example, multiple inheritance is not, and cannot be, a concept in CIP-L [Mol84]. Worse yet, those fixed mechanisms are inadequate for real applications; no existing wide spectrum language provides mechanisms for dealing with such things as persistent data or distributed processing.

Thus we have seen enormous advances in software technology over the past twenty years, but little of that technology is accessible in any usable form to application developers and maintainers. Most of it represents research results that have never been incorporated into practical tools. Practical tools that do exist are inaccessible because they are tied to a particular language, machine or operating system. What useful tools there are, are large monoliths which can seldom be used in cooperation or combination with other tools.

2. Towards a Solution.

We believe that significant progress requires a completely new vision of software development. The key to achieving this vision is what we call a *full spectrum language*, one which provides a base for technology development and integration instead of fixed specification and implementation mechanisms. A sketch of the goals of full spectrum languages and how they relate to wide spectrum languages is presented in section 2.b. In section 2.c, we discuss the key technical requirements for full spectrum languages. The emphasis is on showing how integration of critical technologies is enabled by lifting some of the fundamental but unnecessary restrictions of current languages. In section 3, we discuss the specific activities we plan in pursuit of a practical full spectrum language.

2.a. The Vision.

A number of us have had a vision since the late 1960's that the world of

computation can be different. The vision is of a world in which all aspects of the requirements, design, and implementation of an application are captured in an automated system, and in which new technology can be gradually captured and exploited by the system. We foresee a world in which limitations on our ability to mechanize translations will not limit our use of effective specification mechanisms, and in which the software designer is allowed to contribute to the design at all levels of abstraction, but is required to contribute only at enough levels so that the specifications, in combination with the automated system, are sufficient to produce a correct solution. As a result, new software technology will actually be transferred to practice, and new software tools will typically be better than their predecessors in some way, and more importantly will be as good as their predecessors in all ways.

2.b. Full spectrum Languages.

Full spectrum languages offer the hope of ushering in such a world by exploiting a variety of existing technologies as well as incorporating new technology as it becomes available. They offer the potential for capturing requirements, design and implementation in a common formal framework to the advantage of all manual software activities and automated tools. Finally, they offer the potential for growth to new applications, to new design and specification technology, and to new implementation technology without having to develop additional languages.

Our concept of full spectrum languages rests on the hypothesis that all languages can be composed from a relatively small number of semantic fragments according to certain laws of combination. Soundness of a language stems ultimately from the stability of its structure, according to those laws. Hence we see language design as being akin to chemical engineering, or molecular physics.

A full spectrum language is one that is based on the semantic fragments and laws of combination. *More importantly, these elements are exposed and available so that the language can be expanded and adapted in response to our increasing understanding and knowledge of software processes.* As with natural languages, new notations and forms of abstraction can be incorporated in the language as needed, thereby preventing needless complexity from crippling our ability to solve problems. Also like natural languages, old concepts, notations, and results can be reinterpreted in new contexts, leading to new unifying abstractions. The practical consequence of this capability is dramatically increased potential for sharing and reuse of software knowledge.

Our ideas about full spectrum languages have evolved from our attempts to formalize and consolidate the software development techniques we have been using for building a distributed Ada language system over the past three years. Specifically, the Ada compiler is organized around a collection of knowledge bases containing formal information about a set of abstraction mechanisms and

specialized instances of those abstractions. Some of these define general concepts and mechanisms of computation. Others define specific features of the Ada language, in terms of these general concepts. Still others contain general information about how to derive implementations (and, ultimately, target code) from the combination of Ada source code and the compiler's knowledge of Ada, the target machine, flow analysis, optimization, and so forth.

Our experience with characterizing all parts of the language system in this uniform framework, although somewhat *ad-hoc*, gives us great confidence in the soundness of the basic ideas of the full spectrum language approach to software engineering. Moreover, we have witnessed many of the benefits which we are projecting for full spectrum languages within the narrow confines of the Ada project, including the continual generalization of mechanisms and concepts to broaden their scope of applicability and consequently reduce the size and complexity of the compiler. At the current stage of development, the compiler takes only around 20,000 lines of formal description, and produces code that is comparable to or better than that produced by many commercial optimizing compilers for much simpler languages, such as C and Pascal!

Full spectrum languages are quite different from wide spectrum languages as we know them [DGL*79,DSS81,Che84,GLB*83,Mol84,SS83,Wil83]. A full spectrum language is a vehicle for software technology integration. As such, it need not initially implement any specific technology beyond what is required for a modest core of primitives and integration mechanisms. It must also provide an adequate set of abstraction mechanisms even if their implementation cannot be fully automated now. The primitives must be adequate for synthesizing the technology required by any application, and the integration mechanisms must allow any implementation technology to be absorbed by the language (without change to the language) so that it can be shared and reused.

Wide spectrum languages have different, though complementary, goals. They seek to integrate existing technology to allow specifications at a variety of levels, together with means of analyzing and transforming such specifications both within and between levels. Such technology is very promising, and may eventually lead to significant gains in programmer productivity and reliability of implementations. However, as long as such technology is couched in terms of fixed specification languages, transformation technology, and implementation mechanisms, software practice will not be able to absorb further advances, or economically exploit existing implementation technology. If wide spectrum languages were developed within a full spectrum language, however, each could be of great benefit to the other.

As a simple illustration of how a full spectrum language might be applied, suppose a user wants to use equational program specifications, and suppose that the

type "equation" already exists in the technology library, but there is no existing means of processing a set of equations to get an implementation. The user extracts the relevant algorithms from the literature on equational programming, and writes a function taking sets of equations as input and yielding sets of functions as output. Suppose now that another user has developed an equational simplifier and installed it in the library. The first user can then write applications using equations as the source, and pass them through the simplifier to perform some optimizations before translating them to operational implementations. Still later, another user adds a facility for verifying existing implementations against equational specifications, thereby enabling existing applications to be optimized using the previously developed equational optimization technology.

Notice that there is no language requirement that any existing or future application use any of this equational technology, nor would any user need to learn about it in order to continue using the language as before. But all of it would be available to any programmer who needs it. Moreover, parts of an application might use some of it, and other parts not, at the discretion of the application developer. Most importantly, however, the language implementation is completely indifferent to whether it gets the semantics of a function by compiling a function body or by compiling a set of equations, or by any other means, because all it cares about is the internal semantic representation of functions, which is independent of the surface features used to generate them. The implementation is also indifferent to the source of the transformation rules it applies during optimization, so the user's equational transformations are readily integrated as part of the language implementation. The implementation is also indifferent to the source of the program analysis procedures it carries out to verify the semantic integrity of programs, so the equational specification checker can be integrated as an extension of the usual type-checking mechanism.

Insofar as a full spectrum language enables the formal expression of programming knowledge and has some capacity for "learning", it is a knowledge-based system. It differs from typical knowledge-based systems, however, in that its knowledge is formalized within a strong type discipline. Of course, informal knowledge can still be represented and manipulated by programs written in a full spectrum language, but such knowledge cannot be fully integrated with the language system itself. Put another way, the knowledge base of a full spectrum language consists only of knowledge which is formally expressed and established. There is also a big difference between *representing* knowledge about things which cannot actually be manipulated and *understanding* those which can. For instance, one can easily represent knowledge about concurrent processes in a knowledge representation language like KL/ONE, but no amount of effort will enable a KL/ONE programmer to create a concurrent task, because concurrency is not a basic component of KL/ONE.

Full spectrum languages can also be contrasted with the extensible language developments of the late 1960's and early 1970's. Several of the extensible languages (most notably PPL at Harvard) were quite successful as programming languages, but none of them were successful as extensible languages. Their mistake was to divide the programmer's task into two activities having a very different character and set of concerns: defining syntactic and semantic extensions tailored to the application domain, and writing the application, using those new features in combination with the preexisting ones. Because of this sharp division, the skills and knowledge required for one task could not be readily applied to the other, and it was found to be psychologically impossible to think effectively about both tasks simultaneously. Consequently, the potential benefits of the meta-features were ignored in favor of getting the job at hand done. In contrast, a full spectrum language provides a uniform system in which there is no distinction between the facilities for describing applications and those for describing the descriptions.

2.c. Technical Requirements for the Language.

Any effort to develop a full spectrum language will be primarily one of understanding, interpreting, coordinating, and exploiting large amounts of existing theory and practice from a variety of diverse areas, so that that existing knowledge can be integrated and engineered into the design of a sound and practical full spectrum language. It will involve minimal development of new theory, but will require interpretation of results from a variety of domains including formal types, programming languages, program analysis, design and requirements specification, programming environments, configuration control, object-oriented systems, compiler construction, operating systems, and databases. It will require considerable analytical and empirical investigations of the effectiveness of various technologies and of how they can be used in combination. The design effort will require considerable engineering skill in both language design and compiler construction.

In what follows, we begin by clarifying the intended application environment of our language, stating a number of assumptions we have made about that environment. Following that, we discuss the general requirements of the language design. Lastly, we enumerate some of the specific design goals dictated by the application environment and the general requirements.

2.c.i. The Application Environment. We make the following assumptions about the applications for which the language is intended. Applications are large and may involve hundreds of thousands to many millions of lines of code when implemented in conventional programming languages. Applications will involve many people over many years in their design, implementation and maintenance. Applications will continually undergo changes to their requirements, functionality,

equipment configurations, and design throughout their lifetime. The domain of concern for a full spectrum language, like that of its users, cannot be limited to the internals of a single program or compilation unit, but instead must encompass the entire environment of application development and execution. Neither can the domain of concern be limited to a single stand-alone infinitely reliable uniprocessor, but rather a dynamically changing world of heterogeneous multiprocessors often without shared memory, connected through networks and removable media, and geographically dispersed. It is assumed that the hardware and software components of a system are often unreliable, that the data entering systems are often erroneous, and that applications must function effectively in the presence of such problems. It is assumed that applications do not end at the edges of a program, and instead involve management and control of data and resources that persist beyond the individual programs that manipulate and modify them. It is assumed that applications may run forever, that they must be updated and modified while they are running, and that system and data integrity must be maintained in the presence of such change. Note that these assumptions are consistent with any programming-in-the-large application, and are indistinguishable from what DoD calls embedded computer applications.

2.c.ii. General Requirements for the Language. The technical requirements of a full spectrum language are in many respects similar to those of any other programming language, so in this section we limit ourselves to making an observation and then propose a small set of specific goals for a full spectrum language.

The observation is that a full spectrum language must be concerned not with satisfying the requirements of any given application or set of applications, but instead with ensuring that requirements of any application, whether foreseen or not, can be expressed in the language by its users once its design is completed. Thus, the requirements must reflect (a) the needs common to all applications and (b) the need to encompass unforeseen user requirements. The requirements should not dictate features specific to particular applications or extant technology.

Given our assumptions about the application domain, it must be possible to develop the technology of wide spectrum languages in a full spectrum language. When this is done, however, the width of the spectrum must be limited only by our ingenuity, not by the language itself. The demands of extendible wide spectrum technology thus impose two broad requirements on the design of a full spectrum language. The language must be simultaneously a specification language, an implementation language, and a programming environment. It must also provide binding mechanisms which allow it to be open-ended, permit incomplete specifications at all levels, and enable the implementation to detect and exploit

binding time information to gain efficiency.

A full spectrum language must be broad as well as wide. That is, not only must it span many levels of abstraction, it must also cover all aspects of applications. Current wide spectrum languages ignore or deal only inadequately with issues of concurrent, real-time, and distributed processing, error detection and recovery, and persistent data. These concerns impose a second set of requirements on the design of a full spectrum language. Most prominently, to be practical and useful the language's domain of concern must include the entire environment of application development and execution. It must incorporate a pervasive concern for the integrity of everything within that domain. And, the language must have a generic organizational capability.

2.c.iii. Specification Mechanisms. A full spectrum language must be able to capture the goals and intent of its users in a way that can be understood and exploited by compilers and other automated tools for the language. Over and above the implementation details of an application, a full spectrum language must be adequate to describe its goals, abstract design decisions, and execution environment.

Goals include such things as performance constraints, reliability, and optimization criteria, in addition to functionality.

Abstract design decisions include various kinds of commitments to such things as the decomposition of the solution into components, their logical properties, representation, and the engineering rationale for those decisions, including relevant analyses of alternatives.

Execution environment specifications include expected operating characteristics, including target hardware and software properties, as well as the expected ranges of external data, what action to take for out of range data, and the expected frequency of bad data. To take one example, a certain automated teller system knew enough to check the validity of bank cards and to confiscate cards with invalid numbers, but did not know enough about the application to recognize and report an exception when an inordinate number of invalid cards appeared. Thus, the machine confiscated 2000 cards in less than two hours [Neu85]. We need to be able to describe enough of the expectations of an application's environment that a compiler can automatically provide checking and reporting of statistically unexpected situations.

The point is that the language must enable users to express information serving a variety of purposes. Moreover, the variety and details of specifications are likely to shift continually as new uses for specifications are recognized and the corresponding technologies are invented. As a consequence, it is necessary to allow

information that is usually kept together by syntactic conventions to be factored into several related pieces and assembled as needed. It is also necessary to allow new kinds of information to be introduced as factors in the description of applications.

The current practice in programming languages is to either force certain information to be presented together or to force it to be presented separately. Such restrictions are usually imposed by the concrete syntax of the language. In Ada, for instance, a package is factored into a specification and a body, whereas the declarative part of a package body cannot be separated from its sequence of statements. The factorization of packages has great advantages, for instance in allowing compilation units that depend on a package to be compiled before the package body is defined, and making it unnecessary for them to be recompiled if the body is changed.

However, Ada imposes some restrictions on packages which limits their utility. For instance, there can be at most one body (i.e. implementation) for any package, instead of multiple bodies which could be selected based on efficiency considerations in the application. Ada does not allow additional information, such as axiomatic specifications or performance characteristics to be attached to packages. Instead, some external, difficult to integrate, mechanism such as Anna [LV85] annotations must be used.

Thus the Ada package specification and body are simply two special syntactic forms for specifying particular kinds of semantic information about a certain kind of semantic object. Semantically, a package defines a collection of objects within a scope which regulates their visibility and determines the access of the entities defined within the package to each other and to entities defined in other scopes. There are any number of alternative methods whereby a user could supply additional semantic information to the implementation, and no technical reason for limiting them to a predetermined set of mechanisms. For the user to exploit the alternatives, however, the language must provide access to the basic operations – constructors and selectors – on objects of type *package*. Following this line to its logical conclusion, all semantic concepts of the language must be first class citizens.

When programmers are not restricted in the method of synthesizing semantic components of a program, it becomes immediately possible for them to factor that information in any way. In particular, applications can be specified using any collection of specification languages and mechanisms, provided that these mechanisms include the means of collecting and deriving the information required to develop an implementation from those specifications.

By use of such specification mechanisms it becomes possible to specify requirements, functionality, performance, design and optimization criteria and

decisions, and any other information required to support automated or manual software development techniques. For instance, it should be possible to attach complexity types to code fragments, composing them by the rules of order arithmetic to document and automate, insofar as possible, analysis of the asymptotic performance of applications.

Put as simply as possible, a language can support arbitrary modes of *specification* by not imposing any syntactic restrictions on the form of specifications, while providing access to, and enforcing restrictions on, the semantic structure of all concepts.

In order to relate specifications to computation, the core concepts of the language must include basic *implementation* mechanisms such as function application, assignment, and rendezvous. The essential requirement here is that the language be operationally complete, in the sense that it must provide access to all of the important operational capabilities of computing machines, now and for the foreseeable future. In particular, there must be synchronous and asynchronous mechanisms for concurrent programming, communications, access to hardware exceptions and interrupts, and mechanisms for real-time programming, including timeouts based on deadlines, in addition to the more common mechanisms of sequential processing.

Finally, there must be mechanisms for managing and accessing information, and controlling its definition and application; these are the underlying mechanisms of *programming environments*. Semantically, what is required are the building blocks underlying the visibility, extent, and inheritance rules of languages, but broadened to include the needs of persistent data, in combination with the control mechanisms cited above.

2.c.iv. Binding Mechanisms. The open-endedness of specification mechanisms is only one example of the importance of binding mechanisms in a full spectrum language. We discuss some others here.

A full spectrum language must support incomplete specifications. For instance, it must be possible to compile and execute a program even if it is in some respects incomplete. The desire for incomplete specification arises from three sources. First, it must be possible to test and exercise applications before they are fully designed and implemented.

Second, it must be possible to avoid overspecification when a design or implementation decision is arbitrary. Existing programming languages require that programs be complete. Consequently, it is impossible to leave decisions to the compiler even though the compiler may be able to make a better choice.

Finally, as the level of languages grows, programs in a next generation language can be viewed as incomplete specifications for programs in languages of the previous generation. We wish to provide a language system in which the level of use of the language can grow without having to invent a new language (or class of languages) for each gain in level.

We view incompleteness merely as an extreme form of late binding. That is, the language will not require anything to be bound unless and until it is needed by some computation. When a demand is generated for an unbound entity, an exception is raised which can be handled either by the user (perhaps responding by creating an appropriate binding and continuing) or by a system default action.

The ability to defer commitments indefinitely is especially important for the language *qua* programming environment. Not only do the details of implementations change over time, but so do the characteristics of the devices they are controlling, the machines on which they are implemented, the functional requirements of the application, the general character of their execution environment, and their performance goals. Not only must there be support for orderly change, but the changes must often be accomplished while the implementation remains operational. We cannot shut down an electric power network, a nuclear power plant, a medical life support system, or the environmental control system of a space station while software changes are being made.

On the other hand, certain components of such systems often have severe reliability and performance requirements, and are typically a critical part of a larger system whose primary purpose is not computation, whence the term "embedded computer system". To meet such performance requirements, we must have compilers which can recognize and exploit early binding decisions so that the application does not pay a performance penalty for unused generality (i.e. late binding capability).

Good general techniques for recognizing early binding opportunities have recently been developed [HY86,Jon87] though their applicability to realistic languages has yet to be proved. However, it is clear that effective detection and exploitation of early binding opportunities requires additional information to be supplied to the compiler by the programmer, since without such information binding time analysis is ineffective or intractable.

2.c.v. The Domain is the Environment. Probably the greatest impediment to effective automated systems is lack of accessible information about the application and its intended execution environment. Conventional programming languages

limit their scope of concern to implementation issues that lie within a particular compilation unit. Applications, on the other hand, must be concerned with data objects that persist beyond the invocations of the programs that create and manipulate them, and must deal with execution environments having unreliable hardware, software, and communications. They must cope with multiple processors, distributed networks, and dynamic changes in their requirements, design, functionality, equipment configurations, users, data characteristics, and implementation hardware. These aspects of applications are currently managed outside the program with only the resulting implementation decisions presented within the program. A full spectrum language must encompass the entire environment of the development and operation of applications.

We can identify certain semantic requirements dictated by these concerns, such as mechanisms for concurrency, persistent data, and distribution. Others, such as describing hardware configurations or the role of users, are less clear, as are the ways that such information can be used. An important goal of our research is therefore to clarify the issues in this domain and understand their implications for language design and implementation.

2.c.vi. Pervasive Concern for Integrity. Integrity of all aspects of applications must be a pervasive goal for a full spectrum language. It matters little how good the other aspects of an application are, or how fast it runs, or how much it encompasses, if it produces results that are incorrect or unreliable.

By integrity here we mean something akin to the metaphor of "authentication" with regard to type systems [Mor73]. Specifically, the language must provide a strong, enforced, typing mechanism which applies not only within individual programs, but among programs. Type integrity must be maintained even when data is shared among programs and persists beyond the invocation of the program which created it.

Another form of authentication applies to the integrity of implementations. We take the position that all hardware and software systems are inherently unreliable, and that applications must be designed, implemented, and executed with that understanding. Thus languages and compilers must provide effective error detection and recovery mechanisms. Although a variety of execution errors can be detected and handled by default mechanisms, full integrity of application data cannot in general be guaranteed in such recovery. Consequently, applications must not be denied access to information about any detected errors, and applications must be informed of any errors which may have corrupted their data.

Finally, a full spectrum language must allow mechanisms for authentication (in its usual sense) to be built. It must be possible to provide useful and effective

mechanisms which safeguard the system from deliberate or accidental corruption to the application properties of protection and security.

2.c.vii. A Generic Organizational Capability. Any language which encompasses the environment beyond the bounds of individual compilation units and applications must support the organization and management of large quantities of persistent data. There must be a mechanism for organization and retrieval of such data from the persistent data store. The organizational mechanism (i.e., a logical directory system) must be independent of the types of data stored in it, must be capable of housing values of user-defined types, must be independent of any physical organizational structure, and must be compatible with disjoint and geographically distributed implementations whether of networks or removable media. The directory mechanism must permit logical sharing with the same data appearing in multiple directories. It must be adequate for supporting user-defined organizational mechanisms, as well as a program library mechanism for the language itself. It must provide mechanisms for efficient sharing of and concurrent access to data. It must provide mechanisms for control and management of the physical location of data on peripheral memory and removable media.

Note that neither conventional file systems nor database management systems satisfy these requirements. They tend to be very limited in the types they support. Their logical organization tends to be bound to the physical organization of their implementation, and is usually optimized for very specialized retrieval characteristics which cannot be modified by the user. They often have inadequate concern for data and type integrity. And, finally, they are not well integrated with programming languages.

3. The Agenda.

Our agenda for exploring the realm of full-spectrum languages involves four interrelated activities: a) investigation of key research issues, b) design of the language, c) prototype implementation of the language, and d) review and evaluation. The investigations into key research issues will answer the difficult scientific and engineering questions required for the language design and prototype implementation. The questions themselves will be iteratively refined as they become better understood through feedback from the design and implementation processes. The key research issues and language design efforts will both involve empirical investigations. It is expected that many of the experimental results will be incorporated either directly or with modification into the implementation. The prototype itself will be the primary test of the feasibility, and measure of the cost, of the language constructs.

3.a. Investigation of Key Research Issues.

Most of the key research issues are fairly well understood in isolation, so we expect a minimum of theoretical work in this project. The difficulties arise when the existing results are integrated with other issues in the design and implementation of a language, when the simplifying assumptions of the original work must be removed, and when the requirements are extended to include those of generality and efficiency. On the other hand, we do expect a fundamentally new concept of software development to emerge as a result of our efforts, one which will undoubtedly raise many new theoretical questions.

Some of our current thoughts on several of the key research issues are given in the following subsections. They arise directly from the language requirements set forth above. In particular, research into abstract type mechanisms will help meet the requirements for a specification/implementation language and for open-endedness. Concurrency and real-time issues must be resolved to allow the system to expand its domain into the environment. Error detection and recovery mechanisms are needed to ensure system integrity. Finally, research into the persistent data problem is needed if we are to meet the requirements for a generic organizational capability and persistent type integrity.

3.a.i. Abstract Type Mechanism. A full spectrum language must deal with all of the concepts involved in the engineering of a software product. The primary purpose of the abstract type mechanism is to facilitate the formal definition of concepts and to ensure that concepts are composed in a coherent manner. The formalization of concepts provides, among other things, important information that can be exploited by the language system to optimize applications. We use the term "concept" in this section to avoid the restricting connotations a reader might have for any of the terms "type", "abstract type" or "abstract data type".

The design of a type system for a practical full spectrum language has to strike a balance between formal power and elegance, on the one hand, and effective exploitation of current technologies, on the other. Ensuring that we are not locked into current technology requires certain essential features of a rather formal, theoretical nature. Effective exploitation of current technology requires that these key features be embodied in a set of core concepts which cater to the practical needs of software engineering, language implementation, and computer architecture.

We expect that the core concepts will continually be supplanted and augmented by newer ones, in a completely transparent way. Old applications need not be modified or rewritten; new applications do not have to use the old technology. Achieving this kind of unbounded upward compatibility is the driving force behind

the design of the formal aspects of the type system.

The practical aspects of the design are informed by our experience as software system designers and implementors. In practice, we take Ada as the starting point for language design, because Ada deals with more important issues for programming-in-the-large than any other current language.

The essential theoretical quality of the type system is that it must be reflective. By this we mean that the core concepts are defined internally (i.e. within the language, in terms of the other concepts of the language). It follows from this that all concepts will be first class citizens (i.e. the type system will be higher-order). It also follows that all of the rules of composition (type formation, computation, deduction, etc.) must be first class (i.e., composition is itself an abstract concept; this gives us a categorical outlook).

What is a concept? It is a collection of information. An Ada package, for instance, is a concept. The package specification provides information about the external interface to the package, and the package body provides information about how the entities declared in the specification might be implemented.

All such packages are in turn examples of the higher-level concept *package*. This concept defines the rules of formation for packages in general, including the relationship between specifications and bodies.

The package concept is representative of the best technology for data abstraction that was available at the time Ada was designed. However, we can now see that several exciting capabilities are missing from the Ada language. There is no technical reason why polymorphic type inference could not be applied to a package body to automatically derive a package specification. There is no technical reason not to allow more than one body for a given package, as long as the concepts used to select representations when generating code can be extended appropriately. Nor is there any reason to require a user-defined body when it is possible to derive a representation from an interface specification together with, say, an axiomatic specification. Nor, for that matter, is there any technical reason not to allow dynamic creation of packages and instances *a la* Smalltalk [GR83], perhaps with multiple inheritance, too.

The point is that the concepts and information used to develop an application should be constrained only by the available technology, not by the language. The available technology is embodied in a collection of concepts, each of which is a collection of related bodies of information. In a full spectrum language, the concepts include the abstract notion of "concept", and ways of forming new concepts as

integral extensions.

Our notion of concept is closely related to the Theory of Constructions (TOC) [Coq85, Hue87], in that the formal properties of syntax (their "propositions") are determined by a constructive semantics (their "proofs"). This is a very powerful, but theoretical (read perhaps not practical) base. A similar notion underlies realizability models in logic [Sco87]. However, our system is more powerful than TOC in a number of important ways.

- The syntax of a concept can be any complex structured interface, not just a proposition.
- All concepts are internalized; in TOC, the basic rules of construction are fixed externally.
- As a result, new techniques for defining and manipulating concepts can be introduced by the user.
- We provide a core of concepts which are important for practical software development. Some of these, such as tasking and exceptions, have no place in the simplified world of TOC. Others, such as assignment statements and control structures, make the capabilities of real machines available to the user (TOC is restricted to λ -calculus to make it theoretically tractable). Still others, such as packages and subprograms, are essential tools for managing the complexity of large, long-lived, continuously changing applications. A possible list of core concepts follows.

declarations	types	time
functions	tasks	scopes
packages	exceptions	discrete types
booleans	arrays	records
sets	variables	pointers
lists	persistent data	bindings (in, out, in-out)
numbers (integer, real, complex)		

The concept *concept*² described in this section will guide our research in other areas. In particular, our investigations into concurrency, error recovery, and persistent data are best viewed as type-theoretic investigations into the most prominent concepts of the full spectrum system.

²It is well known that any system with a type "type" is *logically* inconsistent [Coq86,MR86]. This does not particularly bother us, however, since we do not impute any logical content to concepts in general. Rather, logic is itself a concept, one which has many uses in software development. But it does not rule us.

3.a.ii. Concurrency, Real Time, and Distribution. Concurrent execution is required in many applications; it is also an increasingly important programming paradigm, whereby a large task can be decomposed into smaller, communicating, tasks. With concurrency come the issues of real-time behavior and the distribution of applications across a collection of physical processors.

Most research in concurrent language constructs has concentrated on operational mechanisms for synchronization and communication. These range from low-level mechanisms like semaphores to higher-level mechanisms like rendezvous. Abstraction and encapsulation of interacting agents have also been studied, in a number of guises, among them monitors[Hoa74], extended CLU classes[LAB*81], Ada tasks [ALRM83], and CCS [Mil80].

As usual, new technology in these areas cannot be integrated into our current languages because those few languages that have concurrency at all are limited to a fixed set of primitives. In our full spectrum language we will have *concurrency types* which can be composed and manipulated like all other concepts. With concurrency types a programmer can specify the abstract temporal properties of components, and type checking will ensure that temporal concepts are composed only in meaningful ways. This will benefit concurrent programming by drastically reducing the amount of analysis and testing required to validate concurrent software.

A theoretical base for concurrency types is suggested in the recent work of Girard on Linear Logic [Gir86]. We plan to cast these ideas in practical form, drawing heavily on our own experience in the design, analysis, and implementation of a distributed run-time system for Ada.

Our experience with distributed Ada has led us to formulate a number of core concepts for concurrency. These include the separation of tasks and services (entries); extending rendezvous to full transactions (thereby allowing communication between tasks engaged in a rendezvous); asynchronous communication; task abstraction (similar to system abstraction in CCS); and generalized mechanisms for task dynamics and sharing. Early versions of some of these ideas are elaborated in [FW86, FM86].

In the area of real-time behavior, we have developed concepts of observation, time, and reaction which can be used to specify arbitrary protocols for real-time interactions. These concepts are the basis for our current implementation of Ada's real-time components. We believe that most of the current confusion surrounding the real-time features of Ada (cf. [VM86b]) can be eliminated through the use of these concepts.

In the area of distributed systems, the major issues are resource allocation and sharing, error detection and recovery, and persistent data (including removable media). These issues have all been discussed in the foregoing sections, with the exception of resource sharing, which we shall discuss presently. The generalization of these issues to distributed systems and integration into the language will be the focus of our research in these areas.

In this effort we can draw on previous work in distributed databases and operating systems for such things as deadlock detection, extended transactions, rollbacks, dynamic reconfiguration, and so forth. The problem, as always, will be to find the right basic concepts and integrate them into the language core.

The problems of resource sharing in a distributed system have never been adequately addressed in a language. The mapping of software to hardware components is constrained by the operating environment requirements of the software. In particular, the patterns of resource sharing within the software determine how the software can be partitioned.

If two tasks share a variable, for instance, they should be allocated on processors that share a memory, because the only justification for shared variables is high performance, which can never be the result of simulating shared memory. (On the other hand, if two tasks communicate but happen to be placed on the same processor, the runtime system should exploit that fact and use shared memory to speed the communications.) Similarly, subsystems which share a file should be allocated on the same local network as the file server.

Such considerations have led us to the concept of *virtual processors* as basic building blocks of distributed software. Integrating this concept into the language will make it a more effective tool for distributed software design.

3.a.iii. Error Identification, Analysis, and Recovery Mechanisms. A system is reliable to the extent it is correct, makes proper use of computing resources, behaves predictably and appropriately when hardware or software components fail, and can be operated reliably by its users. As in any other engineering discipline, reliability is achieved by applying scientific design principles, by including safeguards and contingency mechanisms in the design, and by testing. Each of these activities contributes to our confidence in a system in different ways, and makes up for some weaknesses of the others.

Each activity has its own style of detecting, analyzing, and recovering from errors. Type checking and related static analysis mechanisms are the tools of scientific design. Exceptions and exception handling mechanisms are used to

safeguard against possible flaws in the design or problems in the operating environment. Various forms of instrumentation (debuggers, performance monitors, psychological experiments) are used for testing operational systems and their components.

In each of the three reliability activities, the same five language issues arise: control, visibility, binding, resource allocation, and abstraction. Moreover, all five language issues relate to all three aspects of error handling: detection, analysis, and recovery.

In instrumentation, for example, control issues arise in each aspect of error handling: the triggering of probes (detection), the ability of probes to alter the control flow of the system being measured (during analysis), and the transfer of control from probes back to the system, when this is meaningful (recovery).

Here are some illustrations of how the language issues of visibility, binding, and resource allocation arise in the context of instrumentation. The environment in which a probe executes determines what user-defined types and data it can access (if any), or whether certain run-time system information is visible. Binding time determines such things as whether breakpoints can be installed interactively, or have to be "compiled in". In performance instrumentation, resources must be apportioned among the observed system, data collection, data reduction and analysis, and presentation and user interaction (if any) so as to minimize intrusiveness.

The most difficult problems here are in the area of abstraction. Ideally, one would like to say "measure the X of system Y", and have any necessary probes, data reduction facilities, etc., generated, installed, and run automatically. Or, better yet, "determine how well system Y's behavior matches hypothesis Q", thereby tying testing back to design specifications. The realization of these ideals requires mechanisms for defining and manipulating abstract properties of systems.

In particular, we need mechanisms for detecting, analyzing, and recovering from abstract errors. The bank card system which confiscated too many cards provides a good illustration of what we mean by an abstract error. We cannot reasonably expect to anticipate every possible misbehavior of a system at every point in its execution and install (often redundant and wasteful) safeguards at each point manually. We *can* hope to describe deviations from a system's expected behavior at the application level, and have safeguards generated and installed at appropriate points automatically.

Clearly, the mechanisms we envision for abstract error definition, detection, analysis, and recovery can and should be defined in terms of the control, visibility,

binding, resource allocation, and abstraction concepts provided by the language core. That is, we should not have to reinvent these concepts. Nor should we have to duplicate the design efforts of those who have already created some good error handling technology [AW85,Cou81,Joh83,KC86,KP82a,LS79,MY86,RLT78,YB85]. Rather, we see these concepts as the key to an unprecedented systematic formalization of error handling ideas, and as the avenue for integrating them into our language.

3.a.iv. Persistent Data and Type Integrity. The full spectrum language will support the production of reliable, efficient and reusable software over a range of applications and will act as a cooperative element in an integrated software development environment. A sophisticated type system such as the one discussed above is needed to formalize the properties of such diverse tools as compilers, compiler generators, debuggers, analyzers and project management assistants, which act on the types comprising languages, programs, specifications, designs, test plans, PERT charts and so forth. The type(s) of a datum determine what tools can create or manipulate it and the relationships in which it can participate.

The concept of object allows instances of values to be named. The object naming mechanism assigns a unique, universal, location independent name to a value to create an object. Thus, persistence is a property (concept) of a particular kind of datum, namely objects. An object is persistent if it outlives the particular invocation of the program or tool that created it. It is the responsibility of object management mechanisms of the full spectrum language to ensure the type integrity of persistent objects and to oversee their creation, destruction and access, both through space (since the environment may be distributed over multiple networks and include removable media) and through time (because some data will be persistent).

Traditional file systems and databases have each addressed some aspects of object management, but each has shortcomings with respect to either persistence or typing. File systems provide persistence, but have no way to enforce type integrity over invocations of tools and manipulations by human users. Programming languages more closely approximate object management typing needs, but provide little support for persistence. Databases achieve typed persistence, but their type systems are inadequate for the variety and complexity of data needed in software applications and software development environments. They especially do not satisfy the need for the typing system to evolve over time and to be self-describing.

The object management aspects of the full spectrum language include a number of requirements, concerns and assumptions beyond those of persistence and typing. The language must provide support for these concerns, of which a sample

follow.

- Persistent data is the key mechanism that underlies the entire lifecycle. There must, therefore, be support of change and the ability to define spheres of activity. Partial information must be tolerated. There must be support for both consistency (e.g. invocation of tools to establish or re-establish some relation among a given set of objects) and inconsistency.
- Persistent data is the key mechanism that underlies a software development environment and variety is the most striking characteristic of the objects in such an environment. This is perhaps especially true when considering object granularity. The object management mechanisms must expect, and remain viable for, objects of a wide range of both physical and logical granularity from the very small to the very large.
- The objects will include active, passive and concurrent entities.
- There will be pre-existing tools and objects for which migration paths must be established, whenever feasible.

3.b. Language Design.

Naturally, good language design practice is required in the design of any language [Wei71, Hoa73, Iron76]. What constitutes good design depends in part on how, by whom, and for what purposes the language will be used. Some design guidelines for full spectrum languages are the following.

Because the applications are varied and many, it is necessary to provide a small number of highly composable mechanisms, instead of a large number of mechanisms specialized to the intended applications. To retain simplicity in the language each primitive mechanism must isolate some unique functionality which is easily composable with the other primitives. Every effort should be made to avoid language features that will lead to psychological ambiguities in programs. The design should emphasize readability over ease of writing programs. It should emphasize the semantic integrity and completeness of the language. It should provide redundancy without duplication. It should avoid default mechanisms that obscure the meaning of programs. It should use syntactic and semantic features, wherever possible, which are compatible with the traditional notations and intuitions of mathematics, engineering, and computer science. The syntax should be conservatively extensible, to allow syntactic extensions that cannot cause confusion by redefining the parsing rules for familiar phrases. And, finally, the design must strive in every way possible to provide features that will in their use (during program execution) be only as expensive as is inherent in the generality of their use.

We can identify four major sources of inspiration and guidance: Ada, functional languages, object-oriented languages, and foundational theory. We look to each of these for specific contributions, as follows.

Ada will influence many of the practical aspects of syntax and program structure. This influence is both positive and negative; we seek to take the best from Ada and avoid its mistakes. Ada is currently the most complete compendium of features needed for practical application development. As such, we can use it as a baseline of capabilities for our full spectrum language; anything which exists in Ada should either be in the new language or preferably be easily defined and integrated. If this goal is achieved, it should be easy for programmers familiar with Ada to switch to the new language with a minimum of retraining.

Ada also contains the echoes of some very good ideas, both in its syntax and semantics. For instance, its near-unification of the syntax of record literals and actual parameter lists reveals an underlying commonality which should be recognized and exploited to make the language smaller and cleaner. The question is: how?

For an answer to this particular question, we turn to functional languages such as ML [GMW79], Amber [Car86a], Ponder [Fai83], and Miranda [Tur85], which have explored a number of alternatives for unifying data and parameter structures. Other desirable aspects of functional languages are the use of highly composable, small-grained components; ease of formal analysis and transformation; the functorial character of abstract data types and modules; and an extensive body of interactive/incremental implementation technology. The most important contribution of functional languages, however, is higher-order programming.

Programmers often face the dilemma of not knowing a good general solution to a problem, though a good solution can be generated for any particular case, given some additional information. Higher-order programming, in a language like common Lisp, enables the designer to implement an algorithm for deriving a good solution, but at the cost of getting an unacceptably inefficient implementation of that solution. A static language like Ada, on the other hand, provides efficient mechanisms which can be combined to obtain an efficient implementation, but at the loss of the general solution.

The problem is that the common Lisp programmer can't convey enough information about the application to the compiler for it to obtain an efficient implementation, while the Ada programmer must convey so much information about the details of a particular solution that the compiler is unable to abstract the general solution. In a full spectrum language, the programmer should be able to

communicate to the compiler, as part of the program, any information it needs to derive an efficient implementation of a specialized solution.

Typed functional languages enable more efficient implementations by including type information in programs. Types constrain the application, promote checking and representation decisions to an earlier point in the computation, and enable a wide class of optimization transformations.

The more intricate a type system is, the more information can be expressed. For instance, dependent types can be used to inform the compiler to represent a list as an array if the length of the list is known to depend on a numeric parameter. In the extreme, virtually any logical property that has constructive significance can be embodied in type information (at which point we say we are doing "logic programming").

The information a compiler needs isn't restricted to functionality, however. To cite a few examples, the criteria to be used in optimization, expected statistical characteristics of input data, and complexity measures of components can all be used to guide the compiler's selection of algorithms and data structures.

As language implementors, we know how to make compiler components that are driven by user-supplied information and are hence open-ended. What is less clear, is what high-level syntactic mechanisms should be supplied to enable the application designer to express information and convey it to the portions of the compiler that need it? This is the most difficult syntax design challenge we face.

Object-oriented languages, operating systems, and databases are currently experiencing the greatest experimental activity in the areas of inheritance mechanisms and persistent data issues, and so we look to them to supply perspectives and mechanisms in these areas. In particular, these languages contribute a third baseline of features, in addition to those found in Ada and functional languages.

The fourth source of inspiration for our design comes from the formal foundations of semantics, type theory, logic, and category theory as they relate to program development. These formal foundations we see as giving the formal outlines to such things as the type system, notions of component composability, computing with semantic components, and the functors relating the various concepts in the language. They give us insight into what is theoretically possible, as well as warnings about potential difficulties. Most importantly, the formal foundations tell us what properties the language as a whole must have in order to assimilate the mechanisms required by tools for formal program manipulation. It is through the application of such tools that we expect the real gains in software

productivity and reliability to be achieved.

Our thesis is that the features we include in the language core should be adequate for any application. As a partial validation of this thesis, the language will be completely self-defining. As a further validation, portions of this self-definition may be incorporated in the prototype implementation. In particular, use of the language to implement its own run-time system could test its ability to support systems programming, and use of the language to support its own development could test its ability to support software engineering. It should be noted that we already have experience with this approach, since we use it in our existing Ada language developments.

3.c. Prototype Implementation.

The purpose of a prototype implementation is twofold. First, it provides an operational description of the language, showing in greater detail than the language reference manual how the primitives of the language behave, how they interact, and how they can be composed. Secondly, it furnishes an existence proof of the language's implementability. As a functioning embodiment of the language's semantics, it confirms that there are no inconsistencies in the design, and gives some indication of the costs associated with implementing the language.

One possible scenario for a prototype implementation is to use common Lisp as a starting point. As more of the features of the full spectrum language become operational, we would hope and expect that parts of the prototype implementation effort would shift into the full spectrum language. However, we insist that the full spectrum language be used appropriately; we would not want to establish the bad precedent of writing Lisp-style programs in the new language, merely in order to obtain a complete bootstrap. The result would be an experimental facility suitable for a wide range of demonstrations of the principles of full spectrum language design.

3.d. Review of Preliminary Language Design.

Formal review by the community can help to raise confidence in the integrity of the language design and to raise confidence that the language meets its requirements. We expect such a review, involving a small number of highly qualified experts representing academia, industry and government, to be a vital part of any full-spectrum language effort. We do not, however, foresee the need or desirability of soliciting comments from the computer science community at large. The comments and suggestions of the reviewers would be incorporated, as appropriate, into the final language design. It is likely that a small number of reviewers would also assist in evaluating the reviews and in weighing comments that have opposing points of view.

Bibliography

Selected Papers by Members of Incremental Systems Technical Staff

- [BFS87a] Baker, D.A., Fisher, D.A. and Shultis, J.C., "Persistence and Type Integrity in a Software Development Environment", in *Persistent Object Systems: their Design, Implementation and Use*, Carrick, R. and Cooper, R., eds., Univ. of Glasgow and Univ. of St. Andrews., August 1987.
- [BFS87b] Baker, D.A., Fisher, D.A. and Shultis, J.C., "A Practical Language to Provide Persistence and a Rich Typing System", *Proc. Workshop on Database Programming Languages*, Roscoff, September 1987 (available as technical report from Univ. of Pennsylvania).
- [BH86] Baker, D.A. and Heimbigner, D.M., *Design Possibilities for Zeus: The Tool/Object Manager for Arcadia*, Technical Report CU-CS-318-86, University of Colorado, February 1986.
- [BHS86] Baker, D.A., Heimbigner, D.M. and Sutton, S.M. Jr., *Providing Programmable Relations over Software Objects in Aspen*, Technical Report, September 1986, University of Colorado.
- [BS86] Baker, D.A. and Sutton, S.M. Jr., *Exception Flow Analysis in Ada*, Technical Report CU-CS-319-86, March 1986.
- [BB84] Baker, D.A. and Baxter, A.Q., "Computer Science Fundamentals II", lecture notes and annotations, instructor's guide, exercises and solutions, IBM IS & CG University Program course, 1984.
- [BO84] Baker, D.A. and Osterweil, L.J. "Critics: An Active Approach to Tools and Environments", Technical Report CU-CS-285-84, University of Colorado, December 1984.
- [Bak83] Baker, D.A., "EASE - An Extensible Abstract Structure Editor", Technical Report CU-CS-250-83, University of Colorado, 1983.
- [Bak82] Baker, D.A., "The Use of Requirements in Rigorous System Design", Ph.D. Dissertation, University of Southern California, 1982.
- [CF82] Carlson, W.E. and Fisher, D.A., "First Complete Ada Compiler Runs on a Micro", *Mini-Micro Systems*, September 1982.
- [CK86] Choi, J.W.C. and Kimura, T.D., "A Compiled Picture Language on Macintosh", *Proceedings of ACM Conference on Personal and Small Computers*, San Francisco, California, December 1986.
- [FFR72] Fisher, D.A., Faber, U. and Reigel, E., "The Interpreter: A Microprogrammable Building Block System", *Proceedings of the Spring Joint Computer Conferences 1972*, AFIPS Vol. 40, May 1972, pp. 705-723.
- [Fis81] Fisher, D.A., "Design Issues for the Ada Program Support Environments, A Catalogue of Issues", *Science Applications, Inc. paper*, SAI-81-289-WA.
- [Fis80] Fisher, D.A., "Ada, The United States Department of Defense High Order Language", *AGARD-ograph on Guidance and Control Software*, Advisory Group on Aerospace Research and Development, North Atlantic Treaty Organization, May 1980, pp. 5.1-5.9.
- [Fis78a] Fisher, D.A., "DoD's Common Programming Language Effort", *IEEE Computer*, Vol. II, No. 3, March 1978, pp. 24-33.

- [Fis78b] Fisher, D.A., "Steelman", Department of Defense Requirements for High Order Computer Programming Languages, High Order Language Working Group (HOLWG) Report, June 1978.
- [Fis77a] Fisher, D.A., "A Common Programming Language for the Department of Defense-- Background, History, and Technical Requirements", *IDA Paper P-1263*, May 1977.
- [Fis77b] Fisher, D.A., "Floating Point Computational Facilities for a Common Programming Language for the DoD", *Proceedings of the 1977 Army Numerical Analysis and Computers Conference*, March 1977, pp. 585-596.
- [Fis77c] Fisher, D.A., "The Common Programming Language Effort of the Department of Defense", *A Collection of Technical Papers from the AIAA/NASA/IEEE/ACM Computers in Aerospace Conference*, November 1977, pp. 297-307; Received Thomas R. Benedict Memorial Award for best paper.
- [Fis75a] Fisher, D.A., "A Common High Order Programming Language for the Department of Defense", *Proceedings of the 10th Anniversary Symposium*, Computer Science Department, Carnegie-Mellon University, October 1975.
- [Fis75b] Fisher, D.A., "Bounded Workspace Garbage Collection in an Address-Order Preserving List Processing Environment", *Information Processing Letters*, July 1975, pp. 29-32.
- [Fis75c] Fisher, D.A., "Copying Cyclic List Structures in Linear Time Using Bounded Workspace", *Communications of the ACM*, Vol. 18, No. 5, May 1975, pp. 251-252.
- [Fis75d] Fisher, D.A., "Programming Language Commonality in the Department of Defense", *Defense Management Journal*, Vol. II No. 4, October 1975, pp. 29-33.
- [Fis74] Fisher, D.A., "Automatic Data Processing Costs in the Defense Department", *IDA Paper P-1046*, October 1974, AD-A004841.
- [Fis72] Fisher, D.A., "A Survey of Control Structures in Programming Languages", *SIGPLAN Notices, Special Issues on Control Structures*, Vol. 7, No. 11, November 1972, pp. 1-13.
- [Fis70a] Fisher, D.A., "Control Structures", *Computer Science Research Review 1970-1971*, Carnegie-Mellon University, September 1971, pp. 21-25.
- [Fis70b] Fisher, D.A., "Control Structures for Programming Languages", Ph.D. Dissertation, Carnegie-Mellon University, May 1970, AD-708 511.
- [Fis67] Fisher, D.A., "Program Analysis for Multiprocessing", M.S.E. Thesis, Moore School of Electrical Engineering, University of Pennsylvania, May 1967.
- [FKRW*78] Fisher, D.A., Kernighan, D.G.B., Reynolds, J., Wetherall, P., and Wulf, W., "Report on the HOL Analyses Coordination Panel", DoD Common High Order Language, Phase I Reports and Analyses, HOLWG Report, June 1978, AD-B950 587.
- [FM86a] Fisher, D.A. and Muzdie, D.A., "Incremental Semantic Analysis and Overload Resolution for Ada", Final Report, Phase I, SBIR, National Science Foundation Award ISI-8560535, August 1986.
- [FM86b] Fisher, D.A. and Muzdie, D.A., "Parallel Processing in Ada", *IEEE Computer*, Vol. 19, No. 8, August 1986, pp. 20-25.
- [FS79] Fisher, D.A. and Standish, T.A., "Initial Thoughts on the Pebbleman Process", Institute for Defense Analyses (IDA) Paper P-1392, June 1979.
- [FW86] Fisher, D.A. and Weatherly, R.M., "Issues in the Design of a Distributed Operating System for Ada", *IEEE Computer*, Vol. 19, No. 5, May 1986, pp. 38-47.

- [FW78] Fisher, D.A. and Wetherall, P.R., "Rationale for Fixed Point and Floating Point Computational Requirements for a Common Programming Language", IDA Paper P-1305, January 1978.
- [Iron76] "Ironman", Department of Defense Requirements for High Order Computer Programming Languages, HOLWG Report, June 1976.
- [GMT*80] Gerhart, S.L., Musser, D.R., Thompson, D.H., Baker, D.A., Bates, R.L., Erickson, R.W., London, R.L., Taylor, D.G., and Wile, D.S., "An Overview of AFFIRM: A Specification and Verification System", *Proceedings IFIP 80*, October 1980.
- [Mun81] Mundie, D.A., "The Integration of the Comecon Computer Industries", A Report Prepared for The National Council for Soviet and East European Research, Master's Thesis, June 1981.
- [Mun80a] Mundie, D.A., "Pascal and the Great Race", *Byte*, September 1980, p. 94.
- [Mun80b] Mundie, D.A., "PILOT/P: Implementing a High-level Language in a Hurry", *Byte*, July 1980, pp. 154-170.
- [Mun79a] Mundie, D.A., "A Computer-Assisted Dieting Program", *The Byte Book of Pascal*, pp. 197-198.
- [Mun79b] Mundie, D.A., "Supermetric: An Automatic Metric Conversion Program", *The Byte Book of Pascal*, pp. 189-196.
- [Mun78] Mundie, D.A., "In Praise of Pascal", *Byte*, 1978. Reprinted in *The Byte Book of Pascal*, Peterborough, New Hampshire, Byte Publications, 1979, pp. 7-12.
- [PW83] Pervin, E.C. and Webb, J.A., "Quaternions in Computer Vision and Robotics", Carnegie-Mellon University, Department of Computer Science CMU-CS-82-150, *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, Washington, D.C., 1983, pp. 382-383.
- [Shu86] Shultis, J.C., "The Design and Implementation of Intuit", *IEEE Conference on Logic in Computer Science*, June 1986.
- [Shu85a] Shultis, J.C., *On the Complexity of Higher-Order Programs*, Technical Report CU-CS-288-85, University of Colorado, February 1985.
- [Shu85b] Shultis, J.C., "What is a Model? A Consumer's Perspective on Semantic Theory", *Proceedings Conference on the Mathematical Foundations of Programming Semantics*; Springer-Verlag, Lecture Notes in Computer Science #239, 1985.
- [Shu83] Shultis, J.C., "A Functional Shell", *Proceedings SIGPLAN '83 Symposium on Programming Language Issue in Software Systems*, June 1983, pp. 202-211.
- [Shu82] Shultis, J.C., "Hierarchical Semantics, Reasoning, and Translation", Ph.D. Dissertation, Department of Computer Science, State University of New York at Stony Brook, August 1982.
- [SK81] Shultis, J.C. and Kieburtz, R.B., "Transformations of FP Program Schemes", *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, October 1981, pp. 41-48.
- [Tad82] Tadman, F.P., "The Arcturus Programming Environment Program Design and Rapid Prototyping Language", Technical Report, Programming Environment Project, University of California, Irvine, California, September 1982.
- [WP84] Webb, J.A. and Pervin, E.C., "The Shape of Subjective Contours", *Proceedings of the Conference on Artificial Intelligence*, Austin, Texas, 1984, pp. 342-343.

Papers in Preparation by Members of Incremental Systems Technical Staff.

- [TBB*87] "Next Generation Software Environments: Principles, Problems, and Research Directions", Taylor, R.N., **Baker, D.A.**, Belz, F.C., Boehm, B.W., Clarke, L.A., **Fisher, D.A.**, Osterweil, J., Selby, R.W., Wileden, J.C., Wolf, A.L., and Young, M, submitted for publication.
- [Shu85] "Imminent Garbage Collection", **Shultis, J.C.**, Technical Report CU-CS-305i-85, University of Colorado, Department of Computer Science, 1985.
- [IRIS] "Iris: An Internal Form for Use in Integrated Environments", **Baker, D.A.**, **Fisher, D.A.**, and **Shultis, J.C.**

Journals and Proceedings Edited by Members of Incremental Systems Technical Staff.

- Proceedings of the IEEE Computer Society Second International Conference on Ada Applications and Environments*, **Fisher, D.A.**, General Chairman; Morris, D., Program Chairman, 1986.
- "Special Issue on Ada Environments and Tools", *IEEE Software*, Urban, J.W. and **Fisher, D.A.**, Guest Editors; Vol. 2, No. 2, March 1985.
- Proceedings of the IEEE Computer Society 1984 Conference on Ada Applications and Environments*, Urban, J.E., General Chairman; **Fisher, D.A.**, Program Chairman.
- "Lecture Notes in Computer Science", Edited by Williams, J.H. and **Fisher, D.A.**, *Design and Implementation of Programming Languages*, Proceedings of a DoD Sponsored Workshop, Ithaca, New York, October 1976; (Publisher: Springer-Verlag Berlin Heidelberg, 1977)

Other References.

- [ALRM83] Ada Joint Program Office, U.S. Department of Defense, *Ada Programming Language Reference Manual*, ANSI/MIL-STD- 1815A-1983, 1983.
- [ABB*86] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A. and Young, M., "Mach: A New Kernel Foundation for UNIX Development", DRAFT, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, May 1986.
- [ABC*83] Atkinson, M.P., Bailey P.J., Chisholm K.J., Cockshott, P.W., and Morrison, R., "An Approach to Persistent Programming", *The Computer Journal*, 1983, 26(4), pp. 360-365.
- [AL81] Anderson, T. and Lee, P.A., "Fault Tolerance: Principles and Practice", Prentice-Hall International, 1981.
- [AMC*83] Andrews, P.B., Miller, D.A., Cohen, E.L. and Pfenning, F., "Automating Higher-Order Logic", Department of Mathematics, Carnegie-Mellon University, Pittsburgh, Pennsylvania, January 1983.
- [And81] Andrews, G.R., "The Distributed Programming Language SR: Mechanism Design and Implementation", *Software Practice and Experience*, 12, No. 8, 1981.
- [AW76] Ashcroft, E.A. and Wadge, W.W., "Lucid--A Formal System for Writing and Proving Programs", *Siam J. Comput.*, Vol. 5, No. 3, September 1976, pp. 336-354
- [AW85] Avrunin, G.S. and Wileden, J.C., "Describing and Analyzing Distributed Software System

- Designs", *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, July 1985, pp. 380-403.
- [Bac80] Back, R.J.R., "Correctness Preserving Program Refinements: Proof Theory and Applications", Ph.D. Dissertation, Math. Cent., Amsterdam, 1980.
- [Bac78] Backus, J., "The History of Fortran I, II, and III", *ACM SIGPLAN Notices*, Vol. 13, No. 8, August 1978, pp. 165-180.
- [Bal81] Balzer, R., "Transformational Implementation: An Example", *IEEE Transactions on Software Engineering*, SE-7, (1), 1981, pp. 3-14.
- [Bar87] Barstow, D., "Artificial Intelligence and Software Engineering", *Proceedings of the Ninth International Conference on Software Engineering*, Monterey, California, pp. 200-211.
- [Bar77] Barstow, D., "Automatic Construction of Algorithms and Data Structures Using a Knowledge Base of Programming Rules", Ph.D. Dissertation, Stanford University, Stanford, California, 1977.
- [BBD77] Bell, T.E., Bixler, D.C. and Dyer, M.E., "An Extendable Approach to Computer-Aided Software Requirements Engineering", *IEEE Transactions on Software Engineering* SE-3, 1, January 1977, pp. 49-60.
- [BBK*82] Bodwin, J.M., Bradley, L., Kanda, K., Litle, D. and Pleban, U.F., "Experience With an Experimental Compiler Generator Based On Denotational Semantics", *Proceedings of SIGPLAN '82 Symposium on Compiler Construction*, June 1982, pp. 216-229.
- [BD77] Burstall, R.M. and Darlington, J., "A Transformation System for Developing Recursive Programs", *JACM* 24, 1, January 1977, pp. 44-67.
- [Ber87] Bernstein, Philip A., "Database System Support for Software Engineering", *Proceedings of the Ninth International Conference on Software Engineering*, IEEE Computer Society Press, Monterey, California, March 1987, pp. 166-178.
- [BG87] Becker, J. and Goettge, R., "Ada Performance Issues for Real-Time Systems", DRAFT, Advanced System Technologies, 1987.
- [BGN84] Balzer, R., Goldman, N. and Neches, B., "Specification-Based Computing Environments for Information Management", *Proceedings of the International Conference on Data Engineering*, Los Angeles, California, April 1984, pp. 454-458.
- [BGW76] Balzer, R., Goldman, N. and Wile, D., "On The Transformational Implementation Approach to Programming", *IEEE Proceedings of Second International Conference on Software Engineering*, San Francisco, October 1976, pp. 337-344.
- [Bir84] Bird, R.S., "The Promotion and Accumulation Strategies in Transformational Programming", *ACM Transactions on Programming Languages and Systems* 6, (4) October 1984, pp. 487-504.
- [Bjo87] Bjorner, D., "On The Use of Formal Methods in Software Development", *Proceedings of the Ninth International Conference on Software Engineering*, IEEE Computer Society Press, Monterey, California, March 1987, pp. 17-29.
- [BKK*86] Bobrow, D.G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M. and Zdybel, F., "CommonLoops: Merging Lisp and Object-Oriented Programming", *OOPSLA '86: Object Oriented Programming Systems, Languages, and Applications*, SIGPLAN Notices 21(11), 1986, pp. 17-29.
- [BL84] Burstall, R. and Lampson B., "A Kernel Language for Modules and Abstract Data Types",

Report #1, Digital System Research Center Reports, September 1984.

- [BM79] Boyer, R. and Moore, J., "A Computational Logic", Academic Press, 1979.
- [BMS80] Burstall, R.M., McQueen D.B. and Sannella, D.T., "HOPE: An Experimental Applicative Language", Internal Rep., Department of Computer Science, Edinburgh University, Scotland, 1980.
- [Boe85] Boehm, B.W., "A Spiral Method of Software Development and Enhancement", *Proceedings of the Second International Software Process Workshop*, Wileden, J.C. and Dowson, M. editors, IEEE Computer Science Press, Coto de Caza, Trabuco Canyon, California, March 1985, pp. 22-42.
- [Boe76] Boehm, B.W., "Software Engineering", *IEEE Transactions on Computers* 25, December 1976, pp. 1226-1241.
- [BP81] Broy, M. and Pepper, P., "Program Development as a Formal Activity", *IEEE Transactions on Software Engineering*, Vol. 7, No. 1, January 1981, pp. 14-22.
- [BPP81] Britton, K.J., Parker, R.A. and Parnas, D.L., "A Procedure for Designing Abstract Interfaces for Device Interface Modules", *Proceedings of the Fifth International Conference on Software Engineering*, San Diego, California, March 1981, pp. 195-204.
- [Bro85] Brookes, S.D., "On the Axiomatic Treatment of Concurrency", CMU-CS-85-106, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1985.
- [But83] Butler, K.D., "DIANA Past, Present and Future", *Lecture Notes in Computer Science Ada Software Tools Interfaces*, Ed. G.Goos and J.Hartmanis, Workshop, Bath: Springer-Verlag, 1983, pp. 3-22.
- [CAIS85] Ada Joint Program Office, U.S. Department of Defense, *Common Ada Programming Support Environment Interface Set*, Proposed MIL-STD CAIS, 1985.
- [Car86a] Cardelli, L., "Amber", *Combinators and Functional Programming Languages*, Lecture Notes in Computer Science #242, Springer-Verlag, 1986.
- [Car86b] Cardelli, L., "The Amber Machine", *Combinators and Functional Programming Languages*, Lecture Notes in Computer Science #242, Springer-Verlag, 1986.
- [Cat80] Cattell, R.G.G., "Automatic Derivation of Code Generators from Machine Descriptions", *ACM Transactions on Programming Languages and Systems* 2, (2), April 1980, pp. 173-190.
- [CC83] Ceri, S., and Crespi-Reghizzi, S., "Relational Data Bases in the Design of Program Construction Systems", *SIGPLAN Notices* 18, 11, November 1983, pp. 34-44.
- [CC77] Cousot, P. and Cousot, R., "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints", *Principles of Programming Languages IV*, ACM, January 1977, pp. 238-252.
- [CDF*86] Carey, M.J., DeWitt, D. J., Frank, D., Goetz, G., Richardson, J.E., Shekita, E. J., and Muralikrishna, M., "The Architecture of the EXODUS Extensible DMBS: A Preliminary Report", Technical Report CS-644, Computer Science Department, University of Wisconsin-Madison, Madison, May 1986.
- [Che84] Cheatham, T.E., Jr., "Reusability Through Program Transformations", *IEEE Transactions on Software Engineering*, Vol. 10, No. 5, September 1984, pp. 589-594.
- [Che83] Cheatham, T.E., Jr., "Harvard Programming Development System (PDS)", *Software Engineering Notes* 8, (5), October 1983, pp. 49-50.

- [Che81] Cheatham, T.E., Jr., "Overview of the Harvard Program Development System", *Software Engineering Environments*, Hünke, H. editor, 1981.
- [CHT81] Cheatham, T.D., Jr., Holloway, G.H. and Townley, J.A., "Program Refinement by Transformation", *IEEE Proceedings of Fifth International Conference on Software Engineering*, San Diego, California, March 1981, pp. 430-437.
- [CK84] Cooper, K.D. and Kennedy, K., "Efficient Computation of Flow Insensitive Interprocedural Summary Information", *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, SIGPLAN Notices 19, 6, June 1984, pp. 247-258.
- [CKT86] Cooper, K.D., Kennedy, K. and Torczon, L., "The Impact of Interprocedural Analysis and Optimization in the R^n Programming Environment", *ACM Transactions on Programming Languages and Systems*, Vol. 8, No. 4, October 1986, pp. 491-523.
- [Cle84] Clemm, G.M., "ODIN - An Extensible Software Environment Report and User's Manual", University of Colorado at Boulder, Computer Science Department Technical Report CU-CS-262-84, May 1984.
- [CO86] Clemm, G.M. and Osterweil, L.J., *A Mechanism for Environment Integration*. Technical Report CU-CS-323-86, University of Colorado, Boulder, January 1986.
- [Coc83] Cockshott, W.P., "Orthogonal Persistence", Thesis CST-21-83, Department of Computer Science, University of Edinburgh, February 1983.
- [Con86] Constable, R.L., et al, "Implementing Mathematics in the NuPr1 System", Prentice-Hall, 1986.
- [Coq86] Coquand, T., "An Analysis of Girard's Paradox", *First Conference on Logic in Computer Science*, Boston, June 1986.
- [Coq85] Coquand, T., "Une Théorie des constructions", Thèse de Troisième Cycle, Université Paris VII, January 1985.
- [Cou81] Cousot, P., "Semantic Foundations of Program Analysis", *Program Flow Analysis: Theory and Applications*, Jones, N.D. and Muchnick, S.S. editors, Prentice-Hall, Englewood Cliffs, 1981.
- [Cur83] Curien, P.L., "Combinateurs Catégoriques, Algorithmes Séquentiels et Programmation Applicative", Thèse de Doctorat d'Etat, Université Paris VII, December 1983.
- [CW85] Cardelli, L. and Wegner, P., "On Understanding Types, Data Abstraction, and Polymorphism", *ACM Computing Surveys* 17, December 1985, pp. 471-522.
- [CWW86] Clarke, L.A., Wileden, J.C. and Wolf, A.L., "Graphite: A Meta-tool for Ada Environment Development", *Proceedings of the IEEE Computer Society Second International Conference on Ada Applications and Environments*, IEEE Computer Society Press, Miami Beach, Florida, April 1986, pp. 81-90.
- [CZ84] Constable, R.L. and Zlatin, D.R., "The Type Theory of PL/CV3", *ACM Transactions on Programming Languages and Systems*, Vol. 6, No. 1, January 1984, pp. 94-117.
- [DB73] Darlington, J. and Burstall, R.M., "A System Which Automatically Improves Programs", *Proceedings of Third International Joint Conference on Artificial Intelligence*, Stanford, California, SRI, Menlo Park, California, 1973, pp. 479-485.
- [deB80] de Bruijn, N.G., "A Survey of the Project Automath", *H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Seldin, J.P. and Hindley, J.R., editors, Academic Press, 1980.

- [Der85] Dershowitz, N., "Program Abstraction and Instantiation", *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, July 1985, pp. 446-447.
- [Der81] Dershowitz, N., "The Evolution of Programs: Program Abstraction and Instantiation", *IEEE Proceedings of Fifth International Conference on Software Engineering*, San Diego, California, March 1981, pp. 79-88.
- [DF80] Davidson, J.W. and Fraser, C.W., "The Design and Application of a Retargetable Peephole Optimizer", *ACM Transactions on Programming Languages and Systems* 2 (2), April 1980, pp. 191-202.
- [DK76] DeRemer, F. and Kron, H.H., "Programming-in-the-Large Versus Programming-in-the-Small", *IEEE Transactions on Software Engineering*, June 1976, SE-2, pp. 80-86.
- [DR79] Davis, A.M. and Rauscher, T.G., "Formal Techniques and Automatic Processing to Ensure Correctness in Requirements Specification", *Proceedings of the Specifications of Reliable Software Conference*, April 1979, pp. 15-35.
- [DR78] Davis, A.M. and Rataj, W.J., "Requirements Language Processing for the Effective Testing of Real Time Systems", *Proceedings of the Software Quality and Assurance Workshop*, November 1978, pp. 61-66.
- [DS87] Dietzen, S.R. and Scherlis, W.L., "Analogy in Program Development", *The Role of Language in Problem Solving 2*, Boudreaux, J.C., Hamill, B.W. and Jernigan, R. editors, North-Holland, 1987, pp. 95-117.
- [DGL*79] Dewar, R.B.K., Grand, A., Liu, S.-C., Schwartz, J.T., and Schonberg, E., "Programming by refinement, as exemplified by the SETL representation sublanguage", *ACM Trans. Program. Lang. Syst.*, vol. 1 no. 1 (July 1979), 27-49.
- [DSS81] Dewar, R.B.K., Schonberg, E., and Schwartz, J.T., "Higher Level Programming: Introduction to the Use of the Set-Theoretic Programming Language SETL", Courant Inst. of Mathematical Sciences, New York Univ., New York, 1981.
- [EP87] Elliott, C. and Pfenning, F., "A Family of Program Derivations for Higher-Order Unification", Extended Abstract, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, April 1987.
- [Ers82] Ershov, A.P., "Mixed Computation: Potential Applications and Problems for Study", *Theoretical Computer Science* 18, 1982, pp. 41-67.
- [Fai83] Fairbairn, J., "Ponder and Its Type System", *Polymorphism*, Vol. 1, No. 2, The ML/LCF/Hope Newsletter, April 1983.
- [Fea82] Feather, M.S., "A System for Assisting Program Transformation", *ACM Transactions of Programming Language Systems* 4, (1), January 1982, pp. 1-20.
- [Fel79] Feldman, S.I., "Make-A Program for Maintaining Computer Programs", *Software - Practice & Experience*, April 1979, 9(4):255-265.
- [Gan86] Ganziner, H. and Jones, N.D., editors, "Programs as Data Objects", (*Workshop Proceedings*), LNCS 217, Springer-Verlag, April 1986.
- [Gan85] "Special Issue on the Gandalf Project", *The Journal of Systems and Software*, May 1985, 5:2.
- [GG78] Glanville, R.S., Graham, S.L., "A New Method for Compiler Code Generation (Extended Abstract)", *Proceedings of the Fifth Annual Principles of Programming Languages*, January 1978, pp. 231-240.
- [GHW85a] Guttag, J.V., Horning, J.J. and Wing, J.M., "Larch in Five Easy Pieces", Report #5,

Digital System Research Center Reports, July 1985.

- [GHW85b] Guttag, J.V., Horning, J.J., and Wing, J.M., "The Larch Family of Specification Languages", *IEEE Software*, September 1985, 2:5, pp. 24-36.
- [Gir86] Girard, J.Y., "Linear Logic", Université Paris, October 1986.
- [Gir70] Girard, J.Y., "Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types, *Proceedings of the Second Scandinavian Logic Symposium*, Fenstad, J.E. editor, North Holland, 1970, pp. 63-92.
- [GLB*83] Green, C., Luckham, D., Balzer, R., Cheatham, T. and Rich, C., "Report on a Knowledge-Based Software Assistant", Technical Report KES.U.83.2, Kestrel Institute, June 1983.
- [GMW79] Gordon, M.J., Milner, A.J. and Wadsworth, C.P., "Edinburgh LCF", *Lecture Notes in Computer Science*, No. 78, Springer-Verlag, Berlin, 1979.
- [Gol86] Goldberg, A.T., "Knowledge-Based Programming: A Survey of Program Design and Construction Techniques", *IEEE Transactions on Software Engineering*, SE-12 (7), July 1986, pp. 752-768.
- [Gol85] Goldsack, S.J., editor, *Ada for Specification: Possibilities and Limitations*, Cambridge University Press, 1985.
- [Goo76] Goodenough, J.B., "Exception Handling Issues and a Proposed Notation", *Communications of the ACM* 18, 12, December 1975, pp. 683-696.
- [GR83] Goldberg, A. and D. Robson, *Smalltalk-80 : The Language and its Implementation*, Addison Wesley, 1983.
- [Gre77] Green, C., "A Summary of the PSI Program Synthesis System", *Proceedings of Fifth International Joint Conference on Artificial Intelligence*, M.I.T., Cambridge, Massachusetts, 1977, pp. 380-381.
- [Gut77] Guttag, J., "Abstract Data Types and the Development of Data Structures", *Communications of the ACM*, June 1977.
- [GT79] Goguen, J.A. and Tardo, J.J., "An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications", *Proceedings of a Conference on Specifications of Reliable Software*, IEEE Computer Society, April 1979, pp. 170-189.
- [Hai86] Hailpern, B., "Multiparadigm Languages and Environments", *IEEE Software*, 3(1), January 1978.
- [Hec77] Hecht, M.S., "Flow Analysis of Computer Program", North-Holland, New York, 1977.
- [HM85] Heimbigner, D. and McLeod, D., "A Federated Architecture for Information Management", *ACM Transactions on Office Information Systems*, July 1985, 3(3):253-278.
- [HN80] Habermann, A.N. and Nassi, I.R., "Efficient Implementation of Ada Tasks", CMU-CS-80-103, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1980.
- [HN86] Habermann, A.N. and Notkin, D., "Gandalf: Software Development Environments", *IEEE Transactions on Software Engineering*, December 1986, 12(12):1117-1127.
- [HO82] Hoffmann, C.M. and O'Donnell, M.J., "Programming with Equations", *ACM Transactions on Programming Languages and Systems*, 4,1, 1982, pp. 83-112.
- [Hoa84] Hoare, C.A.R., *Occam Programming Manual*, Prentice Hall, London, 1984.
- [Hoa73] Hoare, C.A.R., "Hints on Programming Language Design", *SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, October 1973.

- [Hoo84] Hook, J.G., "Understanding Russell, a First Attempt", *Semantics of Data Types*, Lecture Notes in Computer Science 173, Springer-Verlag, 1984, pp. 51-67.
- [How80] Howard, W.A., "The Formulæ-As-Types Notion of Construction", Unpublished manuscript 1969. Reprinted in H.B. Curry: *Essays on Combinatory Logic, Lambda Calculus and Formalism*, Seldin, J.P. and Hindley, J.R. editors, Academic Press, 1980.
- [Hue87] Huet, G., "A Uniform Approach to Type Theory", INRIA, 1987.
- [Hue72] Huet, G., "Constrained Resolution: A Complete Method for Type Theory", Ph.D. Thesis, Jennings Computing Center Report 1117, Case Western Reserve University, 1972.
- [HY86] Hudak, P. and Young, J., "Higher-Order Strictness Analysis in Untyped Lambda Calculus", *Thirteenth Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1986, pp. 97-109.
- [HZ76] Hamilton, M. and Zeldin, S., "Higher Order Software - A Methodology for Defining Software", *IEEE Transactions on Software Engineering SE-2*, (1), March 1976, pp. 9-32.
- [Joh83] Johnson, M.S., editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, Pacific Grove, California, March 1983.
- [Jon87] Jones, N., "The Theory and Practice of Automatic Program Specification", *Third Workshop on the Mathematical Foundations of Programming Language Semantics*, New Orleans, Louisiana, April 1987.
- [Jon86] Jones, N.D., "Flow Analysis of Lazy Higher Order Functional Programs", DIKU, Universitetsparken 1, Copenhagen, 1986.
- [Jon80] Jones, N.D., editor, "Semantics-Directed Compiler Generation (LNCS94)", Springer-Verlag, 1980.
- [JR86] Jones, M.B. and Rashid, R.F., "Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems", *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, Ed. N. Meyrowitz, Association for Computing Machinery, Portland, Oregon, IEEE, November 1986, pp. 67-77.
- [JS86] Jorring, U. and Scherlis, W.L., "Compilers and Staging Transformations", *Thirteenth Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1986, pp. 86-96.
- [JSS85] Jones, N.D., Sestoft, P. and Sondergaard, H., "An Experiment in Partial Evaluation: The Generation of a Compiler Generator", *Rewriting Techniques and Applications, Lecture Notes in Computer Science 202*, Springer-Verlag, 1985, pp. 124-140.
- [KC86] Khoshafian, S.N., and Copeland, G.P., "Object Identity", *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, October 1986, (also *SIGPLAN Notices*, November 1986), pp. 406-416.
- [Kin85] King, R.M., "Knowledge-Based Transformational Synthesis of Efficient Structures for Concurrent Computation", Ph.D. Thesis, Rutgers University, Kestrel Institute Report, KES.U.85.5, April 1985.
- [KNS77] Kibler, D.F., Neighbors, J.M. and Standish, T.A., "Program Manipulation via an Efficient Production System", *Proceedings of Symposium on Artificial Intelligence and Programming Languages*, Rochester, New York, *SIGPLAN Notices (ACM) 12*, 8, August, *SIGART Newsletter ACM 64*, August 1977, pp. 163-173.

- [KP82a] Kieras, D.E. and Polson, P.G., "An Approach to the Formal Analysis of User Complexity", Working Paper No. 2, University of Arizona and University of Colorado, October 1982.
- [KP82b] Kieras, D.E. and Polson, P.G., "An Outline of a Theory of the User Complexity of Devices and Systems", Working Paper No. 1, University of Arizona and University of Colorado, May 1982.
- [Kur86] Kurki-Suonio, R., "Towards Programming with Knowledge Expressions", *Thirteenth Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1986, pp. 140-149.
- [LAB*81] Liskov, B., Atkinson, R., Bloom, T., Moss, E., Schaffert, J.C. and Snyder, A., *CLU Reference Manual*, Springer-Verlag, 1981.
- [Lam83] Lamport, L., "Specifying Concurrent Program Modules", *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 2, 1983, pp. 190-222.
- [Lan66] Landin, P.J., "The Next 700 Programming Languages", *ACM Communications*, Vol. 9, No. 3, March 1966, pp. 157-166.
- [LC84] Leblang, D.B. and Chase, R.P., Jr., "Computer-Aided Software Engineering in a Distributed Workstation Environment", *Proceedings of the ACM Symposium on Practical Software Development Environments*, Pittsburgh, April 1984, pp. 104-112.
- [Lei83] Leivant, D., "Reasoning About Functional Programs and Complexity Classes Associated with Type Disciplines", *Twenty-fourth Annual Symposium on Foundations of Computer Science*, Tucson, Arizona, 1983, pp. 460-496.
- [LF82] London, P.E. and Feather, M.S., "Implementing Specification Freedoms", *Science of Computer Programming*, 2(2), November 1982, pp. 91-131.
- [LHG86] Liskov, B., Herlihy, M. and Gilbert, L., "Limitations of Synchronous Communication with Static Process Structure in Languages for Distributed Computing", *Thirteenth Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1986, pp. 150-159.
- [JHL*77] Lampson, B.W., Horning, J.J., London, R.L., Mitchell, J.G. and Popek, G.J., "Report on the Programming Language Euclid", *SIGPLAN Notices* 12, (2), 1977.
- [Lin84] Linton, M.A., "Implementing Relational Views of Programs", *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Ed. P. Henderson, Association for Computing Machinery, Pittsburgh, Pennsylvania, ACM, May 1984, pp. 132-140.
- [Lis84] Liskov, B., "The ARGUS Language and System", *Advanced Course on Distributed Systems*, Munich, April 1984.
- [LP80] Luckham, D.C. and Polak, W., "Ada Exception Handling: An Axiomatic Approach", *ACM Transactions on Programming Languages and Systems* 2, 2, April 1980, pp. 225-233.
- [LS83] Lampson, B.W. and Schmidt, E.E., "Organizing Software in a Distributed Environment", *Proceedings of the ACM Symposium on Programming Languages Issues in Software Systems*, San Francisco, June 1983, pp. 1-13.
- [Les75] Lesk, M.E., "Lex - A Lexical Analyzer Generator", Computer Science TR #39, Bell Laboratories, Murray Hill, New Jersey, October 1975.
- [LS79] Liskov, B.H. and Snyder, A., "Exception Handling in CLU", *IEEE Transactions on Software Engineering SE-5*, 6, November 1979, pp. 546-558.

- [LSG82] Landwehr, R., St. Jansohn, H. and Goos, G., "Experience with an Automatic Code Generator Generator", *Proceedings of SIGPLAN '82 symposium on Compiler Construction*, June 1982, pp. 56-66.
- [LV85] Luckham, D., and von Henke, F.W., "An Overview of Anna, A Specification Language for Ada", *IEEE Software*, March 1985, 2:2, pp.24-33.
- [Mac71] MacLane, S., "Categories for the Working Mathematician", Springer-Verlag, 1971.
- [Mac86] MacQueen, D., "Using Dependent Types to Express Modular Structure", *Thirteenth Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1986, pp. 277-286.
- [Man87] Manes, E., "Program Expressions in a Category", *Third Workshop on the Mathematical Foundations of Programming Language Semantics*, New Orleans, Louisiana, April 1987.
- [Mar84] Martin-Löf, "Intuitionistic Type Theory", *Studies in Proof Theory*, Bibliopolis, 1984.
- [Mar79] Martin-Löf, P., "Constructive Mathematics and Computer Programming", *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, August 1979.
- [Mau86] Maule, R., "Run-Time Implementation Issues for Real-Time Embedded Ada", *Proceedings of First International Conference on Ada Programming Applications for the NASA Space Station*, June 1986.
- [MB81] MacQueen, D.B. and Burstall, R.M., "Structure and Parameterization in a Typed Functional Language", *MSS*, August 1981.
- [Mey86] Meyrowitz, Norman, editor, *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, ACM SIGPLAN, ACM, Portland, Oregon, September 1986. SIGPLAN Notices, 21(11), November 1986.
- [Mil80] Milner, R., "A Calculus of Communicating Systems", *Lecture Notes in Computer Science*, No.92, Springer-Verlag, 1980.
- [Mit86] Mitchell, J.C., "Representation Independence and Data Abstraction", *Thirteenth Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1986, pp. 263-276.
- [MK83] Magee, J. and Kramer, J., "Dynamic System Configuration for Distributed Real-Time Systems", *IFAC/IFIP Workshop on Real-Time Programming*, Hatfield, March 1983.
- [MN86] Miller, D.A. and Nadathur, G., "Higher-Order Logic Programming", *Third International Conference on Logic Programming*, Imperial College of Science and Technology, London, July 1986.
- [MNR83] McLeod, D., Narayanaswamy, K. and Rao, K.V. B., "An Approach to Information Management for CAD/VLSI Applications", *Proceedings of the ACM SIGMOD International Conference on Databases for Engineering Design*, San Jose, May 1983, pp. 39-50.
- [Mol84] Möller, B., "A Survey of the Project CIP: Computer-Aided, Intuition-Guided Programming", Technical Report TUM-18406, Institut für Informatik der TU München, Munich, West Germany, 1984.
- [Mor73] Morris, J.H., Jr., "Protection in Programming Languages", *Communications of the ACM*, Vol. 16, No. 1, January 1973.
- [MR86] Meyer, A.R. and Reinhold, M.B., "'Type' Is Not a Type", *Thirteenth Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1986, pp. 287-295.

- [MSOP86] Maier, D., Stein, J., Otis, A., and Purdy, A., "Development of an Object-Oriented DBMS", *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, October 1986, (also *SIGPLAN Notices*, November 1986), pp. 472-482.
- [MW80] Manna, Z. and Waldinger, R., "A Deductive Approach to Program Synthesis", *ACM Transactions of Programming Language Systems* 2, (1), January 1980, pp. 90-121.
- [MW77] Manna, Z. and Waldinger, R., "The Automatic Synthesis of Recursive Programs", *Proceedings on Artificial Intelligence and Programming Languages*, Rochester, New York, *SIGPLAN Notices (ACM)* 12, (8), *SIGART Newsletters (ACM)* 64, August 1977, pp. 29-31.
- [MY86] Miller, B.P. and Yang, C.Q., "IPS: An Interactive and Automatic Performance Measurement Tool for Parallel and Distributed Programs", Technical Report 613, Computer Science Department, University of Wisconsin-Madison, Madison, Wisconsin, December 1986.
- [Nau63] Naur, P., editor, "Revised Report on the Algorithmic Language ALGOL 60", *Communications of the ACM*, Vol. 1, No. 17, January 1963.
- [Nei80] Neighbors, J.M., "Software Construction Using Components", Ph.D. Dissertation, Technical Report 160, University of California, Irvine, California, 1980.
- [Neu85] Neumann, P. G., "Letter from the Editor; Risks to the Public", *Software Engineering Notes* 10 (5), October 1985, pp. 4-14.
- [Nie85] Nielson, F., "Program Transformations in a Denotational Setting", *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, July 1985, pp. 359-379.
- [OSD86] Orenstein, Jack A., Sarin, Sunil K., and Dayal, U., "Managing Persistent Objects in Ada", Technical Report CCA-86-03, Computer Corporation of America, Cambridge, Massachusetts, May 1986.
- [Ost87] Osterweil, L., "Software Processes Are Software Too", *Proceedings of the Ninth International Conference on Software Engineering*, Monterey, California, March 1987, pp. 2-13.
- [Ost86] Osterweil, L., "A Program-Object Centered View of Software Environment Architecture", University of Colorado, Department of Computer Science Technical Report CU-CS-332-86, May 1986.
- [PC86] Parnas, D.L. and Clements, P.C., "A Rational Design Process: How and Why to Fake It", *IEEE Transactions on Software Engineering SE-12*, 2, February 1986, pp. 251-257.
- [Phi83] Phillips, J., "Self-Described Programming Environments", Ph.D. Thesis, Stanford University Computer Science Department, Kestrel Institute Report, KES.U.83.1, March 1983.
- [Per87a] Perry, D.E., "Software Interconnection Models", *Proceedings of the Ninth International Conference on Software Engineering*, Monterey, California, March 1987, pp. 61-69.
- [Per87b] Perry, D.E., "Version Control in the Inscape Environment", *Proceedings of the Ninth International Conference on Software Engineering*, Monterey, California, March 1987, pp. 142-149.
- [PL83] Powell, M.L., and Linton, M.A., "Database Support for Programming Environments", *Proceedings of the ACM SIGMOD International Conference on Databases for Engineering Design*, San Jose, May 1983, pp. 63-70.
- [Pre86] Pressburger, T., "An Environment Supporting the Automation of Software Development", Kestrel Institute, KES.U.86.7, September 1986.

- [Ras86] Rashid, R.F., "From RIG to Accent to Mach: The Evolution of a Network Operating System", *Proceedings of the Fall Joint Computer Conference*, November 1986, pp. 1128-1137.
- [Rei87] Reiss, S.P., "A Conceptual Programming Environment", *Proceedings of the Ninth International Conference on Software Engineering*, Monterey, California, March 1987, pp. 225-235.
- [Rei85] Reiss, S.P., "PECAN: Program Development Systems That Support Multiple Views", *IEEE Transactions on Software Engineering SE-11*, 3, March 1985, pp. 276-285.
- [Rei84] Reiner, A., "Cost-Minimization in Register Assignment for Retargetable Compilers", CMU-CS-84-137, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1984.
- [Rep82] Reps, T., "Generating Language-Based Environments", TR82-415, Cornell Computer Science Department, August 1982.
- [Rey85] Reynolds, J.C., "Three Approaches to Type Structure", *TAPSOFT Advanced Seminar on the Role of Semantics in Software Development*, Berlin, March 1985.
- [Rey72] Reynolds, J.C., "Definitional Interpreters for Higher Order Programming Languages", *Proceedings ACM National Conference*, Boston, August 1972, pp. 717-740.
- [RLT78] Randell, B., Lee, P.A. and Treleaven, P.C., "Reliability Issues in Computing System Design", *ACM Computing Surveys* 10, No. 2, 1978.
- [Rob79] Robinson, J.T., "Some Analysis Techniques for Asynchronous Multiprocessor Algorithms", *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 1, January 1979, pp. 24-31.
- [Roc75] Rochkind, M.J., "The Source Code Control System", *IEEE Transactions on Software Engineering*, SE-1, 1975, pp. 364-370.
- [RR77] Robinson, L. and Roubine, O., "SPECIAL - A Specification and Assertion Language", CSL-46, SRI International, June 1977.
- [RS82] Reif, J. and Scherlis, W.L., "Deriving Efficient Graph Algorithms", Carnegie-Mellon University Technical Report, Pittsburgh, Pennsylvania, 1982.
- [RS77] Ross, D.T. and Schoman, K.E., Jr., "Structured Analysis for Requirements Analysis", *IEEE Transactions on Software Engineering SE-3*, (1), January 1977, pp. 6-15.
- [RSW79] Rich, C., Shrobe, H.E., Waters, R.C., "Overview of the Programmer's Apprentice", *Proceedings of Sixth International Joint Conference on Artificial Intelligence*, Tokyo, August, 1979.
- [RTD83] Reps, T., Teitelbaum, T. and Demers, A., "Incremental Context-Dependent Analysis for Language-Based Editors", *ACM Transactions on Programming Languages and Systems* 5, (3), July 1983, pp. 449-477.
- [SB83] Smoliar, S.W. and Barstow, D., "Who Needs Languages, and Why Do They Need them? or, No Matter How High the Level, It's Still Programming", *SIGPLAN Notices* 18, (6), June 1983, pp. 149-157.
- [SB78] Stone, H.S. and Bokhari, S.H., "Control of Distributed Processes", *IEEE Computer*, July 1978, pp. 97-106.
- [Sch85] Scherlis, W.L., "Adapting Abstract Data Types", Carnegie-Mellon University, Pittsburgh, Pennsylvania, September, 1985.
- [Sch81] Scherlis, W., "Program Improvement by Internal Specialization", *Eighth ACM Symposium*

on *Principles of Programming Languages*, ACM, Williamsburg, Virginia, January 1981, pp. 41-49.

- [Sco87] Scott, D., "Domains in the Realizability Universe", *Third Workshop on the Mathematical Foundations of Programming Language Semantics*, New Orleans, Louisiana, April 1987.
- [Sco80] Scott, D., "Relating Theories of the Lambda-Calculus", *H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, Seldin, J.P. and Hindley, J.R. editors, Academic Press, 1980.
- [Sco70] Scott, D., "Constructive Validity", *Symposium on Automatic Demonstration*, Lecture Notes in Mathematics No. 125, Springer-Verlag, 1970, pp. 237-275.
- [SHK*76] Standish, T.A., Harriman, D.C., Kibler, D.F. and Neighbors, J.M., "The Irvine Program Transformation Catalogue", Department of Information and Computer Science, University of California, Irvine, California, 1976.
- [Sin80] Sintzoff, M., "Suggestions for Composing and Specifying Program Design Decisions", *International Symposium on Programming*, Springer-Verlag, Lecture Notes in Computer Science, 1980.
- [Spe87] Spector, Alfred Z., *Distributed Transaction Processing in the Camelot System*, Technical Report CMU-CS-87-100, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, January 1987.
- [SR86] Stonebraker, M. and Rowe, L.A., "The Design of Postgres", *Proceedings of the ACM SIGMOD '86 International Conference on Management of Data*, Washington, D.C., June 1986, pp. 340-355.
- [SS83] Scherlis, W.L. and Scott, D.S., "First Steps Towards Inferential Programming", Carnegie-Mellon University Technical Report CMU-CS-83-142, July 1983.
- [ST84] Standish, T.A. and Taylor, R.N., "Arcturus: A Prototype Advanced Ada Programming Environment", *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Software Engineering Notes, May 1984, 9(3):57-64.
- [ST78] Stevens, S.A. and Tripp, L.A., "Requirements Expression and Verification Aid", *Proceedings of the Third International Conference on Software Engineering*, May 1978, pp. 101-108.
- [Sza78] Szabo, M.E., "Algebra of Proofs", North-Holland, 1978.
- [SZBH86] Swinehart, D.C., Zellweger, P.T., Beach, R.J. and Hagmann, R.B., "A Structural View of the Cedar Programming Environment", *ACM Transactions on Programming Languages and Systems*, Vol. 8, No. 4, October 1986, pp. 419-490.
- [TCO*86] Taylor, R.N., L.A. Clarke, L.J. Osterweil, J.C. Wileden, and M. Young., "Arcadia: A Software Development Environment Research Project", *Second International Conference on Ada Applications and Environments*, April 1986, pp. 137-149.
- [Tei85] Teitelman, W., "A Tour Through Cedar", *IEEE Transactions on Software Engineering*, 1985, SE-11(3):284-302.
- [Tic86] Tichy, Walter F., "Smart Recompilation", *ACM Transactions on Programming Languages*, July 1986, 8:3, pp. 273-291.
- [Tic85] Tichy, Walter F., "RCS - A System for Version Control", *Software - Practice and Experience*, July 1985, 15:7, pp. 637-654.

- [Tic82] Tichy, W.F., "Design, Implementation, and Evaluation of a Revision Control System", *Proceedings of the Sixth International Conference on Software Engineering*, Tokyo, Japan, September 1982, pp. 58-67.
- [Tic79] Tichy, W.F., "Software Development Control Based on Module Interconnection", *Fourth International Conference on Software Engineering*, Munich, September 1979.
- [Tha82] Thall, Richard M., "The KAPSE for the Ada Language System", *Proceedings of the AdaTEC Conference on Ada*, ACM, October 1982, pp. 31-47.
- [TM81] Teitelman, W., and Masinter, L., "The Interlisp Programming Environment", *Computer* 14, 4, April 1981, pp. 25-33.
- [TR81] Teitelbaum, T. and Reps, T., "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", *Communications of the ACM* 24, 9, September 1981, pp. 563-573.
- [Tur85] Turner, D.A., "Miranda: A Non-Strict Functional Language with Polymorphic Types", *Functional Programming Languages and Computer Architecture*, Jouannaud, J.P. editor, Springer-Verlag LNCS 201, 1985 pp. 1-16.
- [VM86a] Volz, R. and Mudge, T., "Instruction Level Mechanisms for Accurate Real-Time Task Scheduling", *Proceedings of Real-Time Systems Symposium*, December 1986.
- [VM86b] Volz, R.A. and Mudge, T.N., "Timing Issues in the Distributed Execution of Ada Programs", *IEEE Transactions on Computers, Parallel and Distributed Processing*, 1986.
- [War77] Warren, D., "Applied Logic - Its Use and Implementations as a Programming Tool", Ph.D. Thesis, University of Edinburgh, 1977.
- [Wat82] Waters, R.C., "The Programmer's Apprentice: Knowledge Based Program Editing", *IEEE Transactions on Software Engineering* SE-8, 1, January 1982, pp. 1-12.
- [Wat81] Waters, R.C., "A Knowledge-Based Program Editor", *Proceedings of Seventh International Joint Conference on Artificial Intelligence*, 1981.
- [WE82] Winchester, J. and Estrin, G., "Requirements Definition and Its Interface to the SARA Design Methodology for Computer-Based Systems", *National Computer Conference*, 1982, pp. 369-379.
- [Wei71] Weinberg, G.M., "The Psychology of Computer Programming", Van Nostrand Reinhold, New York, 1971.
- [Wel86] Wells, M.B., "General Purpose Languages for the Nineties", *Frontiers of Supercomputing*, University of California Press, 1986.
- [Wie86] Wiebe, D., "A Distributed Repository for Immutable Persistent Objects", *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, October 1986, (also *SIGPLAN Notices*, November 1986), pp. 453-465.
- [Wil83] Wile, D.S., "Program Developments: Formal Explanations of Implementations", *Communications of the ACM*, 26(11), November 1983, pp. 902-911.
- [Win87] Winkler, J.F.H., "Version Control in Families of Large Systems", *Proceedings of the Ninth International Conference on Software Engineering*, Monterey, California, March 1987, pp. 150-161.
- [Win79] Winograd, T., "Beyond Programming Languages", *Communications of the ACM* 22, (7), July 1979, pp. 391-401.
- [WPSK86] Wasserman, A.I., Pircher, P.A., Shewmake, D.T. and Kersten, M.L., "Developing Interactive Information Systems with the User Software Engineering Methodology", *IEEE*

- Transactions on Software Engineering SE-12*, February 1986, pp. 326-345.
- [YB85] Yemini, S. and Berry, D.M., "A Modular Verifiable Exception-Handling Mechanism", *ACM Transactions on Programming Languages and Systems* 7, 2, April 1985, pp. 214-243.
- [Zad84] Zadeck, F.K., "Incremental Data Flow Analysis in a Structured Program Editor", *Proceedings of the ACM Sigplan '84 Symposium on Compiler Construction, SIGPLAN Notices* 19, 6, June 1984, pp. 132-143.
- [Zav82] Zave, P., "An Operational Approach to Requirements Specification for Embedded Systems", *IEEE Transactions on Software Engineering SE-6*, (3), March 1982, pp. 253-269.
- [ZCL86] Zadeck, K., Cytron, R. and Lowry, A., "Motion of Control Structures in High-Level Languages", *Thirteenth Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1986, pp. 70-86.
- [Zdo86] Zdonik, S. B., "Maintaining Consistency in a Database with Changing Types", *Object-Oriented Programming Workshop*, June 1986, (also *SIGPLAN Notices*, October 1986), pp. 120-127.
- [ZS86] Zave, P. and Schell, W., "Salient Features of an Executable Specification Language and Its Environment", *IEEE Transactions on Software Engineering*, February 1986, SE-12(2):312-325.
- [ZW86] Zdonik, S.B. and P. Wegner, "Language and Methodology for Object-Oriented Database Environments", *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences*, January 1986, pp. 378-387.
- [ZW85] Zdonik, S.B. and P. Wegner, "A Database Approach to Languages, Libraries and Environments", *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large*, Harwichport, Massachusetts, June 1985, pp. 89-112.

A Conceptual Overview of Prism

Deborah A. Baker
David A. Fisher
David A. Mundie
Jonathan C. Shultis
Frank P. Tadman

*Incremental Systems Corporation
319 South Craig Street
Pittsburgh, Pennsylvania 15213*

February 22, 1991

1 Abstract

Artificial barriers which partition and isolate software activity are inherent in current software development environments. Deliberate and explicit partitioning is evident in the separation of programming languages from operating systems from databases, but more subtle barriers are manifested as limitations to expressiveness forcing overspecification and over-specialization, and barriers to sharing, access, and reuse caused by failure to represent and maintain semantic information about the artifacts produced and maintained by our tools.

The goal of the Prism effort is to produce a common persistent framework in which we can express, capture, reuse, improve, and build on anything that might be relevant to computational activity. Our means of achieving this integrated framework is a language emphasizing expressivity and serving as a medium for dialogue, rather than one-way communication, between user and machine. Highlights of the language include the ability to express and manage incomplete and

¹This work was supported in part by DARPA under contract number MDA 972-88-C-0076.

inconsistent specifications, and a view of semantics that replaces fixed and rigid interpretation of syntactic forms with interpretations that can be imprecise, dynamic, and strongly influenced by history and context.

Though we are somewhat surprised by the seeming radicalness of our approach, we are both convinced that such a departure from tradition is necessary, and encouraged that the goals continue to appear achievable. It is important to recognize, however, that this effort does not expect to solve any particular recognized software problem, but only to characterize an enabling technology that will remove the traditional obstacles to their solution.

2 Why Not Another Programming Language?

Reflection on the history of computer languages reveals a disturbing pattern. Languages are both generated and abandoned at a very high rate. A few languages, like FORTRAN, sh, and SQL, persist, though they are revised periodically in an attempt to improve them. These remarks apply to computer languages invented for a wide variety of purposes, such as operating system and database interfaces, specification languages, requirements languages, and prototyping languages, as well as programming languages. In all cases, the primary cause of longevity in a computer language seems to be not so much that the language has some intrinsic qualities that make it superior, but that there is so much invested in systems written in the language, not to mention the costs of language and environment implementation, and the even higher costs of training and supporting a workforce that has the competence and experience required to use a language effectively. As new paradigms and technologies appear, the number and variety of languages increases, and cults are formed around the idea that one or another family of languages is ultimately superior to all others, which will eventually pass into heathen oblivion, leaving the world of languages a smaller and cleaner place. History, however, tells another story.

Conventional wisdom, as preached by language moderates, is that the diversity and evolution of languages in punctuated equilibria is a natural, perhaps even inevitable, consequence of the diverse and often conflicting demands of their application niches. But is it inevitable? And is it desirable? Although the current situation in computer language engineering bears a strong analogy to biological evolution, the latter does not seem to us to be the best model for an engineering strategy. For one thing, it is slow and inefficient. Perhaps more importantly, it is not directed towards any goals, but merely plays off the (dis)advantages of one mutation against those of others.

Experimental research and innovation are necessary for technological progress, and there is bound to be much trial and error. Edison tested ? materials before finding a satisfactory filament for the electric light bulb. At the end of his experimentation, though, he had a solution to a problem - an answer. Experimentation with computer languages is often seen as being analogous, but the nature of the problem has never been very clearly articulated. If we look closely, we see that each language, or family of languages, is really directed toward a solution to a limited

problem, such as how to specify queries in a language that is psychologically natural and supports efficient retrieval, or how to express algorithms in a language that enhances clarity and reliability.

This kind of activity is very different from an effort to solve some kind of overall "language problem". Instead, it produces many solutions to many problems. What we find disturbing about this is the proliferation of highly duplicative efforts to solve limited problems at great expense, resulting in fragmentary and incohesive interfaces between people and computer systems as a whole. The typical computer user employs several languages for each of a wide variety of tasks, with little uniformity among them, and less ability to share or integrate what is done in one language with what is done in another.

Is there a language problem? Clearly, there is. The problem is to find an effective way of communicating with computer systems. We hypothesize that the problem has a solution because we are equipped with several solutions to the related problem of effective communication among people, namely English, Japanese, Dutch, Swahili, *etc.* These examples of real solutions differ markedly from computer languages generally. Though there is diversity in natural languages, it is unlike the diversity of computer languages. Natural languages evolve gradually, and are infinitely capable of assimilating new paradigms, techniques, and ideas. There are relatively few natural languages, they are relatively stable, and their rates of generation and extinction are extremely low.

At a deeper level, we observe that people have the capacity to be multilingual in a way that transcends translation, because the nature of understanding is universal across all human tongues, including our artificial ones. Thus, despite diversity, the things we do in one language are easily shared and integrated with those we do in another. This is true even when what is done in one language cannot possibly be translated into another, as is the case with poetry and punning.

Have there been any experiments aimed at solving this problem? There has been one, based on formal semiotics as embodied in our books and articles about language design and implementation, and our formal theories of syntax, semantics, and pragmatics. Over the years, this experiment has led to the invention of a great many filaments that are good for special purposes, but none of which sheds the full-spectrum light we need. The general characteristics of artificial and natural languages cited above suggest that this is no accident; that there is something fundamentally flawed in our entire approach to the language problem. It is time for a second experiment.

3 The Goals of Prism

The Prism project is aimed at laying the conceptual and practical foundations for a new kind of language for communicating with computers. The hallmark of Prism is its open-endedness in all dimensions. In the past, languages have tried to overcome various limits by being comprehensive and complete in some respect. The Prism goal is not to be complete, but rather to expunge from the language all varieties and degrees of absoluteness and finality, thereby permitting unbounded

expressiveness.

During the initial phase of the project, which officially started in the Fall of 1988, it became increasingly clear to us that the problem we had set out to solve required some change to the basic structure of language systems, while preserving and accomodating existing language features and implementation techniques. At the most basic level, this has meant replacing the conventional model of language as a means of expressing (naming, identifying, denoting) objects in some semantic domain with a model in which language is used in a more relative and oblique fashion for the mutual orientation of the parties to a dialogue. Thus Prism is a language for conversing with a system that happens to inhabit a computer,² and which has access to and control over the computer's resources. The primary object of a conversation with the system is to have it employ those resources on your behalf.

Some potential misconceptions must be dispelled at once. First and foremost, Prism will not, all by itself, solve any of the specific problems alluded to in the introduction. It will only remove the obstacles to their solution implicit in the platforms on which current languages and systems are designed and implemented.

Secondly, Prism is *not* a programming language, though it can be used to express and discuss programs. It can also, however, be used to communicate about specifications, designs, versions, analyses, requirements, failures, time, problem domains, implementation techniques, notations, representations, transformations, goals, hardware characteristics, operating environments, persistent data, expected behaviors, user models, formal deductive systems – in short, anything that bears on the technical aspects of the development, maintenance, or operation of computer systems.

The ambition is for Prism to be a “full spectrum” language, in a variety of dimensions. First off, it permits specification at arbitrary levels of abstraction. In this it extends the ambition of so-called “wide spectrum” languages, which have attempted to bridge gaps between particular specification languages and implementations. It is not our aim to build any particular bridge or set of bridges, but to design a language in which bridges like these can be built and extended indefinitely.

In another dimension, Prism accomodates incomplete and incorrect specifications as well as complete and correct ones. Tolerance for incompleteness is important not only for incremental refinement, but is a key to avoiding overspecification. Forced overspecification is rampant in current languages, making it difficult to determine which design decisions are important, and which are arbitrary. Many of the difficult and expensive analyses performed by optimizing compilers are aimed at recovering a less committed design so that the compiler can make its own commitments, based on information that is not relevant or readily available to the programmer. As long as the information supplied by the user is sufficient, in combination with the automated system, to obtain the results desired, nothing further is required. Nor is the user prevented from specifying any information that might improve the efficiency or utility of an application. The system's

²The use of the singular should not be construed as limiting the system to a single box. A global network is as much a computer as is an isolated PC.

is to eliminate limits wherever they are found in the existing language framework, and avoid introducing new ones. This is much easier said than done, however, and the number and kind of limits that exist is not always apparent. Moreover, there are strong arguments in favor of limitations, which can serve to guarantee certain degrees of consistency, completeness, or simplicity *a priori*. These advantages, and the technology that goes along with them, can be preserved in the Prism framework, in the guise of contexts within which certain overarching assumptions can be made. In contrast to the current situation, however, sharing and propagation are enabled by a common underlying language, semantic representation, and persistent context.

The most troublesome puzzle is how a common language can be created which not only accommodates all of the useful paradigms, modes of expression, and technologies which currently exist, but also those which have yet to be invented. The current language framework would require us to define a single comprehensive language right now which somehow merges existing languages and anticipates all future developments, and this is clearly not possible. Even if we limit ourselves for the moment to formal logic, the problem, as Gödel's incompleteness result makes abundantly clear, is that one cannot have a single fixed system for any significant purpose which is simultaneously consistent and complete. This seems to force upon us a choice between completeness and consistency, and most reasonable people opt for consistency.

The choice, however, is not forced. There is the option, which seems to have gone unnoticed, of having an open-ended, variable, system. Yet it is easy to see that any kind of systematic, parametric, variability will not suffice, because all one can obtain in that way is a fixed system of higher order. Something more basic has to give.

Natural languages apparently have the kind of open-endedness and flexibility required, but to reduce the problem to that of natural language understanding would be to trivialize it, and quite probably to doom oneself to failure. It is nonetheless reasonable to look to natural language for clues. What one finds is that natural language belies the formal language framework at virtually every point. Even the most basic ideas, that a language is a set of strings, that it has a grammar, and that meaning is compositional, are blasted, to an extent not widely recognized, and often denied, by computational linguists following the lead of such luminaries as Chomsky and Montague.

The surface features of a natural language are in fact merely the expression of a capacity to communicate, shaped by the cultural and personal experiences of each speaker. A language like English is a collective, *a posteriori*, phenomenon that is generated and sustained by a community of speakers.

An analogy with biological taxonomy may help to clarify this idea. Early taxonomists such as Linnæus classified organisms on the basis of their features, or *morphology*, leading to a definition of species as a set of features, corresponding to the definition of language as a set of strings, or grammatical structures.

The advent of genetics and population biology provided an alternative definition of species, as

tolerance for incompleteness and inconsistency is due to a semantics based on intensional objects and supporting intensional reasoning. This support for intensionality implies absolute conceptual freedom, allowing fictives, nonexistents, counterfactuals, and abstractions of all kinds and levels to be expressed.

In yet another dimension, all of the functions ordinarily assigned to operating systems, file systems, and databases are absorbed into the Prism system. All persistent data are contained in the persistent context, which includes all machines, networks, and removable media. When we say *the* persistent context, we mean it; there is only one. (At least, for any given Prism system. There is always the possibility of having multiple systems, much as there are multiple people, but each has its own unique and subjectively comprehensive context.) The system is also responsible for all aspects of control, including resource management and scheduling, requiring all of the elements of the system's grounding in the physical world of the computer to be predefined in the persistent context.

To avoid giving the impression that we regard Prism as some kind of panacæa, we must hasten to point out that the system's representations of its users' intent, and the real semantic content of their utterances, is inherently imperfect and limited. The system will blindly accept any inputs that are consistent with what it has received before, according to whatever tests it has been organized to undertake to verify consistency. We believe these limitations are inherent in all uses of language, even among people. They become even more acute in the case of computer languages, including Prism, because no computer system has any potential for contact with the reality of human concerns such as aesthetics and hunger, which are not part of the natural context of machines. These things can only be reflected in abstract models which necessarily fall short of genuine contact with the human world. Thus the design of systems which are responsive to human desires and needs will remain a human task, requiring great skill and sensitivity, and the responsibility for making decisions that affect human welfare must ultimately remain with people.

The need for a language of Prism's intended scope is clear; we cannot seriously hope to enlist computers as effective partners in computer system engineering without it. The practical impossibility of creating such a language within the tradition of formal languages as we know them is equally clear, and this is the source of a second potential misconception: that a language must be unimaginably complex to achieve our stated goals. In order to respond to this, quite legitimate, concern, we have to explain the basic difference between Prism and conventional computer languages, and how it may solve the problem of fragmentation without being crushed under its own complexity.

4 The Prism Approach to Language

If the fundamental problem with existing languages is that they erect barriers and impose limits restricting our ability to propagate technology and share results, then the solution to the problem

a freely interbreeding population. In this view, what identifies a species is genetic compatibility, which is expressed in a set of similar phenotypes. Phenotypic variation arises from a combination of genetic variation and individual history and environment.

The approach to language we are advocating here corresponds more closely to this latter theory. On this account, English is the (somewhat accidental and ephemeral) expression by a community of speakers of a basic communicative capacity. Differences in capability, errors in transmission, "interbreeding", drift due to cultural isolation, and happenstance account for much of the variability of natural language, but changes may also be stimulated by the appearance of new things in the cognitive landscape. Thus language adapts to changing circumstances and assimilates new ideas.³

Now, a conventional formal language is a closed system, in that its semantics, and in particular its semantic domain, is fixed at the outset by the language definition. As a rule, the more complex the semantic domain, the more complex the syntax that is needed to cover it.⁴

Prism, on the other hand, is an open-ended system, in that its semantics is determined by the contents of the persistent context in which the system is embedded, and the interpretation given to those contents. The persistent context can be viewed as a kind of "knowledge base", which may become arbitrarily complex, incorporating objects from arbitrary domains. For an object from a previously unknown domain to be brought into the context (immigrate), all that is required is to name it, thus entering in the persistent context an *ideograph* consisting of nothing but the identity of the object, *i.e.*, a reference to it. The system may become informed of additional properties of an object in a variety of ways, the most basic of which is to be told about them.⁵ Thus Prism can be used for arbitrarily complex purposes while remaining small and uniform as a language, because it can accomodate semantic extensions without accompanying syntactic extensions.

Actually, the situation with regard to semantic and syntactic extension is more complicated than the rhetoric here would suggest. For one thing, most implementations of programming languages, such as FORTRAN, permit a great deal of semantic extension through externally defined subroutines, and some languages, such as Ada, even include such facilities explicitly in their definition. However, these things can be used to "break" the semantics of the language, as when multitasking is added to FORTRAN by this means. The problem is that the new objects are

³At the risk of drawing the analogy too far, the adaptation of language in direct response to its "environment" is a kind of Lamarckian mechanism, far exceeding in efficiency and appropriateness random mutation, recombination, and natural selection. The extent to which the success of genus *homo* can be attributed to this more effective mode of adaptation cannot be overstated.

⁴As a simple example, consider Scott's LAMBDA [Sco76], in which the product type is "defined" by a term of LAMBDA. What one really gets, of course, is a retract of $\mathcal{P}\omega$, *i.e.*, a set of elements of $\mathcal{P}\omega$ which serve to represent ordered pairs. This is fine as far as it goes, but if one wanted to denote objects from some other domain and use *them* to represent ordered pairs, it would be necessary to add syntax to LAMBDA to denote them, because all of the existing syntax has already been exhausted on $\mathcal{P}\omega$.

⁵Common Lisp exhibits some open-endedness, in that there is a universal type of all objects, and some predefined types, but there is no restriction of types to compositions of predefined types, nor of objects to the objects of the predefined types.

a freely interbreeding population. In this view, what identifies a species is genetic compatibility, which is expressed in a set of similar phenotypes. Phenotypic variation arises from a combination of genetic variation and individual history and environment.

The approach to language we are advocating here corresponds more closely to this latter theory. On this account, English is the (somewhat accidental and ephemeral) expression by a community of speakers of a basic communicative capacity. Differences in capability, errors in transmission, "interbreeding", drift due to cultural isolation, and happenstance account for much of the variability of natural language, but changes may also be stimulated by the appearance of new things in the cognitive landscape. Thus language adapts to changing circumstances and assimilates new ideas.³

Now, a conventional formal language is a closed system, in that its semantics, and in particular its semantic domain, is fixed at the outset by the language definition. As a rule, the more complex the semantic domain, the more complex the syntax that is needed to cover it.⁴

Prism, on the other hand, is an open-ended system, in that its semantics is determined by the contents of the persistent context in which the system is embedded, and the interpretation given to those contents. The persistent context can be viewed as a kind of "knowledge base", which may become arbitrarily complex, incorporating objects from arbitrary domains. For an object from a previously unknown domain to be brought into the context (immigrate), all that is required is to name it, thus entering in the persistent context an *ideograph* consisting of nothing but the identity of the object, i.e., a reference to it. The system may become informed of additional properties of an object in a variety of ways, the most basic of which is to be told about them.⁵ Thus Prism can be used for arbitrarily complex purposes while remaining small and uniform as a language, because it can accommodate semantic extensions without accompanying syntactic extensions.

Actually, the situation with regard to semantic and syntactic extension is more complicated than the rhetoric here would suggest. For one thing, most implementations of programming languages, such as FORTRAN, permit a great deal of semantic extension through externally defined subroutines, and some languages, such as Ada, even include such facilities explicitly in their definition. However, these things can be used to "break" the semantics of the language, as when multitasking is added to FORTRAN by this means. The problem is that the new objects are

³At the risk of drawing the analogy too far, the adaptation of language in direct response to its "environment" is a kind of Lamarckian mechanism, far exceeding in efficiency and appropriateness random mutation, recombination, and natural selection. The extent to which the success of genus *homo* can be attributed to this more effective mode of adaptation cannot be overstated.

⁴As a simple example, consider Scott's LAMBDA [Sco76], in which the product type is "defined" by a term of LAMBDA. What one really gets, of course, is a retract of $\mathcal{P}\omega$, i.e., a set of elements of $\mathcal{P}\omega$ which serve to represent ordered pairs. This is fine as far as it goes, but if one wanted to denote objects from some other domain and use *them* to represent ordered pairs, it would be necessary to add syntax to LAMBDA to denote them, because all of the existing syntax has already been exhausted on $\mathcal{P}\omega$.

⁵Common Lisp exhibits some open-endedness, in that there is a universal type of all objects, and some predefined types, but there is no restriction of types to compositions of predefined types, nor of objects to the objects of the predefined types.

accessed using the syntax of subroutines, but do not conform with the semantics of FORTRAN subroutines, so that the semantics of such things as assignment statements and even simple expression evaluation may be disrupted. In Prism, the use of an extension in a syntactic combination such as function application would be permitted only after establishing either that the extension conforms with the semantics of application (in this case, that it is of type function), or making an appropriate extension to the semantics of application. What Prism permits, that other languages do not, is both of these alternatives, as well as the means to express them. Moreover, it demands that one or the other, or a combination, be accomplished, so that consistency is assured.

The ability to adjust the semantics of operators in Prism hinges on severing the rigid binding of semantics to syntax. This is the primary motivation for introducing the idea of interpretation, whereby the properties associated with an operator, and its scope of applicability, can be modified in response to changing circumstances. Two important kinds of interpretation shift are generalization and restriction. A mathematical example of the former is the generalization of power series on \mathbf{R} to formal power series. An even more striking example is provided by MacLane's account of the genesis of the concept of category, aided by a shift in notation from the set-theoretic $f(X) \subset Y$ for the signature of a function to the less committal $f: X \rightarrow Y$ [Mac71]. Restrictions are equally important, for example when something new shows up on the horizon that violates a formerly general law, which must now be hedged. For example, the unrestricted rule of β -conversion, $(\lambda x. e_1)e_2 \rightarrow e_1[e_2/x]$, has to be restricted when a nondeterministic choice operator is introduced. (In the long run, one would also like a new general rule covering the choice operator, but in the interim the restricted rule justifies retaining results obtained before the disruptive extension was made.)

Of course, the vocabulary and concepts of the language may grow well beyond the capacity of any one person, or machine, to grasp them all, but this is to be expected. Nobody knows all the words in English, nor more than a handful of the concepts that are found in our libraries and the combined minds of all our experts. As the complexity of the persistent context grows, we may expect limitations on the speed and capacity of individual machines to lead to some kind of division of labor amongst them. Burgeoning complexity will also create constant pressure to generalize and abstract knowledge, so that large volumes of specific facts may be replaced by methods for recomputing them on demand.

As remarked earlier, the historical continuity and interpretive sensitivity of Prism allows the system to assimilate new concepts and techniques, and thus become increasingly useful. Assimilation is the key to a cumulative software technology, in which the benefits of new technology may be propagated to existing applications, and the benefits of past technology are automatically conferred on all future applications. This cumulative platform stands in sharp contrast with existing platforms, which are fragmented, duplicative, and lack the ability to propagate existing technology forward, or new technology backward.

The central importance of historical continuity underlying the cumulative strategy forces the language/system designer to focus on conversations instead of isolated utterances. A conversation

users – nothing can be left “outside” the persistent context. This includes inconsistent and incomplete utterances, fictives, and other flavors of nonexistents, such as round squares and flying horses, the reason being that it is impossible to discuss and reason about partial or incorrect specifications if they are excluded from the language.⁶

Although in some respects the Prism system is a kind of “epistemic engine”, and takes some of its inspiration from natural language, it is neither a project in “machine learning”, nor in “natural language understanding”. We have no plans to incorporate, nor contribute any advances to, specific technologies aimed at enabling systems to learn, nor are we attempting to develop a system which converses with the user in natural language. Rather, Prism seeks to enrich formal language with features that have some of the more powerful descriptive capabilities of natural language.

On the other hand, the predefined syntax and notations of Prism exploit a number of features from natural language, complementing and enriching a style of mathematical notation which has some novel aspects of its own. To some extent, the features of natural language are better suited to open-ended, incomplete, and distributed specifications than are their formal counterparts, which are designed to eliminate lexical, syntactic, and semantic ambiguities by restricting expressions to highly stylized syntactic forms. The goal here is to develop a syntax in which one can write in the style of a good mathematical text, in which formulaic notation and English description are intermixed. More will be said about syntax in §10.

Beyond any particular syntax, we require that it be possible for machine learning, natural language, and other AI technology to be programmed and integrated with the system through its use. This requirement obliges us to anticipate and provide the potential for such things in the basic design. After all, the limitations on the dimensions and degrees of language extension imposed by existing languages is at the heart of the problem we are trying to solve.

5 Basic Organization of the Prism Language System

The central focus of the Prism language system is the persistent context, in which all information available to the system is recorded and organized. This includes all information about the system and language itself, as well as all information about applications and the tools used to develop them.

Superimposed on the persistent context is a conversational context, which supplies a parser, generator, and interpretation for the basic constituents of utterances. The parser and interpreter

⁶Programming languages guarantee consistency by excluding inconsistent utterances. This makes consistency into a kind of language property. In Prism, consistency (or lack thereof) is expressly *not* a language property – it is a property of utterances, which they may or may not possess. The problem, as we see it, is not to exclude inconsistency, but to manage it.

typically have to interact as coroutines because parsing may depend on the interpretations given to proper names, anaphoric references, and indexicals generally. Moreover, the parser may contain internal references to parsing rules which may, in principle, vary with context.

Interpretations are synthesized from three basic sources of information: the global persistent context, the local context, and external channels. The local context is structured according to a dialogue model, which determines both how information is gathered and made accessible during the course of a conversation, and what the possible continuations for a conversation are at any given point, including the intent, or role, of a given utterance. In other words, the dialogue model determines a set of possible histories, and the way in which past and future events condition the interpretation of present utterances.⁷

External channels constitute the coupling of the system to its "environment", which is to say, the interface provided by the physical computer. This includes all hardware operations and data representations, including clocks and counters, and access to devices, including processors, memories, networks, and i/o devices interacting with sensors, effectors, storage devices, and users. These things ultimately supply the grounding of all information in the system regarding its "reality". For example, the system's notions of time, which are involved in the specification, analysis, and implementation of concurrent and real-time applications, are grounded in the locally sequential behavior of processors, channels, and clocks, both internal and external.

An important aspect of the system is that all components and concepts of the system itself are represented in the persistent context. These representations provide the necessary "hooks" for reasoning about and enhancing the system, and also provide access to the system components for reuse in arbitrary applications. For example, an application which needs a parser could use the system parser, or derive a new one starting with the system parser as a base.

A necessary consequence, and benefit, of self-description is that nothing in the system, or language, is primitive in the sense of being an unanalyzed, externally defined concept. The advantages of being primitive-less are striking, and first came to our attention when we invented a primitive-less internal representation for programs, known as IRIS, and used it to develop an Ada compiler. IRIS is basically an abstract syntax tree representation, except that there is only one nonterminal class, instead of one for each primitive operator or construct of a particular language. Initially, each nonterminal in a program has an associated operator symbol, and semantic analysis is responsible for replacing each operator symbol with a reference to the declaration of an operator. This analysis is performed in the context of an IRIS tree in which the basic operations of the programming language have been declared, along with any operations used to make

⁷The syntactic arrangement of programs into sections like declarative regions and bodies, or the simple parse-evaluate loop of interactive languages like ML, are rudimentary examples of dialogue models. In these models, past history results in a simple binding environment for names, interpretation of current utterances is purely a matter of looking up the binding of a name, or supplying the fixed interpretation of a symbol or construction, and current interpretations can be related to future events by suspending them and posting an obligation for completion, as in the case of incomplete type declarations in Ada.

those declarations.⁸ This obliterates any distinction between applications of language constructs and applications of user-defined functions, so a single simple, uniform algorithm can be used to resolve and check the semantic composition of the entire program, with no special processing for the basic constructs of the language. Similar simplifications were realized in every phase, resulting in an optimizing Ada compiler which has only a tenth the number of source lines as compilers of comparable speed and code quality. Moreover, the compiler can essentially be used to compile programs in any language whose semantic composition rules are consistent with those of Ada. The only additional information needed is a grammar, and declarations of any constructs which are not specializations or trivial compositions of the generalized Ada constructs defined in the existing language definition package.

Similarly, Prism's persistent context includes a collection of predefined concepts and mechanisms. A relatively small subset of the predefined concepts are essential in that they form a spanning set of concepts, but even these are not to be regarded as primitive. These essential concepts must simply suffice to establish the lowest-level linkage of the system with its environment, and to enable the definition of all other concepts. All other predefined concepts of the language may be regarded more as standard library packages. In selecting and designing standard library packages, we have attempted to confine ourselves to things which are frequently used, and which are too expensive to define from scratch, or for which there are clear advantages to standardization. An example of the former is the type array; an example of the latter is the type Boolean.

Prism includes a variety of abstraction mechanisms, one of which is context abstraction. This can be used to define subcontexts of any context, thereby allowing information about some topic to be packaged and treated as a unit. As with all other concepts in Prism, contexts are semantic entities, which have no fixed relationship to syntax. Hence there are no syntactic primitives, such as syntactic nesting of scopes, which limit the ways in which contexts can be specified and composed. In particular, contexts may overlap lexically, or be entered and exited intermittently.

Another important abstraction mechanism is embodied in the notion of *type*, which enables general reasoning in the system, divorced from the specific details of any particular object or objects. The most prominent features of the Prism type system are outlined in §9.

6 Persistent Ideas

IRIS is a good generalization of abstract syntax that integrates the representation of syntactic and semantic information in a language-independent form. We have found it inadequate as an internal form for Prism, however, because it assumes that all compositions are functional, or at least categorial. The need to handle partial structures, the frequent lack of operator/operand

⁸Thus all information about every operation that is used either in a program or in the language definition is represented uniformly in IRIS form. Mathematically, an IRIS language definition can be viewed as a set of mutually recursive definitions with no ground terms. Taken in isolation, therefore, the IRIS structure constrains the class of possible interpretations, but does not determine a unique interpretation.

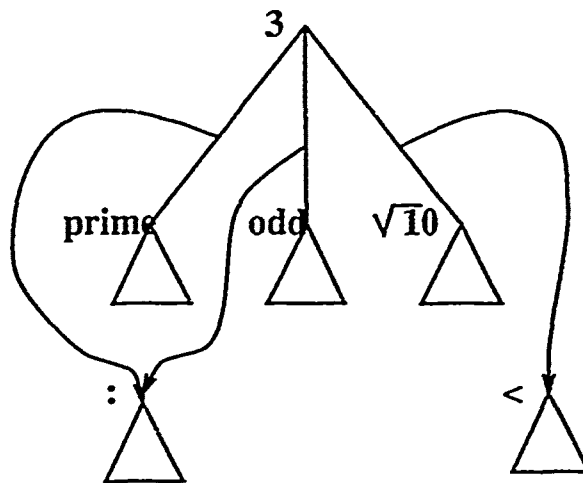
asymmetry, and the breakdown of compositional semantics all contribute to the inadequacy of the functional form of IRIS.

As it turns out, many computational linguists have abandoned trees for similar reasons, replacing them with *feature structures*. [Kni89] is an excellent survey which explains the basic ideas and (for us) the important advantages of feature structures and unification on them. For those unfamiliar with the idea, suffice it to say that a feature structure replaces the rather rigid idea of a distinguished operator dominating some fixed list of operands with an unrestricted list of features, and has much in common with Minsky's frames and related "knowledge representations". Each feature structure represents some partial collection of information, and unification of feature structures requires only that the common parts be reconciled, consistent with any constraints among the parts.

Unfortunately, feature structures, frames, and the like suffer from the same deficiencies that led to the invention of IRIS, namely that the features are labelled with tokens denoting primitive concepts and relations which are externally defined, and about which no information is represented in the system itself. Happily, the remedy is the same: to replace the labels with references to structures of the same kind which represent the semantics of the intended relations. We call these generalized feature structures *ideographs*, with the somewhat immodest and perhaps misleading implication that they are structures representing ideas.

Formally, an ideograph x is a (non well-founded!) set of pairs $\langle r, v \rangle$ where r is an ideograph representing a binary relation, and v is an ideograph representing something that stands in the relation (represented by) r to (the interpretation of) x .

An abstractly equivalent formulation is that ideographs are the nodes of a directed multigraph equipped with a mapping ρ of edges to nodes. An edge $e = \langle x, y \rangle$ represents a binary relation between x and y , where the relation is represented by the ideograph ρe . This provides a convenient way of depicting ideographs as directed graphs with arcs from edges to nodes representing ρ , as shown in the figure below. This sample ideograph represents the number three, which has the properties of exemplifying the types prime and odd ("." is the "exemplifies" relation), and of being less than the square root of ten.



In practice, a wide variety of ideograph representations may be used in the Prism system, providing greater efficiency or convenience for various ideograph types. For example, direct representations may be provided for n -place relations instead of forcing everything to be decomposed into binary relations, or representations may be provided for special composition forms, such as function application, for which a trivial adaptation of IRIS will do.

From an abstract perspective, the persistent context is a network of ideographs. Intuitively, an ideograph is a kind of mental entity having identity and a set of associated properties. The properties need not be consistent, or correspond to any real or even possible object, as in an ideograph combining the contradictory properties of being round and square, which intuitively represents the idea of the round square, a classic nonexistent.

Ideographs play the role in Prism that S-expressions play in Lisp, in that a rudimentary "pure Prism" system can be built exclusively in terms of some standard representation of ideographs, together with an interpreter for a small initial context. As with pure Lisp, an extremely simple syntax can be used to specify ideographs in the standard representation.

The notion of identity in the persistent context is universal, in that no two distinct ideographs can ever have the same identity, anywhere. Nor can the idea of identity be reduced to independent notions, such as location or proper name, because these things are subject to change. In practical terms, this imposes a requirement for each ideograph to have a unique, universal, location-independent identity (see [KC86]). This is one of the fundamental factors in our treatment of persistent object management in Prism, of which more will be said in §8. Another is the degree to which an ideograph is mutable (subject to change).

There is a logic of ideographs which determines when certain ideographs follow from others, leading to notions of type and subtyping. At the most basic level, the logic is consistent with

structural unification, in that the unifier of a set of ideographs implies each of them. Beyond this, there are implications which have to do with the content of individual ideographs, such as that every prime number greater than two is odd. When unification is extended to the whole logic, ideographs are seen to generalize Ait-Kaci's ψ terms, as well [AK84].

Of course, a network of ideographs in isolation doesn't represent anything; it is just a complex structure. Ideographs are infused with meaning by *interpretation*, which projects the universe onto them. We have very little to say in general about the universe, what its internal structure might be, or even if it has internal structure in any meaningful sense. All we can say is that ideographs and their logic are supposed to form an abstract model of the universe, which ignores many distinctions and details, and is thus an approximation at best. We don't even want to claim that every interpreted ideograph, like an ideograph representing the idea of the Statue of Liberty, corresponds to some real object, because that would lead us to puzzle over whether the Statue of Liberty is the same *object* now as it was before its restoration, and similar conundrums. What is constant is the ideograph, and an ideograph has significance in the universe if some aspect of the universe is projected onto it. The round square ideograph, by contrast to an ideograph representing the Statue of Liberty, has no significance in the universe, though its component properties do.

7 Language and Ideas

We have stated briefly that interpretation projects the universe onto ideographs, but how are ideographs related to language? Considered simply as constituents of the universe, linguistic phenomena give rise to ideographs through interpretation just as other phenomena do. These ideographs represent only the linguistic phenomena themselves, however, and not their interpretation. In plainer jargon, the direct projection of an utterance is its *abstract syntax*, and the interpretation process is *parsing*. Other, less common, interpretations include the analysis of spoken language as a sequence of phonemes, or more crudely as a sound wave, or the interpretation of a written text as a pattern of glyphs.

However, the important feature of linguistic phenomena, which distinguishes them and gives them their power, is that they are supposed to stand for something else; they *signify*. For example, the ideograph of the expression " $3 + 4$ " is a structure having three major features: an operator and a left and right operand. This ideograph, in turn, signifies the sum of two numbers, namely three and four.

The interpretation of linguistic phenomena in these two stages is the source of the use/mention distinction, and motivates the factorization of language processing into semantics and syntax. Pragmatics is supposed to complete the semiotic picture by explaining the interpretation and role of utterances in an overall context.

The semiotic factorization is only approximately correct, however, because the concerns of pragmatics are manifested also in the parsing and semantic phases. That is, both the parsing

of an utterance and the interpretation of its significance are influenced by context. Consider the sentence "He is taller than he is old", which most English speakers accept. The interpretation one gives to "taller" in this context is not the usual ordering on heights, but rather a relation between heights and ages according to some unspecified scale. The interpretation is best explained as a context-induced perturbation, or mutation, of a standard interpretation supplied by a lexicon. A more standard, but less satisfying, explanation is that the lexicon supplies a denotation which is somehow parameterized by context in such a way that the meaning in this situation can be derived. The difference between the two views hinges on the question of whether the interpretation in every possible situation has to be anticipated when making entries in the lexicon.

To some extent, the intuition that pragmatics is not separable from syntax and semantics underlies Kamp's discourse representation theory (DRT) of natural language understanding [Kam88], but DRT does not take it far enough. The fact that large parts of the context of English speakers are relatively uniform and stable over time accounts for our ability to communicate, and in fact "the English language" is nothing but this shared context. On this account, the notion of "a language" is not *a priori*, but empirical and derivative.

These considerations led to the novel conceptualization of language which underlies Prism. The basic theme is that the purpose of language is not to denote, but to stimulate ideas. That is, we see language not so much as a vehicle for naming and transmitting semantic objects from one speaker to another, but more as a way for one speaker to provoke a certain kind of reaction from the interpretive capacity of another. The difference is that the response elicited by a given stimulus depends intimately on the recipient's orientation, making meaning relative instead of absolute. As further support for this view we remark that a great deal of conversation is aimed at orientating the participants relative to one another so that a system, or community, is established with a set of regularities and invariants that permits more efficient communication.⁹

At the center of the language process lies the interpreter, which has as its goal the reconciliation, or *unification*, of linguistic data with its interpretation of the universe. That is, the meaning of an utterance is a unifying ideograph which is simultaneously an interpretation of the utterance, and a partial picture of the universe.

To obtain this unifier, the interpreter first "parses" the utterance to obtain a relatively uncommitted abstract structure, with no interpretation. Interpretations for individual components of the initial ideograph are offered by the context, and the interpreter attempts to choose a combination of these which together form an interpretation which is congruent with its interpretation of the rest of the universe.¹⁰ Some of the component interpretations may require that the initial structure be refined, making some features subordinate to others. When ideographs conflict, they may be reconciled by dropping or relaxing some of their features as necessary. Meanings may be inferred for previously unknown words or phrases because they are essentially variables which are

⁹How often have you gotten into a heated debate with someone only to discover after much discussion that you had no real disagreement, but merely misunderstood one another?

¹⁰The astute reader will recognize this as the Prism analogue of overload resolution in Ada!

filled in by unification. By symmetry of the unification process, new information about some part of the system's world model may be inferred. In this way metaphor, analogy, and oblique uses of language become possible. Moreover, because interpretation is naturally partial, overspecification is avoided, and the amount and kind of information that may be supplied to refine an ideograph or its interpretation is not restricted in any way.

8 Persistent Object Management

There are several important requirements for the management of persistent data in Prism. These requirements are concerned primarily with the integrity and efficiency of creation, destruction, storage and access of the persistent data through both space and time. The primary requirements we place on persistent object management in Prism are discussed in the next several paragraphs.

As explained earlier, the persistent data items in Prism are ideographs; they are the "objects" to be managed. From an object management perspective, an object is a container for a data value of arbitrary type (including other objects). There must be no restrictions on the types of values that can be made persistent. Each object must have an identity which is *unique* (to avoid confusing it with other objects), *universal* (so that knowledge of an object's identity is not invalidated in one part of the system when changes are made elsewhere) and *location-independent* (because the location of an object may change in the course of its lifetime).

Integrity is a pervasive goal for Prism; type and identity integrity are central to persistent object management. It matters little how good other aspects of a system are (*e.g.* how fast it runs, or how much it encompasses), if it produces results that are incorrect or unreliable. Because types are used to express the formal properties of data and because of the nature of identity, the persistent object management mechanisms must enforce the typing and identity mechanisms.

Users and developers of software systems must be given control (though not required to exercise it) over the placement of persistent objects as they move between peripheral memories (including removable media) and main memory, as they move across nodes of a network and indeed between networks, and as they are replicated for reasons of efficiency and backup. Controlling the granularity of the data that can be independently placed is essential in achieving performance. Users and developers must have access to mechanisms that remain efficient over the full range of object granularity.

Traditional programming languages, operating systems and databases have addressed some aspects of persistent information management, but each has its shortcomings. The underlying assumptions of operating systems and databases are not valid for the data in a software environment. In violation of the operating systems assumptions, correct and effective management of software development objects requires intimate knowledge of their types, which are expressed (implicitly or explicitly) in the objects themselves, in order to ensure type integrity across tool invocations and manipulations by human users. In violation of the database assumptions, the types of data in

a software development environment at any time are specified by that data, the number of items of information of any given type may range from one to millions, and some objects are virtual (and are only instantiated dynamically). Furthermore, both the user-defined names relied upon by operating systems and the value-oriented names of databases compromise the integrity of object identity.

High-level programming languages are better at managing this kind of complex information. Unfortunately, programming language designers and implementors have never really addressed the problems of managing resources beyond local memory. Instead, they have relied on databases and the file systems of operating systems to manage persistent data.

More recently, research on managing the persistent objects of the entire software activity has been conducted. Much of this work (in areas such as software development environments, CAD/CAE systems, object orientation, databases, database programming languages and persistence) is relevant to the Prism effort in many ways, especially by providing basic techniques for instantiating and realizing our goals.

We should emphasize that similar remarks apply to many other areas of language and systems research and technology, on which the success of the Prism effort depends, and without which its success would be pointless. It is the barriers to accessing and integrating the vast array of existing, potentially useful technology which we decry, not the technology itself!

9 Types

Types in Prism are partial information structures used for general reasoning about Prism programs. A Prism processor's facility at manipulating types is one measure of its "intelligence", or of how well-educated it is. Similar remarks apply to Prism programmers. From this point of view, the Prism type system refracts into two parts: information structures, and the associated logic(s). The issues here are foundational: what are the partial information structures; what and how do they mean; what is the role of logic; and so forth.

From a different angle, the type system appears as a collection of basic abstractions, such as arrays, tasks, and interpretations, together with mechanisms for generalization and specialization. This is the view traditionally used to describe the type systems of programming languages such as Ada and Common Lisp. It is important and useful because it conveys the higher-level syntactic features available for type specification, and the interrelations among those features (*e.g.*, subtype/supertype relations).

A third angle reveals a number of fundamental relationships among types, and strategies for exploiting those relationships. For example, specialization is a basic strategy for deriving subtypes (exploiting the supertype/subtype relation) and class abstraction is a basic strategy for deriving metatypes (exploiting the metatype/supertype or so-called class/instance relation). These basic

strategies are reflected in the generalization and specialization mechanisms alluded to earlier.

From another perspective we perceive a logical spectrum, with varying degrees of richness, complexity, specificity, and intensionality. These reflect the infinite variety of purposeful interpretations. For example, one logic is germane to considering a procedure as a constructive witness of a functional specification; another logic addresses it as a consumer of resources; yet others may be used to evaluate its potential for reuse, or its lack of covert channels. Along the specificity axis we see a tradeoff, between the level of detail and the complexity of automation. Thus a simple propositional logic of functional specifications corresponds to the familiar automatic signature analysis of programming languages like Pascal and Miranda, while the more detailed constructive logic of PRL demands more sophisticated, and necessarily incomplete, automation, and defers to the programmer for many difficult insights.

A fifth point of view regards types in terms of representation and implementation. All too often in programming languages, the notion of type is anchored to these, rather special, relations. The most common mechanism for specifying an "abstract data type" in programming languages is through an isomorphism to some quotient, or, in the vernacular, encapsulated representation. In practice, though, the fact of isomorphism is not taken as an incidental fact used to specify the abstraction, but binds the abstraction inexorably to that representation, to the exclusion of all others. This is a state of affairs which we deplore. Still worse, these mechanisms invariably confound abstraction with presentation [*sic*], so that two "abstractions" which are mathematically isomorphic, but which are specified using different representations, axiomatizations, or even merely different names, are considered distinct! In Prism, we recognize a fundamental relation of *representability* between types, which may be used to specify a type, but which has none of the (over-)committing force of current programming language constructs. We also distinguish between *effective* and *noneffective* representations, depending solely on the computability of the representation functor. An *implementation* is an effective representation in terms of a physically realized type, *i.e.*, the operations and resources of some physical system, such as a computer. To illustrate the distinction, note that many types which are effectively representable are not implementable. In any event, mechanisms for specifying, composing, and deploying representations and implementations are of obvious importance for practical programming.

This brief list of aspects of the Prism type system is by no means exhaustive, but it highlights those issues which have most influenced its design. In order to give a somewhat more instantiated impression of some aspect of Prism, we now provide a few details of Prism types. For more information about the theoretical properties of Prism types, see [Shu89b]. For more information about the standard library types and abstraction mechanisms initially planned for practical applications of Prism, see [Shu89a].

A *type* is a set of properties closed under entailment. That is, if $t \models \phi$ (ϕ a unary predicate) then $\phi \in t$. In order to specify types finitely, we adopt the notation X^{\models} for the closure of X under \models .

For example, let us define *oddprime* to be the type $\bigcirc x \{x \text{ is prime}, x \text{ is odd}\}^{\models}$.¹¹ *oddprime*

¹¹The notation $\bigcirc x$ is meant to identify the variable over which the predicates in the immediately following type

includes the properties " $\odot x(x+1 \text{ is even})$ ", " $\odot x(x > 2)$ ", and " $\odot x(\forall y, m. y < x \wedge m|x \wedge m|y \Rightarrow m = 1)$ " along with infinitely many other consequences of the two properties originally specified.

The subtype relation is defined as follows: $x \preceq y$ if $\forall \phi \in y. x \models \phi$. That is, subtypes specialize their supertypes. An equivalent, if initially counterintuitive, characterization is that $x \preceq y$ if $y \subseteq x$.

$\odot x\{x \text{ is prime}, x \text{ is odd}, x^2 < 10\}^\models$ is a subtype of *oddprime*. Similarly, $\odot x\{x \text{ is odd}\}^\models$ is a supertype of *oddprime*.

To a first approximation, an individual (description) is an example of a given type if it satisfies all of the properties of that type.¹² Formally, $x:t$ if $\forall \phi \in t. \phi(x)$. Actually, we need to refine this definition somewhat, but let us adopt it for the moment, to show what is right and wrong about it. A few important Prism types are the following.

$$\begin{aligned}\diamond &= \emptyset^\models \\ \text{type} &= \odot t\{t \text{ is an } \models\text{-closed set of properties}\}^\models \\ \text{metatype} &= (\text{type} \cup \odot x\{\forall y: x. y: \text{type}\})^\models\end{aligned}$$

The reader can easily verify that these satisfy the following "axioms".

- A1) $\diamond: \text{type}$
- A2) $\text{type}: \text{metatype}$
- A3) $\text{metatype}: \text{metatype}$
- A4) $\text{type} \preceq \diamond$
- A5) $\text{metatype} \preceq \text{type}$

It is also easy to show that the following are consequences of A1-A5.

- C1) $\diamond: \diamond$
- C2) $\text{type}: \text{type}$
- C3) $\text{type}: \diamond$

specification range. In this regard it serves the same purpose as the bound variable in a set specification $\{x|\phi\}$, but the two notations mean quite different things!

¹²I use the word "example", instead of "member", because the members of a type are properties. For example, " $\odot x(x > 2)$ " is a member of *oddprime*; "3" is not. What "3" is is an example of *oddprime*.

C4) *metatype: type*

C5) *metatype: ◇*

C6) *metatype* \preceq \diamond

Proof: C1 follows from A1 and A4 by subsumption. C2 follows from A2 and A5 by subsumption. C3 follows from C2 and A4 by subsumption. C4 follows from A3 and A5 by subsumption. C5 follows from C4 and A4 by subsumption. Finally, C6 follows from A4 and A5 by transitivity. \square

Unfortunately, the type system as presented so far admits the familiar paradoxes. For example, we can define an analogy to the Russell type $R = \odot\{\neg(t:t)\}^{\models}$, and draw the usual contradiction. The problem, alluded to earlier, lies with our notion of examplehood.

The source of the problem is with the requirements for examplehood, which are too loose. Before something can be subject to the general reasoning applicable to a type, it must already exist as an example of some more specific type. Technically, we have the rule

$$\frac{x:y}{\forall\phi \in y. \phi(x)}$$

but the converse of this rule does *not* hold. Instead, we have the following axiom of *unit examplehood*, which explains how we come to have any examples at all.

$$a: \odot x\{x = a\}^{\models}$$

Using these rules, one can prove that the assumption $R: R$ leads to a contradiction, but the assumption that $\neg(R: R)$ leads only to the conclusion that $\forall\phi \in R. \phi(R)$, which is insufficient to establish the contradictory $R: R$.

10 Language and Notation

The goals of Prism impose a number of constraints on the syntax of the language. Although the design of the syntax is far from finished, the requirements for it are clear, and a number of syntactic mechanisms contributing to meeting those requirements have suggested themselves. The fact that Prism is a two-way language, with the system itself generating large portions of the program text, means that readability is at a premium. The user will be obliged to understand and maintain large bodies of code that he did not write, so that traditional *write-once* syntax becomes much more undesirable. Moreover, it is not enough just to allow the user to say what he needs to about a computation; he must also be able to say it easily, and not have to shoehorn

it into s-expressions or function form. The goal of expressivity requires a notation that is at once powerful and natural.

Exploiting today's display technologies promises to go a long way toward achieving this expressivity. The use of graphical representations directly manipulable by the user will reduce the asymmetry of the system, since the same representation can be used on input as on output. Improved typography, the use of multiple typefaces and stylistic variants, and possibly even a two-dimensional syntax, will contribute to readability. Coupled with such use of graphics, lexical extensibility will make the notation more natural by allowing it to conform to existing traditions.

Prism's property-based type system and its use of a generalized feature-structure formalism impose a requirement for a simple and convenient mechanism for talking about properties. This can best be achieved by elevating adjectives and prepositional phrases to first-class status in the language, so that complex compositions of properties can be expressed conveniently.

Prism's view that interacting with a computer is a kind of dialogue, its commitment to supporting intensionality and incompleteness, and its avoidance of overspecification, all require a language quite different from the purely extensional languages of the past. The most promising approach is to integrate a number of features from natural language. For example, the use of anaphora, including both pronouns and anaphoric descriptions, is a natural way of expressing context dependency. The use of variable-free quantification provides a convenient way of avoiding overspecification. The advances made in computational linguistics over the last ten years make us optimistic that these features can be adopted at reasonable cost.

11 Related Work

Although the Prism model of language has been influenced by reading and reflection on linguistics and the philosophy of language, it took shape independent of any particular tradition or set of ideas. At the outset, our goal was simply to remedy a number of shortcomings of conventional programming languages which we saw as standing in the way of significant long term progress in software engineering. What we have ended up with is a fundamental revision of the computer language framework. It is curious to note that our conceptualization was developed in ignorance of the philosophical and critical traditions of phenomenology and hermeneutics, which have only recently come to our attention. The language problems we identified and struggled with were the demons of Brentano, and we are discovering much in common between our conclusions and those of Husserl, Meinong, Heidegger, and others. We also find much that is sympathetic in the more recent work of Zalta [Zal88] and Winograd and Flores [WF87]. A complete discussion of the relationship of our ideas to these others is well beyond the scope of this overview, but a sketch of the main points may help to place the current work in perspective.

Programming languages, modelled after the formal languages of analytic philosophy and mathematical logic, represent a respectable and coherent tradition of thought about the nature of

language that has been remarkably successful in explaining the use of language for such things as mathematical expressions, proofs, and encodings of algorithms and data structures. It has been less successful, however, in explaining natural language, Montague to the contrary notwithstanding. Situation semantics [BP83] was stimulated by the shortcomings of the model-theoretic account of language, even when augmented to handle modality, temporality, and all other forms of intensionality. Unfortunately, situation semantics has problems of its own.

Our concept of language is more in accord with the traditions of phenomenology and hermeneutics, which hold that meaning cannot be attached objectively to words but is to be found in their interpretation and use by a community of speakers. The case for this tradition is made quite eloquently by Winograd and Flores, who conclude that computers cannot understand human language. Although we don't dispute their conclusion, we take issue with its relevance. The question for us isn't whether computers can understand *human* language, but rather, what kind of language is possible in a community composed of people and computer systems?

Among rival theories, Zalta's seems to offer the most complete account of several intensional phenomena, though he doesn't have much to say about others, such as indexicality and time. We have carefully examined each of the issues that Zalta's theory covers and demonstrated to ourselves that our accounts are at least as good in all cases. Unfortunately, the detailed exposition of these results has proved far too lengthy for the current paper, and its publication will have to be deferred.

12 Conclusions and Prospects

Prism seeks to eliminate the barriers inherent in current software technology which inhibit sharing, reuse, and long-term progress. To date, our primary accomplishments are a clear understanding and statement of the problem and its causes, and a clear understanding of what needs to be done to solve it. We have outlined the requirements for a solution, and designed the high-level architecture for a system, namely Prism, which meets those requirements. We have also partially instantiated this design in several areas.

As of this writing, the focus of the project has shifted toward greater instantiation of our ideas, and we expect to accompany this instantiation with some prototype implementations so that we can gain a better understanding of the technical prerequisites of a production implementation. These experiments are also aimed at establishing what is feasible within the limitations of our time, expertise, and other resources. Our findings will influence the initial instantiation of Prism, but of course wherever that initial instantiation lies it will not be the end of the story. Once Prism has been sufficiently primed, it can be enhanced indefinitely.

References

- [AK84] H. Aït-Kaci. *A Lattice Theoretic Approach to Computation Based on a Calculus of partially Ordered Type Structures*. PhD thesis, University of Pennsylvania, Philadelphia, PA, 1984.
- [BP83] J. Barwise and J. Perry. *Situations and Attitudes*. The MIT Press, Cambridge, 1983.
- [Kam88] H. Kamp. Discourse Representation Theory: What it is and where it ought to go. In A. Blaser, editor, *Natural Language at the Computer*, pages 84–111, Heidelberg, February 1988. Springer-Verlag.
- [KC86] S. N. Khoshafian and G. P. Copeland. Object Identity. In *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference (OOPSLA '86)*, pages 406–416. *SIGPLAN Notices*, 21(11), 1986.
- [Kni89] K. Knight. Unification: a Multidisciplinary Survey. *ACM Computing Surveys*, 21(1):93–124, March 1989.
- [Mac71] S. MacLane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1971.
- [Sco76] D. Scott. Data Types as Lattices. *SIAM J. of Computing*, 5:522–587, 1976.
- [Shu89a] Jon Shultis. The Spectrum of Types in Prism. Technical Report DRAFT, Incremental Systems Corp., Pittsburgh, in preparation 1989.
- [Shu89b] Jon Shultis. Towards Types that Make Sense. Technical Report DRAFT, Incremental Systems Corp., Pittsburgh, July 1989.
- [Smi84] B. C. Smith. Reflection and Semantics in Lisp. In *Proceedings of the 11th Symposium on Principles Of Programming Languages*, pages 23–35, Salt Lake City, Utah, January 1984. Association for Computing Machinery.
- [WF87] T. Winograd and F. Flores. *Understanding Computers and Cognition*. Addison Wesley, Reading, MA, 1987.
- [Zal88] E. N. Zalta. *Intensional Logic and the Metaphysics of Intentionality*. MIT Press, Cambridge, 1988.

Development of Research Ideas

for

Languages Beyond Ada and Lisp

Incremental Systems Corporation
319 South Craig Street
Pittsburgh, Pennsylvania

Prism -- Design Notes -- 890109 -- DAF

Design Goals

Property based type system encompassing persistent as well as program data.
Ability to predict behavior of programs.
Full spectrum language encompassing all aspects of a computation.
Distributed specification of computations.
All concept of computation and language first class citizens of language.
Separation of behavioral specification from implementation.
Support for incomplete and inconsistent programs.
Ability to specify efficient implementations independent of a particular program.

Summary of Language

Design Principles

Objects and Values

An *object* is anything subject of a computation, that is, anything that can be computed, described, communicated, queried, modified, shared, or mentioned in a computation. Objects are also called *data structures*.

Atomic objects do not have subcomponents. *Composite* objects have subcomponents. All objects are either atomic or composite, but never both.

atomic : type
composite : type

Assignment is the only operation that modifies objects. It can be applied only to objects that are *assignable*. Each assignable object has an associated target object. The assignment operation replaces the target object. Assignable objects also have a target type that restricts the target objects that may be assigned. The target type of an assignable object may be specified with its type, but becomes a property of each object. If the target type is sharable, the target object will be second argument to the assignment. Otherwise it will be a copy of the second argument.

assignable : function (t: type) return type
assign : ($\forall t$: type) procedure (x : assignable t; y : t) return statement

The dereference operation returns the target object that was last assigned. Dereference often can be implicit in a program's syntax. In particular, if some number of dereferences of an actual argument will produce an object of the required type, then explicit dereferences are not required. Any ambiguity that might be introduced by dereferencing different numbers of levels, is resolved by choosing the solution with the minimal number of dereferences.

dereference : ($\forall t$: type) function (x: assignable t) return t

Immutable objects cannot be modified; that is, they are not assignable and cannot have assignable subcomponents. Immutable objects are also called *values*. Values can be

atomic or composite. Atomic values are called *scalars*. *Mutable* objects can be modified. That is, they are either assignable or have assignable subcomponents. Most objects are values, but much of the complexity in programs arises from mutable objects.

```
immutable : type  
mutable : type
```

{should be moved to type section} Two operations are defined on objects of any type. *Equality* is used to distinguish objects. A *copy* of a value is, or is indistinguishable from, the value itself. When applied to a mutable object, *copy* creates a new object similar to its argument except it and all of its nonshared subcomponents have been replaced in a way distinguishable by equality.

```
eq : function ( x, y: <>) return boolean  
copy : function ( x: <>) return type_of x
```

A copy of an assignable object is also distinguishable because assignment to one will not be reflected in the others. It is sometimes useful to be able to specify whether an assignable object can be shared. Specifically, a *shared* object is one that can be referenced simultaneously by multiple tasks, or that can be simultaneously multiple subcomponents of a composite data structure. All subcomponents of shared objects must be sharable. That is, each subcomponent of a shared object may be immutable or shared, but cannot be a variable. Objects that may have shared subcomponents are called *list structures*. Objects that cannot have shared subcomponents are called *tree structures*.

An assignable object that cannot be shared is called a *variable*. Subcomponents of variables are not restricted; they may be immutable, variable, or shared in any combination. Each assignable object is either a variable or shared, but never both. The type constructors for variables and sharables take a single type valued argument specifying the target type for assignment.

```
variable : function ( t: type) return type  
shared : function shared( t: type) return type
```

The new operation creates an object of the variable or shared type. New also assigns its second argument to the newly created object. The first argument to new specifies the type of the new object. The second argument must be an assignable type, and must designate whether variable or shared. The new object is distinguishable from all previously created objects.

```
new : ( ∀s : type) function ( t: type_of assignable s; y : s) return assignable s
```

Read only is a special property that can be used to prevent assignment to an object and to its subcomponents. The read only operation produces a read only copy of its argument that except for assignment and the read only predicate is indistinguishable from the original. In particular assignments to the original or to subcomponents of the original are reflected in the copy and other non modifying operations produce the same results. Assignment to the read only copy or to any of its subcomponents through the read only copy is illegal. The read only predicate is used to distinguish the copy from the original; immutables are always read only.

```
read_only : (function ( x : <>) return z : type_of x) where is_read_only z  
is_read_only : function ( x : <>) return boolean
```

Types

A *type* defines a set of objects. Each type has a set of properties that distinguish the objects of the type. A type need not be finite or enumerable. The unconstrained type is the set of all objects and is designated by \diamond . Each type is an object. Metatype is the type of all objects that are types, including metatype. Also, metatype is a subtype of type and type is a subtype of \diamond .

```
 $\diamond$  : type
type : metatype
metatype : metatype
```

Scopes and Declarations

A *visibility scope* is a region of a program's execution where certain declarations made within the region can be directly referenced. Visibility scopes can be dynamically embedded in a program's execution and must be lexographically embedded in the program's syntax. The visibility scope operation specifies some of the declarations local to the scope and a computation to be evaluated within the scope. There are other operations that create visibility scopes.

```
scope : function ( dl: list_of declaration; body :  $\diamond$ ) return type_of body
```

Declarations are used to give temporary names to objects within specified regions of a program and to subcomponents of composite data structures. Each declaration has a name and an associated object. Both must be specified at the point of the declaration. The name must be a literal token. Most declarations are on declaration lists, but there is also a declarator for naming other places including statements, scopes, and assertions. Declare operations makes the name visible throughout the most local enclosing scope of the declaration. If the second argument is of a shared type, the associated object as the declaration will be the second argument, otherwise it will be a copy of the second argument.

```
declare : function ( name : literal token; x:  $\diamond$ ) return declaration
declare : function ( name : literal token; x :  $\diamond$ ) return type_of x
```

The target object of a declaration is normally recomputed upon each reentry to its most local enclosing visibility scope. It is sometimes desirable to retain the target object throughout a scope broader than its visibility scope; declarations of this kind are called *own declarations*. Own declarations have an own scope that can be any enclosing visibility scope of the declaration. The target object of an own declaration is recomputed only if the own scope has been reentered since the declaration's visibility scope was last entered. The own specification operation is declaration modifier. It can be applied to single or multiple declarations.

```
own : function ( s : any record_type; d : declaration) return declaration
```

It is illegal to have two declarations of the same name with identical scopes. If the visibility scope of a declaration is embedded in the visibility scope of another declaration of the same name, then the outer declaration is *hidden* within the visibility scope of the inner declaration. A declaration can be directly referenced at any place in a program where it is

visible but not hidden. Direct reference requires only the declarations name. The direct reference operation returns the object associated with the only declaration of the name that is visible but not hidden at the point of the reference.

reference: **function** (name : literal token) **return** <>

Declarations are elaborated in order of their appearance, but there is minimal dependency on the order. There is no requirement that a declaration appear before its references, although such placement often improves readability. It is illegal to elaborate a reference to a declaration before elaborating the declaration. Note, however, that elaboration of recursive declarations (i.e., operations or data types) does not involve elaboration of their bodies. It is also illegal for the elaboration of declarations, including initialization of assignable values, to cause side effects; but the elaboration of declarations may depend on global variables and non pure functions.

In out is a special declarator operation for shared objects. The second argument to an in out declaration must be shared. The object associated with the declaration will be a copy of the second argument. In addition, at each exit from the visibility scope of the declaration, the target value of the object associated with the declaration will be assigned to the shared second argument to in_out.

in_out : **function** (name : literal token; x: shared) **return** declaration

Program Composition

The *abstract syntax* of a program is a composition of calls on operations defined by the language or its users.

Compositions are partitioned into three classes by their result type: declaration, statement, and other types which are called expressions. Declarations are elaborated, statements are executed, and expressions are evaluated.

declaration : type
statement : type
expression : type

Each operation has a *signature* specifying restrictions on the type of each of its arguments. The restrictions specified in the signature for an argument is called the *formal parameter type*. The signature may also specify a type of its result. A composition is legal only if the actual result of each call is an element of the corresponding formal parameter type of the operation to which it is an argument. The signature is part of the type of an operation. A signature is an open scope.

operation : **function** (fpl : list_of fp_decl; result : declaration) **return** type

Unlike other declarations, formal parameters are initialized by calls on the operation. Consequently, the declarator operation for formal parameters specifies only the type of the object being declared and not the object itself. The rules for deriving a formal parameter object from its specified object, are the same as for any other declaration.

fp_spec : **function** (name : literal token; t : type) **return** fp_decl

A *side effect* occurs when a computation modifies an object whose life time (i.e., own scope) is not (dynamically) embedded in the computation. An operation that never causes side effects and whose result depends only on its arguments is called a *function*. An operation that never causes side effects, but whose result may depend on assignable objects with life times broader than its calls, is called a *routine*. An operation that can cause side effects is called a *procedure*. Only statements may have side effects. Thus, the result type of a procedure is always statement and thus need not be specified.

```
function : function ( fpl : list_of fp_decl; result : fp_declaration) return type
routine : function ( fpl : list_of fp_decl; result : declaration) return type
procedure : function ( fpl : list_of fp_decl) return type
```

Declarations immediately within a given visibility scope are elaborated in order of their appearance, but there is minimal dependency on the order. A declaration need not appear before its references, although such placement often improves readability. It is illegal, however, to elaborate a reference to a declaration before elaborating the declaration. Note that elaboration of recursive declarations (of operations or data types) does not involve elaboration of their bodies. It is illegal for the elaboration of declarations, including initialization of assignable values, to cause side effects; but the elaboration of declarations may depend on global variables and routines.

Statements are executed in their order of appearance within a list of statements. Statements are executed only to obtain the side effects they cause. The result often depends on the order of execution.

No evaluation order is specified among arguments that are expressions. Because expressions cannot cause side effects, the result cannot depend on their evaluation order.

Assertions { more to 471053 }

Control Structures

Persistent Data

Tasks and Communication

Implementation + Representation
Implementation Mechanisms

Atomic Data Types

Record Types

A *record type* defines a set of composite data structures of similar structure. Each call on the record type constructor creates new, not necessarily distinct, record types. Each call on the record type constructor is a visibility scope. Each subcomponent must be declared within the scope. The argument to the record type constructor is the list of immediate subcomponent declarations.

```
record_type : metatype
record : function ( dl: list_of declaration) return record_type
```


Each object of a record type is a composite object. It is also a visibility scope. The subcomponent declarations of a record type are replicated for each object of the type. Each call on the record object constructor, `new`, creates a new object of the record type. The object will be distinct unless it is immutable. `New` requires a record type and an aggregate with components corresponding one to one with the declared subcomponents of the record type. `New` returns an object of the record type with subcomponents as specified by the aggregate. The details of aggregate construction and of the correspondence rules are given in section ****. Each subcomponent, `x`, of the resulting record is derived from the corresponding subcomponent, `y`, of the aggregate by the following rules. If `x` is shared and $(\text{type_of } x) = (\text{type_of } y)$, then $x = y$. If `x` is not shared and $(\text{type_of } x) = (\text{type_of } y)$, then $x = \text{copy}(y)$. If $(\text{type_of } x) = (\text{assignable type_of } y)$, then $x = \text{new}(\text{type_of } x, y)$. All other combinations of `y` and `type_of x` are illegal.

`new : function (t : record_type; a : aggregate) return t`

The own scope of a record object created by a `new` operation is normally the visibility scope of the call on the record type constructor that created its type. It is sometimes desirable to further restrict the own scope of a record object. The own scope of a record object may be specified as any visibility scope enclosing the call on the `new` operation that created the record object.

`new : function (t : record_type; x : aggregate; s : any record_type) return t`

Whether by global assignment, by parameter passing, or as an operation result, it is illegal to pass an object out of its own scope. Because bodies of operations are dynamically embedded, objects can be passed to operations declared more globally than the object's own scope.

{move to scope section} It is illegal to declare two record subcomponents of same name unless at least one of them is within an assignable subcomponent of the record. The declaration of a subcomponent of a record object can be selectively referenced at any place in a program where the record object is known, the declaration's name is visible, and the declaration is not within a shared subcomponent of the object. Selective reference requires the record object and the name of the declaration. It returns the object associated with the declaration. Selective reference is also called *dot qualification*.

`reference : function (x : any record_type; name : literal token) return <>`

{move to scope section} The `use` operation specifies that certain declarations within a record object are to act, for purposes of direct reference, as if they had been declared at the point of the call on `use`. The declarations can be referenced by selective reference from the record object and would not cause a name conflict were they declared at the point of use. The latter does not preclude the same declaration being made visible by several `uses` in the same immediate scope.

`use : function (x : any record_type) return declaration`

{move to scope section} The `open scope` operation creates a single record object as a program component. Open scopes are visibility scopes. Like record objects and unlike scopes created using the `scope` operation, subcomponents of open scopes can be referenced from outside using dot qualification. All declarations within an open scope, but not within a more local enclosing visibility scope, are components of the record object created by the `open_scope` operation. They may be referenced by dot qualification on the open scope.

Open scopes are particularly useful for referencing local declarations of assertions when specifying implementations. Open scopes often must be named so that they can be selectively referenced or used.

open_scope : function (dl : list_of declaration; body : \diamond) return type_of body

Outline of a Type System for Prism

Jon Shultis

March 7, 1989

1 Goals

The overall purpose of types in Prism is to facilitate reasoning about the properties of computations and their elements. Consistency checking, bounds checking, verification, derivation, performance and effect analysis are examples of program processing involving types. However, the type system is not beholden to any of these kinds of processing; in particular, we do not require at the outset that any kind of type processing be "static", or even decidable.

At the same time, we acknowledge that most useful processing of type information is made possible by constraints. For example, efficient algorithms for inferring principal types are enabled by the restriction of subtyping to substitutions in type terms. In ML, *int* is a subtype of *tau* by the substitution of *int* for τ , because it serves to specialize the type - provide more information. *prime* is also a subtype of τ , but is not an ML type term. This is why ML types the expression "3" as "*int*", instead of "*prime*", "*odd prime*", or even just "{3}". In sum, the principal type of an expression is the strongest assertion of its properties that can be made when subtyping is limited to substitution.

Historically, designers of formal systems generally and programming languages in particular have set their priorities and developed type systems which are good compromises. In doing so, they have made the tacit assumption that language design and specification are completed by the designer, after which programmers can use it to write software. This assumption forces commitment to a particular set of constraints which achieve the best compromise given the goals and priorities of the language and the available technology.

In Prism, the development of linguistic tools is viewed as an integral part of the process of designing and communicating a software system. This is accomplished by leaving many aspects of the language design incomplete, and providing users with the means to complete them as they see fit. Simultaneously, language features which are traditionally included in order to impose structure and discipline on programmers as well implementation requirements on compilers are demoted to the status of library packages - possible language design extensions that impose constraints in exchange for practical benefits, like static type checking. By the same token, qualified programmers can augment the stock of linguistic tools by contributing their innovations and improvements to the library.

For this reason, the broad outline of the Prism type system is highly ecumenical, but it raises many issues of practical import that will not be addressed in detail here. Suffice it to say that the development

of library extensions which limit the power of the full type system in order to harness it is an important problem which will be taken up elsewhere.

The ensuing exposition serves both as an introduction to the basic ideas of the Prism type system, and as an example (in mathematical English) of Prism programming style. In section 2, we give a partial specification of some of the basic elements of the type system, and proceed to reason about them. What we discover are some apparent difficulties in our prototype type system, which must be rectified. In section 3 additional information is provided about the elements in question; this is an example of what we call "distributed specification". The result is still incomplete, but it solves the problem raised during the analysis of the earlier prototype.

2 A Problem

Having decided that types are supposed to express the properties of things, let us say a bit more about what that entails. To begin with, we need some notation. Let $[x : t]$ signify that x is an example of the type t ; i.e., roughly speaking, it has the properties expressed by t . The notation $[t_1 \preceq t_2]$ will signify that t_1 is a subtype of t_2 , whatever that means.

Although we haven't specified exactly what we mean by these notations, we can state some minimal properties we expect them to have. One is that the two notions are related by the rule of *subsumption*: if x is a y and y is a subtype of z , then x is a z . Another is that the subtype relation is transitive. Formally,

$$\frac{x : y \quad y \preceq z}{x : z} \qquad \frac{x \preceq y \quad y \preceq z}{x \preceq z}$$

Let us turn now to some common sense examples. To begin with, there should be a type of all things, which we shall designate by " \Diamond " (box), such that the assertion " $x : \Diamond$ " conveys no nontrivial information about x . Informally, " \Diamond " corresponds to the English word "anything".

And so we come to our first real assertion, namely, that \Diamond is a type, notated thus: $\Diamond : type$. It follows from this that \Diamond has all of the properties of a type. But what is a type? More directly, is *type* an example of *type*, or does it have some conflicting properties? Let us defer the question as too difficult to answer at present, and assert instead that *type* is certainly an example of *some* type, which for lack of a better name we will call "*metatype*". Formally, $type : metatype$.

But what is a *metatype*? We could defer the question again, claiming that $metatype : metametatype$, and continue in that fashion indefinitely. The alternative is to cut off the infinite regress at some point t , say by asserting $t : t$. Let us explore the consequences of the latter course, tentatively asserting $metatype : metatype$. This amounts to saying that, in whatever sense *type* is a "type of types", *metatype* is also a "type of types".

What about subtypes? Clearly, any example of *type* is something, i.e., is an example of \Diamond , so it should be the case that $type \preceq \Diamond$. Similarly, metatypes seem to be specialized types, though we have been carefully noncommittal to this point about what meaning we should attach to the informal notion of a "type of types". Suppose we take it at face value. Then, every example of *metatype* is also an example of *type*, and hence $metatype \preceq type$. The foregoing is summarized in the following "axioms".

A1) $\Diamond : type$

A2) $type : metatype$

A3) $metatype : metatype$

A4) $type \preceq \diamond$

A5) $metatype \preceq type$

The problem is to explain these notions formally in such a way that they both harmonize with our intuitions and give a logically consistent interpretation to the relations stated above. Here are some of the consequences which, according to these rules, follow from the "axioms" A1-A5.

C1) $\diamond : \diamond$

C2) $type : type$

C3) $type : \diamond$

C4) $metatype : type$

C5) $metatype : \diamond$

C6) $metatype \preceq \diamond$

Proof: C1 follows from A1 and A4 by subsumption. C2 follows from A2 and A5 by subsumption. C3 follows from C2 and A4 by subsumption. C4 follows from A3 and A5 by subsumption. C5 follows from C4 and A4 by subsumption. Finally, C6 follows from A4 and A5 by transitivity. \square

Remark: It does not follow that $\diamond : metatype$, nor is the negation of this statement provable. Hence the axioms admit models in which either \diamond is or is not a *metatype*, and in fact models of both kinds exist.

Now, a number of familiar difficulties arise if we take types to be sets, with " $:$ " being set membership, and " \preceq " being the subset relation. For one thing, we would have situations where two sets belong to each other, as in $\diamond : type$ and $type : \diamond$. Intuitively, *type* contains a copy of \diamond , which contains a copy of *type*, which contains a copy of \diamond , and so forth *ad infinitum* – not your garden-variety set. (Technically, this violates the *axiom of regularity*,¹ which is usually included in any axiomatization of set theory to disallow a number of well-known anomalies.) To make the problem concrete, the reader is challenged to exhibit three sets that satisfy the relations A1-A5 and C1-C6.

Note that without regularity or some other means of restricting the principle of comprehension,² we can apparently form the Russell type $R = \{x | \neg(x : x)\}$. In set theory, membership in a set is equivalent to satisfaction of the characteristic predicate of the set, so we arrive at the contradiction that $R : R \Leftrightarrow \neg(R : R)$.

¹The axiom of regularity disallows infinite regress in the formation of sets. In short, $\dots\{\{\{\}\}\}\dots$ is allowed, but not $\{\{\{\dots\}\}\}$.

²The principle of comprehension states that for any predicate ϕ there exists a set consisting of those things which satisfy ϕ , usually written $\{x | \phi\}$. Russell's original theory of types made peace with the principle of comprehension by imposing an order on predicates, and restricting the membership predicate \in so that the type of the left operand be strictly less than the type of the right. Thus ruled out are locutions such as $x \in x$, thereby making the troublesome Russell class indefinable. This avenue is, however, closed to us because it would banish axiom A3, along with C1 and C2.

3 A Solution

We adopt a subtler interpretation, in which types are sets of unary predicates (equivalently, "properties") closed under entailment. That is, if $t \vdash \phi$ (ϕ a unary predicate) then $\phi \in t$. In order to specify types finitely, we adopt the notation X^+ for the closure of X under \vdash .

For example, let us define *oddprime* to be the type $\{x \text{ is prime, } x \text{ is odd}\}^+$. *oddprime* includes the properties " $x + 1$ is even", " $x > 2$ ", and " $\forall y, m. y < x \wedge m|x \wedge m|y \Rightarrow m = 1$ " along with infinitely many other consequences of the two properties originally specified.

The subtype relation is defined as follows: $x \preceq y$ if $\forall \phi \in y. x \vdash \phi$. That is, subtypes specialize their supertypes. An equivalent, if initially counterintuitive, characterization is that $x \preceq y$ if $y \subseteq x$.

$\{x \text{ is prime, } x \text{ is odd, } x^2 < 10\}^+$ is a subtype of *oddprime*. Similarly, $\{x \text{ is odd}\}^+$ is a supertype of *oddprime*.

To a first approximation, an object is an example of a given type if it satisfies all of the properties of that type.³ Formally, $x : t$ if $\forall \phi \in t. \phi(x)$. Actually, we need to refine this definition somewhat, but let us adopt it for the moment, to show what is right and wrong about it.

We can now give a comprehensible semantics to our trio of entities.

$$\begin{aligned} \diamond &= \emptyset^+ \\ \text{type} &= \{t \text{ is an } \vdash\text{-closed set of properties}\}^+ \\ \text{metatype} &= (\text{type} \cup \{\forall y. x. y : \text{type}\})^+ \end{aligned}$$

It is easy to check the properties A1-A5 and C1-C6 against these definitions. A1 states that \emptyset^+ is an \vdash -closed set of unary predicates; A2 and A3 state that $\{t \text{ is an } \vdash\text{-closed set of properties}\}^+$ and $(\text{type} \cup \{\forall y. x. y : \text{type}\})^+$ are \vdash -closed sets of properties specifying \vdash -closed sets of properties; and so forth, all of which statements are obviously true.

The interpretation of *metatype* is motivated by the intuition that a metatype is a type of types. Note that, with this interpretation, \diamond is not a *metatype* provided that something is not a type. In Prism, we fully expect there to be things which are not types, though it is possible to construct systems consistent with everything that has been said so far in which everything is a type.

Although the proposed interpretation makes the axioms and their consequences seem sensible, it doesn't prevent the paradoxes. We can still define an analogy to the Russell type $R = \{\neg(t : t)\}^+$, and draw the usual contradiction. The problem, alluded to earlier, lies with our notion of examplehood.

I propose to correct the situation by imposing more stringent requirements for examplehood, to wit: examples of a type must be drawn from the examples of its subtypes. This rule seems to me to convey the important part of regularity, which is that types (sets) be populated synthetically. This gives us an analytic theory of synthetically populated types.

Technically, my proposal is that satisfaction of the properties of a type be necessary, but not sufficient, to establish examplehood. Formally,

$$\frac{x : y}{\forall \phi \in y. \phi(x)}$$

³I use the word "example", instead of "member", because the members of a type are properties. For example, " $x > 2$ " is a member of *oddprime*; "3" is not. What "3" is is an example of *oddprime*.

but the converse of this rule does *not* hold. Instead, we have the following axiom of *unit examplehood*, which explains how we come to have any examples at all.

$$a : \{x = a\}^+$$

Using these rules, one can prove that the assumption $R : R$ leads to a contradiction, but the assumption that $\neg(R : R)$ leads only to the conclusion that $\forall \phi \in R. \phi(R)$, which is insufficient to establish the contradictory $R : R$.⁴ So if we are Platonists we can conclude, quite comfortably, that R is not an example of itself. If intuitionists we be, we can remain comfortably agnostic.

The metamathematical status of the foregoing argument is crucial. I did *not* claim that $\neg(R : R)$ is a theorem within the system; I only claimed that R not being an example of itself is consistent with what *can* be proved in the system. If we could prove $\neg(R : R)$ in the system, then we would have that $\{x = R\}^+ \vdash \neg(x : x)$, and hence that $\{x = R\}^+ \preceq R$, which would enable us to conclude the contradictory $R : R$.

To be precise, the metamathematical assertion is $\{x = R\}^+ \not\vdash \neg(x : x)$. If we could internalize this fact, we would have $\{x = R\}^+ \vdash \neg\neg(x : x)$. If this is provable, we can still avoid contradiction if \neg is not an involution (which, of course, is the case in intuitionistic logic).

Note to selves: Remark on tolerance of inconsistent types. Include proof that there are no orphans (i.e., no examples are excluded by the proposed rules). Add some stuff about minimal conditions on entailment. Sharpen it up: what is bound and free in properties; references to context.

4 Prism Categories

We turn now to an application of the ideas presented so far to the problem of explaining modifiers of various kinds, especially adjectives, and their role in specifications. For example, we wish to understand how the use of "odd" in the phrase "odd prime" differs from the use of "red" in the phrase "red seven". The former is taxonomic; some primes are odd. The latter is not; no seven is red. In talking about things like colored numbers we are referring to attributes of objects which are neither colors nor numbers. Nonetheless, we use the same part of speech and the same syntactic structure to specify both types. Our goal is a theory of modifiers which would allow us to use them for type specification in Prism. A rudimentary theory will be presented in section 6. But first, we need to explore some of the structure of the type system and define some new concepts.

A Prism category, or Pcategory,⁵ is a slight generalization of the usual notion of category [MacLane]. It is essentially what MacLane calls a "metacategory".⁶

$$\begin{aligned} Pcategory &\triangleq \mathcal{O}c\{ Ob_c, Ar_c : \Diamond, \\ &\quad dom_c, cod_c : Ar_c \rightarrow Ob_c, \\ &\quad 1^c : Ob_c \rightarrow Ar_c, \\ &\quad dom_c 1_a^c = a, \\ &\quad cod_c 1_a^c = a, \end{aligned}$$

⁴By unit examplehood, $R : \{x = R\}^+$, but without further rules $\{x = R\}^+ \not\preceq R$.

⁵The initial "P" is silent, as in "Pneumonia".

⁶The notation $[\mathcal{O}x]$ is meant to identify the variable over which the predicates in the immediately following type specification range. It is on trial.

$$\begin{aligned}
\forall f, g, h : Ar_c & \vdash f \circ g : Ar_c \Leftrightarrow cod_c f = dom_c g, \\
f \circ (g \circ h) : Ar_c & \Rightarrow f \circ (g \circ h) = (f \circ g) \circ h, \\
f \circ 1_a^c : Ar_c & \Rightarrow f \circ 1_a^c = f, \\
1_a^c \circ g : Ar_c & \Rightarrow 1_a^c \circ g = g \rangle^+
\end{aligned}$$

An ordinary category is just a Pcategory in which the Arrows and Objects are Sets.

$$Category \triangleq \mathcal{O}c\{c : Pcategory, Ob_c, Ar_c : Set\}^+$$

In a similar manner, we lift the set-theoretic restrictions usually imposed on mathematical structures. For example, a *monoid* is a *set* equipped with an associative binary multiplication and an identity; a *Pmonoid* is *anything* equipped with an associative binary multiplication and an identity. We do this because sets in Prism do not have the primacy accorded to them in classical mathematics.⁷

We intend that operations in Prism can be applied to objects of any type. Informally, this means that when something is supplied as an argument, something (perhaps divergence, or an exception) results. A bit more formally, $f : \Diamond \rightarrow \Diamond$ for any f . Because there are no restrictions on the domain of an operation, everything is composable with everything else; i.e., $f \circ g$ is always defined.

The design of Prism also stipulates the existence of an identity on \Diamond . Intuitively, you can leave anything alone. Thus, \Diamond is a Pmonoid.

A Pfunction is a \Diamond -automorphism, i.e. $Pfunction \triangleq \mathcal{O}f\{\forall x : \Diamond. fx : \Diamond\}^+$. A partial Pfunction is the restriction of a Pfunction to a specified domain t , notated $f[t]$. If $x : t$, then $f(x) = f[t](x)$.

The codomain of a Pfunction can be specified using the notation $\rightarrow t$. That is, $f \rightarrow t$ specifies that the result type of f is t . Combining these notations, we can write $f[t_1] \rightarrow t_2$ to indicate that f takes t_1 's to t_2 's.

There is no overloading in Prism, so a given name can refer to at most one Pfunction in any context. However, a Pfunction can be specified piecewise, using restriction. For example,

$$\begin{aligned}
f[x : integer; y : integer] & \rightarrow Boolean \mapsto x < y \dots \\
f[x : integer] & \rightarrow color \mapsto \langle 1 \mapsto red; 2 \mapsto yellow; 3 \mapsto blue; \dots \rangle \dots
\end{aligned}$$

Ada-style overloading would treat these as two different functions with the same name, and would allow the definition of a third function named " f " with domain *integer* and codomain *Ascii*. The expression $f(3)$ would then be ambiguous unless its context dictated an expected result type of either *color* or *Ascii*, which would serve to disambiguate the reference of " f ". In Prism, such ambiguity is impossible, because all we have are distributed specifications of segments of a single function, which cannot have conflicting results on overlapping parts of the domain.

In Prism, $t_1 \rightarrow t_2$ is the type of partial Pfunctions from t_1 to t_2 , defined as follows. $t_1 \rightarrow t_2 \triangleq \{\forall a : t_1. x(a) : t_2\}^+$. As with most objects, Pfunctions can be examples of many types. For example, the following are true assertions of the f partially described above.

⁷The elimination of a central role for sets and set theory in Prism is not without precedent; Lawvere did so as early as 1964 [ref]. The key to the current approach is that types are simply specifications, not the extensions of those specifications. In Prism, " $x : t$ " is an assertion of knowledge about x , that it is an example of the specification (type) t . This says nothing about the collection of all such examples, which (if it exists) is an example of the type $\mathcal{O}x\{x \text{ is a set}, \forall y. y \in x \Leftrightarrow y : t\}^+$. According to this view, we never have to contemplate or account for such things as the extension of \mathcal{O} , though it is easy to show in the standard way that if it exists it is neither a set, nor a class, nor even a meta" class.

$f : \text{integer} \times \text{integer} \rightarrow \text{Boolean}$
 $f : (\text{integer} \times \text{integer}) \vee \text{integer} \rightarrow (\text{Boolean} \vee \text{color})$
 $f : \text{integer} \rightarrow \Diamond$

5 Type Pcategories

There are a number of ways to make Prism types into a Pcategory. One is the *logical* Pcategory, type_{\perp} , in which the arrows are given by the subtype relation, \preceq . It is bi-Cartesian closed, with products, coproducts, and exponents given by the following.

$$\begin{aligned}
 t_1 \wedge t_2 &\triangleq \max_i t_1, t_2 \preceq t \\
 t_1 \vee t_2 &\triangleq \min_i t \preceq t_1, t_2 \\
 t_1 \Rightarrow t_2 &\triangleq \max_i t \wedge t_1 \preceq t_2
 \end{aligned}$$

In words, \wedge forms the greatest common subtype (gcs), \vee forms the least common supertype (lcs), and \Rightarrow forms the least sufficient constraint (on t_1 to verify t_2). The terminal object is \Diamond , and the inconsistent type \rightarrow is initial.

Another type Pcategory is the partial Pfunction Pcategory, $\text{type}_{\rightarrow}$, in which the arrows are the partial Pfunctions. This Pcategory is also bi-Cartesian closed, with products, coproducts and exponents as follows.

$$\begin{aligned}
 t_1 \times t_2 &\triangleq \mathcal{O}x\{\forall f : t_3 \rightarrow t_1, g : t_3 \rightarrow t_2. \exists \alpha : t_3. x = \langle f, g \rangle \alpha \wedge \pi_1 x = f \alpha \wedge \pi_2 x = g \alpha\}^+ \\
 t_1 \oplus t_2 &\triangleq \mathcal{O}x\{\forall f : t_1 \rightarrow t_3, g : t_2 \rightarrow t_3 \\
 &\quad ((\exists \alpha : t_1. t_1 \alpha = x \wedge \langle f, g \rangle x = f \alpha) \\
 &\quad \vee (\exists \beta : t_2. t_2 \beta = x \wedge \langle f, g \rangle x = g \beta))\}^+ \\
 t_1 \rightarrow t_2 &\triangleq \mathcal{O}x\{\forall \alpha : t_1. x \alpha : t_2\}^+
 \end{aligned}$$

These are analogous to the usual Cartesian product, coproduct, and function space types on Set. Here, the type with no examples, \rightarrow , is initial, and any singleton type is terminal.

Fact 5.1 type_{\perp} is a subcategory of $\text{type}_{\rightarrow}$, under the obvious identification of arrows in type_{\perp} with the corresponding inclusion Pfunctions.

Fact 5.2 Every arrow in type_{\perp} is a bimorphism (i.e., epi and mono).

Fact 5.3 In type_{\perp} , sections = retractions = isomorphisms = identities.

Fact 5.4 type_{\perp} is not a topos.

Proof Counterexamples abound. \square

Theorem 5.1 *type_— is a topos. That is, for each type A there is a power object $\mathcal{P}A$ and a membership relation $\iota : \mathcal{E}_A \rightarrowtail A \times \mathcal{P}A$ such that, for any relation $\rho : R \rightarrowtail A \times B$ there is a pair of partial Pfunctions $\beta, \bar{\rho}$ such that the following diagram is a pullback.*

$$\begin{array}{ccc} R & \xrightarrow{\bar{\rho}} & \mathcal{E}_A \\ \downarrow \rho & & \downarrow \iota \\ A \times B & \xrightarrow{1 \times \beta} & A \times \mathcal{P}A \end{array}$$

Proof The required players are defined as follows.

$$\begin{aligned} \mathcal{P}A &\triangleq \mathcal{O}x\{x : \text{type}, \forall y : x. y : A\}^+ \\ \mathcal{E}_A &\triangleq \mathcal{O}x\{x : A \times \mathcal{P}A, \pi_1 x : \pi_2 x\}^+ \\ \iota &\triangleq 1[\mathcal{E}_A] \\ \beta &\triangleq b \mapsto \{\exists r : R. \rho r = \langle x, b \rangle\}^+ \\ \bar{\rho} &\triangleq (\rho \circ (1 \times \beta)) \end{aligned}$$

Given $\sigma : S \rightarrowtail \mathcal{E}_A$, $\hat{\sigma} : S \rightarrowtail A \times B$ making the square commute, the required unique partial Pfunction $u : S \rightarrowtail R$ is given by the correspondence $s \mapsto !r : R. \rho r = \sigma s$, where existence of r is guaranteed by $\bar{\rho}$ and uniqueness by ρ being monic. \square

6 Modifiers

In English, properties are often expressed by modifiers of various sorts, especially adjectives and adverbs. In some cases, the properties are conjunctive, as in “odd prime”, where “odd” serves to identify a subtype of “prime”. We could express this formally as $\mathcal{O}x\{\text{odd}(x), \text{prime}(x)\}^+$.

In other cases, modifiers serve to derive one type from another. For example, a “placeable integer” is not an integer, but an object that has a position and a value, which is an integer. Formally, $\mathcal{O}x\{\text{val}(x) : \text{integer}, \text{place}(x) : \text{position}\}^+$.

Modifiers correspond to Pfunctors on type_—. Examples include \times, \wp , and \rightarrow and the obvious generalizations of \wedge, \vee , and \Rightarrow , as well as *Placeable*, *Colorable*, and so forth. Thus our analysis of “odd prime” is that “odd”, acting as a modifier, is really the Pfunctor $\text{odd} \wedge _$. When used to modify “prime”, it yields the type $\text{odd} \wedge \text{prime}$.

By contrast, the use of “placeable” in “placeable integer” more closely resembles the Pfunctor $\text{place} \times _$, yielding the Cartesian product type $\text{place} \times \text{integer}$ with two components, viz. a place and an integer value.

However, the order of “independent properties” shouldn’t matter, a colorable placeable thing is the same as a placeable colorable thing. Define \cong_{perm} to be equivalence under permutation (e.g., $A \times B \cong_{\text{perm}} B \times A$). The independent combination of properties $\{P_i\}_{i \in I}$ can then be represented by their

product modulo permutation equivalence, $\prod_{i \in I} P_i / \cong_{perm}$. A more appropriate way to eliminate order-dependence is to use an unordered set of names for the component properties, instead of an initial segment of the natural numbers, and treat the independent combination as a context in which each name is bound to a value of the appropriate type, e.g. $\langle val \triangleq 3, place \triangleq var\ position \rangle$. Although it may be appropriate for some purposes, this rendition of independent combination is still not abstract enough to handle placeability, because a placeable placeable thing is the same as a placeable thing, i.e., placeability is idempotent.

What I have tried to do in the preceding two paragraphs is to pick out some of the abstract properties of placeability by contrasting our intuition with various possible representations. So far, my conclusions can be summarized by saying that the placeability functor (as applied to integers) is a commutative and idempotent projective limit (in $type_{\perp}$), called *orthogonal combination* and denoted \perp_{\perp} .

David Mundie has pointed out that a given adjective can be used both conjunctively and orthogonally, depending on what it modifies. For example, the use of "placeable" in "placeable thing" is clearly conjunctive: $placeable \wedge$, while in the previous example of "placeable integer" it means $placeable \perp_{\perp}$. The difference is that things in general can be classified as placeable or not, but in this (conjunctive) sense there are no placeable integers. Hence we construe the latter use of "placeable" as orthogonal.

The obvious hypothesis is that all adjectives are used in one of these two senses, depending on whether the modifier can properly be predicated of the type being modified. If it can, the modifier is conjunctive; if not, it is orthogonal. The obvious question is: is there some similarity between the two classes of Pfunctor which might motivate the use of adjectives for both? Yes, indeed! Both are commutative, idempotent projective limits, one in $type_{\perp}$, the other in $type_{\perp}$.

These are not the only kinds of modifier, or Pfunctor. Prepositional phrases often signal adjunctions, such as "free ring over", "completion of", "quotient of", and "inclusion of". Non-mathematical examples include "school of fish", "network of computers", and "architecture of the house". What these have in common is that they all name structural abstractions which can in some sense be "reversed". We can forget that a house is an integrated structure and treat it as just a bunch of bricks and boards, just as we can forget that $Z/17$ is a ring and treat it as just a bunch of functions on a partition of Z .

The point is that Pfunctors (modifiers) come in a large assortment of flavors: commutative, idempotent, associative, invertible, adjunctive, and so on and so forth. The study and classification of Pfunctors, and the invention of mechanisms for specifying, combining, and manipulating them, can be expected to constitute a large part of the Prism programmer's stock in trade.

To be continued ...

Versions, Configurations and Object Deletion

dab

June 5, 1989

Configurations and versions as well as the problems of controlling them exist in many domains of information acquisition and maintenance. Documents, databases, and the artifacts of software development (source code, test cases, documentation, specifications, designs and so forth) all evolve through multiple versions and both populate configurations and exist as configurations.

1 Versions

Versions arise in information in a variety of ways. *Revisions* of a piece of information are made as mistakes are corrected and missing portions of the information are added. *Parallel* or *alternate* versions arise as the same piece of information is readied for different consumers. Parallel versions of software implementations, for instance, arise when a system, or some portion of it, is developed for multiple environments, in multiple languages and/or using multiple algorithms. *Derivations* are generated automatically. DVI or Postscript versions, outlines and tables of contents, for example, can be derived from the source of text documents.

What each related "version" has in common is that it is a concrete embodiment of some abstract object. Thus, from the same abstract concept of semantic analysis, there could be

- parallel versions for incremental processing, for nonincremental processing, and for expository purposes,
- revisions of each of the parallel versions representing various stages of development, and
- derivations consisting of object code, a dataflow diagram and a cross reference.

2 Configurations *Example*.

Some pieces of information are complex enough that they are built from smaller pieces. A large document, for example, is constructed from some (more or less shallow) hierarchy of chapters, sections, subsections and so forth, as well as the actual text, figures, and tables that populate the sections, and perhaps a style specification to tell the text processor how these pieces fit together and how they should appear when viewed. A computer program is constructed from the source code for its various modules (subprograms, packages or whatever your favorite programming language provides) as well as modules extracted from libraries.

A configuration then is the set of specific versions of the various constituent subcomponents. Configurations, being information, can themselves have versions.

3 Version and Configuration Control

Version control includes knowing when two pieces of data are versions of the same abstraction, knowing when a piece of data is a revision or derivation of another piece of data, and retrieving the "latest" version of a piece of data in some context.

Configuration control includes remembering which versions of which things comprise a configuration, allowing common versions to be shared among configurations, and facilitating the gathering together of the actual components of a configuration.

4 Relative Versioning

Consider the following situation. In a large, ongoing software development project, there are many modules which exist in various revisions, alternates and derivations. Some of the modules are (re)used in multiple software products. In this sort of situation, the "latest" version of a particular software module might be different for the developers of the module, developers of other modules that use it, and actual users (customers). This is equally as true for the version previous to the latest. We maintain that version history is relative.

Figure 1 shows a typical version hierarchy. The circles represent versions and the arrows represent the parent-child relationship. Version a is actually a virtual version. It represents the abstract concept for which each of the others is a version. Versions b, c and d are parallel versions as are versions e and f. Versions e and f are revisions of version b, version g is a revision of version c and so forth.

The important thing to recognize about any version hierarchy is that it is seldom used, or useful, in its entirety. The developers of a parallel version of a module have no need for information concerning experimental versions (revisions) of the other alternatives. The other essential thing to recognize about a version hierarchy such as that in Figure 1 is that it is misleading. For example, for the middle alternative, the "latest" version is k for the developers of that version. For the builders of a program that incorporates the module, however, the experimental revision k is not the "latest" version at all. And, of course, customers probably see yet another "latest" version. Figure 2 shows a more accurate (possible) set of version histories, focusing on just the middle branch of the version hierarchy of Figure 1.

An advantage of relative versioning is that a version hierarchy can be edited without adverse effect on any other view of the hierarchy. For instance, in the situation depicted in Figure 2, perhaps the module users need to delete version c from their version hierarchy. This can be done, as in Figure 3, without the comparable change in the hierarchy of the module developers.

For example, version a in Figures 1 to 3 might be semantic analysis. Parallel version b could then denote an implementation incorporating an incremental algorithm. Parallel version c could denote an implementation incorporating a nonincremental algorithm. Finally, parallel version d could denote an implementation developed for expository purposes. In an ongoing project, we could envision various versions of the semantic analysis module being incorporated into compilers, incremental semantic editors and source-to-internal-representation translators. For the developers of the nonincremental semantic analyzer, the latest version is k. For the developers of the compiler, the latest version is the latest "released" version. And for the users of a particular released version of that compiler, there is probably yet a third version that is the latest.

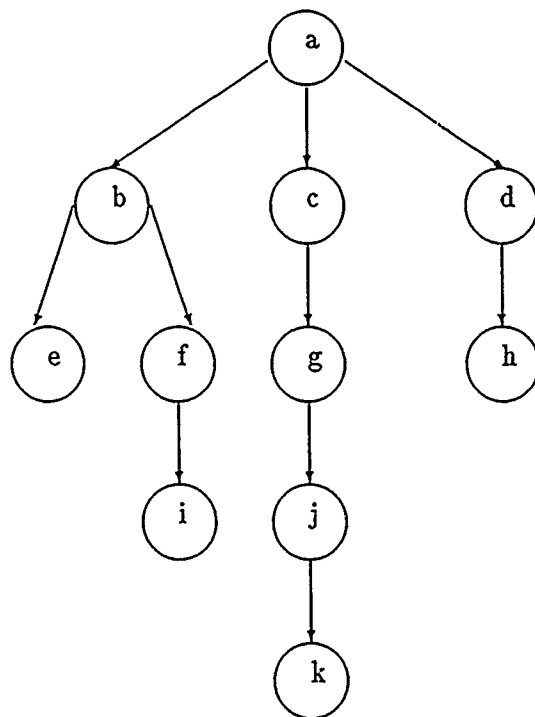


Figure 1: A naive version hierarchy

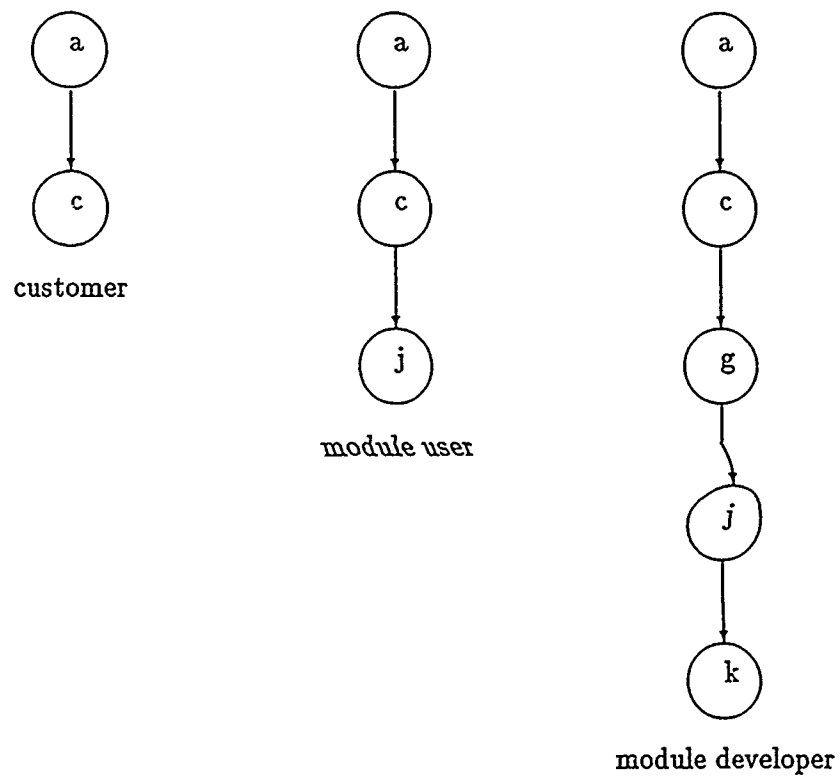


Figure 2: More accurate version hierarchies as seen by ...

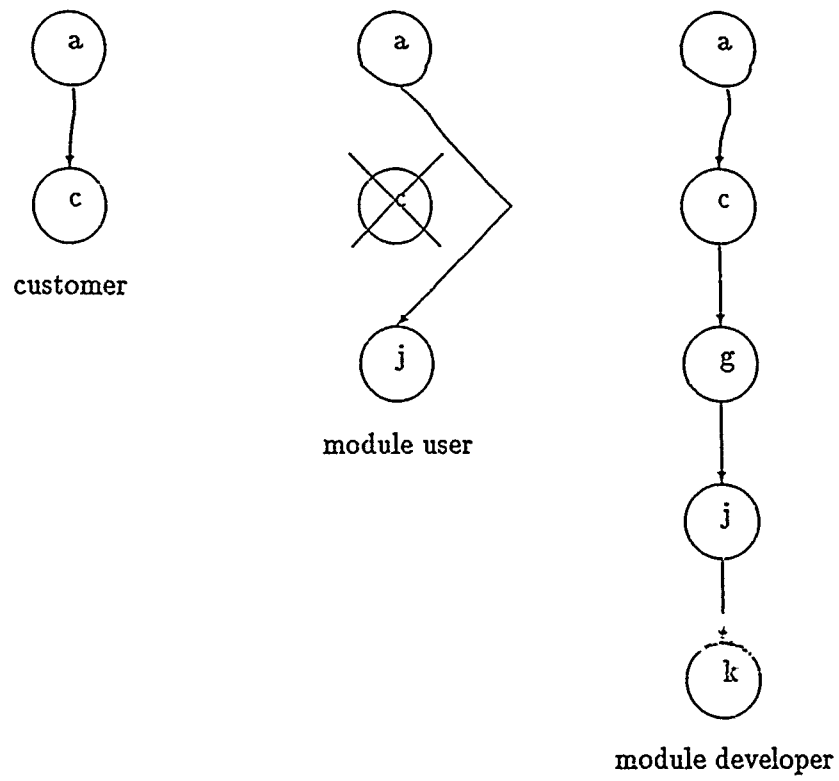


Figure 3: Edited version hierarchies

5 Deletions

There is a question about when objects can be removed from the persistent object system. Traditional reference count mechanisms do not work in the presence of distributed systems, networking and removable media. What we propose is that an object can be deleted only when "someone" says that it can be. The authority and intelligence to initiate object deletion can reside both in human users and in automatic processes. For instance, a human user can elect to delete any object for which the proper permissions are present. *need to discuss permissions somewhere*. The obvious analogy is to file system deletions. There can also be automatic processes (i.e. computer programs) that are allowed to delete objects. A program of this sort will operate under a set of rules such as "there is no need to keep the derived versions of a piece of source code that is obsolete". This is similar to the deletion rule employed in Odin [CO89]. There are obvious tradeoffs in deleting derived versions (such as object code) to make space available when they can be rederived at the cost of some computation.

References

- [CO89] Geoff M. Clemm and Leon J. Osterweil. Odin — an extensible software environment integration mechanism. accepted for publication in *ACM Transactions on Programming Languages and Systems*, 1989.

Untyped λ in Prism 0.4

Jon Shultis

June 8, 1989

1 English

A λ -expression is either a variable, an application, or an abstraction. An application has a rator and a rand, each of which is a λ -expression. An abstraction has a bound variable and a body, each of which is a λ -expression.

The rator and rand of an application are subexpressions of that application. The body of an abstraction is a subexpression of that abstraction. The subexpression relation is a partial order.

A binding of a variable x in an expression e is any subexpression of e which is an abstraction having x as its bound variable.

The scope of a binding of a variable is its body, excluding any binding of that variable.

An occurrence of a variable is free if it is not in a binding of that variable. Otherwise, the occurrence is bound.

The quasi-free substitution of e_2 for x in e_1 is the same as e_1 except that all free occurrences of x are replaced by e_2 . A quasi-free substitution is free if no free occurrence of x in e_1 lies in a binding of a variable that occurs free in e_2 .

An abstraction with bound variable x and body e is α -convertible with any abstraction with bound variable y and body $e[y/x]$ a free substitution. Two λ -expressions are α -convertible if they are the same modulo α -conversion of bindings.

Let e be an application with rator f and rand a , where f is an abstraction with bound variable x and body b . e is β -convertible with any λ -expression $b'[a/x]$ a free substitution, where b' is α -convertible with b .

2 sectionMLish

This following is an annotated rendering in MLish of some parts of the preceding English text.

In ML, structural types are fairly easy to define, although they must be overspecified. In particular, a representation type is required. Here, for example, variables are represented by type symbol, and applications and abstractions are pairs. Other things that are troublesome are that every abstract type specification has to be closed (that's what the double semicolon is for), and all of the clauses relating to the type specification have to be contiguous.

abstype λ -expression \Leftrightarrow [| variable:symbol;

```

                                application: $\lambda$ -expression  $\times$   $\lambda$ -expression;
                                abstraction:symbol  $\times$   $\lambda$ -expression ]]
with   rator(application(e1,e2)) = e1;
and    rand(application(e1,e2)) = e2;
and    bound_variable(abstraction(x,e)) = x;
and    body(abstraction(x,e)) = e;;

```

The subexpression predicate is easily defined, but the property of being a partial order has to be inferred.

syntax infix '<<';

```

let rec e1 << e2 = (e1=e2) or
  case e2 is
  | application(e21,e22): (e1 << e21) or (e1 << e22)
  | abstraction(e2x,e2b): (e1 << e2b)
  | otherwise: false
end case;;

```

In the English version, the phrase "any subexpression of e " introduces a type determined by a predicate. ML doesn't have such types, because they disrupt things like type inference for which ML is famous. Common Lisp does have predicate-restricted types, and I shall pretend that ML does, too, for the sake of this exercise. Another limitation of ML that has to be overcome is the lack of dependent types, where the type is a function of some value. (In ML, types can only be parameterized by other types). With these extensions, we can define the type of subexpressions of an expression as follows.

```
abstype subexpression( $e$ : $\lambda$ -expression) =  $se$ : $\lambda$ -expression satisfying ( $se$  <<  $e$ );
```

Incidentally, property-restricted types provide some of the capability needed to define a type of partial orders, of which we might then specify, or prove, that << is an example. However, there is no way to write an effective procedure to test whether an ML function satisfies a predicate if its domain is infinite, which is the case here. The reason is that one doesn't have access to the function definition, which might be analyzed, and barring that all one can do to test a universal property of the function is to apply it to its entire domain. (Another way of classifying functions is by construction. That is, if there is a set of methods for constructing functions of a particular class, one can define an abstract type with those methods as its constructors. Polynomials are a good example. Unfortunately, I don't know how to do that for partial orders. Even if I did, I would have to specify and verify my construction of partial orders outside of ML. This is the main limitation overcome by constructive type systems.)

The definition of binding is straightforward.

```
let type bindings-of  $x$   $e$  =  $a$ :subexpression( $e$ ) satisfying is-abstraction( $a$ ) and bound-variable( $a$ )= $x$ 
```

The notion of scope is traditionally defined and then used in the definitions of free and bound occurrence, and free substitution. However, I found that it was more trouble than it was worth when it came to giving pure specifications of these latter things. Its main utility seems to be heuristic, aiding in the formulation of algorithms for performing α - and β -substitution. In any event, I left the definition in for the sake of tradition, and because it highlights a shortcoming of existing type specification mechanisms.

Scope is a supertype of λ -expression, where the English specification uses an exclusionary clause to accomplish the generalization. ML does not support this idea, nor have I seen any proposals discussing

it. One approach, not too far removed from current state of the art, would be to specify the supertype from scratch, and then redefine the λ -expression subtype using inheritance, along the following lines.

```
abstype scope  $\Leftarrow$  (| hole:trivial;
                  nonhole:[| variable:symbol;
                           application:scope  $\times$  scope;
                           abstraction:symbol  $\times$  scope || ])
with   rator(application(e1,e2)) = e1;
and    rand(application(e1,e2)) = e2;
and    bound.variable(abstraction(x,e)) = x;
and    body(abstraction(x,e)) = e;;
```

abstype λ -expression = scope without hole;

The "without" operator eliminates one or more variants of a type, in this case the hole variant. A roughly equivalent formulation would be

abstype λ -expression = sc:scope satisfying noholes(sc),
 where "noholes" is the predicate

```
let rec noholes s = if s is-nonhole then
  case s is
    | variable(x): true
    | abstraction(s1,s2): noholes(s1) and noholes(s2)
    | application(sx,sb): noholes(sb)
  end case
else false;;
```

Although this works, it is unsatisfactory. It should be as easy to derive a supertype from a subtype as it is to derive a subtype from a supertype. In any event, having defined the notion of scope, we can proceed to define the scope of a bound variable, as follows.

```
let rec exclude-bindings x e =
  case e is
    | variable(y): y
    | abstraction(e1,e2): abstraction(exclude-bindings x e1, exclude-bindings x e2)
    | application(y,b): if y=x then hole() else application(y,exclude-bindings x b);;

let scope-of(abstraction(x,e)) = application(x,exclude-bindings x e);;
```

The notion of occurrence is that of a correspondence between position and value. Positions are designated by sequences of coordinates (or *paths*, like [12,3,-4], cdaddaddr, .foo.bar.baz.foo, and rator(body(rator(rand(rand. When applied to an object, a path selects the subobject lying at the designated position within the object. Although programming languages typically provide for selection of subobjects, they don't generally support representation of occurrences, which requires pairing objects and paths without applying them. In order to do this, there has to be a first class concept of paths, which doesn't exist in current programming languages in other than rudimentary ways. For example, if

functions are first-class objects then we can use them to represent arbitrary paths, such as $\lambda a(a[12,3,4])$, (function cdaddaddr), $\lambda r(r.foo.bar.baz.foo)$, and $\text{randorandoratorobodyorator}$.

Unfortunately, the function representation only allows for composition and application of paths; we also need decomposition, so that we can traverse the path step by step. To accomplish this, we want sequences of paths. That is, given a path $p = \langle c_1, c_2, \dots, c_n \rangle$, the path sequence p^* is the sequence of prefixes of p , i.e., $p^* = \langle \langle \rangle, \langle c_1 \rangle, \langle c_1, c_2 \rangle, \dots, \langle c_1, c_2, \dots, c_n \rangle \rangle$. Without further lamentation, assume that we have such things as first-class objects in the language. Let us represent occurrences as pairs $\langle p, o \rangle$ where p is a path, and o is an object. Define an occurrence $\langle p, o \rangle$ to be an occurrence of some subobject so if $o[p] = so$, where I have used square brackets as a universal path traverser. With these preliminaries settled, we can now give the MLish specification of free and bound variable occurrences.

```
let free <p,o> =
  let x = p[o] in
    exists binding p*
    where binding path = path[o]:(bindings-of x o);;
```

```
let bound = not o free;;
```

Moving right along,

```
let rec quasi-free-substitution e2 x e1 =
  case e1 is
    variable(y): if x=y then e2 else e1
  | abstraction(y,b): if x=y then e1 else abstraction(y,quasi-free-substitution e2 x b)
  | application(op,arg): application(quasi-free-substitution e2 x op, quasi-free-substitution e2 x arg);;
```

At this point, things become a mite hazy, because the English constructs a type of function applications, a concept which doesn't have any meaning at all in MLish. In particular, it involves reference to the arguments applications and their properties, while in MLish all we have are two values, both λ -expressions. Put another way, I decided to define a function which computes the quasi free substitution, instead of specifying it. This is the norm in MLish, but the English pulls the other way.

Enough. Suffice it to say that this just isn't the way one goes about doing λ -calculus in MLish. Instead, one tends to define some types and some algorithms, and then use them to manipulate things. The specification side is left almost entirely out of the picture.

3

sectionPrism 0.4

A λ -expression is a variable xor an application xor an abstraction.

Descriptive Processes I

Jon Shultis

17 November 1989

1 Introduction

Numerous forms may be used to represent descriptive information. Among forms currently being used are such things as abstract syntax trees, feature structures, and IRIS. Though the theories of each of these descriptive forms are more or less well developed, we lack any kind of general theory of description which would serve to unify them and support analysis and comparison.

The theory of descriptive processes is an attempt at a unifying theory of description, taking as its starting point an unusual, dynamic, view. The basic idea is that descriptions are static representations of descriptive processes. Descriptive processes, in turn, represent objects. When a query is submitted to a descriptive process, another descriptive process, representing an answer to the query, is returned. A descriptive process models an object x if its behavior is consistent with that of x . That is, if each query is interpreted as some operation on the object x , then the process(es) returned from each query should model the results of the corresponding operation on x .

Note that I have turned the traditional semantic paradigm on its head, in that the descriptive process models the object, not the other way around. An important consequence of this reversal is that an object may have many properties which are not described, *i.e.*, not contained in the model. Moreover, a given object may have any number of distinct descriptive models, each of which captures some aspect of the object. The usual semantic view would require either that we equate all of the descriptions, on the grounds that they refer to the same object, or that we populate our semantic domain with distinct objects, one for each distinct description. This distinction is not too important if we restrict our attention to formal languages with well-defined mathematical semantics, and is meaningless when the semantics is fully abstract. However, when

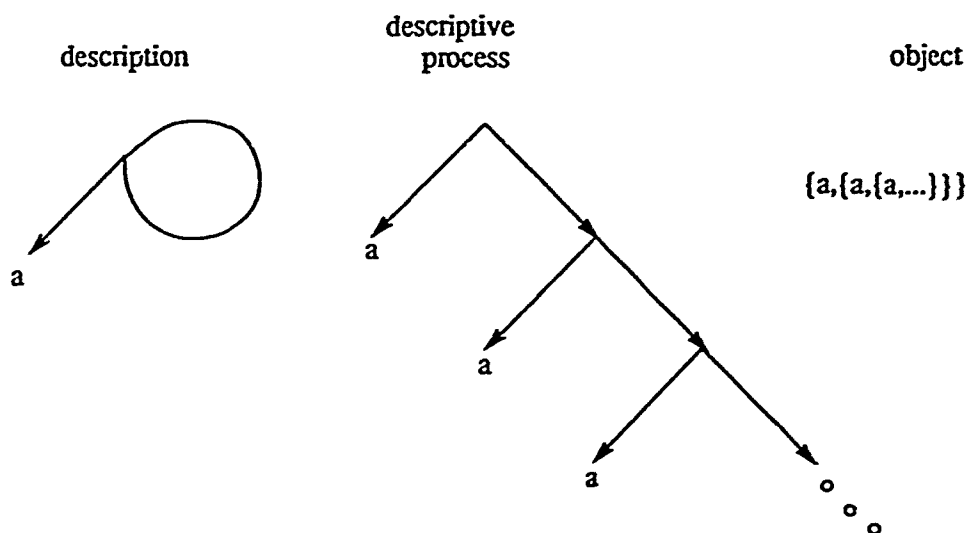


Figure 1: Description of a hyperset

language is used to talk about objects in domains about which we have only partial or imperfect information, such as the real world, the distinction is crucial.¹

For the most part, it is assumed that descriptions are finite, though this is not universal. For example, the possibility of infinite descriptions is entertained in the study of infinitary logic. Another common assumption is that sensible descriptions cannot be circular, though again this is not universal. For example, a tagged circular graph may be regarded as a description of a hyperset, as in figure 1. Moreover, experience with IRIS suggests that circular descriptions have significant engineering advantages over non-circular ones. In any event, I make neither assumption at the outset, though of course I will have to show how the theory works out in these cases.

It is somewhat misleading to call the diagram in the middle of figure 1 a process, because it isn't; it is merely a description of the trace of a process. The static representation can only be understood dynamically by some process of interpretation, and in fact some such process of interpretation is involved in explaining how the description on the left "unfolds" into the description in the middle. Note also that we could give several interpretations to either diagram. The one which results in a model of the hyperset is a "parallel" interpretation, in which all arrows are traversed simultaneously. A different dynamic interpretation of the diagrams is the "indeterministic" one, in which any one arrow can be traversed at any branch point, but it is not specified which one.

¹Of course, if one insists on it then the tail can be made to wag the dog by interposing a domain of senses between expressions and what they denote, but at best there is indexical explosion, and at worst one has to postulate senses which are parameterized by every possibly relevant feature, even though the latter are unknown, which is psychologically implausible in the extreme. What is worse, one has to postulate that the world is in fact describable, i.e., that there are enough properties around to classify things, which is errant nonsense, because the class of properties required to describe all classifications of a set of properties is strictly larger than the original class of properties by Cantor's theorem.

Using that interpretation, the dynamic process is one which makes some indeterminate number of silent transitions, delivers a single "a", and halts.

Note: communicating descriptive processes = dialogue? (Think of the "transition diagrams" used to describe dialogue in Winograd and Flores.)

2 Basic Definitions

For any set-valued function $f: A \rightarrow \mathcal{P}(A)$, define its *finite powers* recursively for any $a \in A$ as follows. $f^0(a) = \emptyset$, and $f^{n+1}(a) = \bigcup_{x \in f^n(a)} f(x)$.

A *preprocess* is a function $p: S \rightarrow \mathcal{P}(S)$ having the property that $\forall s \in S \exists n \in \mathbb{N} (s_0 \in p^n s)$. Note that s_0 belongs to S unless S is empty. The set $S \cup \{s_0\}$ is called the set of *states* of the preprocess, denoted Σ_P , and s_0 is called the *initial state*.

The motivation for having an initial state which roots the process is the intuition that any descriptive process has to start at some time, and at that time it is in some determinate state. The motivation for having every state be at most finitely removed from the initial state is that an infinitely removed state can never be observed, and hence is indistinguishable from the process with that state deleted.

A *process* is an isomorphism class of preprocesses. Formally, let $p_1: S_1 \rightarrow \mathcal{P}(S_1)$ and $p_2: S_2 \rightarrow \mathcal{P}(S_2)$ be preprocesses with initial states $s_{0,1}$ and $s_{0,2}$, respectively. Define $p_1 \cong p_2$ if and only if there is a bijection $\sigma: \Sigma_{p_1} \rightarrow \Sigma_{p_2}$ on the states such that $p_1 \circ \sigma = \sigma \circ p_2$, where the occurrence of σ on the left is the usual extension to sets ("map" in programming language terminology). The reader can readily verify that \cong is an isomorphism. In practice, we shall blur the distinction between preprocesses and processes, using a representative preprocess p to denote its isomorphism class $[p]_{\cong}$.

Descriptive Processes II

Jon Shultis

28 November 1989

This is an internal working document on Prism, and is not intended for distribution outside IncSys.

2 Basic Definitions, Cont'd

(Recall the basic definitions of preprocess and process from 17 Nov.)

A number of properties of processes are immediate, *e.g.*, that every state in a process is the initial state of a subprocess.

The set of *transitions* of a process p is given by $\Delta_p \triangleq \{ \langle s, \hat{s} \rangle \mid s \in p(\hat{s}) \}$. The set of transitions originating at state s is denoted by $s \downarrow S$; the set of transitions terminating at s is denoted $S \downarrow s$.¹

In terms of transitions, the definition of process asserts that every state can be reached from the initial state in a finite number of transitions. This does not, however, rule out the possibility of processes having infinite sequences of transitions between the initial state and other states, as the following example shows.

Example 2.1 Consider the process $p: \mathcal{P}(N) \rightarrow \mathcal{P}(\mathcal{P}(N))$ where $p(X) = \{\emptyset\} \cup \{Y \mid |X - Y| = 1\}$. This process arrives at a given set of numbers X by first selecting an arbitrary subset of X in one step, and then building up the rest of X one element at a time. Any sequence of transitions to an infinite set X which starts by choosing a subset which is missing infinitely many elements of X will be an infinite sequence.

A *labelled process* is a process p together with a function $\Lambda: \Delta_p \rightarrow L$. The elements of L are called *labels*, and Λ is called a *labelling function* or, more simply, a *labelling*. Elements of $S \cap L$, if any, are called *internal labels*.

¹These notations are borrowed from category theory; in the skeletal category generated by taking the reflexive transitive closure of Δ_p , $s \downarrow S$ generates the comma category of objects under s .

Intuitively, labels provide a means of focussing queries, or, equivalently, of guiding processes. If a query can be expressed as a composition of labels, where each label is taken as representing a property (component, feature, ...), then the process can respond to the query by simply traversing the indicated path, and returning the subprocess it finds there (if any). This basic idea can be instantiated in various ways, and has been by various authors. For example, labels can be treated as modal operators in a logic, where processes are taken to represent propositions. The intuition of focus accords with the intuition that modalities serve to constrain the scope, or domain, of a proposition, for example with respect to time, place, type, or number. (Q: what about bisimulation?)

A labelled process $\langle p, \Lambda \rangle$ is said to be *deterministic* if $\Lambda|_s$ is 1-1 for every state s . That is, a process is deterministic if there is at most one transition with a given label leading from any state. Finite deterministic processes correspond to feature structures with shared substructures.

A *tree* is a process in which p is functional. That is, $|ps| = 1$ for every state s . When dealing with trees, we abuse notation slightly and use p to denote the corresponding function, for example writing $ps = \hat{s}$ instead of $ps = \{\hat{s}\}$.

A *record* is a finite, deterministic, labelled tree. Records correspond to what Rounds and Kasper call "trees" in their treatment of propositional feature structure specifications, but we prefer to reserve that term for the more general concept.

Finally, an *ideograph* is a finite labelled process.

Exercise 2.2 Define IRIS.

PRISM

EIGHT QUEENS EXAMPLE

David A. Fisher

Incremental System Corporation
319 South Craig Street
Pittsburgh, Pennsylvania 15213



Incremental

SYSTEMS CORPORATION

Languages Beyond Ada and Lisp (Prism)

- Language
 - mechanism for communication
between user and system
- User
 - requirements
 - conflict resolution
 - design decisions
 - advice on implementation
- System
 - language processing
 - maintaining database
 - consistency checking
 - design analysis
 - guided implementation
 - optimization
 - measurement
 - substitution & transformation



Technology

- Formal Methods -- Precision
- Natural Language Mechanisms --
Avoid Overspecification
- Symbolic Execution
 - partial
 - lazy
 - anticipatory
- Optimization
 - analysis & transformation
 - constraint propagation
 - representation selection
 - abstraction & generalization



Concept

- Single Language

- requirements
 - design
 - implementation
- } specification

- Single System

- analysis
- compilation
- measurement
- persistence

- Design Goals

- expressiveness
- amiable to formal methods
- discourage overspecification
- self reflective
- executable sublanguage
- interactive



Key Language Characteristics

- **Expressiveness**
 - distributed specification
 - all features first class
 - self reflective
- **Amiable to Formal Methods**
 - specification mechanisms
 - property-based type system
 - strong typing
 - no escape hatches
 - automated mechanics
- **Discourage Overspecification**
 - natural language features
 - incompleteness support
 - separation of implementation



Key Language Characteristics

- **Extensible**
 - user definable types
 - few primitive abstractions
 - abstraction mechanism emphasis
- **Persistence**
 - all of language with full integrity
 - integrated with language
 - universal identity
 - location-independent names
 - access control
 - inconsistency management
 - placement control
 - granularity control
 - update without locking



Reply to I.D. Hill

David A. Mundie

1991 Aug 2

How would one communicate with a Prism system? In some sense this is not an interesting question. What matters above all is the internal functionality of the system, the representations it can manipulate and the way in which it manipulates them. Entirely too much time has been spent on the detailed design of the concrete syntax of programming languages—such issues as whether “if” should be balanced by “fi” or “end” or “endif” or “end if”.

Nevertheless, we feel there are good reasons for providing a sophisticated interface to Prism. The principle one is efficiency: constructing ideographs by hand would be a tedious, mind-numbing exercise. The throughput of the system can be magnified many fold by an interface that gives the user a reasonable way of communicating information without descending to the level of internal representation. Because we anticipate that a large portion of the programs in a Prism system will be generated automatically by the system, but will have to be understood by the user, an interface which facilitates human understanding is at a premium.

We take natural language as pointing the way, as the prototypical example of an intensional language with flexible control over commitment. We do not aim for a natural-language understanding system; rather we aim to incorporate those features in natural language which make it such a good vehicle for open-ended, context-dependent communication.

Before we describe the main features of the Prism interface as we see it, we must pause to consider the arguments put forth by I. D. Hill, a leading critic of natural language as a model for programming languages. In a wide-ranging and droll essay entitled “Natural language versus computer language,” Hill claims that natural-language interfaces are not only unachievable, but also *undesirable*. His basic premise that the meaning of computer languages should not be dependent on context strikes at the heart of the Prism effort, with its insistence that meaning must be decoupled from expression.

Hill makes 11 arguments, some of which we actually agree with.

Key Language Characteristics

- Execution Model
 - consistency checking
 - incremental
 - partial evaluation
 - lazy evaluation
 - anticipatory evaluation
- Other Language Concepts
 - scope and visibility
 - control mechanisms
 - arithmetic
 - composite data
 - tasks & concurrency
 - exceptions
 - time
 - communication delay



Property-Based Type System

- Definitions

- a property is a unary predicate.
- a type is a set of properties that is closed under entailment.
- an example is a value of a type.

- Some Examples:

$\text{even} \triangleq \{ (\lambda x \mid x \bmod 2 = 0) \}^\dagger$.

$(\lambda x \mid x \neq 177) \in \text{even}$.

8 is an example of type even.

2 is an even and a prime.

small red dogs = red small dogs.



Noun Phrases

Quantifiers

Anaphora

[quantifier]{premodifier}type_name
[postmodifier][aplicative]

abc

$(2 \times f(abc, 8))$

all integers

each integer in $(1..8)$

some prime integer that is (> 10)

an integer such that $(2 \times it = it+8)$

the blue box on the table

the odd integer 7

7



Declaratives

- Denotational

$f(\text{any integer}) \triangleq \text{the integer} + 3.$

$\pi \triangleq \text{circumference}(\text{any circle}) / \text{diameter}(\text{it}).$

$\text{fido} \triangleq \text{the large cat under the table.}$

- Assertional

$3 \times (a + 2) - a = 2 \times (a + 3).$

all prime integers that are (>2) are odd.

$\text{car}(\text{cons}(\text{any thing}, \text{any thing})) = \text{the first thing.}$

$\text{cdr}(\text{cons}(\text{any thing}, \text{any thing})) = \text{the second thing.}$

- Axiomatic

close t.

- Operational

$+ / \text{any integer} \triangleq [$

$s \triangleq \text{var integer.}$

$s \leftarrow 0.$

for each x, $s \leftarrow s + \text{it.}$

$] s]$.



Incremental
SYSTEMS CORPORATION

Imperatives

verb [name] ['to' name]

verb '(' exp { ',' exp } ')'

name '←' name

rotate the wheel.

assign f(x) to y.z.

move a to the b of c.

print(a, b+c, "xyz").

a ← a+3.

Interrogatives

2 + abc ?

x < y ?

the integer such that it + 3 = 8 ?

the box is on the table ?



The Eight Queens Problem

eight_queens_problem :

a board \triangleq *an array(8, 8) of squares.*

a solution \triangleq *8 queens and 1 board
such that*

each queen has 1 square of the board

and not attack(any queen, any queen).

attack(a : any queen, b : any queen) \triangleq

a \neq b and

(a.square.x = b.square.x or

a.square.y = b.square.y or

abs(a.square.x-b.square.x) =


abs(a.square.y-b.square.y)).



Incremental
SYSTEMS CORPORATION

a solution?

a solution = [

depict(any square) \triangleq .

order the queens.

depict(any queen) \triangleq depict(pos the queen). |

			1				
					2		
							3
	4						
						5	
6							
		7					
				8			

].

implementation a solution ?

a solution IMP

for each queen,

assign the queen to some square.

if attack(any queen, any queen) then fail.

cost = $64 \uparrow 8 \approx 2.8_{10} 14$.



*(a: a queen) \neq (b: a queen) and
(a.square = b.square)
implies attack(a,b)*

implementation a solution ?

a solution IMP

for each queen,

assign the queen to some square such that
not (it has a queen).

if attack(any queen, any queen) then fail.

cost = permutations(64, 8) $\approx 1.8_{18}14$.



Incremental
SYSTEMS CORPORATION

depict(any queen) = depict(any queen).

implementation a solution ?

a solution IMP

order the queens.

order the squares.

for each queen in order,

assign the queen to some square such that

(the queen = the first queen or

the square > the square of pred the queen)

and pos the square \leq pos the last square-

(pos the last queen-pos queen).

if attack(any queen, any queen) then fail.

cost = combinations(64, 8) $\approx 4.4_{18}9$.



Incremental
SYSTEMS CORPORATION

***expand attack in that !
implementation a solution ?***

a solution IMP

order the queens.

$(a : \text{a square}) < (b : \text{a square}) \triangleq a.x < b.x \text{ or}$
 $(a.x = b.x \text{ and } a.y < b.y).$

for each queen in order,

for each 1..8 in order,

assign the queen to

board(pos the queen, the integer).

if (a : (any queen that is

less than the queen).square).y

= (b : (the queen).square).y or

$\text{abs}(a.x - b.x) = \text{abs}(a.y - b.y)$ then fail.


cost \leq factorial 8 = 40320.


that implementation.




Incremental
SYSTEMS CORPORATION

*depict(any square) \triangleq
 case (square.x-square.y) mod 2,*

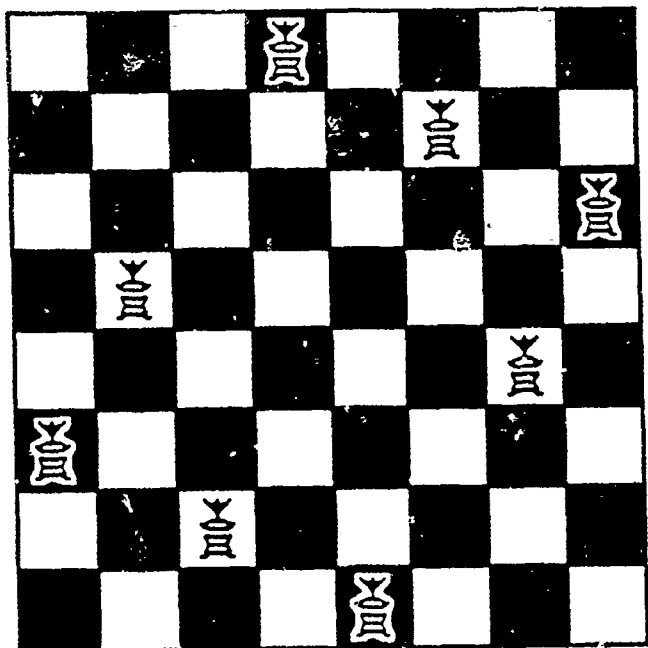
when 0 => .

when 1 => .

depict(any queen) \triangleq .

a solution ?

a solution =



1.



Incremental
 SYSTEMS CORPORATION

Persistence With Integrity and Efficiency

Deborah A. Baker, David A. Fisher and Frank P. Tadman

Proceedings of the Fifth Annual RADC
Knowledge Based Software Assistant Conference

Syracuse, New York
September 1990

Incremental Systems Corporation
Technical Report Number 900901

Persistence with Integrity and Efficiency*

Deborah A. Baker, David A. Fisher and Frank P. Tadman

Incremental Systems Corporation
319 South Craig Street
Pittsburgh, Pennsylvania 15213 U.S.A.
(412) 621-8888

Abstract

Managing persistent data efficiently and conveniently is a difficult task. Persistent object sizes vary from a single bit to many megabytes. The placement of objects in both main memory and secondary storage must be controllable by tools. The integrity of the objects must be maintained while allowing concurrent access. Operating system file systems, databases and software development systems provide partial solutions to the safe, efficient management of persistent data. An Iris-based information management system provides solutions to key persistent data problems heretofore solved only partially, unsatisfactorily, or inefficiently including type integrity for application-defined types, and safe data identification mechanisms.

1 Introduction

There have been many advances in research and development of technology, tools, and environments for design, implementation, and maintenance of large scale software applications during the past 20 years. Even though many of these component technologies are demonstrably effective for some limited aspect of the software process, there has not been any practical way for them to work cooperatively.

*This work was supported in part by the Rome Air Development Center under contract F30602-88-C-0115, the Defense Advanced Research Projects Agency (Arpa Order 5057) monitored by the Department of the Navy, Space and Naval Warfare Systems Command under contract N00039-85-C-0126 and by the Defense Advanced Research Projects Agency (Arpa Order 6487-1) under contract MDA972-88-C-0076.

In this paper we present a conceptual design and an instantiation of a suite of mechanisms that enable sharing and communication of information among the tools and tool components populating a (possibly distributed) software environment. The mechanisms ensure type and object integrity of all persistent information without advance knowledge of their types. They provide the primitive mechanisms required for the higher level imposition of user-defined policies such as those for version control, configuration control, release control, and access control.

In Section 2, the problem is described in more detail, along with the requirements placed on solutions. Section 3 contains a discussion of our model for providing persistence. In Section 4, an Iris instantiation of our model is discussed. Finally, in Section 5, our findings are reviewed and some future work is outlined.

2 Nature of The Problem

Some data may outlive the invocation of the tool or program which created them, in which case they are said to be *persistent*. Persistent data contain information which is important to an application taken as a whole, and which may be needed by several components (or invocations of components) of the application. There are several important requirements for persistent data in distributed software development, maintenance and operational environments that have not been addressed by databases or by file and operating systems. These requirements are concerned primarily with the integrity and efficiency of storage and retrieval of information within such environments.

In software environments, the persistent data obviously includes the requirements, designs, specifications, implementations and execution results of the programs being developed. It also includes representations of those programs in the form of source text, internal representations, unlinked object code, and executable target code. The persistent data also includes artifacts from, and inputs to, analysis, documentation, testing, project management, and maintenance processes such as constraints, rules, histories, and decision trees. All of these data items may exist simultaneously in a variety of versions and configurations. It is also crucial in software environments that the various tools and tool generators of the environment themselves be data objects of the environment.

Integrity for the data in software environments requires that all data be strongly typed with the type protection enforced throughout the persistent object base, not only for a few built-in types, but for those defined later by application developers and by tool builders as well. As in any distributed system, distributed software development, maintenance and operational environments must provide safe mechanisms to either maintain consistency among multiple copies of data objects that are replicated throughout the distributed system

or to detect inconsistency..

Efficiency is critical to the feasibility of any system including distributed software environments. Efficiency issues arise primarily in three areas: delay in accessing data, efficiency in handling inter-object references, and the cost of maintaining the type, version, and configuration integrity of the persistent data. For instance, for our compilation systems, the semantic analyzer must process 3,000 lines per minute if the compiler is to process 1,000 lines per minute. The semantic analyzer for the Ada programming language will make approximately 25,000 access of small grain objects while processing 3,000 lines.

Traditional solutions, whether drawn from operating systems, databases, or software development environments, have typically been inefficient and seldom safe. They assume that the primary location of data does not change, that cross-reference among objects is infrequent or is the responsibility of the user, that type integrity is the responsibility of the user, or that inconsistency can be tolerated when the number of resulting erroneous computations is statistically low.

The Knowledge Based Software Assistant Program is exploring the use of a formally based paradigm, which involves mediation from the software assistant, in the full range of activities associated with software development and maintenance. The functionality associated with each activity is captured in a *facet*, or sub-assistant.

The information that is involved in the KBSA paradigm to date (by virtue of the facets under development) includes such things as requirements, specifications, design history, assumptions, reasons, and rationales[Ele89]. Another vital aspect of the KBSA paradigm is the reuse of this information [Gol89]. For instance, the requirements and specifications developed under their respective facets should be reused by the program development facet to assure that they are satisfied and by the project management facet to assist in scheduling and cost estimation. The most effective way to facilitate this cooperation among the KBSA facets via sharing and reuse of information is to provide efficient mechanisms whereby the information can be identified, stored, accessed, maintained, shared and reused in a distributed environment. It is exactly a substrate of this nature that is the subject of this paper.

Extant Partial Solutions Traditional programming languages, operating systems and databases have each addressed some aspects of persistent information management, but each has shortcomings with respect to our requirements.

Operating systems address problems of resource management and security, providing mechanisms and policies for allocating and sharing basic computing resources. Of concern here are the storage systems provided by operating systems (i.e., file systems); they do not typically ensure type integrity. This is true for both abstract and representation types and

for invocations of tools as well as manipulations by human users. Furthermore, operating systems rely on user-defined names, which compromise identity integrity.

Databases capture some knowledge of the characteristics of the information they manage, and exploit that knowledge to make better use of resources. This knowledge is represented in a variety of ways (e.g. schemata and dependencies) and is used to optimize representation and access to information, to improve the convenience, reliability, and efficiency of maintaining important relationships between items of information, and to drastically reduce the time required to design and implement applications. The success of databases depends on certain assumptions such as there are relatively few schemata and relatively many items per schemata, the schemata are fixed, or change only infrequently, the types of information are closely circumscribed (for instance, relation is not a type in a relational database and therefore a relation can not itself participate in a relation) and they are not dynamic (i.e., types cannot be added arbitrarily), and the granularity of the information is known and fairly uniform. Information in traditional databases is most often distinguished on the basis of some key; such value-oriented names compromise identity integrity.

Unfortunately, the underlying assumptions of operating systems and databases are not valid for the information in a software environment. Software development environments are characterized by a wide variety of objects, with dynamically varying types and relationships. Correct and effective management of these objects requires intimate knowledge of the policies and relationships which are specified (implicitly or explicitly) in the objects themselves. The types of information in the system at any time are specified by that information, and the number of items of information of any given type may range from one to millions. Moreover, some objects are virtual, and are only instantiated dynamically, by applying one body of information to another.

Related research includes databases [Ber87], persistence [AB87, BB87, Coo87], software development environments [SDE88, CAI88, TBC⁺88], and object orientation [KC86, Mey89]. Much of this work is relevant in many ways. However, we have not entirely accepted the requirements, implicit or explicit, of these other projects, and there are, consequently, significant differences between most of the other projects and our own.

The models in object oriented systems are quite specific (e.g., a regime in which the data in an object include methods for responding to messages); furthermore, these systems are typically single user and/or single machine and the efforts to allow sharing among users and distributed machines are not altogether satisfactory. The use of a persistent programming language or a database programming language may be fine, but is not a useful approach for organizations that have mandates to use specific languages, or for organizations that have pre-existing software that they wish to use with as little modification as possible. Unlike many of the research projects on persistence, our system has requirements for strong typing and against restrictions on the types of persistent data. Also, research projects in these

areas have, of necessity, concentrated on attainment of functional requirements, often at the expense of performance or reliability. Successful commercial systems have typically achieved their performance goals by focusing on important but limited application domains.

3 A Model for Providing Persistence

Ensuring integrity and enabling efficiency are design features of our substrate for the management and sharing of information. We present a three layer conceptual model. The first layer provides general mechanisms for large- and small-grain object management. The next layer is implemented in terms of the first and provides attributed information structures. The third layer is Iris: a special kind of attributed information structure that instantiates the design.

Integrity Error-proneness of traditional operating and file systems and databases arises because data can be referenced only by symbolic name, directory structure location, or value. In particular, it is impossible to guarantee *integrity of reference* as the physical location of data changes and to retain integrity of reference to data located on removable media. Our solution provides location-independent internal names that uniquely identify each data object.

The mechanisms developed for this project provide and maintain an *identity* for each type and each data object. The requirements for these identities are dictated by the nature of persistent data. The identity of an object must be *unique* to avoid confusing it with other objects. The identity of an object must be *universal* (i.e., must never change) to avoid invalidating the knowledge of an object's identity in one part of the system when a change is made elsewhere. The identity of an object must be *location-independent* because the location of an object may change in the course of its lifetime.

Integrity is a pervasive goal of an integrated software environment and *type integrity* is central to persistent object management. It matters little how good other aspects of a system are (i.e., how fast it runs, or how much it encompasses), if it produces results that are incorrect or unreliable. Because types are used to express the formal properties of data, object management must include enforcement of the typing mechanisms to ensure integrity. Software applications use such a wide variety of data that it is impossible or impractical to build the complete spectrum into their persistent data system; they are forced to map their types onto the few supported by a database, with loss of integrity and increased error-proneness as the result.

The mechanisms developed for this project support an open-ended type system in which

types can be added at any time and in which individual type properties need not be known to the persistent data system. However, the only way to guarantee type integrity for a piece of information is to have absolute control over all manipulations of the information. This includes determining exactly what operations can be applied to the piece of information. Partial type integrity can be provided when data is being manipulated by the object management mechanisms and by requiring that users of a piece of data have knowledge of its type.

Efficiency Obviously, many characteristics vary with an object's granularity. Examples include expected frequency of access, the complexity and nature of interrelationships with other objects, lifetime, flexibility of the access function set, and the performance requirements on the access functions. The most successful current object managers utilize granularity-based knowledge to tune the overall system performance. As an example, consider the problem of providing complete control over small-grained objects. Allowing them to be independently placeable, independently identifiable, and controlling access to them on an individual basis would be prohibitively expensive and unnecessary for most applications. Different mechanisms, then, are appropriate for different levels of granularity. Careful selection of appropriate granularity for the persistent data of an application is essential in achieving performance.

Current state-of-the-art approaches to object management all distinguish between large and small-grained objects. The distinction is not simply size. Large-grained objects are independent entities, whereas related small-grained objects are grouped into collections which are often represented as a single large-grained object. A file system may be viewed as a structure of large-grained objects (files). Each file is composed of small-grained objects (records or characters), in a certain organization scheme. Object management systems attempt to support these types of relationships in a more general manner.

Conceptual Model The remainder of this section will outline the conceptual model upon which the Iris mechanisms for providing persistence efficiently and with integrity are based.

Figure 1 shows object management from an Iris perspective. The vertical dotted line shows the division between large-grained and small-grained object management components. The two horizontal dotted lines separate general purpose object management (the lowest level), attributed information structures (in the middle), and Iris based persistence (the highest level).

At the lowest level, an *object manager* (left) provides a general set of operations needed to manage large-grained objects and each *item manager* (right) implements a particular

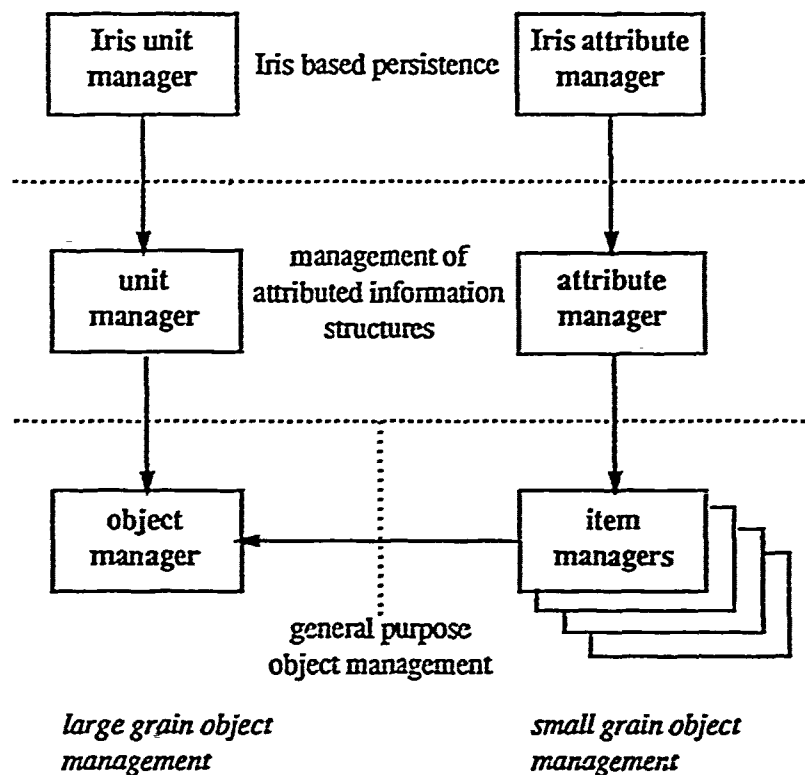


Figure 1: Model of Iris Based Persistence

form of small-grained object management. The item manager exports a data abstraction, the *segment*, which serves as a container for an indexed collection of small-grained objects called *items*. A segment is itself a large-grained object. Operations on (large grain) objects include translations between peripheral storage and main memory. The operations on items include fetch, store and space allocation, all within a segment. Objects (i.e., segments) are independently placeable and identifiable. Items are uniquely identifiable within a segment.

Items can be organized into collections (i.e., segments) in a variety of ways, depending on the properties of the attributed information structure they represent. The designer of a tool must be able to choose a representation for the attributed information structure that exhibits storage utilization and item access times that are appropriate for the application. Item managers vary in the organization of items in a segment because attributed information structures vary in characteristics such as attribute density, attribute size, and uniformity of attribute size.

An *attributed information structure* is a collection of *entities* and information, called *attributes*, about them. The middle level in Figure 1 corresponds to the management of attributed information structures. An entity is a carrier of information. Each entity has an identity and a set of attributes. Each attribute of an entity is a piece of information relevant to the entity; taken as a whole, the attributes of an entity contain all the information concerning it. A *unit* is a collection of entities. The concepts at this second level are built upon those at the lower level.

Every entity is a member of an *entity type*. An entity type includes a list of *attribute definitions*. Each attribute definition specifies the name and value type of an attribute of members of that entity type. If the type of a particular entity has an attribute definition, the entity may or may not actually have that attribute. If it does not, the attribute is said to be *missing*. When a new attribute definition is added to an entity type, the attribute defined is missing for all existing entities of that type, but can be added to some or all members of that type by appropriate attribute manager operations.

Tools do not need to know the entire set of definitions of an entity type, but only about those that are relevant to the function of the tool. A tool's *view of an entity type* is therefore a subset of attribute definitions contained in that entity type. The attribute manager should support views in such a way that changes to the an entity type should affect only those tools which have a view in which that change is visible; tools which have a view in which the change is not visible should not require modification or even recompilation.

An entity collection is an indexed collection of entities of the same type. Entity collections are independently placeable and identifiable. Notice that both objects and entities have unique, universal, location independent identity. Objects are an implementation mechanism; their identities serve to distinguish various chunks of physical storage. Entities are

an abstract mechanism; there is no single chunk of storage that corresponds to an entity. Their identities also distinguish serve to them

The highest level provides Iris-based persistence, discussed in Section 1. Iris unit and Iris attribute management are instantiations of the more general unit and attribute management at the attributed information structure level.

4 Iris Based Persistence

The data objects of a software environment include both composite and atomic objects. They can, therefore, be thought of as utterances in various formal languages. The languages represented include implementation, specification, design, requirements, prototyping, process programming, and constraint languages. Iris¹ provides solutions to the information managements problems of distributed software environments. It is a semantically based system for representing and managing pieces information that can be viewed as utterances in some formal language(s). An Iris system includes a common information structure as well as both small- and large-grain object management. Iris based persistence has been used in an Ada-to-Iris tool, where Iris unit management is equivalent to Ada program library management. An analysis of this design combined with measurement data on earlier prototypes indicates that performance in excess of 50,000 item accesses per second on a Sun 3/60 is achievable. This is well within the performance goals set out in Section 2.

The Iris Information Structure The Iris information structure is a language independent form for representing the sentences of any formal language. It serves as a medium of information exchange and sharing among the tools of a software environment. It is an extensible and open-ended system with respect to the information it can capture and represent.

At an abstract level, the Iris information structure is a tree. Each Iris tree represents a composition or expression consisting of *references* and *applications*. Corresponding to this, an Iris tree is composed of two kinds of nodes: *reference nodes* and *application nodes*. For example, the expression $f(x, g(y, z))$ consists of references to entities named f , x , g , y , and z and applications of f and g . Reference nodes are interpreted as references to declarations that appear elsewhere in the Iris structure. The first child of an application node is its *operator*. The operator identifies an *operation* which is applied to the remaining children, which are called *arguments*. Frequently, the operator is a reference to the declaration of a named operation, but it can be any operation-valued expression represented as an Iris tree.

¹Iris was the Greek goddess of the rainbow and messenger of the gods.

If the reference nodes of Iris are viewed as leaves (terminals) then the Iris representation can also be viewed as an abstract syntax tree with the application nodes acting as nonterminals. Each reference node does contain, however, a reference to a declaration which is itself an application node appearing earlier in (a preorder walk of) the Iris structure.

Iris is unique in that all operators are described within its own structure. It has no primitives. This means that individual tools need recognize and provide special case processing for only those operations that related directly to the functionality of the tool. For example, a semantic analyzer need recognize only operations that are declaration, scope, or type valued but does not have to distinguish between control structures and arithmetic operations. This contributes to the simplicity and small size of Iris based tools.

Iris is also a higher order system in that it provides full support for computed operations. A computed operation may appear either in place at the point of its application (i.e., as another application node which is the operator of the application) or as the value of a declaration which is referenced at the point of call (i.e., as a reference node which is the operator of the application). The combination of internal and higher order specification means Iris can be used to represent any formal language and that Iris based tools can be reconfigured for multiple and evolving languages with little or no change to their components.

To specify the representation of any language L , two things are needed: a *grammar* and a set of *L-standard declarations*. The grammar describes the correspondence between the concrete syntax of the language and its abstract syntax represented as Iris expressions. The *L-standard declarations* specify the built-in operations of the language, i.e., those operations which are available within the language but are not declared within programs of the language (e.g., control structures in implementation languages or invariants in specification languages).

Iris Attributes: Small Grain Object Management Small grain object management in Iris includes item and attribute management. Iris trees are implemented as collections of attributes. Each node in an Iris tree, along with its attributes, is an entity. Each attribute value of each entity is an item.

Units: Iris Large Grain Object Management Figure 2 depicts an entity collection. Each row is an entity and each column is an attribute of the type of the entities in the collection (i.e., an attribute collection). Each square is an attribute of a particular entity in the entity collection, and is represented as an item.

There are two ways to group Iris attributes for storage, as shown in Figure 2. The first is to group all the attributes of a single entity into a unit. This is called *horizontal*

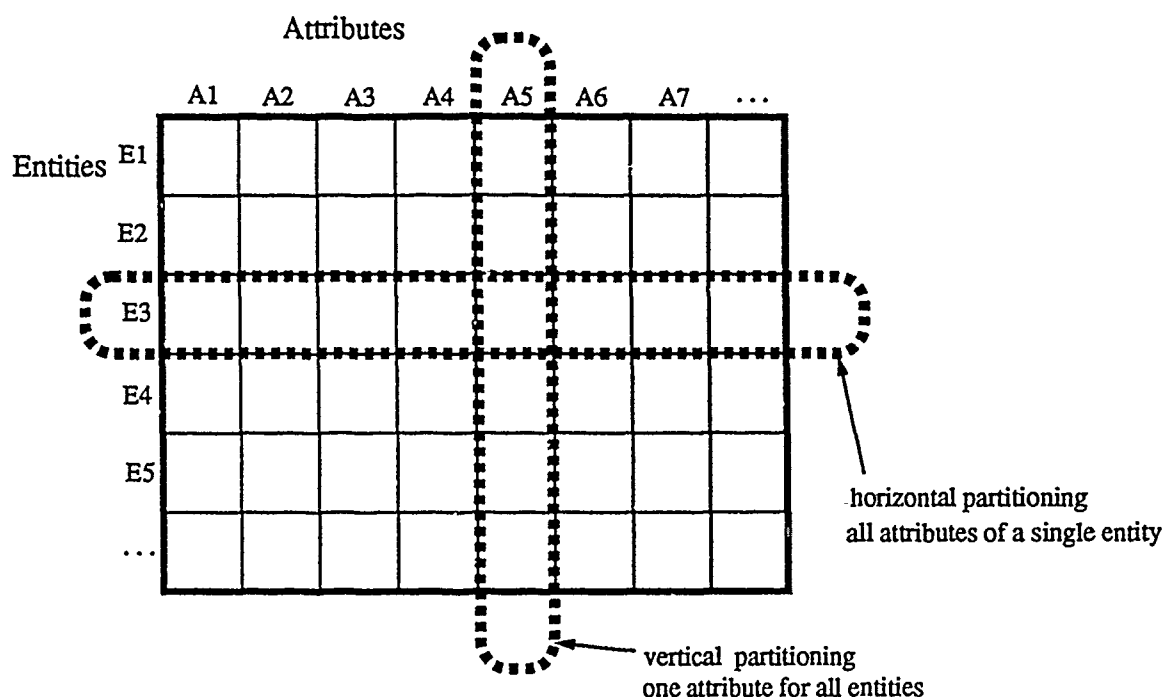


Figure 2: Organization of Iris Attributes for Storage

partitioning. The second is to group a single attribute for a collection of related entities into a unit. This is called *vertical* partitioning. While either partitioning is adequate, Iris uses vertical partitioning. The advantages of vertical partitioning over horizontal are twofold: new attributes can be added without impacting existing attributes or tools and attributes not needed by a tool need not be loaded into memory. The disadvantage of vertical partitioning over horizontal is that accessing attributes is more complex.

5 Summary and Future Work

Integrity and Inconsistency Information in a software environment is generated from many sources (some human interactions, some computations). It is frequently updated, and is subject to change from multiple, independent sources. Consistency of the information is difficult to maintain in such volatile situations. Furthermore, in distributed systems and in the presence of removable media, communication delays necessitate replication of frequently accessed data and preclude complete consistency among all copies.

There are several commonly used approaches that attempt to eliminate inconsistency by periodic, controlled update of changed data. The first, which might be called "lock the

world", is simple in concept and is formally sound, namely: when it is necessary to update a data structure (or set of related data structures) place a lock on the entire database that will prevent both access and reference from any source other than the updating process until the update (of all copies) is complete. Such an approach, of course, can have extremely high cost in performance.

There are several variations that can significantly improve performance. The most obvious is to lock only those portions of the world that are directly affected by the update, thus allowing independent activity to continue in parallel. Another is to select a very small granularity of data for locking in order to maximize the number of independent parallel activities that can be accommodated simultaneously with an update. Such methods can work quite well in small, highly localized databases, but are prohibitively expensive in distributed systems because of the inherent communications delays.

An alternative which overcomes some of the communications delay is to partition the database itself into disjoint partitions (typically the nodes of the distributed network) and then to prohibit interpartition accesses. Locking and update can then be accomplished one partition at a time and propagated throughout the network. This approach reduces the delay as seen by any one node by accomplishing the communication outside the lock. This solution tends to increase the amount of mutable data that must be replicated and imposes a strict requirement for independently managed partitions. The former restriction complicates sharing of data stored at a central location; the latter precludes the use of removable media as a means of sharing and moving mutable data. In any case practical systems based on this approach have generally been those in which update propagation times of the order of one day are acceptable. It then is possible to do local updating within each partition during the day and propagate all the changes at night when there is little or no use of the systems.

The cost that cannot be tolerated in most systems is not the communications delays, but rather the delays imposed by locking. Solutions must either tolerate delays in general system response time caused by the locking, accept the one day update delays that are a consequence of updating only at night, or find a way to update without locking.

Locking, then, is prohibitively expensive and techniques to overcome these expenses are not uniformly applicable. While careful design reduces the adverse consequences of the remaining inconsistency, there are no guarantees against inconsistency nor even that adverse consequences are detectable.

The traditional method of updating without locking is simply to remove the lock for purposes of access or for both access and update. Other means are used to minimize the probability that erroneous or inconsistent data will be accessed, or that the adverse effects of such accesses will not be catastrophic. One of the better known examples of this approach

are airline reservation systems in which there is a single shared central copy of the most current data. All updates must occur directly to this shared copy and are performed there under lock. Additional local copies are used for access purposes and can be arbitrarily far out of date. Similarly, update requests can be delayed (actually queued) arbitrarily long without delaying the updating process. The effect is that sometimes an access will indicate available seats but the update will fail because none are available, or a reservation will be accepted (i.e., queued for update on the assumption that it will succeed) and then later fail due to the effects of other queued requests. The latter results in overbooking.

We find all such approaches unsatisfying. Locking is, in general, prohibitively expensive in distributed systems and the techniques to overcome those expenses are not applicable in many situations. The traditional nonlocking approaches do not guarantee consistency or even detect it, but instead attempt to minimize adverse consequences. Our approach rejects as infeasible avoiding inconsistency. Instead, our approach creates a situation in which inconsistency is safely and efficiently detected and managed. The key to detection of inconsistent data is twofold. First, the various (updated) versions of an object must be distinguishable (universal identities are adequate for this purpose, see Section 3). Secondly, objects or application that are used in combination must maintain records of the identities of the versions or types of the objects with which they must interact. These techniques have been used successfully in our Ada to Iris tool in order to enforce order of compilation.

Summary The solutions outlined here can significantly improve the efficiency, reliability and robustness of applications that use persistent data. Key among these are distributed software development environments, such as that in the KBSA paradigm. The solution consists of efficient mechanisms that form a substrate to facilitate cooperation among KBSA facets via sharing and reuse of information. The substrate permits identifying, storing, accessing, maintaining, sharing and reusing information in a distributed environment.

Some of the current and planned scientific and engineering features that contribute to the efficiency and safety of our mechanisms for the management of persistent objects are:

- Object identity that is unique, location independent and universal.
- Integrity for all types, even those that are user defined, while the data is under the purview of the persistent mechanisms.
- Safe sharing via object identity rather than via user given names or data values.
- Recognition and management of inconsistency in the volatility of software development environments where data is frequently updated from multiple, independent sources.

- Vertical partitioning of attributes.
- Operations optimized for the different demands of small- and large-grain objects.
- Multiple item managers to exploit the various properties of attributed informations structures.

Acknowledgements

We thank our Incremental Systems colleagues David Mundie and Jonathan Shultis, both of whom were instrumental in the development of Iris. We acknowledge the Arcadia Consortium whose members have encouraged this work with their reviews, comments and suggestions.

References

- [AB87] M. P. Atkinson and O. P. Buneman. Types and Persistent Database Programming Languages. *ACM Computing Surveys*, 19(2):105-190, June 1987.
- [BB87] P. Buneman and F. Bancilhon, editors. *Workshop on Database Programming Languages*, Roscoff, France, September 1987.
- [Ber87] P. A. Bernstein. Database System Support for Software Engineering. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 166-178, Monterey, CA, March 1987. IEEE Computer Society.
- [BFS87a] D. A. Baker, D. A. Fisher, and J. C. Shultis. A Practical Language to Provide Persistence and a Rich Typing System. In *Workshop on Database Programming Languages*, Roscoff, France, September 1987.
- [BFS87b] D. A. Baker, D. A. Fisher, and J. C. Shultis. Persistence and Type Integrity in a Software Development Environment. In *Workshop on Persistent Object Systems: their Design, Implementation and Use*, Appin, Scotland, August 1987.
- [BSF88] D. A. Baker, J. C. Shultis, and D. A. Fisher. Mechanisms for Providing Persistence in a Distributed Software Development Environment. In *Proceedings of the 3rd Annual Knowledge Based Software Assistant Conference*. Rome Air Development Center, August 1988.

- [CAI88] U.S. Department of Defense, Proposed Military Standard DOD-STD-1838A (Revision A). *Common Ada Programming Support Environment (APSE) Interface Set (CAIS)*, 1988.
- [Coo87] R. Cooper, editor. *Workshop on Persistent Object Systems: their Design, Implementation and Use*, Appin, Scotland, August 1987. Universities of Glasgow and St. Andrews.
- [Ele89] D. M. Elefante. An Overview of RADC's Knowledge Based Software Assistant Program. In *Proceedings of the 4th Annual Knowledge Based Software Assistant Conference*. Rome Air Development Center, September 1989.
- [Gol89] A. Goldberg. Reusing Software Developments. In *Proceedings of the 4th Annual Knowledge Based Software Assistant Conference*. Rome Air Development Center, September 1989.
- [KC86] S. N. Khoshafian and G. P. Copeland. Object Identity. In *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference (OOPSLA '86)*, pages 406-416. *SIGPLAN Notices*, 21(11), 1986.
- [Mey89] N. Meyrowitz, editor. *Proceedings of the Object-Oriented Programming Systems, Languages, and Applications Conference (OOPSLA '89)*, New Orleans, Louisiana, October 1989. ACM SIGPLAN, *SIGPLAN Notices*, 24(10).
- [SDE88] ACM SIGSOFT. *Third Symposium on Software Development Environments*, Boston, Massachusetts, November 1988. Appeared as *Sigplan Notices* 24(2) and *Software Engineering Notes* 13(5).
- [TBC+88] R. N. Taylor, F. C. Belz, L. A. Clarke, L. Osterweil, R. W. Selby, J. C. Wileden, A. L. Wolf, and M. Young. Foundations for the Arcadia Environment Architecture. In *Proceedings of the Software Engineering Symposium on Practical Software Development Environments*, pages 1-13, Boston, MA, November 1988. ACM SIGSOFT/SIGPLAN. Appeared as *Sigplan Notices* 24(2) and *Software Engineering Notes* 13(5).

Epistemic Type Systems *

Jon Shultis
Incremental Systems Corp.
319 S. Craig St.
Pittsburgh, PA 15213

February 22, 1991

1 Abstract

Current type disciplines reflect firm commitments about the taxonomic and ontological structure of the universe of discourse. Epistemic type systems regard such committed type systems as examples of ideal *epistemic states* representing the beliefs of an agent about the universe. These states are subject to revision and refinement as a result of interactions between the agent and its environment. The focus of attention in epistemic type systems is the mechanism of change, which should be *conservative* in the sense that prior beliefs should be interpretable in any revision. In this talk some basic principles of epistemic type systems are explored, taking Scott's information systems as a point of departure.

2 Opening Remarks

The traditional focus of these workshops on Mathematical Foundations of Programming Semantics has been on problems and applications of mathematics to the semantics of programming languages and related systems as they are today. I am breaking with this tradition by raising questions about the mathematical semantics of language more generally, though with an emphasis on language as it might be used to communicate with computers. Programming languages, and formal languages generally, have the advantage of being "mindless", and hence simpler and easier to describe. But they are rather rigid and impoverished mockeries of language.

My excuse for bothering you with a lot of philosophy and speculation, and disappointingly little mathematics, about aspects of language that are not traditionally considered in this forum, is that

*This work was supported in part by DARPA under contract number MDA 972-88-C-0076.

the critical problems of software engineering cannot be solved without a comprehensive semantic basis for entire computing environments which integrates all components in the environment by transcending the barriers inherent in the local formalisms and conventions used in the development and operation of individual components. The challenge for mathematical semantics is great, but not impossible. As Jon Barwise says in the epilogue to his recent status report on situation semantics, "*Someone* will eventually lay the foundations for the mathematics of meaning" ([Bar89], p. 297). However, the problem has to be understood and taken seriously. It is my hope that some of you will become interested enough in the problem to want to work on it yourselves.

3 Prelude: the greatest prime

The overall goal of the Prism project at Incremental Systems is to lay the foundations for a new generation of computer languages in which to conduct the long-term development, operation, and maintenance of large integrated software systems. Appropriate languages will have to cope with incomplete, imprecise, inconsistent, and continually changing information, including information about the interpretation and use of information. The only examples of languages in common use which exhibit these capabilities are natural languages.

Although we do not believe that anything approaching the complexity and subtlety of natural language is either technically achievable (at present) or necessarily desirable, there are features of natural language which suggest technically achievable revisions to the basic structure of computer languages which would advance our goals. The current paper is a small contribution to the mathematical foundations of one such revision, in the area of types.

The following example of a familiar proof serves to illustrate one respect in which current notions of type are deficient.

Suppose that p is the greatest prime number, and let x be the product of all the primes up to and including p , plus one. Clearly, x is greater than p . It is also prime, because it has no prime factors (upon division by any of the primes, the remainder is one). But then p is not the greatest prime, contrary to our supposition, and therefore there can be no greatest prime.

Everyone understands the foregoing argument, though it is incomplete and lacks rigor. Replace every occurrence of the phrase "greatest prime" with "smallest positive real", however, and it becomes gibberish, though both phrases fail to denote.

Now, "the greatest prime" is a term which purports to refer to the unique inhabitant of what is actually an empty type, *viz.* the type of greatest primes, and similarly for "the smallest positive real" (and, for that matter, " x "). Because the type of greatest primes and the type of smallest positive reals are coextensive, they are, in any extensional account of types, the same type. But

clearly the types are quite different; "greatest prime" confers the properties of being prime, and the greatest such, on any term of that type, whether the term refers or not. That is why the argument makes sense, even though it involves reasoning about a nonexistent. It also explains why coextensive terms are not freely interchangeable in any but the most artificially constrained languages.

Two things seem clear: that reasoning of this sort is extremely common and useful, and that current computer languages don't support it. The historical and philosophical reasons for this are fascinating, but I don't want to go into them here. Suffice it to observe that, if one assumes that only meaningful utterances should be allowed in a language, and that meaning is denotation, the way is blocked.

Of course, this is old stuff to anyone who has more than a passing acquaintance with linguistics, philosophy, or logic beyond the small subset that is heavily used in programming language semantics, and I apologize to my colleagues if I seem to be repeating what everybody knows already. My excuse is that computer scientists at least *seem* to be unaware of the philosophical assumptions underlying programming languages, and the inherent limitations they imply. Considering the dramatic effect that philosophy has had on the overall shape of computer technology, I find it especially disappointing that so many computer scientists dismiss anything resembling philosophy out of hand. End of sermon.

4 From Extension to Intension

To a first approximation, the purpose of types is to express the general properties of classes of objects so that algorithms and other thoughts pertaining indifferently to all examples of the class can be expressed. Generalization, in turn, is the *sine qua non* of computation. Algorithms are useful because they exploit universal properties of relations; *ad hoc* mappings might just as well be tabulated once for all and filed away.

Once we accept this much, two paths are open: to focus on the properties, or on the classes of objects. The former route leads to intensional types; the latter to extensional types. The purpose of this section is to show how the extensional conception, which has thus far dominated in formal semantics, is a special case of the intensional, and can be recovered from it.

Now, it is a commonplace that every consistent theory has a model, so *if* we are interested only in consistent sets of properties, or are willing to identify all inconsistent sets, then there is little harm in identifying each type with its extension. Under these conditions, the distinction between intensional and extensional types vanishes.

Dana Scott's formulation of domains in terms of information systems ([Sco82]) exploits this observation, restricting consideration to *consistent* sets of properties. The (partial) elements of a

domain are taken to be consistent sets closed under consequence¹, and the total elements (proper individuals) are simply maximal elements.

The view advanced in [Sco82] is that types are domains, inhabited by nontotal (*i.e.*, properly partial) and total elements. But what are nontotal elements? Intuitively, a nontotal element x is a set of properties which is insufficient to identify a unique individual; for if x entails all of the properties of an individual, then all of those properties belong to x , and x is thereby an individual (*i.e.*, is maximal). In fact, a nontotal element x consists of all those properties which are common to a *class* of individuals (total elements), namely those which contain x . Put another way, x expresses the general properties of a class of objects, which is what we expect of a type!

As Scott points out, it is a simple matter to construct a domain of domains, by specifying an information system in which the basic properties describe the consistent sets and entailment relations of the internal domains. A two picosecond proof shows that the notion of "subtype" on domains considered as types corresponds to approximation between domains considered as elements of a domain of domains. What this suggests is that the notion of closed consistent set is a more general, and prior, notion of type, of which the domain theory of types is a special case. The fact that the same intuition underlies Smyth's slightly different rendition of domains as sober spaces reinforces this suspicion.

The domain theory of types has some very specific technical goals, *viz.* to guarantee recursive definitions *at* all types, recursive definitions *of* types, to be effective, and to rule out certain sources of error. What domain structure provides is a highly general theory of types satisfying these additional goals.

Although these features of domains are essential for some purposes, they are unnecessary or even inimical to others, as is shown by the example of the greatest prime. (To be explicit, the singleton $\{x \text{ is the greatest prime}\}$ is not consistent, so the property of being the greatest prime is inadmissible as a basic datum in an information system.) What is needed for such purposes is a theory of types which *manages*, rather than eliminates, inconsistency. This requirement militates against any restriction to consistent (or, for that matter, coherent) sets.

Another problem for Prism, which I raised in my casual talk in Boulder two years ago, is that the assumption "... that sufficiently many propositions have been supplied to distinguish between distinct elements" (Scott, *op. cit.*, p. 2) is not reasonable when the intended domain is the real world, or even the small and highly structured part of it known as mathematics. The propositions which we use to distinguish things in the real world arise through a combination of discriminatory observation and model formation; more plainly, we *learn* to classify by noticing differences, and *create* taxonomic structures for the purpose. What this implies is that the set of propositions cannot be stipulated as *a priori* data to the theory.

¹Scott uses the word "entailment", which is properly a semantic relation between syntactic formulae, but he uses the symbol \vdash denoting syntactic provability, and refers to sets closed under this relation as being "deductively closed", thereby encouraging the impression that this is a proof-theoretic notion. However, in taking the elements to be deductively closed consistent sets, he forces the relation to be semantic.

Some of Smyth's work touches on epistemology (see, *e.g.*, [Smy85]), though he too assumes *a priori* properties, and models learning as a kind of monotonic revelation converging on perfect knowledge. The problem with the "topology" of reality is that, not only are the points not there, but the neighborhood systems we create are subject to revision in successive epistemic states. The periodic table we use today is simply not a refinement of "earth, air, fire and water".

Notice, however, that the ancient periodic table is still comprehensible, and even occupies a rather prominent place in our constellation of ideas. However, we have reclassified it, from believed to disbelieved. Now, the only way a point can shift from a neighborhood called "believed" to a disjoint neighborhood called "disbelieved" is if those labels get reassigned to different neighborhoods, whether in the same or another neighborhood system. Thus, if one wants to maintain that properties are neighborhoods, one has to index classificatory terms. On that account, "believed" today refers to one neighborhood; it referred to a different neighborhood in 300 B.C.; and the ancient periodic table lies in the intersection.

The only problem with such reasoning is that it begs the question, because its account of "states of knowledge" and how they evolve is really not about knowledge but about something else which behaves quite differently. The questions before us are, therefore: what properties do we want types to exhibit, and what kind of mathematical structures are suited to modeling a broader notion of intensional, epistemic types? Some preliminary answers to these questions follow.

5 Evidence for the Proposition

In outline, the theory of types developed here is quite simple and familiar. Types are sets of properties, and a type system is basically a partial ordering of types derived from relations of consequence between types and properties. However, unlike Scott's properties, which are "tokens" drawn from a set, or Smyth's neighborhoods, which behave like "semantic tokens", our properties have internal structure which includes references to types. Thus the type hierarchy and its "underlying" properties are intertwined, so that the "field of basic properties" is not fixed, but is part of the developing state of knowledge.

The goal of an open-ended language with the real world as its semantic domain seems to force such interdependency. In our view, epistemic states are models of reality, populated by individuals and articulated beliefs about them. The word "model" is to be understood here in the sense of something which exhibits selected features of interest by abstraction and possibly scaling, as in engineering models and model trains. Properties and types are the bases of individuation and classification, respectively.²

²For those interested in cognitive science I should point out that this entire discussion is limited to the theorizing component of mind, in which abstract concepts are formed and manipulated. It should properly be placed in context of a more complete theory of mind, including distributed-representation sensorimotor and affective components. Alas, that will have to wait for another occasion.

Epistemic change consists in modification of the model, which may involve anything from a simple revision of the properties attributed to an individual to a revision of the type system, which has the potential for radically changing the scheme of individuation and hence the set of recognized individuals. Change is stimulated by the need to assimilate new information, which may be generated externally, say by meeting someone new, or internally, say by proving a theorem in mathematics, or by a combination, as when experiment clashes with prediction. Change is damped by the inertia of the existing model, resulting in a kind of tension between conservative and liberal forces. Rationality, or at least sanity, depends on stable balancing of those forces. (One might even speculate that there is an analog to the Hamiltonian characterizing the trajectories of sane minds. Perhaps the connectionists will discover it someday.)

What, then, is a property? While riding home from the video store recently, my daughter and a friend (both between four and five years of age) started discussing infinity. They agreed that there is no last number, but they disagreed as to the reason. Katie's position was that one can always continue counting starting with any purported largest number (not quite her way of saying it), while her friend held that it is true because "my daddy told me". The curious thing is that neither would accept the other's reason, regardless of the decibel level of its delivery.

The clue offered by this story is that both children seemed to regard the defining feature of the property in question to be the evidence they could advance to support its predication of an individual. Lacking agreement about reasons, there was mutual suspicion about content and understanding. This suggests that we take the set of evidence for belief as the meaning of a proposition.

Let $\vdash^b p(x)$ denote that the (implicit) agent stands in a relation \vdash^b of *belief* to the proposition $p(x)$ (which, intuitively, asserts that x exhibits property p). If we have occasion to make the agent explicit, it will appear to the left, as in $I \vdash^b p(x)$.

Although the treatment of belief as a simple binary relation has precedents in the literature on propositional attitudes, real belief is subject to shading and uncertainty. We therefore associate a *strength* with each belief, lying in the real interval $[0, 1]$. In general, a belief strength s will be written below the belief symbol, thus: $\vdash_s^b p(x)$. For any belief $b = \vdash_s^b p(x)$, we write b_p to denote the proposition believed, i.e. $p(x)$, and b_s for the strength of the belief.

When no strength is indicated, as in $\vdash^b p(x)$, it is assumed to be exactly 1; and when the strength of belief is 0, we will use the familiar negation $\nmid^b p(x)$. Note, however, the difference between $\nmid^b p(x)$ and $\vdash^b \neg p(x)$! The former says that $p(x)$ is not believed; the latter, that $p(x)$ is believed to be false.

Now, if an agent holds a belief b in context Γ , it does so by virtue of possessing a collection of evidence which it construes as supporting \vdash . The particles of evidence are themselves beliefs,

with their own content and strength, and they bear on other beliefs *via* weighting functions.³ Altogether, then, the epistemic state of an agent is characterized by a set of *epistemic factoids*, which may be expressed using the notation $\Gamma \xrightarrow[e]{b} p(x)$, to be read as "in context Γ the evidence e encourages the agent to believe $p(x)$ with strength $s = w(e_s)$ ".

The goal of any evidential semantics, of which Brouwer's intuitionism and Girard's phase semantics for linear logic are more or less familiar examples, is to reduce the content of propositions to evidence; that is, to equate *what* is believed with the possible *reasons* for belief. It is then possible to dispense with propositions *per se*; it suffices to name the beliefs and the witnesses (units of evidence), and describe how the latter give rise to the former.

In the current situation, this enables us to eliminate propositions from the expression of epistemic factoids, thus: $\Gamma \xrightarrow[e]{b_s} b$, where $b_s = w(e_s)$. Doing so frees us from any obligation to give a linguistic account of propositional content, thereby avoiding the conundrums of reference, compositionality, and extension which plague sentential/propositional theories of knowledge.

Of what use, then, are propositions? Articulated propositions e_p, b_p are useful primarily for (inexact) communication and for forming theories which *justify* the agent's reasoning, taking b_p as the starting point of a systematic explication of why the agent is justified in taking e_p as evidence for b_p .

In the sentential/propositional theories alluded to earlier, the *structure* of propositions is central to the account of how evidence bears on belief. An intuitionistic proof of $\phi \wedge \psi$, for instance, is a pair of proofs $\langle x, y \rangle$ where x is a proof of ϕ and y is a proof of ψ . In classical logic, also, the content of propositions is given by a recursive rule over their structure, and the relationships among those contents are used to justify syntactic inferences. That is model theory in a nutshell.

Now, formal logic in all its variety and splendor is a wonderful and useful invention, but its applicability to everyday knowledge and reasoning has been overestimated. In theories of language, mind, and the behavior of intelligent (as opposed to so-called "ideally rational") agents, it is absolutely critical to maintain the distinction between why an agent draws a conclusion, and why an agent may be justified in drawing that conclusion. An agent draws conclusions because its machinery is attuned to respond in a certain way to stimuli. It need not, and generally will not, respect logical propriety. Indeed, if it attempts to do so it will be outpaced by events in its environment, and thereby fail.

Similarly, one must maintain a distinction between uniformities in an agent's behavior and the

³There is an obvious threat of infinite regress in this account. If beliefs are supported by evidence, and the units of evidence are beliefs, what supports the evidence? My answer is that certain beliefs are stimulated spontaneously, such as the beliefs represented by the activation levels of cones and rods in the retina of the eye, which our theories of biology and physics attribute to such things as frequency-selective photochemical reactions. Given the impossibility of telling whether such attributions are correct, or if we are being manipulated by Cartesian demons (read: artificial reality puppeteers) the content of these "immediate sensory stimulus beliefs" — that is, the evidence which supports them — is beyond knowing, though we may (and do) form various theories about it, and call it "reality". However, spontaneous beliefs may also be hallucinatory, conjectural, or explicitly fictional.

causes of those uniformities, on the one hand, and a justifiable system of rules (*i.e.*, a theory) which more or less closely accords with the agent's behavior, on the other. At best, the latter is a functionalist explanation for the degree of "success" enjoyed by an agent in its environment. It is a grave, but all too common, error to attempt to forge such theories into implementations of intelligent agents.

Then the meaning of a property p satisfies the equation

$$\llbracket p \rrbracket \Gamma x = \{e | \Gamma \overset{e}{\sim} \vdash^b p(x)\}.$$

As a sanity check, note that if we assume a truth value $\hat{p}(x, \Gamma)$ for each proposition $p(x)$ in context Γ , and further assume that $\Gamma \overset{e}{\sim} \vdash^b p(x)$ *iff* the evidence e is $\hat{p}(x, \Gamma) = \text{true}$, then the meaning of p reduces to a characteristic function $\llbracket p \rrbracket \Gamma = \lambda x(\hat{p}(x, \Gamma))$.

As an aside, I recently noticed that the foregoing account bears a strong similarity to the phase semantics for linear logic [Gir86], in which the meaning of a proposition p is the set of "phases" (evidences) which "verify" (lead to belief in) p , written $\llbracket p \rrbracket = \{e | e \models p\}$. The main difference is the subjectivity of the "observer" in the epistemic account, and the consequent dependence on the observer's epistemic state, Γ .

For purposes of the current paper, this explication of properties and propositions should suffice. However, it neglects the internal structure of properties, which is involved in the processing of evidence. In the example of the greatest prime, the relevant structure is a conjunction of two other properties, *viz.* being prime and being the greatest such. The parallel argument for the least positive real is nonsense (to me) because there is no reasonable (to me) rule of logic which licenses the inference that something is prime from the premise that it is a positive real and the least such.⁴ Hence the argument is not admissible as evidence for the proposition that the least positive real does not exist; it is not part of what that proposition means (to me).

Returning briefly to the infinity debate, we can now see that the meaning of "doesn't exist" is different for Katie and her friend, because their evidence sets differ. This position may seem unpalatable at first; surely there is some sense in which the children mean the same thing by the proposition, "there is no last number". My claim is that they mean something similar, but not exactly the same.

Another example may help to clarify this point. Suppose that George and Mikhail have been conversing for years and using the term "the sun" in apparently the same way, when one day Maggie appears on the scene and declares "There's the sun!" while pointing at a bright light in

⁴An obvious rejoinder to this assertion is that, in fact, many familiar logics have rules which explicitly license such inferences, and they are validated by appropriate models. However, the decision to identify nonexistents in a model, whether motivated by a *a priori* philosophy or merely a desire to validate a formal system, is not necessary, and there is nothing to prevent one from populating models with distinct points representing distinct modes of nonexistence. To the extent that such distinctions are useful and important, and it seems irrefutable that they are, models (and logics) which ignore them are "unreasonable", no matter how common and familiar they may be today.

the sky. Now it happens that George believes everything Maggie says, but Mikhail disbelieves everything she says, so when each is asked whether the apparition in question is, or is not, the sun, they give different answers. Because they differ on the extension of the term, they cannot mean the same thing by it, and there is no difference in the facts of the situation which can account for their disagreement; there is only a difference in their evidence sets. (Shades of late Wittgenstein...)

Nonetheless, if we examine the intersection of George and Mikhail's evidence sets we find they have a good deal in common, and this may lead them to believe that their meanings are the same, despite their conflict over Maggie's testimony. Of course, doing so may cause them to cast doubt on other beliefs, *e.g.*, about Maggie's trustworthiness, or the target of her gesture.

Topologically speaking, suppose we treat evidence sets as open neighborhoods, drawn from two different neighborhood systems. As long as the intersection of the two neighborhoods is again a neighborhood in each system, one can maintain the impression that there is an objective shared meaning contained in a sequence of intersecting neighborhoods, and things become reminiscent of the complete prime filter construction of domains. A theory of "local sobriety" would help out here.

Another example which sheds some light on the problem of how meanings are related across changes in epistemic state is an embellishment of the Richard-Soames problem, due to Nathan Salmon [SS88]. The premise is an ancient astronomer *A* who refers to Venus as "Hesperus" when it appears in the evening sky, and "Phosphorus" in the morning, and is unaware of their material identity. The astronomer takes measurements and calculates the mass of each, makes an error, and declares their masses unequal; in symbols, $A \vdash^b m(H) \neq m(P)$. Salmon's puzzle has to do with the principle of direct reference, which would lead one to infer that $A \vdash^b m(H) \neq m(H)$, which is not the case. In the evidence theory of propositions, the fatal inference would require, not the "objective fact" that "Hesperus" and "Phosphorus" corefer, but that *A* believe that they do, which she does not. On the other hand, given that $I \vdash^b H = P$, we can infer that $I \vdash^b ((A \vdash^b m(H) \neq m(P)) \wedge m(H) = m(P))$, which is true; *I* think the astronomer is mistaken! So the Richard-Soames species of anomaly, *per se*, is extinct in the evidence theory.

As usual, such formulas have to be composed and read carefully. In general, $\exists p (I \vdash^b (A \vdash^b p) \wedge \not\vdash^b p)$ expresses that there is some specific thing about which *I* believe *A* to be mistaken, in contrast to $I \vdash^b \exists p ((A \vdash^b p) \wedge \not\vdash^b p)$, which says that *I* believe *A* to be mistaken about some undetermined thing.

The real puzzle is to explain what happens when the astronomer learns that Hesperus is Phosphorus, and corrects her calculations. Suppose that the original mass calculated for Phosphorus was *p*, and this was incorrect. Then the epistemic transition involves a change in the meaning of the proposition that the mass of Phosphorus is *p*; that is, $\llbracket m(P) = p \rrbracket \Gamma_1 \neq \llbracket m(P) = p \rrbracket \Gamma_2$. *A* would likely object, however, to the suggestion that there had been any change in the meaning of the proposition. What I claim underlies such intuitions is the fact that *both* evidence sets contain

hypotheticals on the order of

$$(\vdash^b OK(d)) \wedge (\vdash^b OK(d) \Rightarrow r(d, \phi) \leadsto \vdash^b \phi) \wedge (\vdash^b r(d, \phi))$$

where $OK(d)$ means that the measuring device d is functioning properly, $r(d, \phi)$ means that d reports that ϕ , and ϕ is the proposition $m(P) = p$. That is, if it is believed that the device is functioning correctly, and it is believed that the report of a correctly functioning device should be believed, and it is believed that the device is reporting ϕ , then this is evidence for believing ϕ , in any “imaginable” epistemic state. Sets of such “decontextualized” evidences, insofar as they are invariant over a range of epistemic states, behave locally like ideal, “objective” concepts.

Incidentally, it goes without saying that the principle of logical omniscience has no place in this scheme, because the meaning of a proposition is the set of evidence which the agent is ready to accept in its support, which frequently excludes most of the evidence which the agent is *capable* of collecting. In general, one expects that an agent will draw out consequences of its beliefs only to the extent of its capabilities and inclination. Its capabilities, of course, are determined in part by what it believes about valid inference and how it can be conducted, and its inclination is influenced by its beliefs about the value of any such activity. Both, of course, can be externally influenced, and in the case of computers there is an opportunity to inculcate certain ideals of rationality into a system, at least at the outset. Over time, however, it might turn out that becoming somewhat irrational is a rational strategy for dealing with the world, which would lead to an attenuation of those ideals.

6 Types

As a first approximation, define a *type* to be a set of properties. This definition immediately raises several questions: Are logically equivalent sets of properties the same type? What about contingently equivalent sets of properties ($\{x = 9\}$ and $\{x = |planets|\}$, to borrow Quine's example)? How are types affected by context? By changes of epistemic state? How are types related to one another?

To answer these questions, I have to bring out some points that are implicit in the foregoing discussion of the semantics of properties. To begin with, I have been assuming that each property has a unique identity which is invariant over all epistemic states and contexts. This identity is not syntactic; a given property may be designated by arbitrarily many terms, and any term may designate any number of properties, depending on context. Nor is it necessary for the reference of a term to be unambiguously resolved; if a passage is meaningful under several available interpretations of a term, it may legitimately be used to convey multiple thoughts. And, because the meaning of a property can change, its identity is not semantic, either. Rather, its identity is that of a point in a domain of mental representations. In general, we call such points “individuals”.

An immediate consequence of taking properties to be “mental tokens” is that, not only can

the meaning of a property vary, but distinct properties can have the same meaning in certain epistemic states. A number of problems having to do with contingently coincident descriptions, such as "Aristotle" and "the teacher of Alexander" are thereby solved. The properties of being Aristotle and of being the teacher of Alexander are different, though their meanings are (for me) the same.

Now, consider the allegation $x:t$, that x is an example of type t . In developing software, we want to be able to develop t over time; for example, we may assert the relation between x and t , and then proceed to state the properties in t gradually. Therefore, as with properties, we take t to designate an individual; the epistemic state determines what properties are attributed as members of t . What makes t be a type is the attribution that it is a set of properties.

For emphasis, the proper definition of type is: an individual which has the property of being a set of properties. *Which* set of properties is not necessarily determined, or fixed, however.

It is clear that in certain epistemic states distinct types may have the same, or merely logically equivalent, sets of properties attributed to them as members. So logical equivalence alone is not sufficient grounds for believing that two types are the same.⁵ Additional beliefs, such as that neither type has any unspecified properties, are required to warrant the conclusion that their differences are merely apparent.

Having explained and motivated these preliminary definitions and concepts, let us fix an epistemic state, and explore the mathematics of its types. Assume that, as in any epistemic state of interest, there is a relation \vdash of *deducibility* between sets of propositions and propositions. The details of this relation depend on the state, but we assume it satisfies some reasonableness conditions, which echo the axioms for "entailment" in information systems.

1. $\frac{\phi \in t}{t \vdash \phi}$
2. $\frac{\forall \phi \in t_2 (t_1 \vdash \phi), \quad t_2 \vdash \psi}{t_1 \vdash \psi}$

Unlike information systems, no distinguished proposition is assumed to play the part of the "least informative" Δ , nor do we assume any special classification of types or sets of properties. On the other hand, nothing prevents there being any number of types having types, or sets of properties, as their examples, and satisfying closure under subsetting and entailment, thereby carving any number of information systems (and corresponding domains) out of the epistemic state.

The most basic relationship among types is *structural subtyping* (" \sqsubseteq "), whereby $t_1 \sqsubseteq t_2$ iff $t_2 \subseteq t_1$. Intuitively, anything which exhibits all of the properties in t_1 exhibits all subsets of those properties. Given its definition, the properties of structural subtyping are obvious.

⁵An interesting question is. should an agent decide that two types *are* the same, what can it do to remove the duplicate individual? A mechanistic answer is that it can consolidate its information around one "node", and attribute the other as being equal to the first. Future references to the duplicate node will be redirected to the chosen representative, and at some point the duplicate will decay (say, when all references have been redirected, or when there is sufficiently little activity to support its continued existence).

A more interesting relation is *deductive subtyping* (" \sqsubseteq_{\vdash} "), defined as follows: $t_1 \sqsubseteq_{\vdash} t_2$ iff $\forall \phi \in t_2 (t_1 \vdash \phi)$. Now, take the set *Prop* of all properties in the epistemic state, and form its powerset; call it *type*. Define deductive equivalence in the obvious way, and take the quotient $type/\equiv_{\vdash}$. Then, \sqsubseteq_{\vdash} is simply structural subtyping on $type/\equiv_{\vdash}$.

Some interesting structure arises if the property of being deductively closed exists in the state (and, of course, satisfies the obvious semantic constraint: that $\Gamma \xrightarrow{e} \vdash^b dc(t)$ iff $(t \vdash \phi) \Rightarrow (\phi \in t)$). In that case, $type/\equiv_{\vdash}$ contains a type *ty* definable as the deductive closure of the property of being a deductively closed set of properties. Clearly, *ty* is an example of itself, i.e., $ty:ty$.

Considered as a relation on $type/\equiv_{\vdash}$, the deductive subtype relation supplies the morphisms of a bi-Cartesian closed skeletal category, in which the equivalence class of the deductive closure of the empty set is initial, and the equivalence class of *Prop* is terminal. An even more interesting structure — a topos — emerges if we take "restricted functions" between types as our morphisms. The required definitions and proofs are not difficult, but they would take us too far afield here.

7 Platonic Types: closed sets

Having decided that types are supposed to express general properties, let us say a bit more about what that entails. To begin with, we need some notation. Let $x:t$ signify that x is an example of the type t ; i.e., roughly speaking, it has the properties expressed by t . The notation $t_1 \preceq t_2$ will signify that t_1 is a subtype of t_2 .

Although I haven't specified exactly what I mean by these notations, I can state some minimal properties I expect them to have. One is that the two notions be related by the rule of *subsumption*: if x is a y and y is a subtype of z , then x is a z . Another is that the subtype relation is transitive. Formally,

$$\frac{x:y \quad y \preceq z}{x:z} \qquad \frac{x \preceq y \quad y \preceq z}{x \preceq z}$$

remark

It is instructive to reflect on what has been done to this point. I have introduced some uninterpreted symbols, and told you some rules that apply to them. I have not, and need never, give them extension: they are, and may remain, open to interpretation. Moreover, this is a perfectly reasonable thing to do, and is completely within the bounds of our language, regardless of whether any interpretation is possible at all! If it turns out that all of the descriptions I've given are totally inconsistent, and so could never properly describe anything, I am not thereby prevented from presenting them and reasoning about them. In fact, were it impossible to do that, we could never make inconsistent utterances, much less prove them inconsistent!

The point is that description is prior to interpretation, and can stand on its own. The fact

that I have attributed certain properties to $:$ and \preceq , for example that they are relations, conveys that certain inferences can be applied to them, *viz.* those that you associate with the description "relation". None of this requires that relations "exist", or that we have the same descriptions, though it is my hope that our shared background will enable you to follow the ensuing argument.

kramer

Let us turn now to some common sense examples. To begin with, there should be a type of all things, which we shall designate by " \Diamond " (box), such that the assertion " $x:\Diamond$ " conveys no nontrivial information about x . Informally, " \Diamond " corresponds to the English word "anything".

And so we come to our first real assertion, namely, that \Diamond is a type, notated thus: $\Diamond:type$. It follows from this that \Diamond has all of the properties of a type. But what is a type? More directly, is *type* an example of *type*, or does it have some conflicting properties? Let us defer the question as too difficult to answer at present, and assert instead that *type* is certainly an example of *some* type, which for lack of a better name we will call "*metatype*". Formally, $type:metatype$.

But what is a *metatype*? We *could* defer the question again, claiming that $metatype:metametype$, and continue in that fashion indefinitely. The alternative is to cut off the infinite regress at some point t , say by asserting $t:t$. Let us explore the consequences of the latter course, tentatively asserting $metatype:metatype$. This amounts to saying that, in whatever sense *type* is a "type of types", *metatype* is also a "type of types".

What about subtypes? Clearly, any example of *type* is something, *i.e.*, is an example of \Diamond , so it should be the case that $type \preceq \Diamond$. Similarly, metatypes seem to be specialized types, though we have been carefully noncommittal to this point about what meaning we should attach to the informal notion of a "type of types". Suppose we take it at face value. Then, every example of *metatype* is also an example of *type*, and hence $metatype \preceq type$. The foregoing is summarized in the following "axioms".

A1) $\Diamond:type$

A2) $type:metatype$

A3) $metatype:metatype$

A4) $type \preceq \Diamond$

A5) $metatype \preceq type$

The problem is to explain these notions formally in such a way that they both harmonize with our intuitions and give a logically consistent interpretation to the relations stated above. That is, we want to *make sense* of them. Here are some of the consequences which, according to our rules, follow from the "axioms" A1-A5.

C1) $\Diamond : \Diamond$

C2) $type : type$

C3) $type : \Diamond$

C4) $metatype : type$

C5) $metatype : \Diamond$

C6) $metatype \preceq \Diamond$

Proof: C1 follows from A1 and A4 by subsumption. C2 follows from A2 and A5 by subsumption. C3 follows from C2 and A4 by subsumption. C4 follows from A3 and A5 by subsumption. C5 follows from C4 and A4 by subsumption. Finally, C6 follows from A4 and A5 by transitivity. \square

Remark: It does not follow that $\Diamond : metatype$, nor is the negation of this statement provable. Hence the axioms admit models in which either \Diamond is or is not a *metatype*, and in fact models of both kinds exist.

Now, a number of familiar difficulties arise if we take types to be sets, with “:” being set membership, and “ \preceq ” being the subset relation. For one thing, we would have situations where two sets belong to each other, as in $\Diamond : type$ and $type : \Diamond$. Intuitively, *type* contains a copy of \Diamond , which contains a copy of *type*, which contains a copy of \Diamond , and so forth *ad infinitum* – not your garden-variety set. (Technically, this violates the *axiom of regularity*,⁶ which is usually included in any axiomatization of set theory to disallow a number of well-known anomalies.) To make the problem concrete, the reader is challenged to exhibit three sets that satisfy the relations A1-A5 and C1-C6.

Note that without regularity or some other means of restricting the principle of comprehension,⁷ we could apparently form the Russell type $R = \{x | \neg(x : x)\}$. In set theory, membership in a set is equivalent to satisfaction of the characteristic predicate of the set, so we arrive at the contradiction that $R : R \Leftrightarrow \neg(R : R)$.

⁶The axiom of regularity disallows infinite regress in the formation of sets. In short, $\dots\{\{\{\}\}\}\dots$ is allowed, but not $\{\{\{\cdot\}\}\}$. Regularity is also known as the axiom of foundation (von Neumann’s *Fundierung*), to the effect that membership is a well-founded relation. Peter Aczel’s hypersets hinge on a rejection of the axiom of foundation. Indeed, hypersets provide one solution to the current puzzle, but not the one I want.

⁷The principle of comprehension states that for any predicate ϕ there exists a *set* consisting of those things which satisfy ϕ , usually written $\{x | \phi\}$. Russell’s original theory of types made peace with the principle of comprehension by imposing an order on predicates, and restricting the membership predicate \in so that the type of the left operand be strictly less than the type of the right. Thus ruled out are locutions such as $x \in x$, thereby making the troublesome Russell class indefinable. This avenue is, however, closed to us because it would banish axiom A3, along with C1 and C2.

Because the extensional interpretation of types won't do, we adopt a subtler interpretation, in which types are sets of unary predicates (equivalently, "properties") closed under entailment. That is, if $t \models \phi$ (ϕ a unary predicate) then $\phi \in t$. In order to specify types finitely, we adopt the notation X^\models for the closure of X under \models .

For example, let us define *oddprime* to be the type $\mathcal{O}x\{x \text{ is prime, } x \text{ is odd}\}^\models$.⁸ *oddprime* includes the properties " $\mathcal{O}x(x+1 \text{ is even})$ ", " $\mathcal{O}x(x > 2)$ ", and " $\mathcal{O}x(\forall y, m. y < x \wedge m|x \wedge m|y \Rightarrow m = 1)$ " along with infinitely many other consequences of the two properties originally specified.

The subtype relation is defined as follows: $x \preceq y$ if and only if $\forall \phi \in y. x \models \phi$. That is, subtypes specialize their supertypes. An equivalent, if initially counterintuitive, characterization is that $x \preceq y$ whenever $y \subseteq x$ (note the reversal!).

Some examples: $\mathcal{O}x\{x \text{ is prime, } x \text{ is odd, } x^2 < 10\}^\models$ is a subtype of *oddprime*. Similarly, $\mathcal{O}x\{x \text{ is odd}\}^\models$ is a supertype of *oddprime*.

To a first approximation, an individual (description) is an example of a given type if it satisfies all of the properties of that type.⁹ Formally, $x:t$ if and only if $\forall \phi \in t. \phi(x)$. Actually, we need to refine this definition somewhat, but let us adopt it for the moment, to show what is right and wrong about it.

With these definitions in hand, we propose the following equivalences between descriptions as a possible "model" of the symbols introduced earlier.¹⁰

$$\begin{aligned}\diamond &= \emptyset^\models \\ \text{type} &= \mathcal{O}t\{t \text{ is an } \models\text{-closed set of properties}\}^\models \\ \text{metatype} &= (\text{type} \cup \mathcal{O}x\{\forall y: x. y: \text{type}\})^\models\end{aligned}$$

It is easy to check the properties A1-A5 and C1-C6 against these definitions. A1 states that \emptyset^\models is an \models -closed set of unary predicates; A2 and A3 state that $\mathcal{O}t\{t \text{ is an } \models\text{-closed set of properties}\}^\models$ and $(\text{type} \cup \{\forall y: x. y: \text{type}\})^\models$ are \models -closed sets of properties specifying \models -closed sets of properties; and so forth, all of which statements are obviously true.

⁸The notation $\mathcal{O}x$ is meant to identify the variable over which the predicates in the immediately following type specification range. In this regard it serves the same purpose as the bound variable in a set specification $\{x|\phi\}$, but the two notations mean quite different things!

⁹I use the word "example", instead of "member", because the members of a type are properties. For example, " $\mathcal{O}x(x > 2)$ " is a member of *oddprime*; 3 is not. What 3 is is an example of *oddprime*.

¹⁰This is, in fact, what becomes of model theory here. certain descriptions are made to refer to other descriptions. Note that reference to anything other than a description is impossible, because reference is itself a property, i.e. a relation from descriptions to descriptions. This accounts for the model-theoretic ontology of formalism, and supports Brouwer's contention that mathematical objects are mental constructions. These mental constructions, or descriptions, can be used in turn to "model" reality, but reality is not like them.

The interpretation of *metatype* is motivated by the intuition that a metatype is a type of types. Note that, with this interpretation, \Diamond is not a *metatype* provided that something is not a type. In Prism, we fully expect there to be things which are not types, though it is possible to construct systems consistent with everything that has been said so far in which everything is a type. (Think of reflexive domains defined in terms of information systems!)

Although the proposed interpretation makes the axioms and their consequences seem sensible, it doesn't prevent the paradoxes. We can still define an analogy to the Russell type $R = \mathcal{O}t\{\neg(t:t)\}^\models$, and draw the usual contradiction. The problem, alluded to earlier, lies with our notion of examplehood.

The source of the problem is with the requirements for examplehood, which are too loose. Before something can be subject to the general reasoning applicable to a type, it must already exist as an example of some more specific type. That is, it must be an individual description already, with properties that entail those of the more general type. This captures the important part of regularity, which is that types (sets) be populated synthetically. Thus we arrive at an analytic theory of synthetically populated types.

Technically, my proposal is that satisfaction of the properties of a type be necessary, but not sufficient, to establish examplehood. In symbols,

$$\frac{x:y}{\forall \phi \in y. \phi(x)}$$

but the converse of this rule does *not* hold. Instead, we have the following axiom of *unit examplehood*, which explains how we come to have any examples at all.

$$a: \mathcal{O}x\{x = a\}^\models$$

Using these rules, one can prove that the assumption $R: R$ leads to a contradiction, but the assumption that $\neg(R: R)$ leads only to the conclusion that $\forall \phi \in R. \phi(R)$, which is insufficient to establish the contradictory $R: R$.¹¹ So if we are Platonists we can conclude, quite comfortably, that R is not an example of itself. If intuitionists we be, we can remain comfortably agnostic.

The metamathematical status of the foregoing argument is crucial. I did *not* claim that $\neg(R: R)$ is a theorem within the system; I only claimed that R not being an example of itself is consistent with what *can* be proved in the system. If we could prove $\neg(R: R)$ in the system, then we would have that $\mathcal{O}x\{x = R\}^\models \models \neg(x: x)$, and hence that $\mathcal{O}x\{x = R\}^\models \not\preceq R$, which would enable us to conclude the contradictory $R: R$.

To be precise, the metamathematical assertion is $\mathcal{O}x\{x = R\}^\models \not\models \neg(x: x)$. If we could internalize this fact, we would have $\mathcal{O}x\{x = R\}^\models \models \neg\neg(x: x)$. If this is provable, we can still avoid contradiction if \neg is not an involution (which, of course, is the case in intuitionistic logic). This indicates some of the boundaries within which we will have to play in formulating specific inference rules for Prism.

¹¹By unit examplehood, $R: \mathcal{O}x\{x = R\}^\models$, but without further rules $\mathcal{O}x\{x = R\}^\models \not\preceq R$.

In closing this section, let us examine the question of whether the proposed restriction (a kind of weak extensionality) excludes any examples. That is, is it possible that the rules we've proposed are insufficient to prove the examplehood of some example, leading to orphans? The answer is "no", as the following argument shows.

If a is an example of t , then it satisfies all of the properties of t . It follows that all of the properties of t are included in the singleton type $\{x = a\}^{\models}$. By unit examplehood and subsumption, $a : t$. End of story.

8 Prism Categories

A Prism category, or Pcategory, ¹² is a slight generalization of the usual notion of category [Mac71]. It is essentially what MacLane calls a "metacategory". As an aside, here is a description of the type Pcategory, which may be illuminating to those who are wondering how complex types can be defined as sets of properties.

$$\begin{aligned}
 \text{Pcategory} \triangleq \bigcirc c \{ & Ob_c, Ar_c : \Diamond, \\
 & dom_c, cod_c : Ar_c \rightarrow Ob_c, \\
 & 1_c^c : Ob_c \rightarrow Ar_c, \\
 & dom_c 1_a^c = a, \\
 & cod_c 1_a^c = a, \\
 & \forall f, g, h : Ar_c < f \circ g : Ar_c \Leftrightarrow cod_c f = dom_c g, \\
 & \quad f \circ (g \circ h) : Ar_c \Rightarrow f \circ (g \circ h) = (f \circ g) \circ h, \\
 & \quad f \circ 1_a^c : Ar_c \Rightarrow f \circ 1_a^c = f, \\
 & \quad 1_a^c \circ g : Ar_c \Rightarrow 1_a^c \circ g = g > \}^{\models}
 \end{aligned}$$

An ordinary category is just a Pcategory in which the Arrows and Objects are Sets.

$$\text{Category} \triangleq \bigcirc c \{ c : \text{Pcategory}, \\
 Ob_c, Ar_c : \text{Set} \}^{\models}$$

In a similar manner, we can lift the set-theoretic restrictions usually imposed on any mathematical structure. For example, a *monoid* is a *set* equipped with an associative binary multiplication and an identity; a *Pmonoid* is *anything* equipped with an associative binary multiplication and an identity. We do this because sets in Prism do not have the primacy accorded to them in classical mathematics. ¹³

¹²The initial "P" is silent, as in "Pneumonia" and "Prangler".

¹³The key to the current approach is that types are simply specifications, not the extensions of those specifications. In Prism, " $x.t$ " is a belief (assertion of knowledge) about x , that it is an example of the specification (type)

We intend that operations in Prism can be applied to objects of any type. Informally, this means that when something is supplied as an argument, something (perhaps divergence, or an exception) results. A bit more formally, $f: \Diamond \rightarrow \Diamond$ for any f . Because there are no restrictions on the domain of an operation, everything is composable with everything else; *i.e.*, $f \circ g$ is always defined.

The design of Prism also stipulates the existence of an identity on \Diamond . Intuitively, you can leave anything alone. Thus, \Diamond is a Pmonoid.

A *Pfunction* is a \Diamond -automorphism, *i.e.*, $Pfunction \triangleq \bigcirc f\{\forall x: \Diamond. fx: \Diamond\}^F$. A *restricted Pfunction* is an equivalence type of Pfunctions, namely those that are indistinguishable on a specified domain t , notated $f[t]$. Application of restricted Pfunctions is defined in the obvious way, so that $f[t](x) = f(x)$ whenever $x: t$, and is undefined otherwise.

The codomain of an operation can be specified using the notation $\rightarrow t$. That is, $f \rightarrow t$ specifies that the result type of f is t . Combining these notations, we can write $f[t_1] \rightarrow t_2$ to indicate that f takes t_1 's to t_2 's.

There is no overloading in Prism, so a given name can refer to at most one Pfunction in any context. However, a Pfunction can be specified piecewise, using restriction. For example,

$$\begin{aligned} f[x: integer; y: integer] \rightarrow Boolean &\mapsto x < y \\ f[x: integer] \rightarrow color &\mapsto < 1 \mapsto red; 2 \mapsto yellow; 3 \mapsto blue; \dots > \end{aligned}$$

Ada-style overloading would treat these as two different functions with the same name, and would allow the definition of a third function named " f " with domain *integer* and codomain *Ascii*. The expression $f(3)$ would then be ambiguous unless its context dictated an expected result type of either *color* or *Ascii*, which would serve to disambiguate the reference of " f ". In Prism, such ambiguity is impossible, because all we have are distributed specifications of segments of a single function, which cannot have conflicting results on overlapping parts of the domain.

In Prism, $t_1 \rightarrow t_2$ is the type of Pfunctions from t_1 to t_2 , defined as follows. $t_1 \rightarrow t_2 \triangleq \bigcirc x\{\forall a: t_1. x(a): t_2\}^F$. For comparison, the corresponding type of restricted Pfunctions is

$$[t_1 \rightarrow t_2] \triangleq \bigcirc x\{x: type, \forall f, g: t_1 \rightarrow t_2. f: x \wedge g: x \Rightarrow (\forall y: t_1. fy = gy)\}^F$$

As with most objects, Pfunctions can be examples of many types. For example, the following assertions are true of the f partially described above.

$f: integer \times integer \rightarrow Boolean$

t . This says nothing about the collection of all such examples, which (if it exists) is an example of the type $\bigcirc x\{x \text{ is a set}, \forall y. y \in x \Leftrightarrow y. t\}^F$. According to this view, we never have to contemplate or account for such things as the extension of \Diamond , though it is easy to show in the standard way that if it exists it is neither a set, nor a class, nor even a *meta*ⁿ class.

$f: (integer \times integer) \vee integer \rightarrow (Boolean \vee color)$
 $f: integer \rightarrow \Diamond$

Note, however, that it does not follow from $f: t_1 \rightarrow t_2$ that $f: [t_1 \rightarrow t_2]$! The correct conclusion is that $f[t_1]: [t_1 \rightarrow t_2]$.

9 Type Pcategories

There are a number of ways to make Prism types into a Pcategory. One is the *logical* Pcategory, $type_{\preceq}$, in which the arrows are given by the subtype relation, \preceq . It is bi-Cartesian closed, with products coproducts, and exponents given by the following.

$$\begin{aligned}
 t_1 \wedge t_2 &\triangleq \max_t & t &\preceq t_1, t_2 \\
 t_1 \vee t_2 &\triangleq \min_t & t_1, t_2 &\preceq t \\
 t_1 \Rightarrow t_2 &\triangleq \max_t & t \wedge t_1 &\preceq t_2
 \end{aligned}$$

In words, \wedge forms the greatest common subtype (gcs), \vee forms the least common supertype (lcs), and \Rightarrow forms the least sufficient constraint on t_1 required to verify t_2 . The terminal object is \Diamond , and the inconsistent type \ast is initial.

Another type Pcategory is the restricted Pfunction Pcategory, $type_{\rightarrow}$, in which the arrows are the restricted Pfunctions. This Pcategory is also bi-Cartesian closed, with products, coproducts and exponents as follows.

$$\begin{aligned}
 t_1 \times t_2 &\triangleq \mathcal{O}x\{\pi_1 x: t_1, \pi_2 x: t_2\}^{\models} \\
 t_1 \uplus t_2 &\triangleq \mathcal{O}x\{(\exists y_1: t_1. \iota_1 y_1 = x) \vee (\exists y_2: t_2. \iota_2 y_2 = x)\}^{\models} \\
 [t_1 \rightarrow t_2] &\triangleq \mathcal{O}x\{x: type, \forall f, g: t_1 \rightarrow t_2. f: x \wedge g: x \Rightarrow (\forall y: t_1. fy = gy)\}^{\models}
 \end{aligned}$$

These are analogous to the usual Cartesian product, coproduct, and function space types on Set. Here, the type with no examples, \ast , is initial, and any singleton type is terminal.

Incidentally, the axioms for projections and the like, which play an important part in determining what is entailed by the properties cited in a type specification, are usually specified separately. For example,

$$\forall f: t_3 \rightarrow t_1, g: t_3 \rightarrow t_2. \exists! h: t_3 \rightarrow t_1 \times t_2. h \circ \pi_1 = f \wedge h \circ \pi_2 = g.$$

Of course, it is a good idea to specify universal concepts like “product” as a type, and confer the universal properties of products on all instances by inheritance. An appropriately abstract definition using only the rather primitive \models -closure mechanism is rather complicated, but a good start is

$$\begin{aligned} \text{product}(c: Pcategory) \triangleq & \mathcal{O} p \{ \exists A, B : Ob_c. \pi_1: p \rightarrow A \wedge \pi_2: p \rightarrow B \wedge \\ & \forall C: Ob_c. \forall f: C \rightarrow A, g: C \rightarrow B. \exists! h: C \rightarrow p. \\ & h \circ \pi_1 = f \wedge h \circ \pi_2 = g \}^{\models}. \end{aligned}$$

We can then assert $\forall t_1, t_2: Ob_{type_}. t_1 \times t_2: \text{product}(type_)$.

Fact 9.1 $type_{\leq}$ is a subcategory of $type_{=}$, under the obvious identification of arrows in $type_{\leq}$ with the corresponding inclusion Pfunctions.

Fact 9.2 Every arrow in $type_{\leq}$ is a bimorphism (i.e., epi and mono).

Fact 9.3 In $type_{\leq}$, sections = retractions = isomorphisms = identities.

Fact 9.4 $type_{\leq}$ is not a topos.

Proof Counterexamples abound. \square

Theorem 9.5 $type_{=}$ is a topos. That is, for each type A there is a power type $\mathcal{P}A$ and a membership relation $\iota: \in_A \rightarrow A \times \mathcal{P}A$ such that, for any relation $\rho: R \rightarrow A \times B$ there is a pair of restricted Pfunctions $\beta, \hat{\rho}$ such that the following diagram is a pullback.

$$\begin{array}{ccc} R & \xrightarrow{\hat{\rho}} & \in_A \\ \downarrow \rho & & \downarrow \iota \\ A \times B & \xrightarrow{1 \times \beta} & A \times \mathcal{P}A \end{array}$$

Proof The required players are defined as follows.

$$\begin{aligned} \mathcal{P}A & \triangleq \mathcal{O}x \{x: type, \forall y: x. y: A\}^{\models} \\ \in_A & \triangleq \mathcal{O}x \{x: A \times \mathcal{P}A, \pi_1 x: \pi_2 x\}^{\models} \\ \iota & \triangleq 1[\in_A] \\ \beta & \triangleq b \mapsto \mathcal{O}x \{ \exists r: R. \rho r = \langle x, b \rangle \}^{\models} \\ \hat{\rho} & \triangleq (\rho \circ (1 \times \beta)) \end{aligned}$$

Given $\sigma: S \rightarrow \in_A$, $\hat{\sigma}: S \rightarrow A \times B$ making the square commute, the required unique restricted Pfunction $u: S \rightarrow R$ is given by the correspondence $s \mapsto !r: R. \rho r = \sigma s$, where existence of r is guaranteed by $\hat{\rho}$ and uniqueness by ρ being monic. \square

The detailed calculations involved in the foregoing proof are tedious, and not too interesting, but it may be helpful to sketch them out. From $\hat{\sigma}s: \in_A$ infer that $\pi_1(\hat{\sigma}): \pi_2(\hat{\sigma})$. By the definition of β and the assumption of commutativity, $\pi_2(\hat{\sigma}s) = \mathcal{O}x\{\exists r: R. \rho r = \langle x, b \rangle\}^F$, whence $\exists r: R. \rho r = \langle \pi_1(\hat{\sigma}s), \pi_2(\hat{\sigma}s) \rangle$. But again, commutativity forces $\pi_1(\hat{\sigma}s) = \pi_1(\sigma s)$, and so $\exists r: R. \rho r = \sigma s$. A similar "diagram chase" can be used to show the uniqueness of r , though with a few lemmas the whole thing can be done in a more purely categorical style.

An exhaustive formal study of the Prism type system is well beyond the scope of this brief overview, but the few elementary results obtained here are sufficient to give a general feel for its structure, and to reassure us that its semantic base does not force anything terribly unnatural on us.

10 Abstractness

This section points out the lack of abstractness in the representation-based "abstract data type" facilities found in current programming languages, and indicates how Platonic types can be used to remedy that defect. The basic point is that *what* is specified should be divorced from the form and especially the details of specifications (such as choice of names, order of presentation, or choice of axioms). One thing that this permits is the definition of multiple representations for the same type. Another is the ability to recognize subtype relations that don't depend on mechanistic inheritance; for example, the integers are a group under both addition and multiplication, but this fact does not (and should not) depend on any *construction* of the integers explicitly involving the type of groups.

11 Sets

Set is an interesting example of a Prism type, because it provides insight into the interplay between intension and extension. Of particular note is the axiom of replacement, which allows examples of types to be collected into sets by Pfunctions, though there may be no Pfunction which collects all examples of a given type. Types that *can* be completely collected are called *sets* in the category-theoretic sense.

12 Epistemology: monotonic revision of Platonic types

In this section a simple model of taxonomic learning is presented, in which examples and counterexamples of taxa induce monotonic refinement of Platonic type lattices. Examples generalize taxa to least upper bounds, and counterexamples split taxa at greatest lower bounds. "Ad-hoc"

collections are modelled as internal coproducts of taxa. Something else to consider, somewhere, is indeterminates. For example, the type "heap" refers to an undetermined type lying between (more) determined types of "strong heaps" and "strong non-heaps". Epistemology comes into play in regulating increases in the definiteness of the description "heap". That is, e.g., when something that was not previously acknowledged to be a strong heap is declared to be a heap, the type of strong heaps shifts to the least common supertype of strong heap and the new heap, or something like that, thereby constraining "heap" further.

13 Intensionality: partial types, partial deduction, and indexicality

The basic problem with deductively closed sets (Platonic types) is that all inconsistent sets are identified. Distinct inconsistent types may be obtained if deduction is allowed to be incomplete (i.e., if there is incomplete information about entailment). It is also necessary, however, to admit non-closed sets as types, as shown by the greatest prime example (because it doesn't become unintelligible as soon as the conclusion is reached). A third intensional issue is indexicality and how to handle it. The solution to these problems is basically to distinguish between specifications and types, with the valuation of specifications in "context space" being taken in the domain of Platonic types.

14 Real World Semantics

Prism represents a major paradigm shift in language design, because its domain is the real world, not just formal abstractions of real world situations (though of course formal abstractions are an important part of the real world, and are therefore in the domain of Prism). In doing so, Prism challenges a pervasive attitude: that computer languages deal only in formal abstractions, and it is the job of some human conjurer to do the abstracting. We see this attitude as a byproduct of a linguistic theory developed in the narrow context of formal languages, in particular formal logic, mathematics, and computation. In order to manage the long-term operation and maintenance of software systems, however, it is necessary to capture and manipulate information relating formal artifacts, such as algorithms and specifications, to their purposes and the real world situations they represent. The crux of the problem is to maintain the distinction between the machine's internal *description* of a real world situation (which is a formal artifact) and the situation it describes (which is not). We accomplish this by a semantics in which descriptions are partial models of real world situations. Moreover, this semantics allows descriptions to be composed of meaningful parts which are nonetheless meaningless (or fictional, or ...) in combination.

15 Open Problems

The purpose of this section is to point out the major deficiencies of Platonic type theory and epistemology, and to indicate the direction of continuing research.

References

- [Bar89] J. Barwise. *The Situation in Logic*, volume 17 of *CSLI Lecture Notes*. Center for the Study of Language and Information, Stanford, 1989.
- [Gir86] J. Y. Girard. *Linear Logic*. Technical report, Université Paris VII, 1986.
- [Mac71] S. MacLane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1971.
- [Sco82] D. Scott. Domains for Denotational Semantics. In *Proceedings of ICALP '82 (LNCS 140)*, pages 577–613. Springer-Verlag, 1982.
- [Smy85] M. Smyth. Finite Approximation of Spaces. In *Category Theory and Computer Programming, LNCS 240*, pages 225–241. Springer-Verlag, 1985.
- [SS88] N. Salmon and S. Soames, editors. *Propositions and Attitudes*. Oxford University Press, 1988.

Conclusions

for

Languages Beyond Ada and Lisp

Incremental Systems Corporation
319 South Craig Street
Pittsburgh, Pennsylvania

Unnatural Languages

David Mundie

1991 Aug 2

Introduction

Programming languages have as twin progenitors the formal languages of mathematics and the natural languages of everyday life. If Prism is to achieve its goals of redefining the look and feel of programming languages, and of defining a language which will not be made obsolete by the next innovation in compiler technology, it behooves us to look carefully at what those two progenitors have contributed. In this report we examine the natural language side of things.

Our thesis is that programming languages are less satisfying than they could be, not because they are formal (non-natural) languages, but rather because they are *unnatural* languages, lacking many of the characteristics which make natural languages comfortable. Using techniques developed in the computational linguistics community during the last twenty years, we feel the time is ripe to incorporate some of those characteristics in a formal language.

Let us state clearly and unambiguously at the outset that we are in no way shape or form proposing a natural-language understanding system. It is clear that the state of the art is hopelessly far from such a system¹, and there are theoretical grounds for suspecting that it will remain so². This does not imply, however, that the baby need

¹ See for example Hans Kamp, "".

² For example those advanced by Winograd and Flores.

Unnatural Languages

-2-

be thrown out with the bath water. We feel that many language designers, bolstered by the almost comforting fact that they could not implement English, have felt licensed to impose unnecessarily awkward syntactic rules and restrictions on their language. For example, just because anaphora in the most general case is difficult, it does not follow that a limited, easily-analyzed form of anaphora is not useful in programming languages.

Two objections. There are two objections to this whole approach which we would like to address before beginning. The first is based on what for the lack of a better term we shall call linguistic parallax: the confusion that can arise when old words are used in new ways. Whenever a name for one object is applied to a new object with different properties, the language user can become confused by thinking that the new object has some of the properties of the old when in fact it does not. For example, the user who does not realize that in programming languages the symbol '+' is used for modular-arithmetic '+', and therefore ignores overflow, is a victim of linguistic parallax.

The argument against using natural-language constructs in programming languages, then, says that there will inevitably be so much parallax that the resulting confusion will outweigh the advantages of familiarity. Better to introduce completely unfamiliar features, this theory goes; that way no confusion arises—the resulting system will be less learnable, but no less usable.

We make the following observations about this argument. (1) Linguistic parallax is an endemic fact of life, in both natural and programming languages. It occurs in natural languages whenever an old term is used a new way, and that is very frequently indeed. In programming languages vast majority of linguistic features have been drawn from either natural language or from mathematics, and have almost never been precisely equivalent to their counterparts. The art of language design is to try to capture the "essential" features of the old usage, so that the new use seems as natural as possible. This is not always an easy task, but it is one that cannot be avoided. In short, we are not denying that confusion bred from parallax can be a problem, but we feel the assessment cannot be made globally, in advance; rather, each case must be examined separately. (2)

Unnatural Languages

-3-

Abandoning familiar notations to avoid parallax is a dubious strategy. One could, obviously, substitute '§' for '+' in programming languages. Would the language user thereby be spared the problems of parallax? Probably not. For how would he interpret '§'? It seems clear that he would say to himself that "§ is just +, except that it overflows when the result is greater than 2^{16} ." But this is exactly the same sequence the user of traditional languages goes through in learning about the '+' of those languages. (3) Historically, accusations of parallax have often been made by entrenched partisans of existing technology. Early claims that users would be baffled by the ways in which the computer "desktop" differed from a real desktop have not been born out by experience. (5) Learnability is a major concern.

The second argument against making Prism more natural is that the ordinary linguistic habits of the programmer will be corrupted by interference from Prism. We do not dispute the phenomenon: students of French do end up speaking Franglais; parents of small children do lapse into baby talk. What we dispute is the gravity of the situation. It is no argument against learning French or having children. Multilingualism is a natural human state, and there is no documented case of an English expression being killed off by baby talk. We welcome Prismish, and would regard it as a measure of our success.

Related Research. It is surprising how exiguous the literature on this sort of approach appears to be. Most treatments of formal languages begin with a terrifying glance at the ambiguities of natural language and then quickly settle down to a comfortable consideration of the joys of context freedom. A subset of the computational linguistics community has explored the construction of practical natural-language understanding systems, but the nature of their endeavour makes the approach we are taking irrelevant: if your goal is to analyze an enormous corpus of German law reports³, the opportunity of writing those reports in a formal language is not open to you. One might think the most closely related research is that in the field of natural language interfaces to databases⁴ but the emphasis there is on making the languages as English-like as

³ See Hans Kamp et al., "The LEX System."

⁴ References.

Unnatural Languages

-4-

possible, rather than using the features of natural languages in a formal way; and such systems have been for the most part special-purpose query languages rather than general-purpose programming languages.

The original inspiration for this project came from Montague semantics⁵. Over fifteen years ago Montague showed how a substantial subset of English, including the notoriously difficult features of quantification and intensional contexts, could be treated as a formal language, using a semantic analyzer proceeding in lockstep with a syntactic analyzer. His system took English sentences and generated first-order predicate calculus equivalents. His system has been extended in a variety of ways by a plethora of research efforts which both incorporate into it ever greater portions of English (e.g. modal operators and general quantifiers⁶) and improve the mechanism itself.

Our original intent, then, was to see whether a Montague grammar could be used as a front end to a predicate-calculus-based formal system. The system we ended up with, however, takes quite a different approach: instead of translating everything into the predicate calculus, we instead use primitives which correspond much more closely to those of natural language itself. For instance, instead of translating the variable-free quantification of natural language into the variable-ridden quantification of the predicate calculus, we instead treat variable-free quantifiers as first-class citizens of our system.

We would be hesitant to suggest our approach as the basis for a practical system without three supporting technologies that have been the focus of intense research efforts in the last five years: unification grammars, intensional logic, and discourse semantics. Unification grammars and their generalizations at last promise a powerful and flexible formalism for expressing the constraints on language which are an essential part of its semantics (see Incremental Systems Technical Report 89-??? for a discussion of unification-based semantics), and for describing phenomena such as non-contiguous conjunction⁷, which have resisted treatment by

⁵ References.

⁶ References.

less powerful formalisms. Recent developments in intensional logic⁸ hold forth the hope that logic may at last be able to formalize the Husserlian noesis so as to capture the directedness of natural language. Finally, discourse structure has quite recently been the focus of much research⁹; although no adequate general theory exists, there is room to hope that a framework adequate for a reasonable discourse structure for a formal language can be developed.

Overview. The remainder of this report is divided into three sections. In "Properties of Natural Language" we summarize those features of natural language which we feel have been left out of programming language, and which it might be worthwhile investigating as candidates for inclusion in Prism. In "A Natural-Language View of the Prism Type System" we approach the topic from the other end, arguing that the property-based type system of Prism can be thought of as an attempt to come closer to the type system used by natural language. A separate document (Technical Report 89-???) describes the results of our applying this approach to the design of a grammar for Prism.

Properties of Natural Language

Consider the following thought experiment. Suppose at the end of an introductory programming course the students were asked to say which of the following fragments was from a natural language and which from a formal language:

(a) The long sighs of autumnal violins break my heart
with a monotonous languor.

(b) `pzd & sqr (@j = ^45) & \ssp$$`

There is little doubt that the student would identify (a) as the

⁷ References.

⁸ References.

⁹ References.

Unnatural Languages

-6-

natural-language fragment. Yet (a) conforms to a simple grammar that can be rigorously described in a few lines of BNF, while fragment (b) consists of an almost random sequence of tokens, written down haphazardly.

This example is meant to suggest that the differences between programming languages and natural languages have little to do with their grammars. It is as though the failure to achieve full natural-language processing capability has been used as an excuse to introduce into programming languages a host of arcane features which may or may not have independent merit, but which surely distance them from our native tongues. "Since I cannot let him say what he means, I will make him start all string variable names with dollar signs."¹⁰

Both natural and programming languages can be described using the same grammatical techniques.

Noun ::= Adjective Noun

Term := UnaryOperator Term

Why then do we end up in a situation where a simple request such as:

Choose thirteen cards from a standard deck. Sort them in descending order, then print them on the Laserwriter.

comes out as

```
print(sort(chooseCards(std, 13, randomOrder),
descending, "", 3), true, lw, -1)
```

or something similar? This is the question we shall ponder here.

Freedom from Overspecification. One of the goals of Prism is to design a language that allows programmers to specify only as many

¹⁰ The example is of course *not* meant to suggest that natural language is the preferred notation for all tasks. "Add b to c and store the value in a" is arguably more opaque than "a := b + c;" we advocate a wide spectrum of notations.

Unnatural Languages

-7-

details of his problem as are actually relevant to finding a solution. As a trivial example, consider the problem of a number greater than five. Traditional languages will not let you simply write "x is > 5" or "x > 5" at all. Instead you must overspecify the situation by choosing some *particular* number and assigning it to x: "x := 99" or "x := 9999" or "x := 32768" or the like¹¹. The compiler is consequently unable to tell whether this particular value was necessary to your solution or not.

It is difficult to overestimate the price we pay for this overspecification in programming languages. (1) It impedes the generation of optimized code, since the compiler cannot tell what things can be optimized away and which were essential. (2) It greatly increases the time consumed in the programming task itself, since many of the hardest decisions in programming are the *arbitrary* ones where the problem imposes no choice but the programming language does. Psychologically speaking, nothing is *really* arbitrary, so the programmer falls back on secondary criteria such as how fast he thinks the code the compiler will generate will be or what looks nicest, rather than being able to concentrate on the problem at hand. (3) Formal methods such as proofs of correctness become much more difficult when they must be applied to extraneous information rather than just the problem statement. (4) The ability of the system to learn from the user's input is severely limited, since it cannot know how to generalize from what is given. (5) Maintainability suffers because programmers studying old code have to learn every irrelevant choice the previous programmer made. (6) It is arguable that the complexities imposed by overspecification necessitate an empirical rather than an analytic approach to debugging.

Now the sort of benign underspecification we feel is needed abounds in natural language. A good example is presented by non-deterministic quantifiers such as "some". "Some dog is barking" does not specify which dog it is, it just states that there is one. The use of tenses to express relative temporal relations is another example. "John ate his dinner" is very different from "eat(John,

¹¹ It is true that lazy functional languages like Miranda allow you to talk about the *sequence* of numbers greater than 5 as "[6,7,8...]." This is not a generalized language feature, however.

Unnatural Languages

-8-

dinner, 19:00)."

Extension through juxtaposition. The extreme form of this underspecification is the free-form way in which natural languages allow one to build up specifications, tacking them on one by one until the required degree of specificity is achieved. For example, starting with a simple "sort!" one can move to "sort the cards" and then on to "sort the cards in descending order using a bubble sort algorithm while discarding duplicates and treating the jacks as high cards."

It is important to realize that this underspecification is *not* to be thought of in terms of default parameters. The simple "sort!" is not a call on a fully-specified

sort(what: any ordered things; descending: boolean; how: algorithm; discarding duplicates: boolean; jacks high: boolean)

with all the parameters unspecified. It is rather the case that the relevant specifications are discovered and added on the fly. The lexicographer who wrote the entry "sort" in Webster's Collegiate never even contemplated the possibility that jacks might be considered high card; this is a requirement I added afterwards.

This sort of "action at a distance" is analogous to the "interoperability" towards which the software industry is striving. It is unclear how far along this path Prism could go. I see three approaches to implementing it: (a) Use reflection¹² to simulate it. Thus when the interpreter processed "make jacks high," it might modify *itself* so that when it later interpreted ">" in the body of the sort algorithm, it would treat kings as less than jacks. (b) Some analogue to inter-process communication might be used; when doing the comparisons, the sort algorithm could "communicate with" the jacks-high "process" to ensure the proper ordering. (c) Severe limits could be placed on the freedom with which specifications could be piled on one another. For example, a conceptual framework could be imposed specifying exactly what kinds of modifiers are allowed. These special cases could be built into the interpreter. For example, knowledge about location could be built in, so that *any* prepositional

¹²Brian Cantwell Smith, ""

Unnatural Languages

-9-

phrase specifying a location could be combined with any verb. (d) The most promising approach, however, is none of the above, but rather is to be found in an execution model in which the *context* is taken into account in every act of interpretation. It is just such an execution model which we have been investigating in Unification Semantics (Incremental Systems Technical Report 89-???.)

Parts of Speech. Part of the conceptual baggage of natural language users is the set of composition rules for the language, encoded in its parts of speech. Programming languages are quite impoverished from this point of view, typically supporting just nouns, verbs, and a few built-in conjunctions. This severely distorts the way things are said in programming languages. It would be particularly regrettable in Prism, where programming will consist more of providing assertions to the interpreter than of constructing algorithms.

As an example, let us take the case of adjectives. Adjectives play a major role in Prism's property-based type system, and in specifications in general. How are they handled in traditional programming languages?

If the adjective is a constant property of the object, it will typically be built into the object's name as an inseparable part, meaning that it gets repeated every time the object is mentioned:

```
big_juicy_red_delicious_apple  
largest_common_denominator  
maximum_thrust_rate
```

If the property is not a constant, two possibilities exist. One is to create a subcomponent (subnoun) of the object to store the value of the property:

```
apple.color := red
```

Another is to store a boolean value recording whether the object has the property:

```
apple.is_red := true
```

Neither of these is a particularly convincing rendering of "The apple is red," but the *real* problem is that neither one allows the composition of noun phrases using the adjective. Consider trying to write an assertion of the form "Smart monkeys love big red juicy apples" in this notation. One ends up with something like

$$\forall x \forall y. x.type = \text{apple} \ \& \ x.color = \text{red} \ \& \ x.size = \text{big} \ \& \ x.juicy = \text{true} \ \& \ y.type = \text{monkey} \ \& \ x.smart \rightarrow \text{loves}(x,y)^{13}$$

The very natural on-the-fly creation of subtypes through use of adjectives is made very cumbersome because adjectives are not treated as first-class citizens in traditional programming languages.

There is one area where this shabby treatment of adjectives causes so many problems that proposals for reform surface periodically in the literature: dimensional analysis. In natural languages numbers are not dimensionless nouns, as in traditional programming languages; they are *adjectives applied to units*. The advantages of treating them that way in programming languages, and the obvious drawbacks of trying to simulate dimensional analysis using only the features of traditional programming languages, are well known¹⁴.

Parts of speech may be thought of as signatures which define the composition rules of natural language, in much the same way as the signatures of Ada functions define the legal compositions in that language. Setting aside the many and obvious complexities, we could set forth the following as the nine basic structuring rules for English:

¹³Prolog does little better: $X \text{ loves } Y :- \text{smart}(X), \text{monkey}(X), \text{big}(Y), \text{red}(Y), \text{juicy}(Y), \text{apple}(Y)$. The property lists of Lisp come close to providing the subtyping semantics of adjectives, but with too little syntactic support.

¹⁴References. We are not suggesting that units can be handled like any other adjectives, rather we are trying to reinforce our claim that adjectives have gotten a raw deal.

Unnatural Languages

-11-

Exclamation $\rightarrow S$

$N \times V \rightarrow S$

$V \times Adv \rightarrow V$

$Adj \times N \rightarrow N$

$Pron \rightarrow N$

$Art \rightarrow Adj$

$Prep \times N \rightarrow Adj$

$Prep \times N \rightarrow Adv$

$X \times Conj \times X \rightarrow \text{type of } X$

where "Adj", "N" and so forth are the lexical categories of the language. Recursion is introduced by the third and fourth of these rules, but it is not felt as nesting: "the big fat red Winesap apple" is parsed as "the big fat red Winesap (apple)", not "the (big (fat (red (Winesap (apple))))))". Only the last, atypical rule introduces nesting, and then only in the case of a very small number of *subordinating* conjunctions.

There is a natural-language rule which facilitates parsing. Parts of speech are traditionally divided into "function words", consisting of prepositions, pronouns, conjunctions, exclamations, and articles, and "content words," consisting of nouns, verbs, adverbs, and adjectives. The rule states that *there shall be no user-defined function words*. Thus when parsing "frumtious bismorky" I can be sure that such interpretations as article + noun and preposition + noun are ruled out. (An analagous rule in programming language forbids user-defined keywords.)

Given a programming language which supported a larger set of parts of speech, an important design decision would be whether also to emulate the fluid conversions natural languages support between parts of speech. I see no clear-cut answer here. Certainly there are some conversions which we would want to provide: that between adjectives and their corresponding nouns is one (from "the apple is red" to "red is a color.") If the crippled discourse structure of traditional programming languages is retained, the need for many of these conversions disappears. For example, consider a task with a "print" verb in it. In traditional environments one is not allowed to

talk *about* such tasks, but only to rendezvous with them. If one were to allow dialogues about them, however, it would be very natural to support participle formation (noun to adjective conversions), so that the user could say things like "if my task is printing...". More thought needs to be given to this topic.

Paucity of punctuation. The great abundance of punctuation in traditional programming languages seems an important impediment to their legibility. There are two reasons for such overpunctuation. Too often, extra delimiters and operators have been used to guide the parser and facilitate syntactic disambiguation; this is a form of punctuation we hope to do without. On the other hand, punctuation is also used in programming language to avoid having multiple implicit operators, and this form of punctuation we intend to retain. This would put us close to the natural-language system, where on at least one interpretation there is just one implicit operator, function composition governed by parts of speech, and punctuation serves the useful function of cluing the reader as to large groupings within the text. We propose function composition as our single implicit operator.

Brevity without abbreviation. A chronic dilemma in programming languages is the identifier length problem: long identifiers convey more meaning, but can clutter up the program and impair its legibility; abbreviated identifiers can reveal more clearly the program structure, but obscure the semantics. Natural languages have a variety of mechanisms for attaining brevity without resorting to abbreviation. The principal such mechanism is anaphora, where a context is retained and short co-referential terms are used to refer to elements of that context which have been more fully described earlier. The most conspicuous example is the use of pronouns to avoid restating the entire noun phrase it represents, but there are many others, such as repeating nouns without the adjectives originally used to describe them (first using "the total amount of income received in 1988," then simply "the total"), or using short adverbial phrases to recapitulate long ones ("sort the second deck *in the same way*".)

It is often revealing to see what becomes of natural language fragments when we apply programming-language conventions on them. Consider what happens to a recipe when we combine the no-

adjectives rule with the no-pronoun rule:

Take 250-grams-of-large-fresh-ripe-tomatoes. Peel the 250-grams-of-large-fresh-ripe-tomatoes. Chop the 250-grams-of-large-fresh-ripe-tomatoes. Sauté the 250-grams-of-large-fresh-ripe-tomatoes in oil.

What do the following examples tell us about the sanity of the assignment statement?

"Let the roast be the roast minus the carrots."

"Let the contents of the bowl be the contents of the bowl plus a cup of milk."

It is largely the absence of anaphora which makes abbreviations so essential in programming languages. The past few years have seen the introduction of a timid, severely limited form of anaphora in a few recently-designed languages (e.g. "it" in ML and Hypertalk.) In Prism we propose a more extensive anaphoric analyzer which makes limited use of semantic information to handle a wider range of types of anaphora. The complexities of dealing with full-blown anaphora in natural languages are well known¹⁵; fortunately, it seems to us that they are easily avoidable in a formal language.

Discourse Structure. During the 1950's computational linguistics focused on the syntax of natural languages, since it was felt that its semantics was intractable. Then Montague came along and showed how to incorporate the semantics in the system as well. It is increasingly felt in the computational linguistics community that we are today in the same situation with regard to *pragmatics*, i.e. that further progress in our understanding and formal analysis of language depends on bringing into the analysis the dialogue within which natural language discourse is always embedded. Taking speech-act theory as a point of departure, a number of researchers have attempted to provide a formalism for discourse structure¹⁶.

¹⁵For a view of the state of the art, see Hans Kamp et al., "The LEX C 'stem."

¹⁶ See for example David Evans, "Situations and Speech Acts."

The relevance of this for Prism are obvious. Traditional computer systems have a seriously crippled model of discourse: the user types commands and the computer utters truths. The system has no understanding of what the user's goals are, and does not consider the history of the discourse in giving its answers. You could type "2+2" at your machine all day long, and it would keep on typing back "4", never once asking you why you kept on asking the same question. (As with anaphora, there have been a few faint attempts to make the discourse structure more sophisticated. The Dialogue Manager in the Macintosh, for example, allows graduated error messages depending on how many times the error has been committed¹⁷.)

The importance of a system that has a dialogue structure and is sensitive to the history of the dialogue is of course of much greater importance than the trivial "2+2" example indicates; it is relevant to a large class of problems in the real world. One thinks, for example, of the "Mellon Bank problem" in which all the automated tellers of a large city ate all the bank cards of their customers for an entire day. If the system had had a reasonable model of the dialogue it was engaged in, it would have been able to reflect on what it was doing and realize that it was not behaving appropriately.

Crucial to any attempt to mimic natural-language dialogue structure will be the identification of contexts. Conventional wisdom says that natural-language contexts are mostly undelimited, yet some researchers have found context markers like "anyway" which have gone largely overlooked¹⁸. Engineering a flexible yet unambiguous context structure will be a major challenge.

Nesting. Closely related to the discourse structure is the question of nesting. Natural language avoids deeply nested structures like the plague, and the psychological dangers of deep nesting are well known. (This applies only to *local* nesting, however. A sentence might well be embedded six levels deep within a global structure—volume 1, chapter 2, section 3, subsection 4, paragraph 5, sentence 6. But it is unlikely to have six levels of embedding within it.) Two features of natural language account for the shallowness of its structures. On the one hand, its composition rules are such that

¹⁷ *Inside Macintosh, Chapter ??.*

¹⁸ References.

nested structures are impossible in all but a few severely restricted situations. On the other hand, it provides mechanisms (analagous to subroutine calls in programming languages) to provide the effect of nesting without the nesting itself.

Aliasing. We close this section with an enigma. Why is it that aliasing is so manageable in natural language and so disastrous in programming languages. I have no trouble dealing with a situation where "Chris," "Wicks," "my son," and "the blighter" all denote the same person; why is it so awful to have "the largest element in x" and "x[i]" refer to the same memory element?

A Natural-Language View of Prism's Type System

The Prism type system is intended to accord better with our everyday intuitions than have previous formal typing systems. In this section we take a look at some of the basic aspects of the Prism typing system and attempt to find analogues for them in natural language.

The Prism view of types as collections of properties certainly is closer to our everyday intensional notion of taxonomy than to the extensional view of types as collections of values. We define birds as "feathered bipedal egg-laying warm-blooded animals," not as "this crow and that crow and ... this nuthatch and that nuthatch and ..."

The doctrine that all functions are piecemeal-defined functions from any type to any type merely reflects the open-ended nature of natural language, which does not rule out any combinations of nouns and verbs *a priori*. I am always free to extend the language by defining what it would mean to behead exponentiation, no matter how unexpected that combination of noun and verb may seem.

The requirement that functions return the same results when applied to subtypes as they do when applied to supertypes with the same value reflects a common-sense feeling that properties should be invariant along the generality continuum. Somehow we feel that the meaning of "rich" in "rich man," "rich doctor," and "rich pediatrician" should be the same.

Unnatural Languages

-16-

It is easy to think of apparent counterexamples, of course. The sense of "yellow-throated" in "yellow-throated birds" is quite different, for example, from its meaning in "yellow-throated warbler": it is the difference between "having a yellow throat" and "belonging to the species *dominica*". The verb "to drink" has quite a different connotation when applied to "a liquid" and to "some aquavit," in spite of the fact that aquavit is a liquid. The Prism analysis of these situations is to claim that the more general senses have all the special ones bound up inside them. "To drink" under this analysis might mean "to sip slowly if it is a high-proof alcoholic beverage, and to chug in big mouthfuls otherwise." In this way the apparent contradiction is avoided.

When one is expanding the domain of a function to include a new type that is not a subtype of any of the types already in the range, there are two distinct possibilities. One is to define the expanded function recursively on some component of the type in question—presuming that the function has already been defined on the type of that component. This corresponds in natural language to *synecdoche*—the substitution of the whole for the part. Thus I might want to define a "prime" function on a record type as

$$\text{prime}(R) = \text{prime}(R.i)$$

where *prime* has already been defined on integer and where *i* is an integer field of *R*. This is exactly the mechanism at work in *synecdochal* expressions such as "to feed the army," where the sense of feeding the whole is derived from the sense of feeding the part. When functions are expanded in this way, the new properties added to the type are "conjunctive" in the sense defined in "Prism Categories, Adjectives, and Functors" (Incremental Systems Technical Report 89-???). In the example above, primeness is now a conjunctive property of *R*.

Symmetry demands that there be an inverse process where the meaning of the function applied to the part is derived from its meaning when applied to the whole. Thus in natural language the sense of "to bring home the bacon" is derived from the sense of "to bring home food in general." In programming languages it is difficult

to think of an example because programming languages typically provide no way of going from components to their totalities. One would never write "print A[3]" when one meant "print A". It is interesting to speculate as to why this whole/part synecdoche seems less valuable in programming than part/whole synecdoche.

The second possibility when expanding the domain a function is not to call the function recursively, but rather to define a new body for the function that (presumably) bears some similarity to the function on previously defined types. This is the natural-language phenomenon of *metaphor*. If I hear someone say that "the price of gold is skyrocketing," I do not look up into the sky, but rather realize that he is using the general metaphor schema "up is more" [ref] to extend the domain of skyrocketing to gold prices. The properties defined by these new functions are just the "orthogonal" properties described in "Prism Categories, Adjectives, and Functors." The statement that orthogonal properties are not type properties is simply the statement that one does not use metaphor when doing taxonomy. If I sat down to do a taxonomy of gold prices, I would not divide them into "skyrocketing" and "non-skyrocketing."

Let us take an example that shows the difference between synecdochal and metaphoric function expansion. Suppose one hears a person X refer to "exponential toothpicks" and asks X to explain what he meant. X might answer that he had just bought some toothpicks made by a company which had experienced exponential growth over the last five years. This would be a synecdochal expansion of "exponential", involving no shift in meaning but just a movement from a part (the toothpicks) to the whole (the company). On the other hand, X might answer that he just meant they were great toothpicks. Here would be a metaphorical expansion of meaning using some "fast is good" metaphor schema, and we would say that the property of being exponential is "orthogonal" to the other properties of toothpicks. One would continue to categorize toothpicks as flat or round, wooden or plastic, long or short—not as exponential or not.

One troublesome point in this account is that it cannot account for "white metaphors," the metaphors that have fixed to the point that they are taxonomic. In the beginning, "splitting headache" was

a metaphor exploiting the physical sense of splitting. Now however it has solidified into a "white metaphor" and we talk about splitting headaches without images of axes coming to mind. As a result, it would seem to be quite reasonable to use the term taxonomically—one can imagine a doctor trying to diagnose your illness by asking you if your headache was splitting or not. I see no parallel in Prism to this evolution of language

Typing in natural language. In natural languages, legitimate types for actuals are usually determined by a semantic analysis of the operations performed, not by a formal type. We do not say "decapitate(x: animal). To remove x's head." Rather we say "decapitate *v.t.*: to remove the head of." It does not matter what the definer of the function foresaw its uses to be; if it makes semantic sense, we should interpret it that way. We do not refuse to interpret "decapitate the pin" just because a pin is not an animal. Prism allows just this sort of specification.

A Prism Primer

dam

1991 Aug 2

Noun Phrases.

Prism's property-based type system is incarnated in the use of modifiers and quantifiers to construct flexible, ephemeral subtypes on the fly. These subtypes are designated by noun phrases, which are constructed in Prism by starting with a type, restricting it by adding properties which the examples of the type must satisfy, and then quantifying the result. For example, the noun phrase "any (<5000) Mersenne prime integer that ends in '9'" starts with the type integer, adds the properties of being prime, of being Mersenne, of being less than five thousand, and of ending in nine, and then quantifies with "any."

```
name => np
```

```
quantifier {premodifier} type_literal [postmodifier] => np
```

Note that only one postmodifier is allowed; this is to allow disambiguation of noun phrases such as "the cat on the hat on the mat on the bat," which will be parsed as "the cat on (the hat on(the mat on the bat))." This is no functional limitation, since parentheses and conjunctions may be used to combine postmodifiers: the cat (on the mat & that loves salmon) is hungry.

Names.

Names in Prism constitute a primitive syntactic category. Names designate values of any type. If the type of the value is type "type", the value designated by the name is itself a type, and may enter into quantified expressions.

```
text_literal | graphic_literal | pronoun => name
```



```
name [" " name] => name
```

```
'(' np ')' => name
```


```
type_name '[' np '[' np ']' ']' => name1
```

¹ I have deleted the "type qualification" production (type_name " " name => name) since it seems to serve the same function as the use of type names as premodifiers. Are not "the dog Fido" and "dog'Fido" one and the same?

Text Literals. The most familiar kind of name is a text literal consisting of characters. Special characters that do not have other meanings in Prism may be used as text literals. *Examples.* Fido. Integer. 3. 3.48×10^{123} . π . θ . \clubsuit . $\$$.

Graphics Literals. In addition to special characters, Prism allows the use of graphics as literals. In this way the display image seen by the programmer and that seen by the user of the program coincide. *Examples.*  is a zoom box.  is a rook.


Pronouns. The third form of name is the pronoun. These are like text literals except that they designate values anaphorically. That is, like quantified phrases using the definite article, they are coreferential with another noun phrase occurring earlier in the program. *Examples.* Square all (<10) primes. Print them.

Note that unlike natural language, Prism permits the use of pronouns in quantified expression and in compound names. *Example.* It'top. Them'  Every green it. (In this last example, "it" must have a type value as its antecedent.)

Apostrophe Qualification. Prism provides a genitive construction using the preposition "of", but, in common with many programming languages, it also provides a shorthand notation. This is provided by the genitive apostrophe. *Example.* John'hat'brim (= the brim of the hat of John or "John's hat's brim").

Noun Phrases as Names. The end result of a noun phrase constructed with quantification and modification designates one or more values of the type being quantified and modified. As such, it may be used as a name. To avoid ambiguity, however, such noun phrases must be parenthesized. *Examples.* (Every AK-47 assault rifle)'bullets. (Some zoom boxes)'boundingRectangle.

Type Constructors. In addition to text literals and graphics literals, literals for compound types may be constructed from component values. This is done by enclosing the component values in square brackets and prefixing it with the name of the type.

Examples. List[it'top, every prime integer such that it ends in '3', any character, ]. Employee[Ogden, Smith, 21000 \$, 1956, October, 23].

Quantification.

Plurals. Few traditional programming languages support the formation of plurals; typically iteration or repetition must be used instead. Compare "Type the letters" with "for $i := 1$ to $n_letters$ do type(letters[i])". Or compare "type the letter, the report, and the résumé" with "type(the letter); type(the report); type(the résumé)".

Prism allows plurals. The default meaning of plural constructions is the so-called "distributive" interpretation; i.e., operations applied to plurals and predicates asserted of plurals are interpreted as being applied and predicated of each element of the plural.

Examples. Tom, Dick, and Harry are employees. Register the employees. Print i, j,

and k.

The "group" interpretation, indicating that operations and predicates apply to all the elements taken together, is obtained by preceding the plural with "the group of."

Examples. The group of Tom, Dick, and Harry has three elements. The group of i, j, and k contains one prime.

Quantifiers. To the extent that traditional programming languages support quantification, they do so by means of the quantification variables inherited from the predicate calculus. Compare "All dogs are mammals" with " $\forall x, \text{dog}(x) \supset \text{mammal}(x)$."

Prism supports a number of forms of the variable-free quantification of natural language. They may be thought of as "selectors" or "filters" which select examples from the type or subtype specified by the rest of the noun phrase.

This selection is of two distinct kinds. Some quantifiers determine in and of themselves which examples are to be selected, in which case we say the selection is "forced". For example, in "every green dog," once the subtype "green dogs" has been constructed, we have no choice but to select *all* the examples of green dogs in forming the quantified expression. On the other hand, other quantifiers leave the choice of examples open, in which case the selection is "free". Thus in "some green dog" the selection is not determined completely by the choice of subtype and quantifier, but depends on other elements of the context.

Free selection is also of two types, arbitrary and nondeterministic. In arbitrary selection, the choice of examples is completely unconstrained and random, as in "any (old) book will do". In nondeterministic selection, the choice of examples is constrained by the requirement that things "come out right" in a quasi-game-theoretic sense. Thus in "some green dogs are barking" the choice of green dogs is constrained by the contextual requirement that the dogs selected must make the assertion true. We call these nondeterministic quantifiers because the selection is not done at the time the quantification is processed, but only when the result is needed. For example consider the following Prism fragment:

Some numbers are prime. Print them. 2,3,5,7,11...

Clearly, if the processor stopped at the first statement to compute all the prime numbers, it would never reach the print statement. It should be noted that in imperative contexts, the contextual information may be inadequate to fully determine the selection, in which case arbitrary selection takes over. Thus "print some (>5) integer" and "print any (>5) integer" produce the same results.

We now consider the different types of quantifiers in the current version of Prism.

Numeric quantification. Numbers may be used to quantify noun phrases

nondeterministically, with or without range constructors such as “at least” and “at most”. “Exactly” is redundant in Prism; “3 squares” and “exactly 3 squares” mean the same thing.

[‘at least’ | ‘at most’ | ‘exactly’] number_name => quantifier

Examples. 2 chess pieces are rooks. At most 6 employees are architects. Print at least 3 letters.

Relative quantification. Instead of giving the number of examples in the quantification absolutely, one may specify it only relative to the collection as a whole.

‘some’ | ‘few’ | ‘many’ | ‘most’ => quantifier

Examples. Some document windows have go-away boxes. Most users give correct passwords.

These are also nondeterministic quantifiers. One might at first wonder about their utility in programming languages, but they are important tools for avoiding overspecification in Prism, where a major goal is the provision of reasonableness checking. Forcing one to choose an exact figure and write “if wrong passwords > 4% or wrong passwords < 0.005%” denies the processor crucial information about the program—viz. exactly the information that *most* answers should be correct.

Total quantification. Prism provides three² quantifiers which in different ways select the totality of the domain.

‘all’ [number_name] => quantifier

‘every’ [ordinal] => quantifier

‘any’ [number_name] => quantifier

“All” and “every” are forced quantifiers which specify that every example in the collection is to be selected.³ The only difference between them, apart from the syntactic

² I have omitted “each”, since Grammar 0.4 provides no justification for it—“every” can always be used instead.

³ The comments in Grammar 0.4 state that quantifications formed with “all” can have the group interpretation, but I see no advantage to this: one can always say “the group of all employees,” can one not? Grammar 0.4 seems to suggest that the difference between “all” and “every” is that between a group and a distributive interpretation. I am rejecting that in favor of the simpler rule that all quantifiers have the distributive

issue that "every" is a singular, is in the use of the accompanying numeric expression. The optional numeric value with "all" is a redundant specification of the number of examples that result. *Examples.* (a) All modal dialogues have plain windows. (b) All 3 (<10) primes are divisors of 105! *4 primes are <10, not 3.*

The optional ordinal with "every" provides a filter which restricts the selection. *Examples.* Every shark is a fish. Every 3rd line starts with "Alas!".

Note that because the default interpretation is distributive, multiple quantifiers result in the cross product of their individual numbers. Thus, "All 4 girls kiss all 7 boys" denotes 28 acts of kissing. "The group of all 4 girls kisses the group of all 7 boys" denotes 1 act of collective kissing. "All 4 girls kiss the group of 7 boys" denotes 4 acts of semi-collective kissing, and so on.

"Any" is an arbitrary quantifier used to specify that no matter which examples are chosen, the operation will succeed. Although "any" does not select every example in the collection, it is classified with the total quantifiers because if a *predicate* is applied to a quantification constructed with "any", since the selection is arbitrary, the result is the same as if "every" had been used. That is, "Any prime is an integer" is the same as "Every prime is an integer." This is *not* the case with other operations, however: "Print any prime" is quite different from "Print every prime." The optional number_name specifies the size of the resulting selection. *Examples.* The sum of any 2 (<10) primes is <13. Report on any 5 salesmen. Any 6 weapons will protect the tank.

Interrogative quantification. The interrogative "how many" is used to form questions about quantification.

Examples. How many primes are <10? 4. How many users give correct passwords? Most.

Note that the current version of Prism does not provide for quantification of mass nouns: "A little water is in the bottle," etc. Consequently, there is no need for a "how much" interrogative.

Articles. Prism includes versions of the definite and indefinite articles.

'a' | 'an' | 'the' [number_name] => quantifier

"A" and "an" are of limited utility; they specify that a single example is to be selected, but do not distinguish between arbitrary and nondeterministic selection. Thus in "a bird has landed on the porch," the article is being used nondeterministically to pick out the particular feathered friend who has graced our home with its presence, while in "a bird is a feathered biped," it is being used to denote arbitrary selection, and is thus synonymous (in Prism) with "any bird is a feathered biped"⁴.

interpretation.

The definite article is a nondeterministic quantifier which in Prism differs from "some" only in that the resulting noun phrase must be coreferential with a preceding noun phrase, and is thus a primary mechanism for forming anaphoric phrases. Consider for example: "Print any integer. $Z \leftarrow$ the integer + 5". Just as "some green dog is barking" picks out a green dog which satisfies the barking property, "the integer" picks out an integer which has the property that it has been previously referred to, in this case by "any integer". The numeric expression functions as with "all" to redundantly specify the number of resulting examples. *Example.* All 4 (<10) primes.

Modification.

Properties. The fundamental primitive of the Prism type system is the property, which is defined to be a Boolean function of one argument.⁵ Properties thus partition the objects to which they are applied into two classes.

Prism provides three ways of denoting properties: by names, by prepositional phrases, and by relative clauses. Thus "red" and "prime" are properties denoted with names; "in the active window" is a property denoted by a prepositional phrase; and "that is printing" is a property denoted by a relative clause.

Types and Clusters. In Prism there are two quite different ways of combining properties into larger groupings. The first method creates *clusters* by *disjoining* a set of properties. Thus in Prism "color" is a cluster meaning "red or orange or yellow or green or blue or indigo or violet."

The second method of combining properties produces *types* by *conjoining* a set of properties. Thus "featherless rational biped" is a type meaning "featherless and rational and biped." Unlike clusters, types are closed under entailment, so that "legged" and "sentient" are properties of the foregoing type, since they are entailed by "biped" and "rational" respectively.⁶

⁴ This ambiguity is quite problematic, and raise the issue of whether the indefinite article should not be eliminated from the language altogether. The current thinking is that it is acceptable in limited contexts. For example, inside definitions—which are in any case the strongest argument for retaining the arbitrary article—it could be interpreted as arbitrary, and as nondeterministic elsewhere. *Example;* A bird =_{def} a feathered biped.

⁵ This is (according to DAF) in accordance with programming-language tradition, but conflicts (according to DAM) with philosophical tradition, which holds that such things as "color," "mass," and "odor" are properties.

⁶ Types are denoted using braces and the superscript turnstile. The notation for clusters is as yet undecided.

Premodifiers. Named properties in Prism are simply placed before the nouns (named types) to which they apply.

`property_name | type_name => premodifier7`

Examples. Green dogs. Prime integers. Large dialogue boxes. AK-47 assault rifles.

Literals in Prism are conceived of as types consisting of a single property, viz. the property of bearing a name. Thus "Fido" is the singleton set consisting of the property "has the name 'Fido'". Since many individuals can bear the same name, Prism allows for using type names as premodifiers to pick out a particular individual or set of individuals.

Examples. The dog Fido. The toucan Fido. The movie Fido. The prime 5. The integer 5.

Units. Measurements are a special kind of property consisting of a number and a unit. In natural language there is often an elision of the property being measured; one talks of a "9 mm bullet" rather than "a bullet with a 9 mm diameter." Prism does *not* support this elision.

The unit name is optional, so that unitless measurements can be handled.

`number_name [unit_np] => measure_np`
`measure_np {x measure_np} => premodifier`

Examples. A 250 °C temperature. A 3 m length. A 4 m × 4 m area. A 4 length. A 4 × 4 area. A window of 200 pixel × 300 pixel area. A warhead of 250 °C temperature. An array of 100 × 2 area.⁸

Prepositions. Another special form of property is that denoted by prepositional phrases. These properties express relations (in the everyday, nontechnical sense) between noun phrases, and occur as postmodifiers.

`preposition noun_phrase => prepositional_phrase`
`prepositional_phrase => postmodifier`
`'in' ['reverse'] [binaryBooleanFunction_name] 'order' => postmodifier`

Examples. Any green dog under any pink lion. Every prime in {3, 5, 7, 11, 24}. Each line

⁷ Prism0.4 also allows for "value names" as premodifiers, but I can no longer make any sense of that.

⁸ We need to provide for and explain expressions such as "a liquid with 450 °C boiling point," "a circle with 35 mm diameter." (I.e. the introduction of specific measurements with known dimensionality.)

in the active window. All missiles on the aircraft.

Note that in formal languages the desire for succinct notation has fostered a tradition of treating relational expression as producing not the individuals which satisfy the relation, but rather a truth value indicating whether or not the stated individuals in fact satisfy it. Thus in such languages, to write "every $x > y$ " one must use a circumlocution such as $\{x \mid x > y\}$, but can write "if $x > y$ " as a shorthand for "if x is $> y$." Prism follows this tradition for mathematical symbols such as the relational operators, but breaks with it for prepositions. Thus "every prime in $\{3,5,7,11,24\}$ " returns 3, 5, 7, and 11, not "false." The Prism equivalent to Pascal's "if x in s " is "if x is in s ." Although one would not write "every $x > y$ " in Prism (since it would mean "every true" or "every false" depending on the values of x and y), one can achieve the same effect by converting the relation into a prefix modifier, and writing "every $(>y) x$."⁹

The use of prepositions can greatly simplify algorithm expression. Consider the following Prism fragment and its Pascal equivalent:

Display the window for 30 s or until a mouse click.

```
showWindow(theWindow);
c := clock;
while (clock - c) < 30 and not mouseClicked do;
closeWindow(theWindow);
```

A special form of prepositional phrase allows specification of the *order* in which examples are to be chosen when forming quantifications. The binary Boolean function name specifies¹⁰ which ' $<$ ' operator is to be used in ordering the examples. *Examples.* Print every word from the dictionary in reverse lexicographic order. Place every (<100) prime in numeric order into A.

Relatives. The final form of property allowed in Prism is the relative clause. Like the nondeterministic quantifiers, relative clauses do not specify how the examples are to be chosen, but merely gives a condition which must hold; it is up to the system to choose examples which make the whole phrase turn out right.

'such that' claim => postmodifier

⁹ We may want an extension of this adjective formation mechanism which allows using verb phrases as adjectives, preferably in conjunction with participle formation: "every working employee." This is all syntactic sugar, however: "every $(>y) x$ " \equiv "every x that is $> y$ "; "every working employee" \equiv "every employee that works."

¹⁰ Just how one specifies that an order is lexicographic is left unspecified for the time being.

'that' vp => postmodifier

The second form is pure syntactic sugar; "the mouse that roared" is simply a more convenient form for "the mouse such that it roared." *Examples.* Every fibonacci number that divides 120. Every x, y : integer such that $x + y > 54$. Every spacestation such that $\text{distance}(\text{it}, \text{moon}) < 100,000$ m length.

Functions and Expressions.

Claims and Declarations¹¹.

Claims are expressions which yield truth values when evaluated.

Boolean_np => claim
 np vp => claim
 claim '.' => declaration

The two primary forms of claims are boolean expressions and declarative sentences. *Examples.* $x > y$. Red and green are colors. The printer is busy.

A declaration is a speech act consisting of the assertion of a claim. That is, the action I expect of the system when evaluating a declaration is to compare the claim I have asserted to its database, and to notify me if an inconsistency is detected. *Example.* 3 is a color! *No it is not.*

Claims can be used in other ways, however—in the protasis of a conditional sentence, for example. When I utter "if the printer is busy, use the speaker instead," I clearly am not asserting that the printer is busy, and do not want the system to give me a warning if it is free.

Special forms of claims. Two frequently-occurring verbs are given special syntax for the sake of succinctness.

np '=def' np => claim
 np '=imp' np => claim

The verb '=def' is used to introduce definitions. It makes the claim that the value of the first noun phrase is defined by the value of the second noun phrase. *Example.* $x! =_{\text{def}} x \times (x-1) \times (x-2) \times \dots \times 1$. $x^i =_{\text{def}} x \times x^{i-1}$. To zoom a window =_{def} to make the bounding

¹¹ I have used "declaration" instead of "declarative" because claims are declarative sentences; using the word both ways is confusing.

rectangle of it = the size rectangle of the screen.¹²

Analogously, the verb '=imp' is used to introduce implementations. It makes the claim that the value of the first noun phrase can be implemented by the second noun phrase. *Examples.* $x! =_{\text{imp}} \text{if } x \leq 1 \text{ then } 1 \text{ else } x \times (x-1)!$. To zoom a window $=_{\text{imp}} \text{resize(it, (the screen)'bounding rectangle)}$.

Note that since these are verbs, they may be used wherever claims are allowed. Thus it is allowed to write "if $x! =_{\text{def}} x \times (x-1) \times (x-2) \times \dots \times 1 \text{ then} \dots$ " or "if to zoom a window $=_{\text{imp}} \text{resize(it, (the screen)'bounding rectangle)}$ then..." The claim is true if and only if the system has been given a definition or implementation which exactly (in some sense) matches the one specified.

Verb Phrases and Imperations¹³.

A verb phrase is a partial application of a verb, with every slot except the subject filled in. It is thus analogous to the partial evaluation of a function with one argument left unsaturated. An imperation is a speech act requesting an action or state change.

From a linguistic point of view, Prism's treatment of verbs is rather *ad hoc*. It exploits some idiosyncrasies of English to simplify its processing. In particular, it divides verbs into action verbs and copulatives, and disallows any imperative forms of the latter, since that would require special treatment of imperatives like "be." It allows only second-person imperatives, with the exception of the assignment imperative. (It is a peculiarity of English that the uninflected root, the non-third-person-singular forms of the present declarative, and the second-person imperative are all the same for most verbs.)

Declaratives. There are two forms of verb usage in Prism. Intransitive, transitive, and bitransitive verbs can simply juxtapose the verb with its objects. Verbs with other parameters must use function form syntax. Note that the optional prepositional phrase is *not* a parameter to the verb, but rather an environmental modifier which changes the behavior of the verb "behind its back". The prepositional phrase is placed *before* the objects to distinguish it from any postposition prepositional phrases modifying the objects of the verb.

```
action_verb [prepositional phrase] '(' [np [' np] ] ')' => action_vp
action_verb [prepositional phrase] [np ['to' np]] => action_vp14
```

¹² We need examples of nonanaphoric name introduction.

¹³ I use the word "imperation" to distinguish clearly between grammatical category and speech act. This gives the series of grammatical categories imperative / interrogative / declarative and the parallel series of speech acts imperation / interrogation / declaration.

¹⁴ I have my doubts about distinguishing bitransitives as Prism0.4 does. In linguistics they must be treated specially because of constructions like "give him the ball"

action_vp => vp
 copulative [np | premodifier | postmodifier] => vp

Examples (including subject noun phrases). (1) Intransitives: Every green warhead shines¹⁵. Some Chrome-57 atoms decay in 27 s. That subroutine sorts with the Quicksort algorithm. Zoom boxes appear at the upper right-hand corner. (2) Transitives: That laser printer prints at 1000 dots/cm the reports. That algorithm computes in $O(n \times \log(n))$ time it's result. (3) Bitransitives: Each accountant pays on (the last day of every month) a check to each employee. (4) Copulatives. The file with the report is in (the directory such that it's name = "reports"). All ($>x$) integers are $>(x+1)$. The earth appears small from 1000 km height. (5) Function form: Each planet attracts(sun, (the planet)'mass, distance(sun, the planet)). The F-16 flies through the sky(5000 km, 800 km/h, 1826 L/100 km).

Imperatives. As mentioned above, in Prism imperatives are syntactically indistinguishable from action verb phrases.

action_vp => imperation
 variable ' \leftarrow ' np => imperation

Examples. (1) Intransitives: Halt. Initialize. Start. (2) Transitives: Print every letter that has a ($>89/08/01$) date. (3) Bitransitives: Show some employees such that them's salary > 20000 \$ to Joe. Send any monthly report to the Houston office. (4) Assignment. (Each employee)'salary \leftarrow it + $20\% \times$ it. The group of bad printers \leftarrow it + Old_Faithful.

Conjunctions.

Conjunctions are used to join two or more instances of the same syntactic form. Prism provides two distinct forms of conjunction. The first, more concise, form is only available for "and" and "or," since it requires that its arguments be associative.

premodifier | postmodifier | np | vp | claim => form
 form ',' {form ','} ['and' | 'or'] form => type of form
 form conjunction form => type of form

(where the "to" is omitted), but Prism does not allow such constructions, so there is nothing special about bitransitives for us. I think we should either provide a more general prepositional-parameter mechanism, or else require function form for verbs with more than one parameter.

¹⁵ To what extent subject-verb agreement will be supported is still undecided, but the examples assume it will be supported to some degree.

'and' | 'or' | 'xor' | 'iff' | 'implies' => conjunction

'not' form => form

Examples. Send the message to Tom, Dick, and Harry. Socrates is a Greek implies Socrates is a man implies Socrates is mortal and not Socrates is feathered. x iff a xor b iff not c or d iff e .

Control Mechanisms.

Computations. Prism recognizes four types of computations: noun phrases, declarations, imperations, and interrogations. The language's control mechanisms operate indifferently on each of those four types. Thus the same syntax is used to build a conditional data structure as is used to build a conditional statement. Computations may be given names, and may be evaluated in foreign scopes through the use of the 'with' operator.

```
np | declaration | imperation | interrogation => computation
'«' simple_name »' computation => type of computation
'with' np 'do' computation => type of computation
```

Note that interrogations are allowed only in interactive dialogs, not in programs.

Examples. (1) If «empty» stack'top = 0 then push any integer. Not empty. *Here empty is an anaphoric name for "stack'top = 0."* (2) With Venus do print mass. (3) With math_package do print σ of list[3, 8, 15, 7, 22, 4].

Branching control. The fundamental branching statement in Prism is the case statement, which occurs in two forms. The if statement is provided as a more succinct special case.

```
'case' ['when' claim '=>' computation] => type of computation
'case' np ['when' np '=>' computation] => type of computation
'others' => claim | np
'if' claim 'then' computation => type of computation16
```

Examples.

(1) Case

```
when the animal is a lion, a tiger, or an elephant => run
when it is a dog or a cat => pet it
when it is a sowbug => feed it to the lizard.
```

(2) Print case

```
when  $x \geq \infty$  => 'Operand too large.'
when not some printer is free => 'All printers busy.'
others => any integer! 17.
```

¹⁶ I had a 'whenever claim then computation' production, but then deleted it. Do we want one? with daemon semantics?

(3) Case Joe, John, and Jim

when every employee that earns (<10000 \$) \Rightarrow «reward» give a raise to them
 when every salesman such that their sales are (>1000000 \$) \Rightarrow reward
 others \Rightarrow fire them.

Iterative control. For the moment Prism has but a single iterative construct, which iterates through noun phrases.

'for' np 'do' computation \Rightarrow type of computation

Examples. (1) Print for every prime x such that $3 \leq x \leq 7$ do (x . x^2 . x^3 .)

3	9	27
5	25	125
7	49	343

(2) For a few employees such that they'birth $> 79.01.01$ do print they'name. *Tiny Tim, Lil Youngling, Pete Petit.*

Sentences.

Prism recognizes two classes of sentences: those that are entered interactively and acted upon immediately by the system, and those that are entered into programs which are executed at times possibly remote from the time of their creation. Most of the examples of declarations and imperations we have seen up to now have been taken from programs.

declaration | imperation \Rightarrow sentence

The interactive versions of declarations and imperations differ only in the use of the exclamation point rather than the period:

claim '!' \Rightarrow interactive_declaration
 vp '!' \Rightarrow interactive_imperation

Examples. (1) Print a few employees'names! *Joe Blow, Fred Flintstone, Barney Miller.*

(2) 567 is prime! *False.* Explain! $567 = 9 \times 9 \times 7$.

In addition to these sentences, Prism allows interrogation and calculation in interactive mode. Interrogation is used to query whether a claim is true (i.e. consistent with the database) or not, while calculation is a shorthand for displaying noun phrases (i.e. " x !" is equivalent to "display x !").

claim '?' => interactive_interrogation
np '!' => interactive_calculation

interactive_calculation |
interactive_declaration |
interactive_imperation |
interactive_interrogation => sentence

Examples. (1) Airplane with biggest mass! *B-52*. (2) Temperature of boiling silicon! *527 °C*. (2) Barney Miller works in what¹⁷ precinct? *57th*. What divisions have (> 50000 \$/quarter) earnings? *Northeast, Southeast, Northwest*.

Scoping.

{tbw}

¹⁷ Need to put this and other interrogatives in the grammar, or document them *somewhere*. In general the number of interrogatives should be minimized; there are often several ways to ask a question, and we do not need to support them all. E. g. "Employees that have (> 20000 \$) salaries!" = "What employees have (>20000 \$) salaries?"

Observations on Prism.

1. I sure would like a solution to the "spaces in identifiers" problem. I want to be able to write "(every month)'last day", not the required "(every month)'last_day". Cf. "the last day of every month".

2. Holes: function and verb declarations. Name introduction. Scoping. Grouping and indenting.

Appendix A: Collected Syntax

```

name => np
quantifier {premodifier} type_literal {postmodifier} => np
text_literal | graphic_literal | pronoun => name
name [" name] => name
'(' np ')' => name
type_name '[' np (',' np) ']' => name
['at least' | 'at most' | 'exactly'] number_name => quantifier
'some' | 'few' | 'many' | 'most' => quantifier
'all' [number_name] => quantifier
'every' [ordinal] => quantifier
'any' [number_name] => quantifier
'a' | 'an' | 'the' [number_name] => quantifier
property_name | type_name => premodifier
number_name [unit_np] => measure_np
measure_np (x measure_np) => premodifier
preposition noun_phrase => prepositional_phrase
prepositional_phrase => postmodifier
'in' ['reverse'] [binaryBooleanFunction_name] 'order' => postmodifier
'such that' claim => postmodifier
'that' vp => postmodifier
Boolean_np => claim
np vp => claim
claim '.' => declaration
np '=def' np => claim
np '=imp' np => claim
np '=def' np => claim
np '=imp' np => claim
action_verb [prepositional_phrase] '(' [np (',' np) ] ')' => action_vp
action_verb [prepositional_phrase] [np ['to' np]] => action_vp
action_vp => vp
copulative [np | premodifier | postmodifier] => vp
action_vp => imperation
variable '<' np => imperation
premodifier | postmodifier | np | vp | claim => form
form ',' (form ',') ['and' | 'or'] form => type of form
form conjunction form => type of form
'and' | 'or' | 'xor' | 'iff' | 'implies' => conjunction

```

'not' form => form
 np | declaration | imperation | interrogation => computation
 '«' simple_name '»' computation => type of computation
 'with' np 'do' computation => type of computation
 'case' ('when' claim '=>' computation) => type of computation
 'case' np ('when' np '=>' computation) => type of computation
 'others' => claim | np
 'if' claim 'then' computation => type of computation
 'for' np 'do' computation => type of computation
 declaration | imperation => sentence
 claim '!' => interactive_declaration
 vp '!' => interactive_imperation
 claim '?' => interactive_interrogation
 np '!' => interactive_calculation
 interactive_calculation |
 interactive_declaration |
 interactive_imperation |
 interactive_interrogation => sentence

Grammar 0.5

dam

1991 Aug 2

General

(1) To simplify processing¹ the grammar has been kept simple. One way to think of it is as baby talk: verbs are restricted to second-person imperatives and third-person declaratives. First- and second-person declaratives are expressed in the third person, as, "Daddy wants a greatest common denominator."

Quantifiers

Quantifiers are used to specify how many examples of the type are to be selected and how those examples are to be interpreted (by the parent operation). Quantifiers do not specify which examples are to be selected. The selection is generally nonarbitrary.

nclass number_name => quantifier The number of examples may be specified absolutely.

'at least' | 'at most' | ['exactly'] => nclass

'some' | 'few' | 'many' | 'most' => quantifier

The number of examples may be specified relatively.

'any' [number_name] => quantifier

"Any" is used to specify that the selection is arbitrary and that the default numeric value is one.

'each' => quantifier

"Each" is used to specify that every example is to be selected each one at a time. Two or more independent arguments quantified by "each" specify cross product computations. Like "each", but ordinals may be used to narrow the selection.

'every' [ordinal] => quantifier

'how many'

"How many" is the interrogative quantifier.

'the group of' quantifier => quantifier

"The group of" is used to specify that every example is to be selected together as a single group.

¹ DAF also believes there are psychological benefits (i.e. that this is good language design), a point DAM disputes.

'all' [number_name] => quantifier

"All" specifies that every example is to be selected but does not distinguish between one at a time and group interpretation. The numeric value is a redundant specification of the number of examples.

'a' | 'an' => quantifier

"A" and "an" specify that one example is to be selected but does not distinguish between arbitrary and nonarbitrary selection.

empty => quantifier

The absence of the quantifier specifies that the unquantified type itself is intended.

'the' [number_name] => quantifier

"The" is used to specify anaphoric reference to the intended quantity of examples. The numeric value is a redundant specification of the number of examples.

quantifier 'of' primary => primary

Quantifiers may be cascaded, This form cannot be used with the quantifiers "a", "an" and "the".

Modifiers

Modifiers restrict the subtype or position of examples. Properties are boolean-valued functions of one argument. Premodifiers must be names. Postmodifiers may be prepositional or relative phrases. A prepositional phrase specifies a relational feature and the object to which it is to be applied. Premodifiers and relative phrases specify properties that restrict the subtype of the examples; these properties are easily computed from the modifiers.

property_name => premodifier

This form of modifier restricts the examples to those satisfying the given property. By definition a property is any boolean-valued unary function. Note that verb phrases are not properties: (1) Unlike functions they have side effects. (2) When applied, instead of producing boolean values, they make assertions.²

² Although DAF and DAM believe they are in essential agreement on the

value_name => premodifier	<i>This form of modifier restricts the examples to those having a feature with the given value.</i>
type_name => premodifier	<i>This form of modifier restricts the examples to the named type.</i>
measure_np { 'x' measure_np } => premodifier	<i>This form of modifier restricts the examples to those that have the same number as the measure when measured in the same units.</i>
preposition primary => postmodifier	<i>This form of modifier restricts the examples to values of a particular feature of the object np. The preposition specifies the feature as some positional relationship.</i>
'such that' claim => postmodifier	<i>This form of modifier restricts the examples to those for which the claim is true, The claim typically contains anaphoric references</i>

facts, they have persistent differences as to how the facts should be conceptualized. DAM's version is as follows. (1) Verbs only have side effects when they are used imperatively. I can sit around all day saying "Paul is blowing up Heinz Hall" without so much as chipping the paint on the building. Consequently we can continue to use the traditional Tarski-style view of statements as boolean-valued functions without harm. (2) As for the difference between assertions and boolean-valued expressions, the situation seems pretty dubious. DAF's contention is that assertions are put into the data base unevaluated, while functions are always evaluated immediately. It seems to me that lazy evaluation and immediate evaluation are equally available in both cases. In Miranda-style notation we might write " $\#[2, 4, 6...] < 3$ " and the processor would not go off and immediately compute the length of the infinite sequence, but this does not mean that Miranda's "<" is a verb rather than a function. Contrariwise, whether or not an utterance gets immediately interpreted seems irrelevant to the nature of its speech act. If you say "It is raining outside" I may either look out the window to evaluate your statement or wait until later to do so. I do not see why we should restrict verbs to later-evaluated items.

JCS believes that the issue is neither lazy evaluation nor side-effects, but rather Frege's notion of judgment. DAM agrees that there is such a notion, but denies that it corresponds to any difference between verbs and functions.

to the examples.

'that' vp => postmodifier

This form of modifier restricts the examples to those satisfying the verb phrase.

'in' ['reverse'] [binaryBooleanFunction_name] 'order' => postmodifier

For "each", for "every", and for plurals the order of selection of examples may be specified. The order may be specified as '<', '>' or by name. The default is any consistent order.

number_name [unit_np] => measure_np

Each measure has a numeric value and a unit, arithmetic operations and functions can be applied to any measure or number in any combination, Each unit is a postfix operator that is defined by its relation to other units. [This may not be correct.] The unitless case is provided for e.g. 3×3 arrays.

'first' | 'last' | ordinal => property_literal

These are examples of properties. They are defined for all finite ordered types. Ordinals are clumsy to specify in BNF but simple to implement in Lex.

'of' | 'in' | 'above' | 'below' | 'with' | 'in order to' | 'for' | ... => preposition

Some examples of prepositions.

Names

A name is a syntactic primitive describing a value of any type.

literal => name

A value can have a literal name.

(' np ') => name

Any value can be named by a parenthesized expression that computes the value.

name ['.' name] => name

This is a tightly binding form of the preposition 'of' with the arguments reversed. It is also known as dot qualification.

graphic => name

An atomic graphic can be used as the name of the value of a type.

type_name '[' np {' np '}]' => name

An example from a composite type can be named by an application of the constructor

type_name " name => name operation for the type.
The type of an name can be specified
explicitly for disambiguation or emphasis.

'that' | 'it' | 'they' | 'itself' | 'themselves' | 'this' => name
Simple anaphoric and deictic reference is
allowed.

Noun phrases

Noun phrases describe quantified values
from some type. A primary describes the
quantified values from which expressions
are constructed.

name => primary
Any value can be referenced by its name
alone.

quantifier {premodifier} type_literal['s'] [postmodifier] [value_name] => primary
This is the general form for a primary noun
phrase.

type_literal ==> type
The type_literal specifies the type of the
intended value.

premodifier type ==> type
A premodifier may be used to restrict the
subtype of the value.

type postmodifier ==> type
The subtype may be further restricted by a
prepositional or relative phrase.

quantifier type ==> value
All values are quantified examples of some
type

value's' ==> value
The marker 's' indicates that the quantified
value may be plural.

value value_name ==> value
Apposition with the name of the value is
also allowed.

Functions and expressions

A functions is a parameterized side effect
free computation of a value. An expression
is an application of a function, It describes a
value. Functions may be applied to plural
values with the effect of arbitrary, parallel, or
cross product computation.

expression {relop expression} => np
Chained relationals are abbreviations for the
conjunction of the individual relations:
3<x<10 is short for 3<x and x<10.

addop expression => expression	<i>Adding operations may be used in unary form.</i>
term [addop expression] => expression	
factor [mpyop term] => term	
primary [expop factor] => factor	
function_primary '(' np {' np } ')' => primary	<i>Function form syntax can be used for application of any named function.</i>
function_primary name => primary	<i>Function form without the parentheses is allowed for unary functions. Primary is the only left recursive form. This treatment gives notational to functionals.</i>
'(' op ')' => function_name	<i>A parenthesized operator symbol is a function name.</i>
'(' binaryFunction_op np ')' => unaryFunction_name	<i>An infix operator symbol and its right argument can be converted to a unary function value.</i>
expression => primary	<i>Any expression can be used as a primary iff the function of the expression has a result type different from the type of all of its arguments. this makes the grammar ambiguous, but eliminates the need for parentheses where the structure can be determined from the types of the arguments.</i>
leftpara np {' np } rightpara => name	<i>Several parentheses pairs are provided. they may be used to name any function including type value constructors.</i>
'{' np {' np } '}' => name	<i>This form may be used to construct a type from its properties.</i>
'λ' np {' np } '!' name => primary	<i>anonymous functions can be constructed.</i>
<i>Claims and declaratives</i>	<i>Claims have truth values. A declaration asserts that a claim is true.</i>
boolean_np => claim	<i>A boolean expression is a claim by definition.</i>
np vp => claim	<i>A noun phrase verb phrase pair is a claim that the np value is the agent for the vp</i>

np ' <u>△</u> ' np => claim	<p>action or state change, or the subject in the case of copulatives.</p> <p>A definition is a claim that the value of the first argument is defined by the value of the second argument. Either side of a definition may introduce new nonanaphoric parameterized names. If the left side is a literal it is interpreted as a new local nonanaphoric name.</p>
np ' <u>IMP</u> ' np => claim	<p>An implementation is a claim that the value of the first argument can be implemented by the algorithm or data representation of the second argument.</p>
claim => declarative	<p>A claim may be used in a context where a declarative is required to assert that the claim is true.</p>
'whenever' claim 'then' declarative => declarative	<p>Any declaration may have a precondition.</p> <p>Note: we may also want an imperative generalization of this.</p>
{ declarative '.' } declarative => declarative	<p>Claims may be composed in any order.</p>
'[[declarative '.' ']]' => declarative	<p>Declaratives may be parenthesized for disambiguation and emphasis.</p>
<i>Verb phrases and imperatives</i>	<p>A verb is a parameterized side effect causing computation.³ A verb phrase is a partial application of a verb without the agent or patient. An imperative prescribes an action or state change.</p>
noncopverb_name [pp] '(' np {',' np} ')' => ncvp	<p>Function form syntax can be used for imperative application of any noncopulative verb.</p>
noncopverb_name [pp] [primary ['to' primary]] => ncvp	<p>A simpler syntactic form may be used for application of verbs with zero, one or two</p>

³ Vide footnote 2.

ncvp => imperative	arguments. Any noncopulative verb phrase can be used as an imperative with the machine as the agent.
ncvp => vp	Noncopulative verb phrases are verb phrases.
copulative_verh [premodifier postmodifier] => vp	Copulative verb phrases are verb phrases which connect the patient to an attribute.
variable_name '←' np '→' imperative	The assignment imperative has an alternative syntax equivalent to: 'assign' np 'to' variable_name.
'if' claim 'then' imperative => imperative	Any imperative may be conditional.
{ imperative '.' } imperative => imperative	Imperatives may be composed in the sequential order of their evaluation.
'[' imperative '.' ']' => imperative	Imperatives may be parenthesized for disambiguation and emphasis.
declarative => imperative	declaratives can be used in imperative contexts to make assertions about the state at that point in the imperative sequence.
declarative '.' '[' imperative => imperative	Declarations that are true throughout an imperative sequence also can be specified.
'to' ncvp => np	A noncopulative verb phrase can be converted to a noun.
'[' imperative '.' np ']' => primary	The value of this expression is the value of np were the imperative to be executed before np is evaluated. This expression however does not have side effects.

Conjunctions

Conjunctions are used to join two or more instances of the same syntactic form.

premodifier | postmodifier | np | vp | claim => form

Conjunctions can be applied to a variety of syntactic forms.

form ',' {form ','} conjunction form => form

The most general form requires that the

	<i>conjunction be associative. It can be used only with and and or.</i>
form conjunction form => form	<i>The binary form is not restricted.</i>
'and' 'or' 'xor' 'iff' 'implies' => conjunction	<i>Not complements the meaning of a form.</i>
'not' form => form	<i>Forms can be parenthesized for disambiguation and emphasis.</i>
'(' form ')' => form	
Control mechanisms	<i>Several mechanisms are available for any computation. !!! tasks and exceptions may also require some special case syntax!!!</i>
np declarative imperative => computation	<i>There are three classes of computations.</i>
[name ':'] computation => computation	<i>Any computation may be given an anaphoric name. The name labels the primary, the declarative, or the point immediately before the imperative.</i>
'with' context_np 'do' computation => computation	<i>With allows a computation to be evaluated in a foreign context.</i>
'case' {'wher' claim '=>' computation} => computation	<i>A computation with a true claim is selected.</i>
'case' np {'when' np '=>' computation} => computation	<i>A computation with a possible plural associated np that contains all the examples of the first np is selected.</i>
'others' => claim np	<i>can be used after when in case's to specify all other cases.</i>
'for' np 'do' computation => computation	<i>This form allows anaphoric reference to np within the computation.</i>
'quote' computation => comp_name	<i>A computation can be quoted to obtain the complete computational description including the associated context, in unevaluated form. the computation here must be a fully parenthesized form.</i>
'eval' comp_name => computation	<i>the eval operation of one argument will evaluate the quoted computation.</i>

'**r**' computation '**r**' => iris_name *a computation can be quoted to obtain the
computational description as a data value.*
'self' => context_name *self references the current environment.*
'eval' '(' iris_np ',' context_np ')' => computation

*The two argument eval operation will
evaluate an iris value in a given context.*

Sentences

*Sentences are interactive and are acted upon
immediately.*

claim '?' => sentence

Users can ask whether a claim is true.

np '?' => sentence

*Users can ask for the value of any
expression.*

imperative '!' => sentence

*Users may request an immediate action or
state change.*

declarative '!' => sentence

*Users may make assertions in the context of
the interaction.*

Lexicon

Quantifiers: the, a, all, some, no, numbers, any, how many, another

CoordConj: and, or

SubordConj: if, whenever, loop

Preposition: at, of, on, in, above, below, with, in order to, for

Adjective: mass, intransitive, colinear, local, true, false, estimated, first

N: unit, temperature, noun, vt, method, case, candidate, successor, fail

Unit:

Pronoun: it, this, that

Vi: are, is, explain, move, exist, =, \neg

Vt: have, mean, show, try, place, return, raise, move

Vd: move

BinaryOp: +, =, \neq

FuncFormOp: abs*Lexicon*

lexical items: the binding strengths of lexical symbols are given here in roughly increasing	order of binding strength:
. ? !	the sentence constructors.
;	the semicolon separator. semicolon
may be used instead of comma in any form above.	
: \triangle \equiv \Leftarrow when verb when \Rightarrow	the verbs and open form key words. other separators.
,	the comma separator.
and or iff xor implies	the conjunctors.
not	not.
$= \neq < \leq \geq >$	the primary relational operators,
$\approx \neq r < r \leq r \geq \dots$	the secondary relational operators,
$\int \! / \! \int \ e \ \in \ @$	alternative relational operators.
$+ - \vee \bar{\vee} \sim \cup \sim$	the primary and secondary adding
operators,	
$\& \approx \subset$	alternative adding operators,
$" \$ ^ \beta$	other operator symbols.
$\yen / \prod \wedge \bar{\wedge} \ll$	the primary and secondary multiplying
operators,	
$* \setminus f \bullet \blacklozenge \blacksquare$	alternative multiplying operators.
$\boxtimes \boxdot \uparrow$	the exponentiation operators.
func() type[] λ	function form and literal constructors,
quote eval	some function and verb names.
	quantifiers.
of	prepositions and similar words.
$^{\circ}\text{C } ^{\circ}\text{F}$	units specification.
' "	names constructors.
() [] [] []	the primary parentheses.
$\langle \rangle \langle \rangle \langle \rangle \{ \} \{ \} \{ \}$	other parentheses.
""	character string quotes.
` `	program quotes.
alpha{alphanum} digit{digit} "{char}"	identifiers, numbers, string literals, and
icons.	
self \circ • box circle diamond	other literals.
	the character set for strings.

Prismatic Samples

David Mundie

1991 Aug 2

Introduction

The purpose of this report is to illustrate some features of the grammar for Prism currently under development at Incremental Systems. The method used is the technique known in Comparative Literature as textual analysis; by juxtaposing similar texts written in different languages we hope to be able to bring out the similarities and differences between them.

A separate Technical Report ("Unnatural Languages") presents the rationale for some of the design decisions illustrated here.

Sample 1: °C and °F in Ada and Prism

Ada version:

```
function C(F: integer) return integer is
begin
    return (F-32)*5/9;
end;
```

```
function F(C: integer) return integer is
begin
    return (C*9/5) + 32;
end;
```

```
put(C(400)); put(F(-40));
```

Prism version:

"°C" and "°F" are units of temperature.

"Water" and "bread" are mass nouns.

"Boil", "freeze" and "bake" are intransitive verbs.

Water boils at 100 °C and it freezes at 0 °C.

Water boils at 212 °F and it freezes at 32 °F.

Bread bakes at 400 °F. It bakes at how many °C?

°C and °F are colinear?

That is true.

Bread bakes at 200 °C.

Bread bakes at how many °C?

Bread still bakes at 200°C.

-40 °C = how many °F?

-40 °C = -40 °F.

Explain that.

$k1 \text{ °C} + k2 = \text{°F}$.

$k1 \times 100 + k2 = 212$.

$k1 \times 0 + k2 = 32$.

$k2 = 32$.

Prismatic Samples

-3-

$$k1 \times 100 + 32 = 212.$$

$$k1 = 1.8.$$

$$1.8 \times -40 + 32 = -40.$$

Comments:

1. The difference in philosophy between traditional languages and Prism becomes immediately apparent even in a simple example like this. Where in traditional languages the programmer must specify the *implementation* of the solution to his problem, in Prism he is encouraged to engage in a *dialogue* with the system. In the course of that dialogue the programmer conveys information about his problem to the computer, and the computer does what it can to help. In particular, it provides algorithms and data structures which can then be optimized by the programmer.

Thus in the example the Ada programmer had to "presolve" the Celsius/Fahrenheit problem by figuring out the formulæ required. The Prism programmer simply sat down and gave the information he knew, namely the boiling and freezing points in the two systems. The Prism processor uses this information to figure out the conversion formula, after requesting verification of an assumption it has to make about the colinearity of the units.

2. Unlike traditional programming languages, extensive support for and knowledge of units is built in to Prism at the lowest level. The processor knows what it is to be a unit of temperature, and allows the natural-language form of unit notation.

3. The first three lines are a way of entering the new terms into the lexicon. The example is written using double quotes to distinguish between use and mention, although DAF prefers distinguishing the two by context, as is done in most traditional programming languages.

4. Double quotes aside, and barring errors in hand-parsing, it is believed that the Prism Grammar 0.4 will process this example as it stands.

5. Note that the processor is sensitive to the history of the conversation, letting the user know that it has noticed that he has repeated the question ("still".)

Sample 2: Trees in Pascal and Prism

Pascal version:

```

type tree = ^node;
  node = record
    left, right: tree;
    item: string; end;
procedure enter(var p: tree; s: string);
begin
  if p = nil then begin
    new(p); p^.left := nil; p^.right := nil; p^.item := s; end
  else if p^.item < s then enter(p^.left, s)
  else enter(p^.right, s);
end;
procedure print(p: tree);
begin
  if p <> nil then begin
    print(p^.left); writeln(p^.item); print(p^.right); end;
end;
enter("abc", root); enter("father",root); enter("child", root);
enter("man", root); enter("abrecadabra", root);
print(root);

```

Prism version:

"Tree" is a local noun.

Trees are empty or non-empty.

Each non-empty tree has a string and ("left" and "right": tree).

To insert a string into a tree is

 If the tree is empty,

 Make it non-empty. Put the string into it. Make
 its left and right empty. Return.

 If the string = the string of the tree, return.

 If the string > the string of the tree,

 enter it into the left of the tree,

 otherwise enter it into the right of the tree.

To print a tree is

 Print its left, print its string, then print its right.

Prismatic Samples

-5-

"Root" is a local, empty tree.

Enter "abc", "father", "child", "man", and "abrecadabra" into root.

Print root.

Comments:

1. Ordinarily we would expect a Prism programmer to define a sort program by giving a set of sorting axioms and allowing the Prism processor to suggest a data structure and algorithm to do the job. We include this example simply to show what a traditional implementation program *would* look like in Prism.

2. Notice how the need for explicit dynamic record structures dissolves in the presence of the common-sense natural-language concepts of things having components and properties.

3. The Prism version has far fewer names, since anaphoric references ("it", "the tree", "the string") take their place. Names are only needed where there are two items of the same type ("left" and "right").

4. Allowing plural formation, as in the request to enter the strings into the tree, greatly streamlines the notation.

Sample 3: The Eight Queens in Miranda and in Prism

In Miranda:

```
queens 0 = []
queens (n+1) = [q:b | q <- [0..7]; b <- queens n; safe q b]
safe q b = and [~checks q b i | i <- [0..#b-1]
checks q b i = q==b!i \ / abs(q - b!i) = i
```

In Prism:

Queens of 0 \rightarrow [].

Queens of "n": any (>0) integer \rightarrow ["q": each 0..7] & b: queens of (n-1) such that q is safe on b.

Any queen is safe on b: any list \rightarrow not (the queen checks on b at some of (0...#b - 1)).

Q: any 0..7 checks on b: any list at i: any 0..7 \rightarrow q = b_i or abs(q-b_i) = i.

Comments:

1. Here again, this is not the way we would expect a Prism programmer to approach the Eight Queens problem, since by the time these programs are written, the problem has been predigested by a large amount of analysis. The example is included simply to illustrate that Prism can approach Miranda's terseness.

2. Analogously to what we saw in the previous example with records, the use of natural-language-like quantification ("some of (0...#b - 1)") all but eliminates the need for explicit list manipulation. The single explicit list operation uses a syntax which is not part of Prism per se, viz. square brackets for list construction and ampersands for concatenation. The example assumes that this syntax has been declared in some package in the environment. Similarly for the "#" operator.

3. Note the freedom with which the Prism programmer can construct adjectives from predicates: "any (>0) integer," i.e. "any greater-than-zero integer." An equivalent but more cumbersome paraphrase would be "any x: integer such that $x>0$."

Let us examine them in turn.

1. Hill's central thesis seems to be that natural languages are perniciously ambiguous, requiring contextual information for interpretation, whereas formal languages are not. He points to three general sources of ambiguity in natural languages:

- Phrasal lexical items: in "now make the pay half as much again", the phrase "half as much again" could be either the phrasal lexical item meaning 150% or simply a phrase composed of "half as much" and "again".

- Lack of precise binding and scoping rules: for example, group versus distributive plural interpretations of "The operas *Cavalleria Rusticana* and *I Pagliacci* will be performed simultaneously on Radio 3 and BBC2 television this evening.

- Difficulties of anaphora resolution: as in an advertisement for a statistician "to be responsible for: 1. the analysis of experiments on sheep and pigs, and 2. collaborating in their design."

Much of the current work has been devoted to arguing that meaning *must* be dependent on context if Prism is to achieve its goals, so it is clear that we are in fundamental disagreement with Hill on this point. Far from believing that context-dependency leads to a quagmire of ambiguity which would engulf programming languages, we see it as liberating them from an unnecessary straightjacket.

We feel that Hill's criticism stems from an overly simplistic view of programming languages. Hill seems to think that ambiguities simply do not arise in programming languages, because context is not allowed to determine meaning, and that that is a good thing. In point of fact, of course, ambiguities *do* arise in programming languages, and context *does* determine meaning. A good portion of the progress made in such languages over the years can be seen as the introduction of increasingly more sophisticated mechanisms for dealing with ambiguities and allowing ambiguity resolution to be performed using more and more contextual information. To take a simple example, the statement "i = junk" in FORTRAN may actually come close to Hill's ideal; the naming rules are such that from the statement itself one can conclude that i and junk are integers and that assignment is being performed on them. With Algol and related languages, this is no longer true, since the

programmer can establish a declarative scope which must be consulted when interpreting an expression. To avoid ambiguity, such languages impose the rule that there cannot be more than one entity with a given name in the same scope, but that restriction disappears with the polymorphism provided by mechanisms such as Ada's overloading, ML's type inferencing, or the inheritance mechanisms of object-oriented programming, all of which allow ever-larger amounts of context to be taken into account when resolving ambiguities. Seen from this perspective, the proposal that interpretation in Prism should depend on the totality of the context of the expression is not woolly-headed wishful thinking, but rather the logical next step in the evolution of programming languages.

As for the specific sources of ambiguity which Hill cites, we take a non-doctrinaire, engineering viewpoint towards them. Since our aim is not to process natural language in its entirety, we feel free to make engineering trade-offs. Where techniques exist to allow efficient ambiguity resolution, we believe they should be used. For example, anaphora resolution is now close to being a solved problem (ref), close enough at any rate so that the benefits from using it and ridding Prism of the crippling effects of anaphora-free languages far outweigh its costs. Where no such techniques exist, as seems to be the case with the different interpretations of plurals, we have no trouble requiring that the speaker be consistent.

We are not claiming that ambiguity resolution in natural language is easy. What we are claiming is that to conclude, as Hill does, that therefore we should not tolerate *any* ambiguity, is to throw out the baby with the bathwater.

Ironically, Hill actually makes a much weaker case than he could have. The cases he considers by and large involve choosing between several well-defined, preexisting meanings, as with the prisoners in the metal shop learning the art of "copper beating," which depends on the two lexical entries for the word "copper" (viz. the metal and policeman). If this were all there were to it, one could imagine a whole series of ever-more-sophisticated algorithms for choosing among the alternatives, including pun-evaluators which, as in the copper example, leave multiple meanings intact.

But if this is all one does, the result will have the same brittleness, the same lack of extensibility, as current languages. The hard case is not just ambiguity, but rather completely new uses of language. A proud

parent at a spelling bee, when his child takes first place, exclaims "He's a real buccaneer!" This is not an ambiguity; there is no lexical entry for "buccaneer" meaning "good speller". This is a completely new, *ad hoc* use of language, whose correct interpretation depends on the sememe "first place" shared by the man's son and the Pirates baseball team in 1990. As was seen in the chapter on Unification Semantics, it is insufficient to adopt a passive strategy, working on a fixed set of inputs to see which one was meant; rather the Prism processor must actively "make" sense of the utterance, using all the means at its disposal. Once this can be done, ambiguity loses its bite.

2. Sometimes natural language expressions are meant to be taken literally, and sometimes not, but the rules for when to take things literally are hard to specify. Example: some speakers use double negatives as the normal form of negation. From an engineering perspective, we might very well disallow pleonastic double negatives in Prism on the grounds that they provide too little bang for the buck, but this does not prevent us from being in radical disagreement with Hill on his conclusion that programming languages should be forever condemned to strict literalness. We agree that "specifying" exactly when to interpret an utterance literally is a doomed, Sisyphean enterprise; rather, as always the interpretation must be guided by Unification Semantics. You hand a large glass of water to Sam and he shakes his head saying "I do not want no water." You do not interpret this as "I want water," not because you have some hard-and-fast rule about Sam and pleonastic negatives, but rather because it does not integrate into the linguistic context.

3. Natural language does not always obey the rules of Boolean algebra. Example: The substitution of "true" for "not false" in "If this is not false then that must be" inverts the meaning because of the ellipsis. Once again we view this as an engineering problem: the rule should be to handle as much ellipsis as possible. However, we reject for many reasons the claim that the substitution rules for Prism should be the simple substitution rules of purely extensional logic; for one thing, this would mean that the language could not handle intensionality at all.

4. Consciousness is probably a requirement for natural-language understanding. This is close to the conclusion reached by Winograd and Flores in their study of artificial intelligence. It is important to distinguish between two separate claims: (a) Even if a machine displayed complete natural-language understanding, it still would not

count because the machine is not conscious. (b) The implementation of natural-language understanding systems is fundamentally irrelevant, but as a point of fact the only possible implementation given our world is one involving brain cells and consciousness.

We reject claim (a), because we accept Dennett's claim that when I interact with you, the exact details of your brain chemistry are immaterial to the intensional stance I take towards you. It really does not matter to me whether you store the word "banana" using one molecule of acetylcholine or two, and by a garden path argument I cannot therefore care whether you store it using a charge on an iron oxide film. As for claim (b), it may turn out to be true, but we feel it is somewhat premature to judge.

5. Temporality is simple in programming languages, but not in natural languages. Conversational implicature. Hill's affection for the simplicity of good old FORTRAN is touching. He claims that the temporality of computer programs is exhausted by the distinction between code to be executed "later" (i.e. subprogram declarations) and code to be executed "now" (i.e. in-line code). This of course hardly does justice to concurrency in computer languages. His argument rests on the bugaboo of a robot entering a conference, seeing "Take a taxi to the hotel" written on the board, and leaving immediately to follow out the instructions, not realizing that there was an implied "after the meetings". Instead of decrying the sorry state of robotics, where imperatives are reduced to a subhuman, slavish interpretation of "do exactly this immediately," he embraces this idiocy with fervor. We on the other hand hope that we can enlarge the range of speech acts of which the computer is capable, to the point that the normal mode of interaction is not unthinking obedience, but rather dialogue.

6. Bugs are problems not of language, but of logical precision. We confess not to understand Hill's point here. He gives several examples of how illogical thinking can be expressed in natural language, for example instructions for getting to a place by bus which leaves the hapless instructee no closer to his destination than when he started. If we had ever claimed that Prism would prevent one from writing illogical programs, this would be a telling point, but we have of course made no such claim. Clear thinking is a virtue, period. Does Hill believe this shows we should all be writing in assembly code?

7. Legal language shows the folly of trying to make natural language precise. On this and the following point we agree with Hill completely. Despite an amazing recent proposal that litigation should be taken as a

new paradigm for computer programming (ref), we do not take legalese as a reasonable first step from natural languages to programming languages, and for all the reasons Hill avers: the conglomeration of proviso upon proviso, without any consideration such basics of language design as scoping and precedence. But does this mean that no formalization of any kind could succeed? Hardly.

8. Formal languages could improve natural languages. Hill cites a number of cases where a simple mathematical formula is worth a mountain of verbiage. Where word languages such as mathematical notation are good, they are very good, and in fact one of the goals of the Prism interface is to allow the harmonious use of traditional two-dimensional notation alongside linear text. We are by no means advocating "add two to four and put the result in j" over " $j := 2 + 4$ ". To each tool its proper usage.

9. Numbers are not always used mathematically in natural language. Often the units must be inferred from the context. These are just special cases of problems we have already considered. The contrast between "USAir buys 3 jets" and "USAir buys 707 jets" is simply a lexical ambiguity, like "half as much again". Inferring the units from the context is just a specific example of the general problem of dealing with ellipsis.

10. Voice input makes the problem even harder, because of homophones. Hill is on shaky ground here. He is not arguing that extracting information from speech is difficult (it clearly is), but rather that even once it has been extracted, it is inferior to written language. He implicitly assumes that homophones outnumber homographs, which they well may, but he should at least have made the assumption explicit. "Resume Scandal" the headline cries, with an ambiguity absent from a spoken rendition.

The real point, however, is philosophical, and has to do with what is relevant to communication. We view written language as an immensely sophisticated distillate of more general underlying communicative acts, and feel that the larger the spectrum of evidence we can bring to bear on interpretation, the better. A case in point is the example Hill cites: prosody. Recent research in computational linguistics has shown that prosodic information, far from complicating matters, can actually guide parsing (ref). "Mary prefers *corduroy*", "Mary *prefers* corduroy", and "Mary prefers corduroy", where italics convey prosodic stress, each provide information that is useful in

interpreting, viz., whether the question being answered is "What does Mary prefer?", "How does Mary feel about corduroy?", or "Who prefers corduroy?" Certainly, humans acquire language with prosodic information intact, and learning to get along without it in reading is a long drawn-out process. It may well be that Prism systems have to undergo a similar evolution.

11. The state of the art in natural-language understanding is pathetically short of the mark. This again is a point made forcefully by Winograd and Flores, and it is a point with which we have to agree.

Why do we think that Prism can succeed where so much other research has failed? For starters, we are not attacking the natural language problem *per se*. Our goal is more modest. We believe we can succeed because: (1) Get a reason from Shultis. (2) The vast majority of the research in natural-language research has been just that: research directed at the full natural-language problem. Very little of it has been devoted to how to transfer techniques for partial solutions of that problem into techniques useful in programming languages. (3) The computational linguistic community is itself prey to exactly the kinds of interoperability problems and the balkanization that Prism is designed to solve. Just like the computer scientist who devises a hot new algorithm for code optimization, the linguist who comes up with a great new algorithm for, say, avoiding the complexities of conversational implicature faces very high entry costs. He cannot just run off and test his idea; first he must piece together a parser, a semantic analyzer, a data base processor, &c. Once he has expended the enormous effort to construct this test bed, he cannot share it with his colleagues, since they have all run off and implemented incompatible systems. The net result is the same: enormous numbers of ideas wilt on the vine for want of a fertile medium of exchange. It would be a gratifying result if Prism ultimately provided a mechanism for communication among computational linguists as well as among computer scientists.

Signature Grammars

The boundary between what is considered syntax and what is considered semantics is of course variable; one definition has it that syntax is just lower-level semantics, i.e. the unstructured substrate from which semantics emerges (ref?). One can always attempt to capture semantic information through increasingly picayune syntactic categories. To take a hoary example, consider the constraint that the subordinate clause in an "if" imperation must be a statement.¹ Computer science calls statements "boolean expressions", and lumps boolean expressions with arithmetic expressions, so often one has:

`if_statement ::= if expression then ...`

One can however attempt to capture the fact that "if" requires not just an expression, but a boolean expression, in the grammar:

`if_statement ::= if boolean_expression then ...`

There are two problems with this. The first is that it requires doubling a good portion of the grammar, viz. the portion that captures the syntactic commonality of boolean expressions and arithmetic expressions. Alongside productions for normal (arithmetic) terms, factors, etc. we would have to have similar productions for boolean terms, factors, etc. It was considerations of this sort that drove computational linguists to embrace unification grammars.

More serious, however, is the fact that this strategy is, to say the least, inconvenient for user-defined semantics. Suppose I the user introduce new types S and T, variable a of type S, and variable b of type T.

¹ The use of parts of speech indicators in programming languages is an interesting one. Typically what are called "statements" in e.g. Ada are not statements at all, but rather imperatives (what I call imperations). The "statements" are contained in the declarations (cf. declarative verbs) on the one hand, and on "boolean valued expressions" on the other. I confess to being puzzled as to how the confounding of sentences and noun phrases first arose. I suspect it has to do with the feeling that things expressed using symbols must share some commonality. No one would be tempted to place "The dog dwarfs the rat" and "The silver-edged cloud" in the same syntactic category; why are we so willing to do so with "x>3" and "x+3"?

There is now a semantic constraint against writing "a := b". In theory it might be possible to handle this by modifying the grammar to include a "T_expression" production and a "T_assignment" operator, but in practice the only reasonable path is to use a polymorphic assignment operator and to detect semantic constraints independently of syntactic ones.

The Iris-Ada grammar in use at Incremental Systems carries this principle to its extreme. The grammar it uses contains a single non-terminal category

The fact that Prism is a two-way language, with the system itself generating large portions of the program text, means that readability is at a premium. The user will be obliged to understand and maintain large bodies of code that he did not write, so that traditional "write-once" syntax becomes much more undesirable. Moreover, it is not enough just to allow the user to say what he needs to about a computation; he must also be able to say it easily, and not have to shoehorn it into s-expressions or function form. The goal of expressivity requires a notation that is at once powerful and natural.

Exploiting today's display technologies promises to go a long ways towards achieving this expressivity. The use of graphical representations directly manipulable by the user will reduce the asymmetry of the system, since the same representation can be used on input as on output. Improved typography—the use of multiple typefaces and stylistic variants, and possibly even a two-dimensional syntax—will contribute to readability. Coupled with such use of graphics, lexical extensibility will make the notation more natural by allowing it to conform to existing traditions.

Prism's property-based type-system and its use of a generalized feature-structure formalism impose a requirement for a simple and convenient mechanism for talking about properties. This can perhaps be achieved by elevating adjectives and prepositional phrases to first-class status in the language, so that complex compositions of properties can be expressed conveniently.

Prism's view that interacting with a computer is a kind of dialog, its commitment to supporting intensionality and incompleteness, and its avoidance of overspecification, all require a language quite different from the purely extensional languages of the past. The most promising approach is to integrate a number of features from natural language. For example, the use of anaphora, including both pronouns and anaphoric descriptions, is a natural way of expressing context-dependency. The use of variable-free quantification provides a convenient way of avoiding overspecification. The advances made in computational linguistics over the last ten years make us optimistic that these features can be adopted at reasonable cost.

Proceedings of the Workshop on Informal Computing

Santa Cruz, California
1991 May 29-31

Edited by
David A. Mundie and Jonathan C. Shultis
Incremental Systems Corporation

1991 July 11

Sponsored by
Defense Advanced Research Projects Agency
Information Science and Technology Office
Project: "Languages Beyond Ada and Lisp"
ARPA Order No. 6487-1; Program Code No.9E20
Issued by DARPA/CMO under contract number
MDA972-88-C-0076

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Contents

Introduction

Editors' Introduction and Overview

David Mundie and Jon Shultis	1
<i>Reflections on the Informal Computing Workshop</i>	
Catherine Harris	6
<i>The First Workshop on Informal Computing: A Somewhat Personal View</i>	
David Littman.....	7

Wednesday, May 29: Conversational Computing and Adaptive Languages

Opening Remarks

Jon Shultis	9
<i>Natural and Unnatural Languages</i>	
David Mundie	13
<i>Building and Exploiting a User Model in Natural Language Information Systems</i>	
Sandra Carberry	17
<i>Informalism in Interfaces</i>	
Larry Reeker	24
<i>Natural Language Programming in Solving Problems of Search</i>	
Alan Biermann.....	33
<i>Linguistic Structure from a Cognitive Grammar Perspective</i>	
Karen van Hoek.....	41
<i>Notational Formalisms, Computational Mechanisms: Models or Metaphors?</i>	
Catherine Harris	45
<i>Wednesday Discussion</i>	51

Thursday, May 30: Informal Knowledge and Reasoning

What is Informalism?

David Fisher	83
<i>Reaction in Real-Time Decision Making</i>	
Bruce D'Ambrosio	90
<i>Decision Making with Informal, Plausible Reasoning</i>	
David Littman.....	107
<i>Informality and the Epistemology of Computer Programming</i>	
Tim Standish.....	114
<i>Intensional Logic and the Metaphysics of Intentionality</i>	
Edward Zalta	124
<i>Connecting Object to Symbol in Modeling Cognition</i>	
Stevan Harnad.....	126
<i>Thursday Discussion</i>	131

Friday, May 31: Modeling and Interpretation

Mathematical Modeling of Digital Systems

Donald Good.....	155
<i>Ideographs, Epistemic Types, and Interpretive Semantics</i>	
Jon Shultis	166
<i>A Model for Informality in Knowledge Representation and Acquisition</i>	
Timothy Lethbridge.....	175
<i>Friday Discussion</i>	179

Editors' Introduction and Overview

David Mundie and Jon Shultis, Incremental Systems Corporation

The Workshop on Informal Computing was held by Incremental Systems Corporation under DARPA sponsorship from 1991 May 29-31 at the Inn at Passatiempo in Santa Cruz, California. It brought together researchers in computer science, linguistics, psychology, and philosophy to examine the limits of the formalist approach to problem solving and to make an initial attempt at defining ways to overcome those limits. The workshop was an outgrowth of the Prism project at Incremental Systems, a two-year effort to explore ways to eliminate the barriers which current programming languages impose on the software development process.

The participants at the workshop were:

Alan Biermann, *Duke University*
 Yung-O Biq, *San Francisco State University*
 Sandra Carberry, *University of Delaware*
 Bruce D'Ambrosio, *Oregon State University*
 David Fisher, *Incremental Systems Corporation*
 Don Good, *Computational Logic, Incorporated*
 Cordell Green, *Kestrel Institute*
 Stevan Harnad, *Princeton University*
 Catherine Harris, *University of California at San Diego*
 John Kozma, *Tulane University*
 Timothy Lethbridge, *University of Ottawa*
 David Littman, *George Mason University*
 David Mundie, *Incremental Systems Corporation*
 Larry Reeker, *Institute for Defense Analyses*
 Yvonne Rogers, *University of California at San Diego*
 Anton Schwartz, *Stanford University*
 Jon Shultis, *Incremental Systems Corporation*
 Tim Standish, *University of California at Irvine*
 Karen van Hoek, *University of California at San Diego*
 Stephen Wight, *AGFA Compugraphics Division*
 Ed Zalta, *Stanford University*

The present volume is a record of the presentations and discussions at the workshop. The viewgraphs and handouts for each of the talks are included, along with transcripts of the discussion sessions and reflections on the workshop by three of its participants.

Themes of the workshop. Not surprisingly, a good deal of discussion time was spent attempting to articulate the distinction between formal and informal computing. Broadly speaking, three independent though mutually reinforcing views of formalism emerged, each of which leads to a corresponding view of informalism.

According to the first view, formal means *syntactic*; that is, a formal system consists essentially of a finite set of transformation rules applied to a finite set of tokens whose identity is determined solely by reference to their physical shape. Negating this position leads to the view that informalism is essentially semantically-based computation, that the transformation rules in an informal system may depend in an essential way on what the tokens they manipulate signify. This point of view underlies, for example, Littman's insistence that informal reasoning uses task-specific reasoning techniques rather than generic ones, and van Hoek's claim that natural language is inherently encyclopedic, relying on general world knowledge for its processing.

On the second view, most ardently espoused by Harnad, what is essential in formal systems is that they are *systematically interpretable*. By negating this view, a number of the workshop participants came to conclude that informal systems are those which do not have a fixed interpretation, but are rather permanently open to reinterpretation, like the U.S. Constitution or the Ada Reference Manual.

The third view claims that what characterizes formal systems is that they are *complete and consistent*. Negating this leads to a view of informalism as reasoning in the presence of incomplete and inconsistent information. This view was reflected in Standish's catalogue of informal reasoning techniques (probabilistic reasoning, buggy causality, superstition, &c.) and in Biermann's claim that informal means underspecified. Often holders of this view see informalism as a precursor to formalism, as witnessed in Reeker's comments on heuristics, or in van Hoek's charge that a problem with chomskian linguistics is that it indulges in premature formalization, although most participants agreed that reality is not formalizable in general.

One view of formalism that was explicitly *rejected* by the workshop is the one which says formal means machine-processible, although the exact relationship between informalism and computers generated a lot of debate. Standish started this theme off with his early question on the finite symbol system hypothesis, to the effect that intelligent beings, whether organisms or machines, can only exhibit intelligence by means of finite symbol systems transformed by discrete rules, since that is all they have at their disposal. Several participants maintained that informalism is relative, and that a computational system which is purely formal at one level may very well appear informal at a higher level—neural networks being a favorite candidate. Littman even went so far as to suggest that informal computing might not have any informalism in the computer at all—that formal computational support for informal reasoning in humans might be the right place to begin implementing informal systems.

Language Day. The first day of the workshop was devoted to language, as the opening wedge into the subject of informality. The choice of language as a focus was somewhat arbitrary, assuming the increasingly plausible hypothesis that the cognitive apparatus is not modular; we could easily have started with a day about dance, or perception, or emotion.

Informalism draws inspiration and sustenance from the fact that a surprisingly large number of contemporary thinkers have launched nearly simultaneous challenges to the underpinnings of the formalist enterprise, from Lakotos's exposé of formalist mathematics to the critiques of objectivist linguistics by Lakoff, Givón, and Langacker, and Putnam's and Schiffer's attacks on philosophical functionalism. In the opening presentation David Mundie attempted to situate informalism within the context of the history of the philosophy of language and mind, arguing that in some respects the debate between connectionism and symbolic AI can be traced back to the Hellenic era.

An informal computing environment as envisaged, for example, by the Prism project will require radical changes in the nature of the discourse model used for man-machine interaction, and Wednesday morning's speakers all described aspects of those changes. At the end of his talk, Mundie described a programming language developed as part of the Prism effort which incorporates advances in computational linguistics to make the language more natural for humans. Alan Biermann described a similar project he has worked on, called natural language programming, and suggested that such a language would be a powerful front-end for KIDS, the sophisticated formal design environment developed at Kestrel Institute, allowing programmers to express the complex, very-high-level software designs and concepts used in KIDS in a natural way.

Informal computing will require the computer to take on a more active role in the man-machine dialogue, and to be more flexible and adaptive in response to user variation. Sandra Carberry addressed the first of these requirements in her talk, giving an overview of the crucial field of user modeling and then describing her current project to develop computerized consultants based on dialogues driven by models of users' goals and planning strategies. Lary Reeker addressed the second area, explaining how linguistic interfaces can be designed to conform to the idiolects and system models of their users, and how informal visual cues can be used in graphics-based problem-solving systems.

Chomsky's generative grammar approach is the epitome of formalism in linguistics, and has dominated the field for the last thirty years, although as Michael Scriven has said, Chomsky may have gotten out of linguistics and into politics just in the nick of time. Wednesday afternoon saw talks by two cognitive linguists who challenge the chomskian paradigm. Karen van Hoek summarized Ronald Langacker's theory of cognitive grammar, which is based on the notion that language is an instrument of general cognition, and is not a separable module. Although still in its formative stages, cognitive grammar promises to account for many of the phenomena which severely strain generative grammar. Catherine Harris examined the factors which made Chomsky's approach initially appealing, using it as a textbook example of the role of formalism in science, and concluded by placing connectionism in its historical context as a formalism for modeling complex phenomena which are not easily captured by other formalisms.

Discussion 1. The discussion that first day revolved around the question of what "informality" means, and how to characterize it. In the process, the group attempted to define "formality", and found it surprisingly elusive; it is only in retrospect that the three major viewpoints described above became clear. Other important issues that were discussed on the first day were incrementality, lack of specificity, incompleteness, and implementation.

Reasoning Day. On the second day, we broadened the scope of the talks to informal reasoning in general. Once again, there was a great variety of topics, with some common themes running through them.

A good way to think about informal computing is to ask what features we would like to see that current, formal systems do not possess. In the opening talk on the second day, David Fisher enumerated many desirable characteristics of informal systems, and suggest ways in which these might be realized in computer systems. He presented the vision of informal computing that evolved from the Prism project, portraying informalism as a way of extending the problem-solving capabilities of computers. Bruce D'Ambrosio concentrated on one specific limitation of formal systems: their tendency to fail under severe real-time constraints. In a sense, real-time constraints are what force informality on us; given unlimited time and no pressure, we could afford to be as precise and formal as we cared to be about anything. But, we live instead in a state of continuous action, forced to make decisions in the face of incomplete knowledge; we are, as Heidegger put it, "thrown". D'Ambrosio's recent work in decision-making under time constraints attempts to find good approximations to probabilistic decision functions.

As mentioned above, human beings are our best example of informal systems, and the remaining speakers on the second day explored insights to be gleaned from analyzing the way people reason about problems. David Littman talked about ways of forming and refining plans, and ways in which informal reasoning might be used in the exploration of problem and solution spaces. Tim Standish talked about abstraction and reasoning in what he calls the "collateral reasoning domains", which people use to explore problem spaces and formulate solutions. Their utility is that they help to control the conceptual complexity of problems and solutions, while being generally good guides towards solutions. Ed Zalta gave his analysis of intentionality, and of how it can be captured in intensional logic—an important prerequisite if informalism is to break out of the extensional straightjacket. His analysis is noteworthy because it explains, in a formal system, many of the intentional phenomena that have led people to wonder about the formalizability of ordinary language and meaning. Finally, Steve Harnad talked about his work in perception and categorization, and the role of non-arbitrary representations in deformalizing cognitive models.

Discussion 2. The discussion on this day revolved around trying to identify tasks in which informal processing is exhibited or would be useful or necessary. Key concepts that were discussed include approximate representation and reasoning; granularity; the tension between phenomena and their interpretation, on the one hand, and the mechanisms which produce them, on the other; concept formation; grounding; interpolation and extrapolation; the advantages of informal reasoning for controlling the complexity of a problem space; the methodological recommendation to tackle informality in specific domains and tasks; incrementality;

and adaptability (and the importance of parameterized theories as a formal version of concept adaptation).

Modeling and Interpretation Day. A constant theme of the workshop was that any discussion of formalism or informalism very quickly leads to questions of modeling and interpretation, and on the last morning there were two talks on that subject. The first talk, by Don Good, recast recursive function theory in the role of the mathematics used to model digital systems. Don pointed out that a digital system can be interpreted and understood at many different levels, and he explained how the mathematics of recursive functions could be used to map between the levels and to show that these mappings guarantee that the properties of the system at each level are faithfully reflected in the other levels.

Jon Shultis wrapped up the talks with an account of modeling and interpretation, and explored a spectrum of representations, from abstract syntax to grounded connectionist networks, as points in a continuum of model types, comparing and contrasting them along several dimensions of importance for cognitive modeling.

A third talk on modeling and interpretation, by Jeff Rothenberg, regrettably had to be canceled.

Discussion 3. The discussion on the last day consisted of a small, informal experiment, in which the participants attempted to solve a problem, and later analyzed what they were doing as a means of generating some concrete examples of informal reasoning.

Acknowledgments and Disclaimers. The workshop organizers would like to thank all of the participants, and especially the speakers, for contributing their time and talent, and for making this a most interesting and informative workshop. Carmen Hoecker deserves special thanks for doing all of the really hard work of local arrangements, and for taking care of the many, many details and problems that inevitably accompany any such event.

Funding for the workshop was provided by the Defense Advanced Research Projects Agency (DARPA), Information Sciences and Technology Office (ISTO) of the United States Department of Defense, under contract MDA 972-88-C-0076, entitled "Languages Beyond Ada and Lisp".

The transcripts of the discussion sessions have been lightly edited to improve readability, but due to time pressures have not been reviewed by the speakers, and should therefore not be taken as an accurate account of what was said.

Reflections on the Informal Computing Workshop

Catherine Harris, University of California at San Diego

The workshop had the side effect of being a consciousness-raising session about the nature of informality. Pre-workshop, I thought of informality as evidence of lack of rigor or lack of clear thinking, which might be necessary in the early stages of work, but which was something a researcher should be embarrassed by, something to play down in published work. This attitude is widespread in mainstream scientific thought. I just read a review of sociologist Bruno Latour's *Science in Action*. He shows that once the formalism is worked out or the molecule structure is identified, scientists rewrite the immediate history of the fuzzy, trial-and-error or other "informal" steps leading up to the discovery. Post-workshop, I've felt on a few occasions that it would be beneficial to describe the informal steps that I was taking to explain a currently mysteriously phenomena. Furthermore, I had the urge to do so *without apology*, to simply inform readers that the description was informal, and that although the phenomena might be amenable to descriptions by formalisms a, b or c, each was inadequate in various aspects. The problem with apologetic presentations of informal ideas is that the apology often takes the form of either de-emphasizing the informal idea, or trying, as much as possible, to dress up the new thoughts in the trappings of established formalisms (using jargon terms, selective description). Other consciousness raising along similar lines: I've read recently a few papers where scientists note upfront that what they are doing is not susceptible to complete, precise description, but that this doesn't mean science is impossible. An example is Marian Dawkin's 1990 BBS article "From an animal's point of view: Motivation, fitness and animal welfare".

The First Workshop on Informal Computing: A Somewhat Personal View

David Littman, George Mason University

This short note is intended to convey my impressions of what I consider to be a very interesting, successful workshop on the difficult topic of informal computing, hosted by DARPA and Incremental Systems. When we all sat down at the conference table the first day, I believe that we had different ideas about what informal computing is. Over the course of the three-day workshop, two camps—very friendly to one another I must point out—emerged. The two camps might be informally labelled the “philosophical camp” and the “psychological” camp. I, a computer scientist and cognitive psychologist, count myself in the latter and the remainder of this note reflects that fact. The questions that I had in my mind from the time that I was invited to the workshop until this moment are these: “How do we develop a theory of the representations and reasoning processes that humans use when they reason informally?” and “How can we build artificially intelligent computer programs that can 1) help humans reason informally and 2) reason informally themselves?” Let me give an example. I have worked for several years on the problem of trying to understand how novices and experts do problem solving tasks for which they do not have an off-the-shelf, stock solution. Without a stock solution, a problem solver is almost invariably forced to fall back on what we call common sense, or background knowledge. For example, both Tim Standish and I have devoted considerable effort to understanding how computer programmers figure out how to solve problems that they have never solved before. We both have come to the conclusion—along with others who study programmers—that in such situations programmers think about how to solve the problem using what can only be called informal reasoning. For example, a student who is just learning to write programming loops might imagine how he or she would control the actions that are supposed to be done in the loop if he or she were responsible for controlling the execution of the loop statements. The transformation of this informal representation of the loop into a semantically and syntactically correct loop is a significant problem solving accomplishment and it is crucial, in my opinion, to keep clearly in mind that the long path to the student's working loop started off with informal reasoning. Tim Standish talked about several fascinating informal reasoning techniques that novice and expert programmers use and I discussed some of the informal reasoning processes that I have seen in my studies of robot designers. This thread of activity at the workshop convinced me that we really can have a science of informal computing and that such a science should, initially, dispense with philosophical arguments about whether it is possible to formalize informalism (I think that it is); whether we can

have computer programs that reason informally (I think that we can) and the like. Indeed, I believe that one of the major positive outcomes of the workshop on informal reasoning was that there really IS a domain of study that can be called informal computing. Instead of obsessing about such philosophical conundrums we should, in my opinion, set about our research empirically. We should attempt to identify the representations and reasoning processes that humans use—and machines might use—to engage in problem solving that is based in informalism. Heuristically, this type of problem solving is almost always seen in two situations: First, during the initial phases of problem understanding and solution generation and, second, during the acquisition of problem solving skills. Many of the efforts to build intelligent educational software and intelligent problem solving environments could, in my view, take great advantage of and contribute to such work. This is where, I am convinced, we should direct our efforts and, if I have my say, this topic will be the focus of the Second Workshop on Informal Computing!

Opening Remarks

Jon Shultis

Logicians have too much neglected the study of vagueness, not suspecting the important part it plays in mathematical thought.

– C. S. Pierce

[T]his bleak alternative between the rationalism of a machine and the irrationalism of blind guessing does not hold for live mathematics: an investigation of informal mathematics will yield a rich situational logic for working mathematicians, a situational logic which is neither mechanical nor irrational, but which cannot be recognised and still less, stimulated, by the formalist philosophy.

– Imre Lakatos, Proofs and Refutations

We all realize that we cannot hope to mechanize interpretation. The dream of formalizing interpretation is as utopian as the dream of formalizing nonparadigmatic rationality itself. Not only is interpretation a highly informal activity, guided by few, if any, settled rules or methods, but it is one that involves much more than linear propositional reasoning. It involves our imagination, our feelings — in short, our full sensibility.

– H. Putnam, The Craving for Objectivity.

... the meaning of an expression is not determined in any unique or mechanical way from the nature of the objective situation it describes. The same situation can be described by a variety of semantically distinct expressions that embody different ways of construing or structuring it. Our ability to impose alternate structurings on a conceived phenomenon is fundamental to lexical and grammatical variability.

– R. Langacker, Foundations of Cognitive Grammar I, p. 107

Facts just twist the truth around.

– David Byrne

Welcome to the Workshop on Informal Computing — the *first* Workshop on Informal Computing! I am very excited to be here, and gratified that such an impressive and enthusiastic group was willing to come here on such short notice.

Why are we here? We are here because we sense that a good deal of human understanding and communication is informal, and that informality underlies much of our ability to adapt and function in the real world.

We want to understand this. What is it? What does it tell us about ourselves and our world? How can we have a science about it? What part does it play in scientific understanding? And finally, how can we build machines that are more like us? Machines that sense, and reason, and communicate, and create, and feel, the way we do? Informally...as well as formally.

In one way, our being here speaks of growing dissatisfaction with the standard litany of formal software development. It is an acknowledgement of the criticisms that have been leveled over the past decade or so at the widespread assumption within computer science and related fields that, because computers *can be described* as formal symbol manipulators, that that is *what they are*. This explains why, to most computer scientists, the phrase "informal computing" rings loudly of the absurd; for them, it is an oxymoron. But even though a screwdriver *can be described* as a device for applying simultaneous torque and pressure to the head of a screw, it is also handy for prying open lids and gouging small holes in things.

Preconceptions like this have far-reaching consequences. Consider Gödel's celebrated incompleteness theorem, which I believe is more often cited than studied. It reads:

For every ω -consistent recursive class κ of FORMULAS there are recursive CLASS SIGNS r such that neither $v \text{ Gen } r$ nor $Neg(v \text{ Gen } r)$ belongs to $Flg(\kappa)$ (where v is the FREE VARIABLE of r).

Notice that the theorem applies to a very special kind of mathematical structure: ω -consistent recursive classes of formulas. Notice also that the undecidability of $v \text{ Gen } r$ pertains only to the closure of κ under the relation of immediate consequence (that is what $Flg(\kappa)$ denotes). If these restrictions are lifted or even just relaxed somewhat, the conclusion is undermined. Yet the Church-Turing thesis, coupled with the assumption that the universe is reducible to formal terms, makes these restrictions seem benign because inevitable.

This is really quite remarkable. It is as though a geometer were to conclude, upon seeing a demonstration of the impossibility of trisecting an angle with a straight edge and compass, that it is impossible to trisect an angle at all! Yet this is exactly the kind of inference which is invoked to support the symbolic thought hypothesis, which has dominated AI for over 30 years.

Informal computing is mandated by the need to make computers more responsive to human needs in the context of the real world, and the failure of formalism (rationalism and analytic philosophy generally) as a theory of reality, human under-

standing, and language. We represent the growing number of people who have come to accept that, not only are natural language, understanding, and reasoning unlike their formal counterparts, but they are intrinsically more valid and faithful to reality, and real purposes.

Goals

We are here to give birth to a new discipline: the study of the informal, not as something to be repaired, replaced, or reduced, but on its own terms.

The goals of this workshop are programmatic. We want to establish ourselves as a community in which we can have discussions and make progress unhampered by the need to explain or argue about the basic problems, issues, and validity of informality. By doing this, we hope to bring greater focus to all our efforts, and accelerate the pace of research. Once a body of strong results and successes begins to appear, the broader scientific community will become more receptive to our philosophical arguments. For now, we need to spend less time defending ourselves to the rest of the world and get on with it.

For this community, we want to

- set goals,
- identify key problems and approaches:
 - long,
 - medium,
 - short term
- establish methodological principles for
 - theory,
 - experiment
- initiate a series of publications, meetings, etc.

Program Notes

Themes

- Adaptive languages and conversational computing: this is the first place where we see a strong impulse to depart from rigid, formal interfaces.

- Informal reasoning and knowledge: natural language is just one manifestation of general cognition, and is inextricably involved with pragmatics, epistemology, robotics, analogy, metaphor, and other forms of informal reasoning. Basically, the doctrines of pragmatism and meaning holism imply that natural language is not separable from general intelligence, so we can't have adaptive languages and conversational computing without informal reasoning and knowledge.
- Modeling and interpretation (hermeneutics): these are central to implementation — the computational/physical/mechanical realization — of the information processing components of cognitive agents.

Approaches

- broken formalisms
- emergent cognitivism/epiphenomenalism
- intentionality/dialog/hermeneutics

Discussion goals

- Clarify, reconcile, and consolidate approaches within and across themes on successive days.
- Research agenda: goals, problems, program, and plans.
- Stimulate discussions leading to cooperative research among participants.

Natural and Unnatural Languages

David A. Mundle
Incremental Systems
310 South Craig Street
Pittsburgh PA 15213-3720

Trail Map



Historical Perspective

RATIONALISTS	PRAGMATISTS	EMPIRICISTS
•Pythagoreans	•Kant	•Aristotle
•Plato	•Pierce	•Crates
•Aristarchus	•James	•Locke
•Descartes	•Husserl	•Berkeley
•Leibniz	•Heidegger	•Hume
•Fregoe	•Wittgenstein	•Quine
•Russell	•Putnam	•Churchland
•Chomsky	•Lakatos	
•Todor		

What does software engineering have to do with the philosophy of language?

What is formalism?

Curry:

Discrete symbols manipulated by a finite set of rules

Symbols differentiated by formal features

Syntax

Unambiguous syntax and semantics

Machine-processible

Axiomatic (complete and consistent)

Invented

Unnatural

The long sighs of the violins of autumn injure my heart with a monotonous languor.

1849 JRA 88\$ 4*(R3322()*(1)40\$

Three Dogmas and a Claim

Dogmas

The mind is like that
The world is like that (objectivism)
Computers are machines to process formal systems

"It is of the essence of being a machine, that it should be a concrete instantiation of a formal system." ~J.R. Lucas, "Minds, machines, and Gödel"

Claim

Formal systems do not exist

The Collapse of Functionalism

Aristotelian tradition:

Words are associated with mental representations
Synonyms have same mental representation
The mental representations determine reference

Functionalism: mental events are classified by their causal roles, independent of their physical constitution

The Collapse of Functionalism (Cont'd)

The qualia objections: inverted, absent
Heidegger, Ryle, Dreyfus & Co.: some information remains in the environment and is not represented

Ants in their tunnels

The world is the best of all databases

Schiffer: The Remnants of Meaning

The Gricean IBS program: reduce token-meaning to speaker-meaning, then speaker-meaning to physics

Upshot: Ontological physicalism, sentimental dualism

Putnam: Representation and Reality

Two reasons mentalism cannot be right

Meaning is holistic

Meaning depends on general intelligence
Principle of charity: meanings are preserved under normal belief fixation

Division of linguistic labor

The contribution of the environment

Putnam's "Internal Realism"

Physicalist accounts of the world are incomplete

Meanings are not "in the head"

Kant: we are unlikely to be able to give an account of "schematism" in natural scientific terms

For Brentano, Husserl: "scientific" meant "mechanics" of the brain, associationist psychology of "ideas"

Today: "scientific" means computer science

Intentionality

Not a relation, because of non-existent referents

Frege: sense vs. reference

Formal models make the problem hard to solve, because they pass over the world (Heidegger)

Closest they come: causal theories

But: error; misrepresentation; nonexistent objects

Problem: symbols are atomic and arbitrary

There is nothing about a symbol which determines its referent

Another symptom: context freedom

Connectionism and Intentionality

Mode of processing within the system is continuous with processes occurring in the external world (cf. Harnad)

Elucidates Heideggerian thrownness

In recognizing regularities and categorizing, leads to failure of substitutivity

Connectionism and Intentionality (Cont'd)

Representations not arbitrary, but rather "chosen" as part of learning in interaction with the world

Teleology results from accommodation to environment

Not context-free: no assumption that the world consists of a body of facts

Nonexistent objects: activations brought about by activity in the network itself

Getting there from here

Full-blown NL interface not feasible

Major stumbling block: metaphor

Nonetheless, computational linguistics has made progress towards handling context-dependency

Anaphora resolution

Conversational implicature

Speech act theory & user modeling

Lexical structure a la Pustejovsky

Proposal: incremental engineering of progressively more intentional systems

The Awfulness of Programming Languages

What happened here?

Choose thirteen cards from a standard deck. Sort them in descending order, then print them on a Laserwriter.

print(sort((choose-cards(13, random-order), descending, " ", 3), true, 1w, -1);

History of programming languages is towards more context-dependency

Desirable Properties of Natural Languages

Nonphrasal lexical items

Extension through juxtaposition (interoperability)

Anaphora

Sam bought a bright blue Cadillac with power steering. Then he drove the bright blue Cadillac with power steering downtown to get floor mats for the bright blue Cadillac with power steering and to have the bright blue Cadillac with power steering rustproofed.

Let the contents of the pot become the contents of the pot plus an onion.

Quantification without variables

More Desirable Properties of Natural Languages

A type system based on properties & reflected in nominals

Nouns

Adjectives (both predicate and nominative)

Quantifiers

Grounding terms

Frequent use of juxtaposition

Infrequent punctuation

Recursion without nesting

the(crisp(fresh(juicy(red(delicious(apple))))))

Support for incompleteness, inconsistency, underspecification

Discourse structure making H&U tolerable

Building and Exploiting a User Model In Natural Language Information Systems

By

Sandra Carberry
Department of Computer Science
University of Delaware
Newark, Delaware

INFORMATION SEEKING DIALOGUE

1. Participants

Information-Seeker

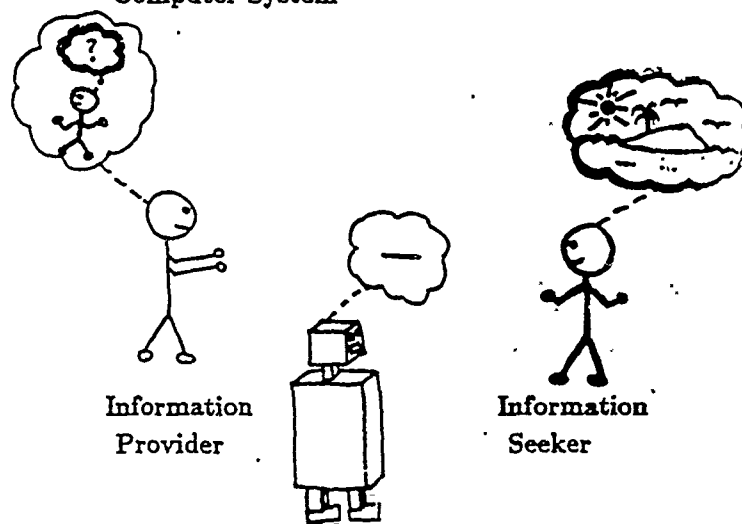
Information-Provider

2. Underlying Task

3. How is dialogue understood and used

Human Being

Computer System



Modeling the User

1. Beliefs and knowledge: Kass, Ballim&Wilks

2. Level of expertise: Wilensky, Chin, Crosby&Stelovsky

3. Preferences

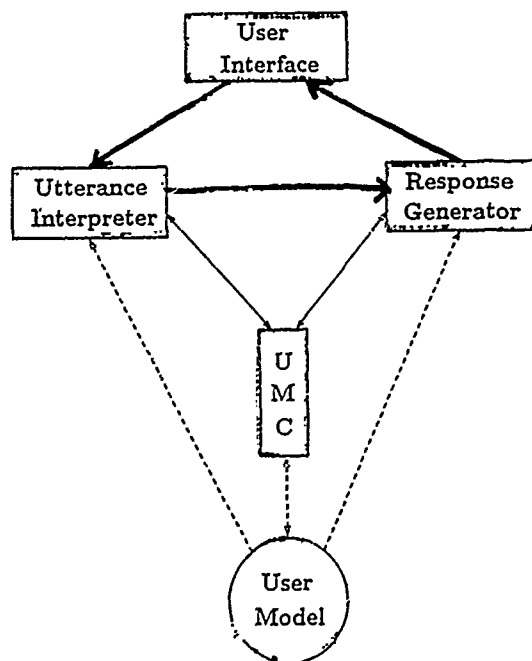
4. Information receptivity

5. Goals and Plans

A. Domain: Schank, Allen, Carberry, Pollack
Lochbaum&Grosz&Sidner

B. Plan-construction: Wilensky, Litman, Ramshaw

C. Communicative: Lambert&Carberry



Exploiting a User Model

Understanding

Indirect speech acts: Allen
Ill-formedness: Carberry, Ramshaw
Ellipsis: Allen, Carberry, Litman

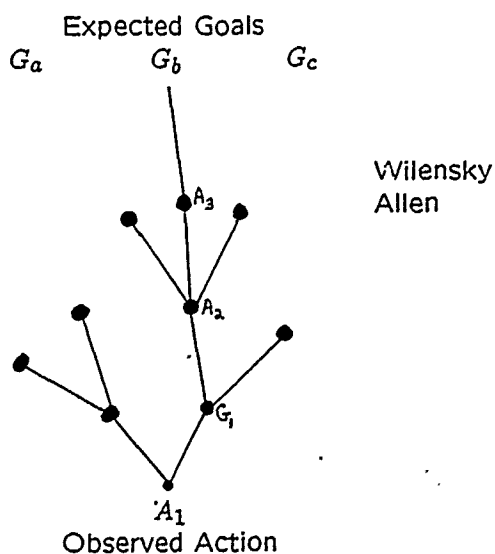
Response Generation

Extended responses: Allen, McKeown,
Joshi&Webber&Weischedel,
van Beek&Cohen
Misconceptions: Pollack, Quilici
Descriptions: Paris
Bateman&Paris
Definitions: Sarner&Carberry

Outline

1. Modeling the user's plans and goals
2. Understanding pragmatically ill-formed utterances
3. Generating tailored definitions
4. Problems and research directions

Overview of Plan Recognition



Incremental Plan Inference

- Build a model of the user's plan incrementally as the dialogue progresses
- Maintain global and local context
- Domain-independent reasoning strategies
- Domain-dependent knowledge of goals and plans

Hierarchical Plan Representation

- A. Applicability Conditions
- B. Preconditions
- C. Body
- D. Effects

Sample Plan: Declare-Major(_agent, _dept)

Applicability Conditions:

Admitted-University(_agent)

Preconditions:

Have-GPA(_agent, _ave)

where $GE(_ave, 2.2)$

Have-Interview(_agent, _fac)

where Undergrad-Advisor(_fac, _dept)

Body:

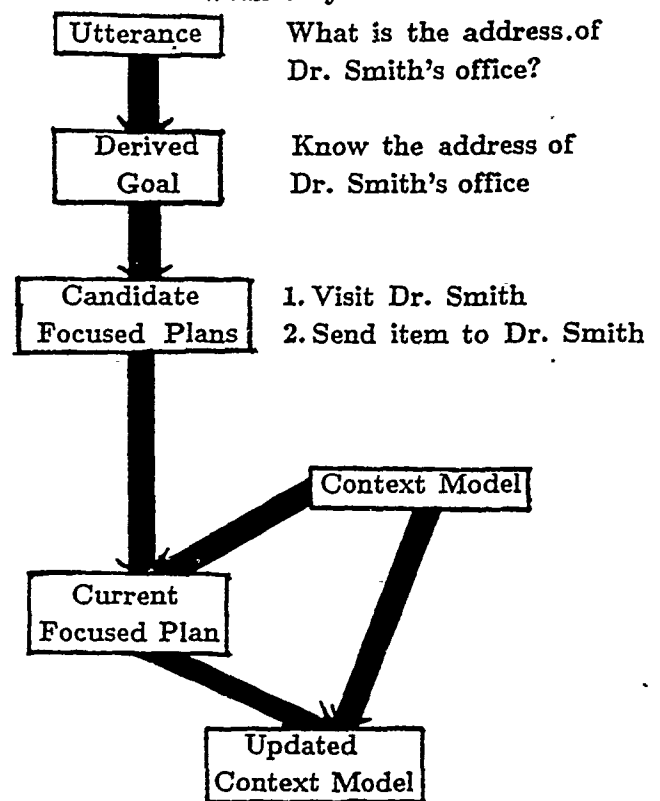
1. Obtain-Acceptance(_agent, _dept)

2. Submit-Change-Form(_agent, _dept)

Effects:

Declared-Major(_agent, _dept)

Wednesday Presentations 19



Plan Identification Heuristics

- If the user wants to know the values of a term for which a proposition P is true, then those plans containing P or $\neg P$ are candidate focused plans.

Sample Plan: Declare-Major(_agent, _dept)

Applicability Conditions:

Admitted-University(_agent)

Preconditions:

Have-GPA(_agent, _ave)

where $GE(_ave, 2.2)$

Have-Interview(_agent, _fac)

where Undergrad-Advisor(_fac, _dept)

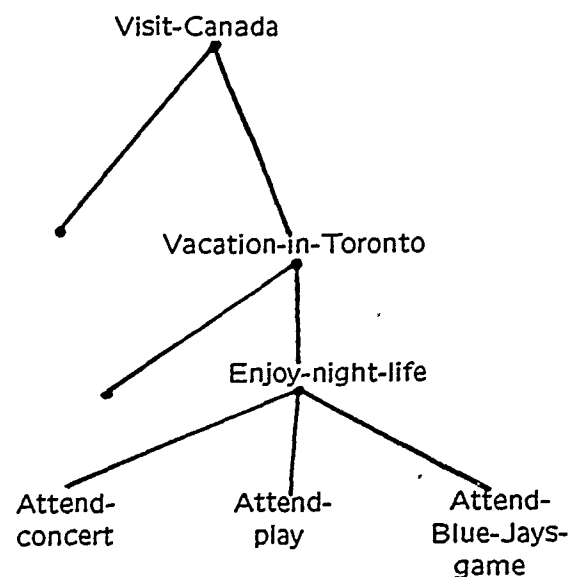
Body:

1. Obtain-Acceptance(_agent, _dept)

2. Submit-Change-Form(_agent, _dept)

Effects:

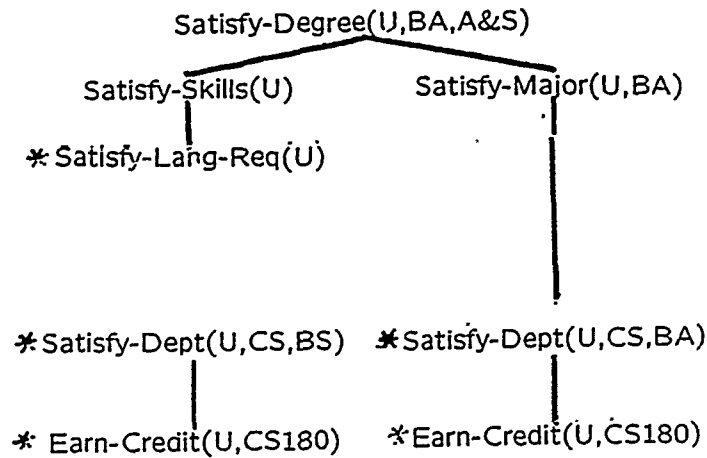
Declared-Major(_agent, _dept)



"I want to major in computer science" CS180?"
"What are the prerequisites for taking CS180?"

Wednesday Presentations 20

"What courses must I take to satisfy the foreign language requirement?"



Pragmatic Ill-Formedness

Utterance is

1. syntactically and semantically well-formed
2. violates pragmatic rules of world model (Intensional violation)

Examples

U: "I'd like to own my residence,
but I don't like a lot of maintenance.
Which apartments are for sale?"

Sale-Status(x:Apartment, For-Sale)

U: "Who is teaching CS105?"

S: "Dr. Smith, Dr. Brown, and Dr. Jones."

U: "When's Smith meet?"

Meet-Time(Smith, t:Time)

Causes of Pragmatic Ill-Formedness

1. Improper use of language
2. Short-cut use of language
3. Erroneous beliefs about the world

Criteria For Responding

1. Discrepancy between speaker and listener beliefs
2. Seriousness with which discrepancy is viewed
3. Faith in correctness of own beliefs

Ways of Responding

1. Explicitly correct speaker's misconceptions
2. Negotiation dialogue to "square away" discrepancies
3. Address speaker's perceived intent in making the utterance

1. GOAL: Address speaker's intent

2. MOTIVATION

- A. Gricean Theory of Meaning
- B. Gricean Maxim of Relevance

3. STRATEGY

- A. Infer speaker's underlying task-related plan from the preceding dialogue
- B. Use inferred plan to suggest substitutions for the erroneous proposition
- C. Evaluate suggestions on semantic and relevance criteria

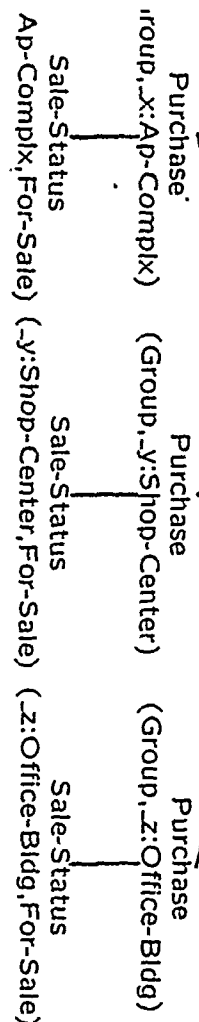
"We'd like to invest between 50 and 80 million dollars."
"Which apartments are for sale?"

Sale-Status(_x:Apartment, For-Sale)

Expand-Investments(Group)

Precondition:

Want-Invest(Group, _amt:Money)
where GE(_amt:Money, \$50,000,000)



Suggestion Mechanism

Inferred plan suggests substitutions for the erroneous proposition.

1. Simple Substitutions

A. Object class substitution

U: "We'd like to invest between 50 and 80 million dollars."
Which apartments are for sale?"

Sale-Status(_x:Apartment, For-Sale)

Enroll-Section(U, _s:Section)
where Is-Section-Of(_s:Section, CS105)

Learn-Material(U, _s:Section)

Learn-From(U, _s:Section, _f:Faculty)
where Teach(_f:Faculty, _s:Section)

Attend-Class(U, _p:Cls-Room, _tm:Time)
where Meet-Place(_s:Section, _p:Cls-Room)
Meet-Time(_s:Section, _tm:Time)

Meet-Time(Smith, _t:Time)

Teach(Smith, _s:Section) *

Meet-Time(_s:Section, _tm:Time)

Suggestion Mechanism

Inferred plan suggests substitutions for the erroneous proposition.

2. Expanded Path Substitutions

U: "Who are teaching sections of CS105?"
S: "Dr. Smith, Dr. Brown, and Dr. Jones."
U: "When's Smith meet?"

Meet-Time(Smith, _t:Time)

Selection Mechanism

Evaluate suggested revised queries

1. Similarity

- (a) Operation producing substitution
- (b) Semantic similarity of terms

2. Relevance

- (a) Shift from current focus of attention

- Process model for interpreting pragmatically ill-formed queries
- Based on Gricean theory of meaning and maxim of relevance
- Relies on established dialogue context as primary mechanism for suggesting revised queries
- Uses semantics and relevance to evaluate suggested revisions and identify appropriate interpretations
- Identifies user intent or at least satisfies perceived needs

Generating Tailored Definitions

Factors Influencing Definitions

- 1. User's domain knowledge
- 2. User's receptivity to different kinds of information
- 3. Existing dialogue context

"What's baking soda"

Tailored Definitions

"What's baking soda?"

Make-light-texture(U, _x:B-GOOD, _y:MIXTURE)

Add(U, _z:RISING-AGENT, _y:MIXTURE)

where Acidity(_z:RISING-AGENT, BASIC)

Make-light-texture(U, _x:B-GOOD, _y:MIXTURE)

Precondition:

Acidity(_y:MIXTURE, ACID)

Effect:

Light-texture(_x:B-GOOD)

"Baking soda is a basic rising agent. If the mixture is acidic, then adding baking soda will make the texture light."

Critical Assumptions

1. All domain goals are a priori equally likely
2. The user's knowledge is incomplete but not erroneous (Exception: Pollack)
3. The user's statements are correct and not misleading
4. The system's inference mechanisms do not introduce errors
5. The user's queries address aspects of the task within the system's limited knowledge

Is this realistic?

Problems and Current Research

Default Inferences:	Carberry
Probabilistic model:	Goldman&Charniak
Abduction:	Konolige&Appelt&Pollack
Disparate models:	Eller&Carberry
Novel plans:	Kass, Pope&Carberry
Problem-solving plans:	Ramshaw
Communicative plans:	Lambert&Carberry
Collaboration:	Lochbaum&Grosz&Sidner

Summary

1. Plan Inference

- Incremental as dialogue progresses
- Exploit model of user's plan for robust understanding and cooperative response generation

2. Other aspects of a user model

- Beliefs and knowledge
- Preferences
- Information receptivity
- Level of expertise

Larry Reek, Informalism in Interfaces WHY ADAPTIVE?

1. BROAD RANGE OF USERS (NEEDS VARY)
2. NATURAL OR ARTIFICIAL LANGUAGE
3. PERSONALIZED COMPUTER
4. AUTOMATIC USER CUSTOMIZATION
5. HABITABILITY / MACHINE MEMORY

EXPERIMENTAL EVIDENCE (JILL LEHMAN)

- 1) Consistent single-user interaction.
- 2) Differs significantly among users.
- 3) Improves performance.

(CHAMP SYSTEM)

TWO ADAPTIVE INTERFACE DESIGNS:

CHAMP (LEHMAN)

parse by adaptive

parser

deviations cause changes

Semantics in grammar

meaning determined

by least deviant

parse

AID (REEKER)

Parse by (partial) chart

parser

Deviations lead to transformations

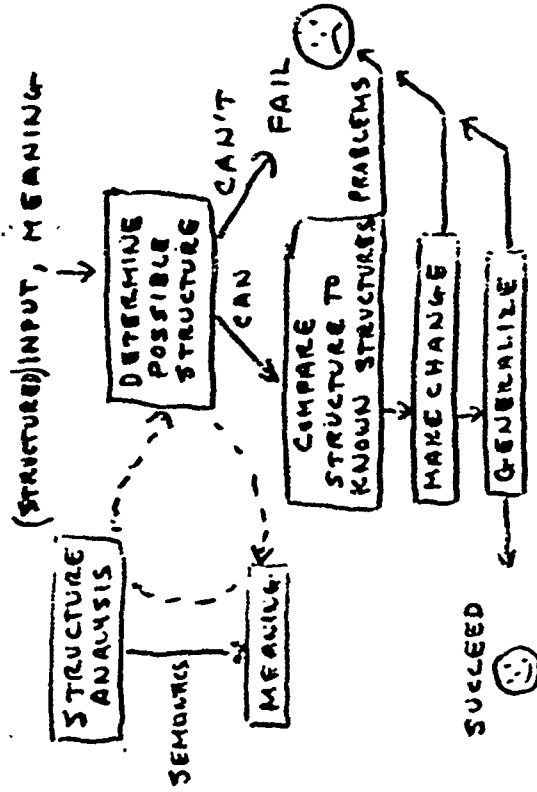
Semantics in grammar + transformations

meaning determined

by heuristics or

dialogue.

STRUCTURAL LEARNING BY SUCCESSIVE APPROXIMATION



DIALOGUE WITH AN EXPERT SYSTEM ON INSECT IDENTIFICATION

(Demonstration for AFHRL/ID)

USER: "Feelers are small"

SYSTEM: (Not recognizing "feelers", but achieving a partial match on "small")
"DO YOU MEAN 'FRONT WINGS ARE SMALLER THAN HIND WINGS'?"
[PRESS SPACE BAR TO ANSWER 'YES'; PRESS RETURN KEY TO ANSWER 'NO']

USER: (Presses RETURN key)

SYSTEM: (Not finding any other close matches)
"THE MOST LIKELY ASSERTIONS AT THIS STAGE ARE
_HEAD IS PROLONGED TO FORM BEAKLIKE STRUCTURE
_SHORT ANTENNAE
_HAS SUCKING MOUTHPARTS."

IF ONE OF THESE MATCHES, PLACE 'X' NEXT TO IT."

USER: (Places 'X' next to "short antennae")

(It is Necessary to Determine What the User Means - In the System's Terms - before Learning Can Take Place. This can be Determined by Querying the User Based on Various Associations - of Lexical Items, Synonymous Lexical Items, etc. - or by Metrics of "Likelihood" with User Choice.)

"DOES 'SMALL' ALWAYS MEAN THE SAME AS 'SHORT'?"

PRESS SPACE BAR TO ANSWER 'YES'; PRESS RETURN KEY TO ANSWER 'NO'."

USER: (Presses space bar)

SYSTEM: "DOES 'FEELERS' ALWAYS MEAN THE SAME AS 'ANTENNAE'?"

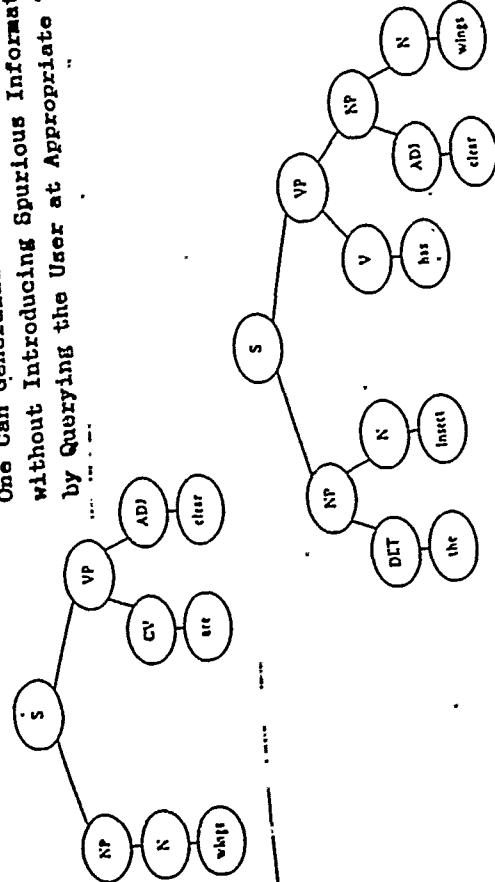
[PRESS SPACE BAR TO ANSWER 'YES'; PRESS RETURN KEY TO ANSWER 'NO'."

USER: (Presses space bar)

CONTINUED ON NEXT PAGE

(The User is Queried to Determine Identity among Lexical Items. Note That These May Not Be the Same as Semantic Identities within the Language as a Whole. They May be Identities relative to (1) the System Context and (2) the User.)

One Can Generalize Transformations without Introducing Spurious Information by Querying the User at Appropriate Times



(Pictures from manuscript "Machine Learning for Information Retrieval", by Dr. Roy Anderson, National Library of Medicine, describing ML adaptive interface proposed by Dr. Larry Rasmussen, IBM Corporation.)

SYSTEM: "IS 'FEELERS' ALWAYS GOING TO MEAN 'SHORT ANTENNAE'?"
[PRESS SPACE BAR TO ANSWER 'YES'; PRESS RETURN KEY TO ANSWER 'NO'"]

USER: (Presses space bar)

SYSTEM: (Finding that "feelers" is a noun in its second-stage lexicon, and already having available the fact that "small" is an adjective in its first-stage lexicon, since it is used in other assertions, it

S/NP/N/feelers/VP/COP/are/ADJ/small/]]

and compares this to the parse

S/NP/ADJ/short/N/antennae/]]

It then formulates a possible transformation of

S/NP/N/VP/COP ADJ/]]

to

S/NP/ADJ N/]]

(In Order to Learn a Transformation, the System Needs to Make Sure that No Mistake Has Been Made, as "Unlearning" is More Difficult... Once a Transformation Involving Particular Lexical Items Has Been Learned, it is Then Important to Consider a More Generalized Version.)

... DIALOGUE CONTINUED

USER (sometime later): "Feelers small"

SYSTEM: (Uses its knowledge to transform this to "Feelers are short", then to "Short antennae")
"IS 'FEELERS' SMALL" TO BE USED TO MEAN 'SHORT ANTENNAE'?"
[PRESS SPACE BAR TO ANSWER 'YES'; PRESS RETURN KEY TO ANSWER 'NO'"]

USER: (Presses space bar)

...

(The Necessity of Dialogue May Seem Somewhat Tedious; but as Transformations are Formulated, it Can Rapidly Adapt to the User, in Ways That a General-Purpose NLI Would Not Do, and with Less Time-Overhead on Each Query. The Combinatorial Growth of Inputs Covered Quickly Converges on a "Habitable" System.)

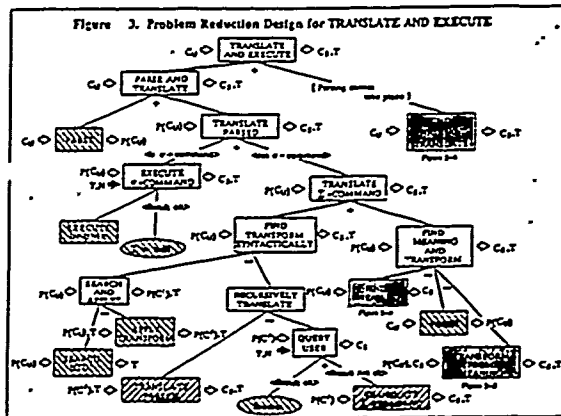
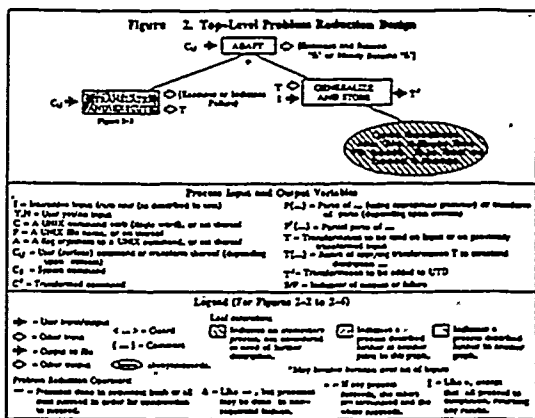


FIGURE 1. SUMMARY OF PROCESSES IN AIUI
(When the User Command is Not a System Command)

Case I: User Command Can be Parsed

Case I-A: Transformation in UTD

- Apply Transformation
 - Match
 - Transform

Case I-B: Transformation Not in UTD

Case I-B-1: Meaning of User Assertion Known to AIUI

- Formulate Transformation
 - Specific Tree Mapping
 - Category Generalization
 - "Risky Generalization"

Case I-B-2: Meaning of User Assertion Not Known

- Heuristics
- Partial transformation

Case II: User Command Cannot be Parsed

Case II-A: All Lexical Items Are Known

- Partial parses
- Formulating New Transformations
- Reformulating Grammars

Case II-B: There Are Unknown Lexical Items

- Assign new categories
- Merge categories that use same Transformations

Case II-C: Create Transformation for Given String Only

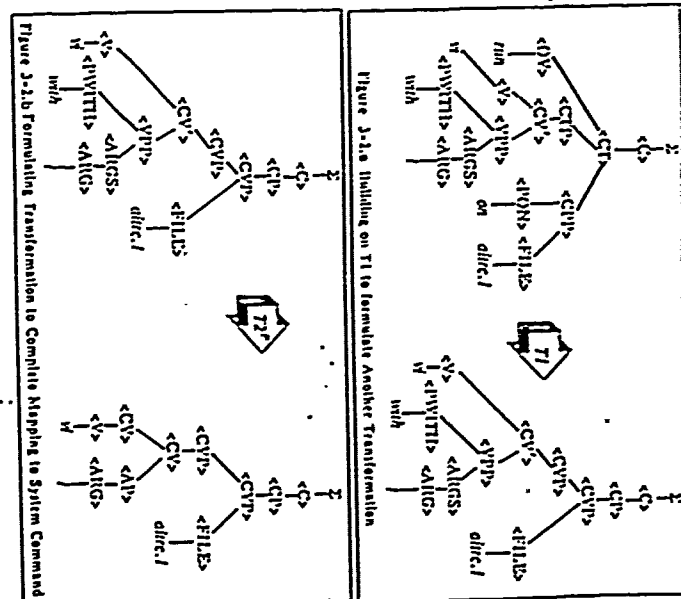
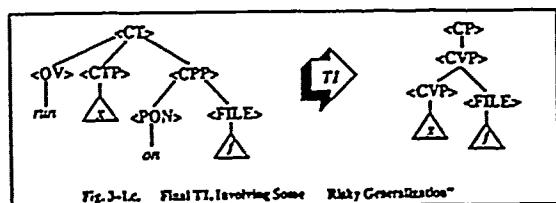
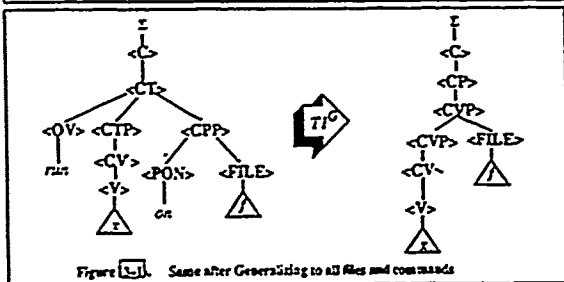
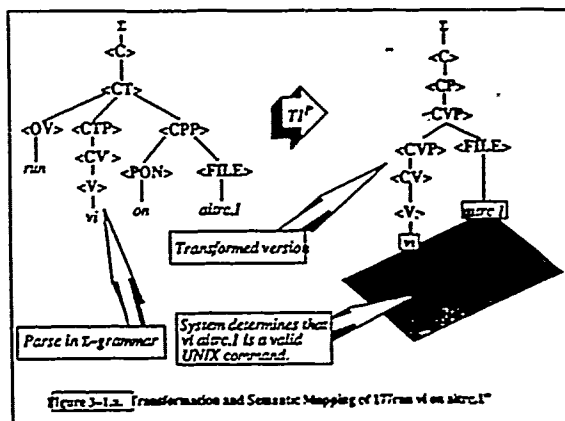


Figure 3-2b. Formulating Transformation to Complete Mapping to System Command

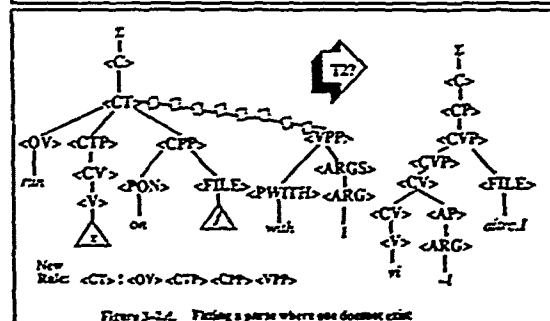
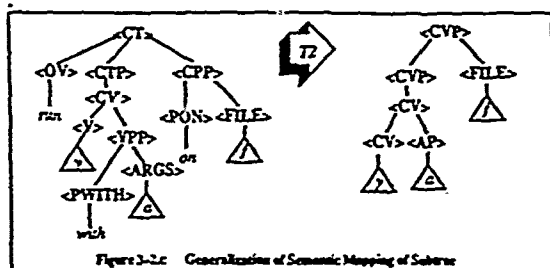
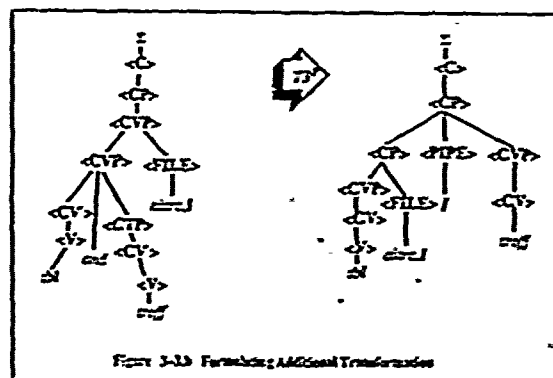
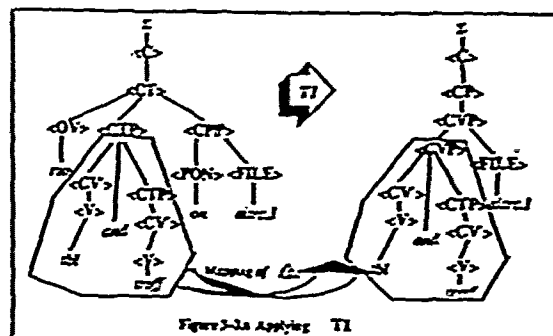
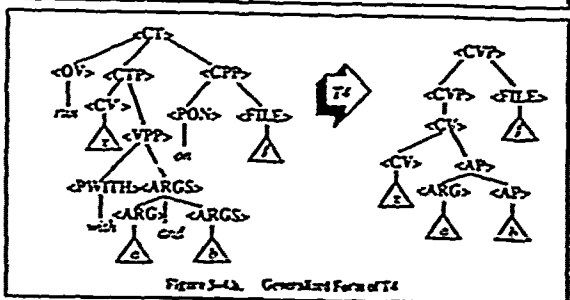
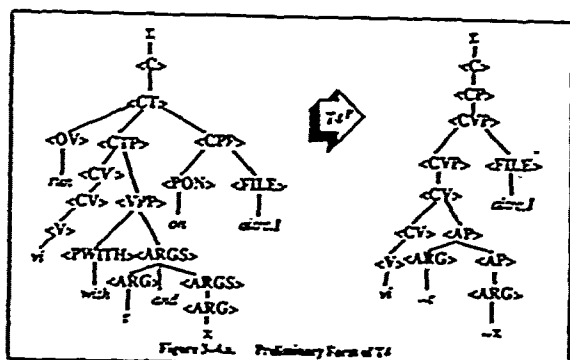


Table 1. User Commands and Meanings (System Commands)	
User Command	Meaning
(1) run vi on <i>slim</i> .1	vi <i>slim</i> .1
(2) run vi with l on <i>slim</i> .1	vi -l <i>slim</i> .1
(3) run <i>tbl</i> and <i>woff</i> on <i>slim</i> .1	<i>tbl</i> <i>slim</i> .1 <i>woff</i>
(4) run vi with r and x on <i>slim</i> .1	vi -r -x <i>slim</i> .1
(5) run ls with l and grep with c'e' on <i>slim</i> .1	ls -l <i>slim</i> .1 grep -c'e'
(6) run <i>woff</i> on <i>slim</i> .1 and append to <i>slim</i> .2	<i>woff</i> <i>slim</i> .1 >> <i>slim</i> .2
(7) run ls with l and grep with c'e' on <i>slim</i> .1 and append to <i>slim</i> .2	ls -l <i>slim</i> .1 grep -c'e' >> <i>slim</i> .2

[illegible]

CVV → s: >>>	GVG → w e _
GVG →	VAG → - < _
GV → s e _	GV → s e _
GVw b → w e e _	VAG → - < _
GV → s e e e _	GV → s e _
GV → s e e e _	

Some of the comparisons in this system are defined by morphology or by function (e.g., whether a file or directory is in the currently accessible directory system), rather than by function in the lexicon.



Wednesday Presentations 28

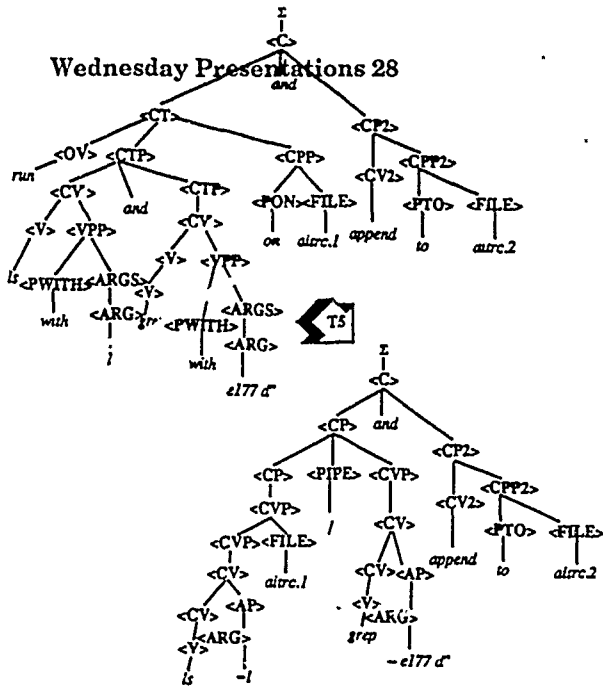


Figure 2-7a. Transformation of `^n` is with `|` and `grep` with `^177 d"` on `altrc.1` and append to `altrc.2"` to `"is -l altrc.1|grep ^177 d" >> altrc.2"` — First Stage: application of TS.

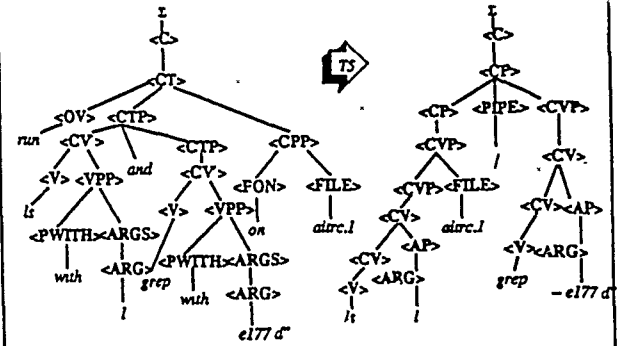


Figure 3-5a. Transformation of "1071s with 1 as 1 grip with el 77 d" on airc.1".
Before Generalization.

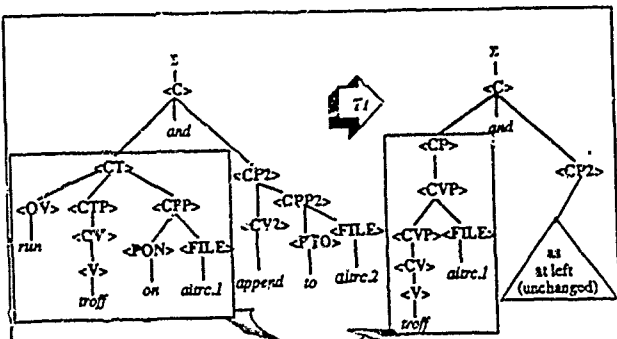


Figure 3-6a. Transformation of 1/True troll on airc.1 and append to airc.2.
First Stage is application of T1 to its fully generated form.

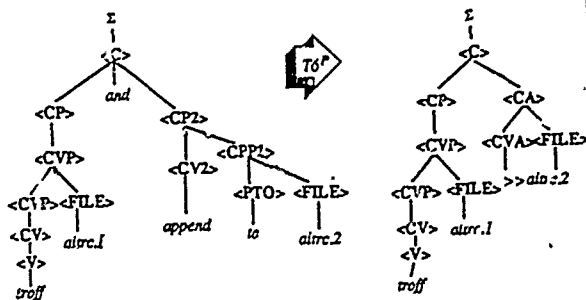
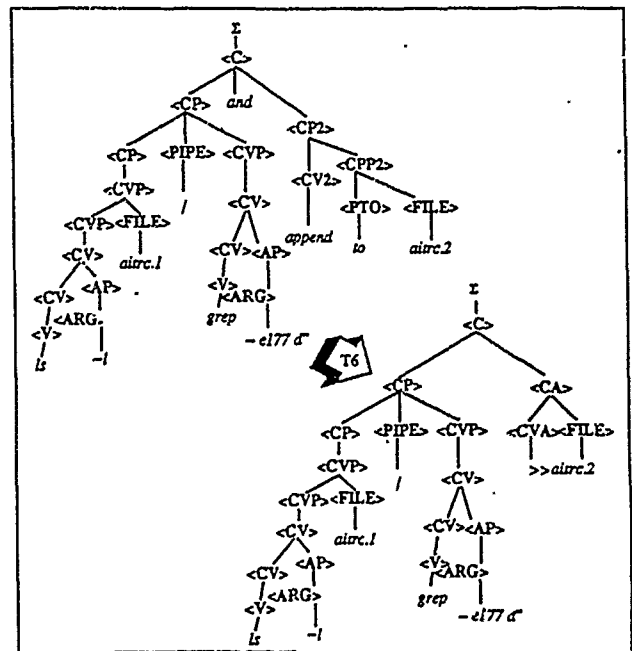


Figure 3-12. Transformation of 17TTHUJ.TUJ into T and applied to store.
Second Stage—application of T_1 .



Figures 3-7.b. Transformation of is with | and grep with -e177 d" on a1trc.1 and append to a1trc.2" to "ls. | a1trc.1 | grep -e177 d" >> a1trc.2" -- Second stage: application of T6.

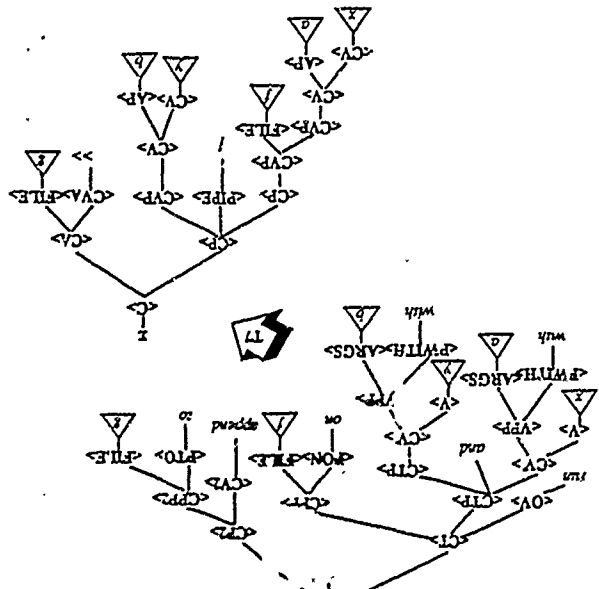


Figure 7-5c. Transition of _____ as it shifts from "is with" end group with 0.77 at "no more," and spread to "more" so "is - more" | [grp - 0.77] >> "less," accomplished by T5 and T6, as generalized.

NEURISTICS

run vi en aire.! (as vi aire.!)
morphological clue to file.
current dictionary may contain file

div-1/div-2/div-3

morphological clues to find
(but could be confused \rightarrow check to see if conventional)

trifft etwa. 1. und place in y (x ... > y)
 ↑ transform this part first
 if this is file, generate from action (2, 2), 1, ...

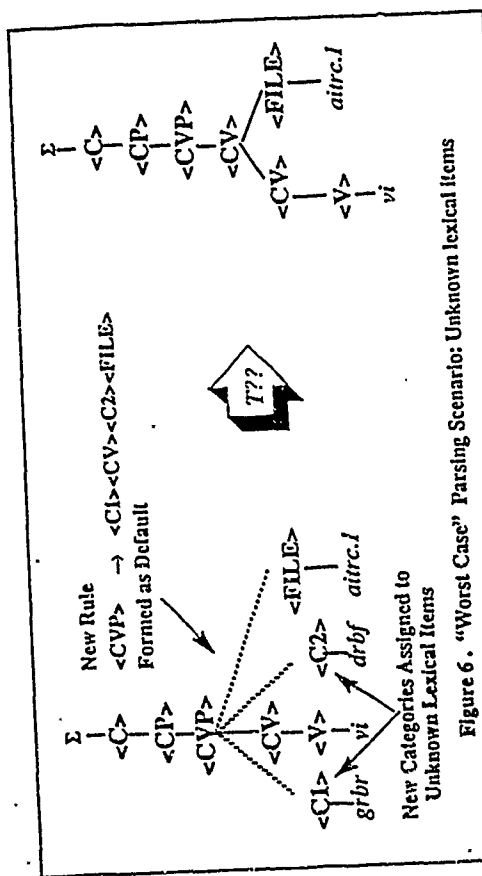


Figure 6. "Worst Case" Parsing Scenario: Unknown lexical items

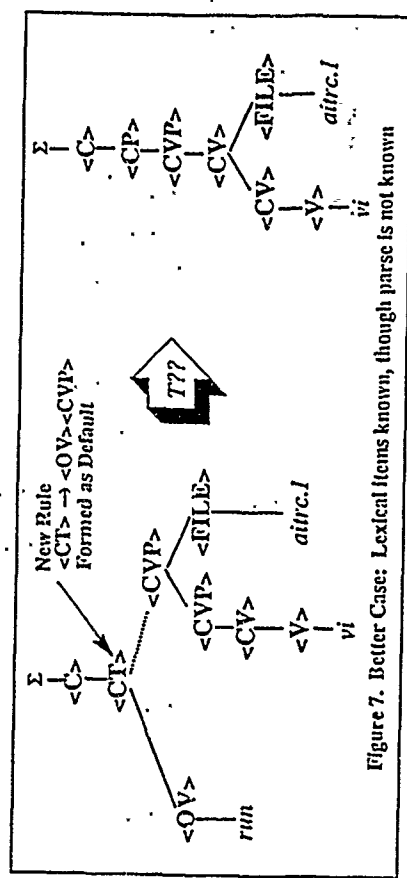
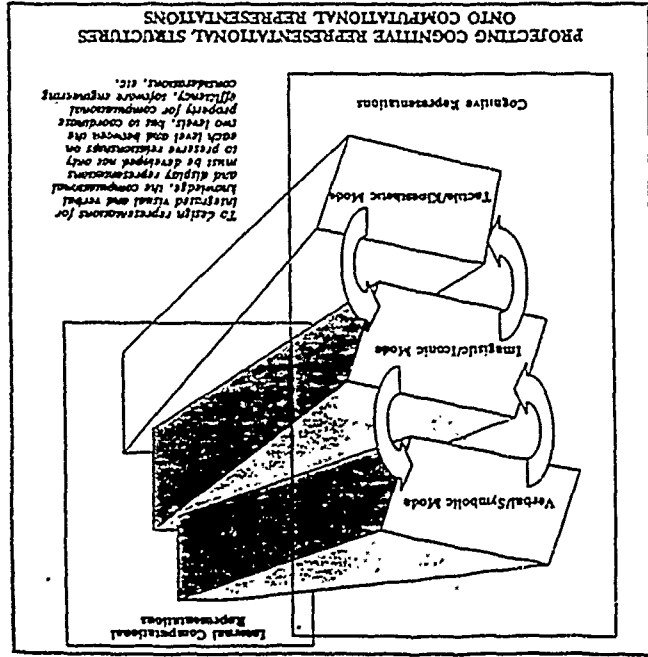
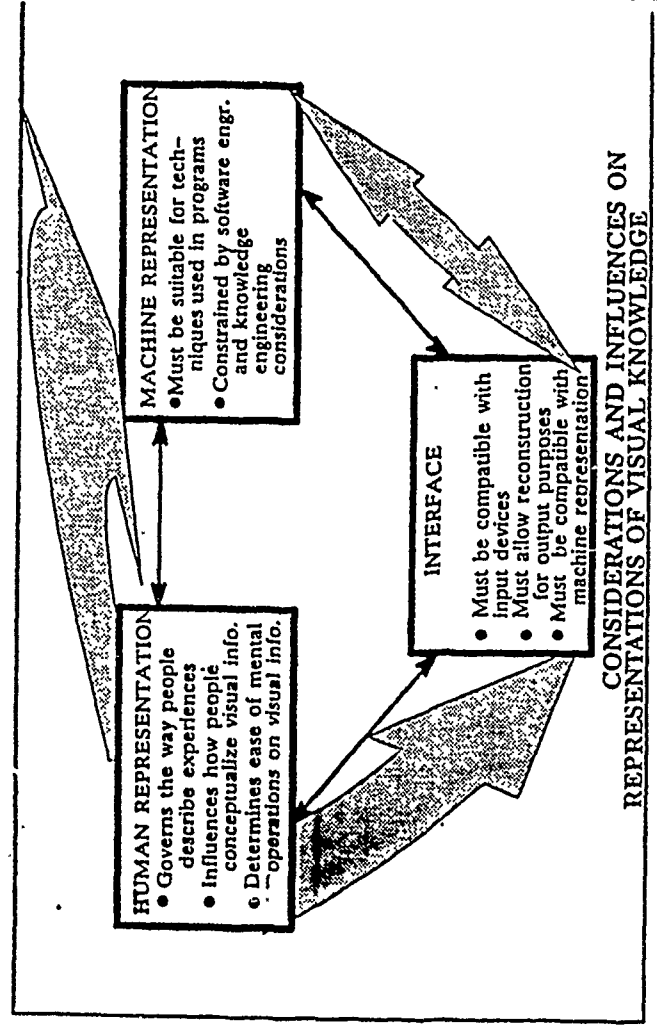
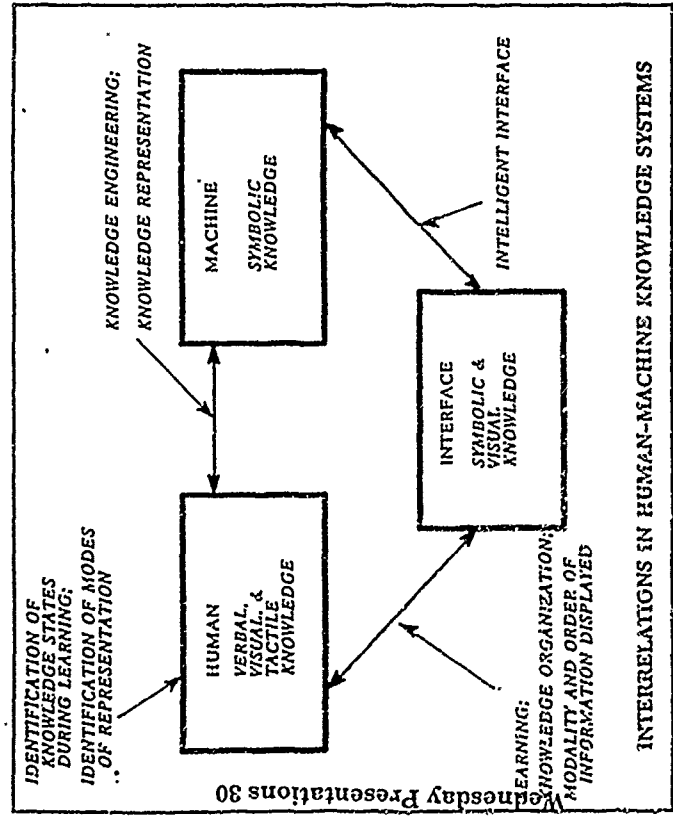


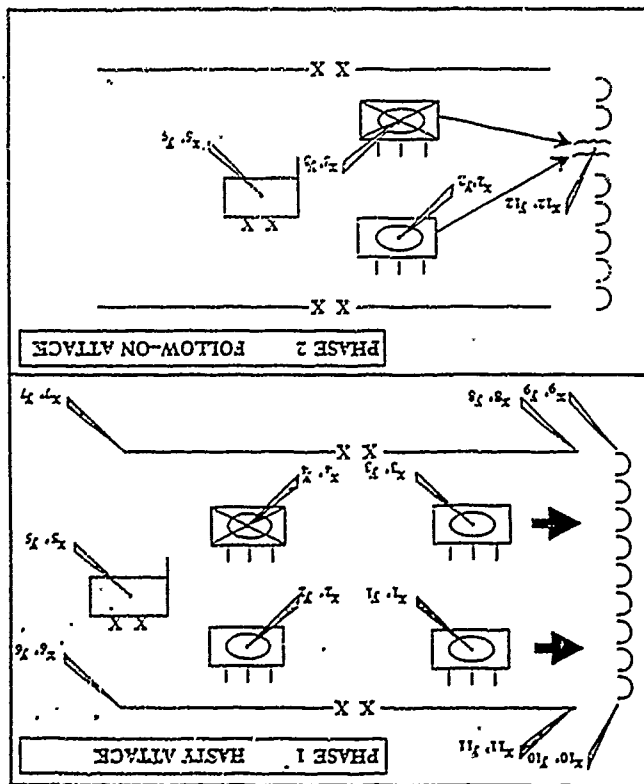
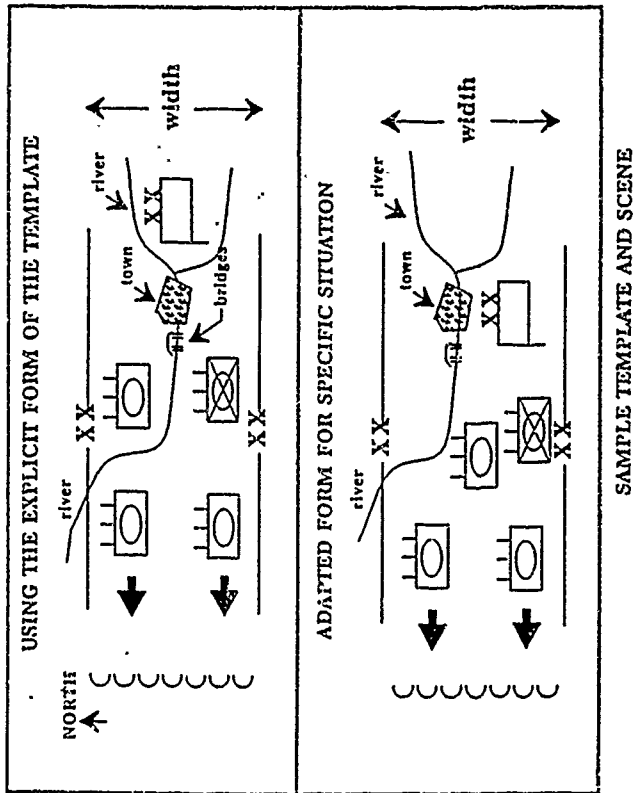
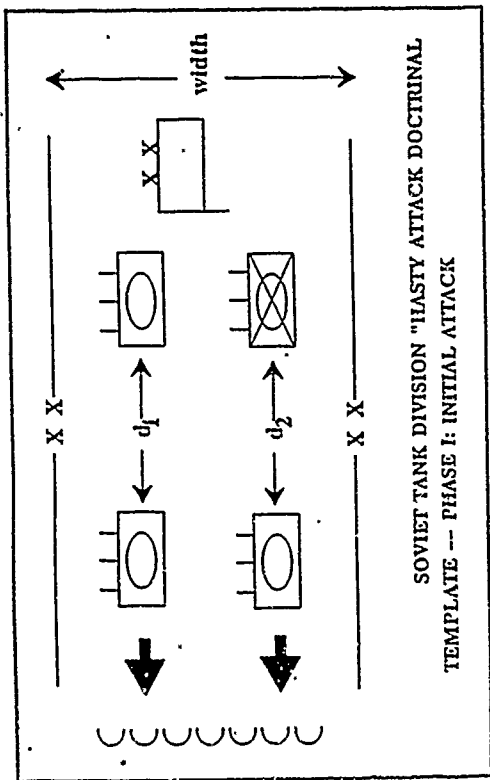
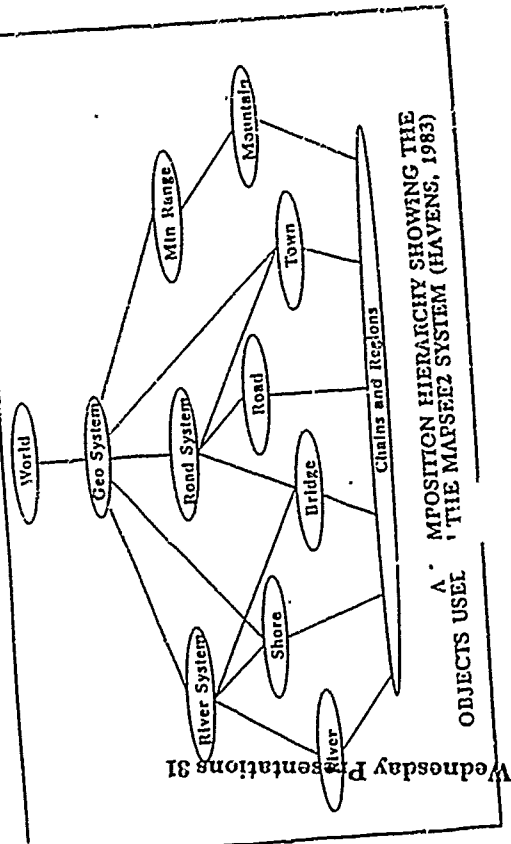
Figure 7. Better Case: Lexical items known, though parse is not known ^{1/2}

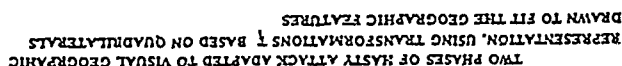
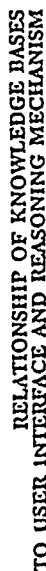


COMPUTER ENCODED GRAPHIC DATA

VA,	
(9,2,0)	
(9,2,8)	
(6,2,8,16)	
(9,2,8)	
(4,2,32,1,500405,3,297169,1,707699,2,738909,0,0,0)	
(4,3,0,1,707699,2,738909,1,79074,2,51515,17,1,0)	
(4,4,0,1,79074,2,51515,2,355828,3,913778,17,1,0)	
(4,5,0,2,355828,3,913778,1,416383,3,523558,17,1,0)	
(4,6,0,1,416383,3,523558,1,500405,3,297169,17,1,0)	
(6,8,0,16)	
(9,8,8)	
(4,8,32,1,500405,3,297231,1,707699,2,738909,0,0,0)	
(4,9,0,1,707699,2,738909,1,627013,2,639111,17,1,0)	
(4,10,0,1,627013,2,639111,1,549216,2,535053,17,1,0)	
...(etc.)...	







Natural Language Programming in Solving Programs of Search

Alan W. Biermann
Duke University

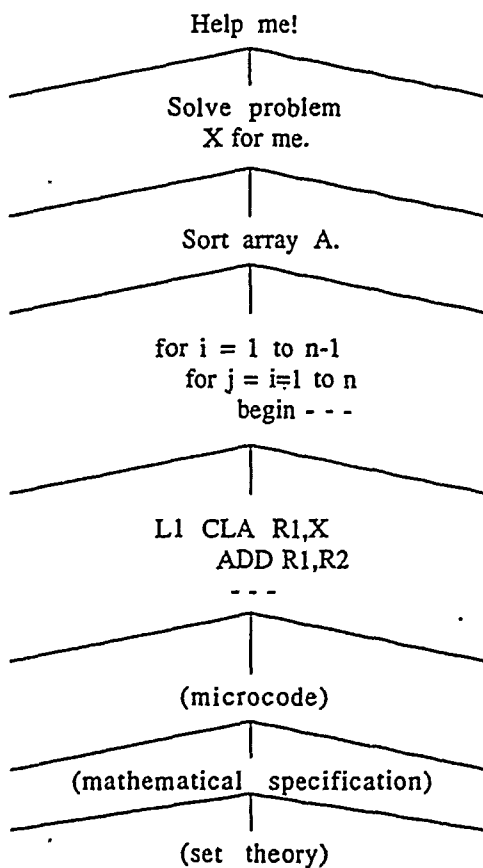
29 May 1991

A definition of "informality":

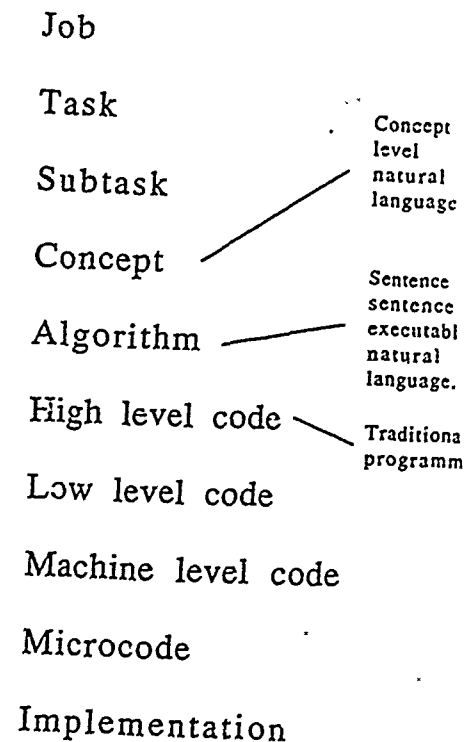
Wednesday Presentations 33

The degree of informality is inversely proportional to the degree of specification.

Levels of specification:



Levels of granularity:



This talk:

Natural Language Programming

A Generation Methodology for
Search Programs

Using Natural Language to Generate
Search Programs

A fortual language:

- (1) A subset of natural language.
- (2) Approximately learnable from
a small amount of instruction.
- (3) A formal definition.

A natural language program:

Here is a way to sort an array.

Exchange the i-th smallest entry with
the i-th entry for each entry i.

Example usage:

Sort that array.

Traditional program:

```
procedure sort(A:array of integer,  
              n: integer);
```

```
var i,j:integer;
```

```
begin
```

```
for i:= 1 to n-1 do
```

```
  for j := i+1 to n do
```

```
    if A[i] > A[j] then
```

```
      begin
```

```
        temp := A[i];
```

```
        A[i] := A[j];
```

```
        A[j] := temp
```

```
      end;
```

```
end;
```

Compiling the natural language program:

Definition of verb sort. Argument: array

For each entry i:

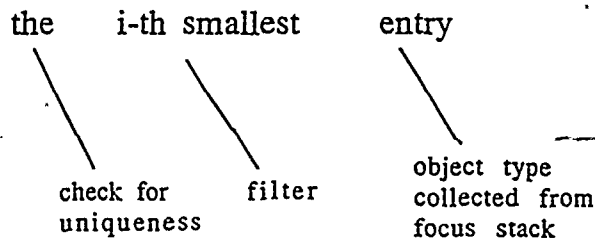
Exchange:

the i-th smallest entry

With:

the i-th entry

Noun phrase syntax:



apply(ordinal, i-th smallest,
typegen(entry, in,
containerof(entry))))

collect(X)

Example: Finding the second
smallest entry in [9, 3, 7, 10]
e1 e2 e3 e4

Noun phrase semantics:

for X in apply(artdef, the
apply(ordinal, i-th smallest,
typegen(entry, in,
containerof(entry))))
collect(X)

apply(artdef, the e3:7
apply(ordinal, 2-nd smallest, e3: 7
typegen(entry, in, e1:9 e2:3 e3:7 e4:10
containerof(entry)))) [9,3,7,10]
collect(X)

Imperative verb execution:

Exchange e3 with e2.

Complete computation:

For each entry i:
Exchange:
the i-th smallest entry
With:
the i-th entry.

The natural language program:

Here is a way to sort an array.
Exchange the i-th smallest entry with
the i-th entry for each entry i.

A natural language program:

Display a 4 by 5 matrix and call it testmat.
Fill the matrix with random values.
Choose an entry and call it p.
Define a method to pivot testmat about p.

Choose an entry not in the p row and not in
the p column and call it q.
Compute the product of the entry which
corresponds to q in the p row and the entry
which corresponds to q in the p column.
Divide the result by p and subtract this result
from q.
Repeat for all other entries not in the p row and
not in the p column.
Divide each entry except p in the p row by p
and negate those entries.
Divide each entry except p in the p column by p.
Put the reciprocal of p into p.
End the definition.

The equivalent in PL/I code:

```

EXCHANGE;
/* Wednesday Presentations 36
PROCEDURE(MATRIX,PIVROW,PIVCOL);
  DECLARE (MATRIX(*,*)PIVOT) FLOAT,
  (PIVROW,PIVCOL,ROWS,COLMNS,I,J)
  FIXED BINARY;
  /* DETERMINE THE NUMBER OF ROWS
  AND COLUMNS */
  ROWS = HBOUND(MATRIX,1);
  COLMNS = HBOUND(MATRIX,2);
  /* NAME THE PIVOT ELEMENT */
  PIVOT = MATRIX(PIVROW,PIVCOL);
  /* APPLY THE "RECTANGLE RULE" */
  DO I = 1 TO PIVROW-1, PIVROW+1 TO ROWS;
  DO J = 1 TO PIVCOL-1, PIVCOL+1 TO COLMNS;
  MATRIX(I,J) = MATRIX(I,J)
  - MATRIX(I,PIVCOL) *
  MATRIX(PIVROW,J)/PIVOT;
  END;
END;
/* CHANGE THE OLD PIVOT ROW */
DO J = 1 TO PIVCOL-1
  PIVCOL+1 TO COLMNS;
  MATRIX (PIVROW,J) =
  - MATRIX(PIVROW,J) / PIVOT;
END;
/* CHANGE THE OLD PIVOT COLUMN */
DO I = 1 TO PIVROW-1,
  PIVROW+1 TO ROWS;
  MATRIX(I,PIVCOL) =
  MATRIX(I,PIVCOL) / PIVOT;
END;
/* CHANGE THE PIVOT */
MATRIX(PIVROW,PIVCOL) = 1 / PIVOT;
ENDEXCHANGE;

```

Success rates using NLC:

Number of subjects		
(a) achieving success	10	5+2
(b) reporting success		
system failure	1	1
(c) unable to proceed	1	0
(d) unable to finish on time	0	3
Total number of subjects	12	11

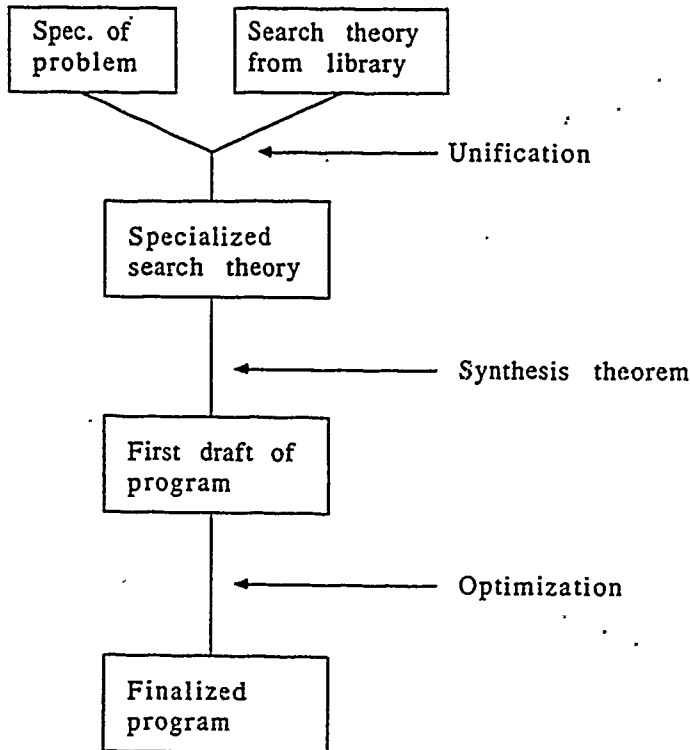
Sentence processing statistics for NLC:

Sentence type	Number	Percent
Correct	1283	81.2
User Error	18	1.1
Fault of English	2	0.1
User Sloppiness	141	8.9
Unimplemented	107	6.8
System Error	30	1.9
	1581	100.0

A generation methodology for search problems:

1. Specification of the problem.
2. Selection of a theory from a library.
3. Specialization of the theory to the current problem.
4. Instantiation of the theory into a first draft program.
5. Refinement of the draft program.
6. Compilation into object code.

Specification of the problem:



F Ord Search

D $\{ \langle h, A, \text{key} \rangle \mid h \text{ in Nat and } 1 \leq h \text{ and } A \text{ in map}(\{1..h\}, \text{Nat}) \text{ and Ordered}(A) \text{ and key in Nat} \}$

R $\{ \text{index} \mid \text{index in Nat} \}$

O $\text{lambda} \langle h, A, \text{key} \rangle, \text{index. } A(\text{index}) = \text{key and index in } \{1..h\}$

Unification:

Specializing the search theory:

A search theory:

F gs-binary-split-of-integer-subrange

D $\langle m, n \rangle \mid m \text{ in integer and } n \text{ in integer and } m \leq n$

R $\{ k \mid k \text{ in integer} \}$

O $\text{lambda} \langle m, n \rangle, k. k \text{ in } \{m..n\}$

R' $\text{lambda} \langle m, n \rangle. \{ \langle i, j \rangle \mid i \text{ in integer and } j \text{ in integer and } m \leq i \leq j \leq n \}$

Satisfies $\text{lambda } k, \langle i, j \rangle. i \leq k \leq j$

r0' $\text{lambda } \langle m, n \rangle. \langle m, n \rangle$

Split $\text{lambda } \langle i, j \rangle, \langle i', j' \rangle. i < j \text{ and } \langle i', j' \rangle = \langle i, (i+j) \text{ div } 2 \rangle \text{ or } \langle i', j' \rangle = \langle 1 + (i+j) \text{ div } 2, j \rangle$

Extract $\text{lambda } k, \langle i, j \rangle. i = j \text{ and } k = i$

for all x in D_p
 there is y in D_T
 for all z in R_p
 $[O_p(x, z) \implies Q_T(y, z)]$

A synthesis theorem: Wednesday Presentations 38

```
function Fp(x:Dp): set(Rp)
  returns {z | Op(x,z)}
  = F-gs(x,r0'(x))
```

```
function Fgs(x: Dp, r':R') :set(FT)
  returns {z | Satisfies(z,r')
    and Op(x,z)}
  = {z | Extract (z,r') and
    Qp(x,z)} union
    {Fgs(x,s) | Split(x,r',s)}
```

Substituting into the theorem:

```
function OrdSearch (h:Nat,A:map({1..h},Nat),key:Nat):
  set(Nat)
  returns {index | A(index) = key and
    index in {1..h}}
  = Ordsearch_gs(h,A,key,1,h)
```

```
function Ordsearch_gs(h:Nat,A:map({1..h},Nat),
  key:Nat,i:Nat,j:set(Nat))
  returns {index | A(index) = key and
    index in {i..j}}
  = {index | i=j and index=i and A(index)=key}
    U {index | i<j and <i',j'>=<i,(i+j) div 2> and
    index in Ordsearch_gs(h,A,key,i',j')}
    U {index | i<j and <i',j'>=<1+(i+j) div 2,j>
    and
    index in Ordsearch_gs(h,A,key,i',j')}
```

Efficiency: Accounting for order

$1 \leq i \leq j \leq h \implies A(i) \leq A(j)$.

Therefore:

$A(i) \leq A(\text{index}) \leq A(j)$

$A(i) \leq \text{key} \leq A(j)$

Adding the efficiency guards:

```
function OrdSearch (h:Nat,A:map({1..h},Nat),key:Nat):
  set(Nat)
  returns {index | A(index) = key and
    index in {1..h}}
  = Ordsearch_gs(h,A,key,1,h)
```

```
function Ordsearch_gs(h:Nat,A:map({1..h},Nat),
  key:Nat,i:Nat,j:set(Nat))
  returns {index | A(index) = key and
    index in {i..j}}
  = {index | i=j and index=i and A(index)=key}
    U {index | i<j and <i',j'>=<i,(i+j) div 2> and
    A(i') <= key <= A(j') and
    index in Ordsearch_gs(h,A,key,i',j')}
    U {index | i<j and <i',j'>=<1+(i+j) div 2,j>
    and A(i') <= key <= A(j') and
    index in Ordsearch_gs(h,A,key,i',j')}
```

Using natural language to generate search programs:

Create a program called OrdSearch.

The inputs to the program are h , a positive natural number, A , a sorted array of entries indexed from 1 to h , and key , a natural number.

The program is to find index such that $A(\text{index}) = \text{key}$.

Accessing a theory:

1. by name.
2. by "plan recognition".
3. by self invocation.

Accessing a theory by name:

"Use binary search."

Accessing a theory by "plan recognition" :

Select interval $[1..h]$.

If $1 = h$ then
 if $A(1) = \text{key}$ then return $\{1\}$
 else return $\{\}$

Divide interval into
 interval $[1..h \text{ div } 2]$ and
 interval $[h \text{ div } 2 + 1 .. h]$.

Etc.

(Pattern match against all search theories.)

"Process A from i to j only if

$A(i) \leq \text{key} \leq A(j)$ ".

What natural language offers:

Mechanisms for specifying
complex processes

Utilization of context

Variable granularity

Analogy

Stability

Universality

Speakability

Conclusions:

1. We can do natural language programming which is in some sense analogous to traditional programming.
2. There are some powerful methodologies around for program generation.
3. We may be able to link natural language into them to produce substantially better natural language programming.

Karen van Hoek

University of California, San Diego

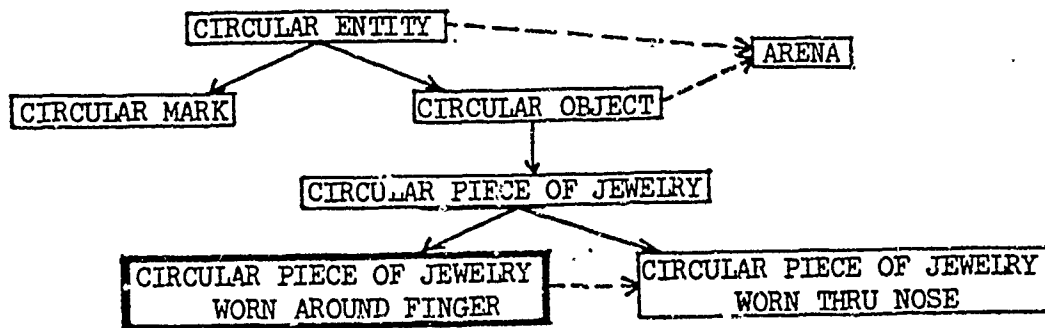
- (1) Standard assumptions
 - a. A linguistic system comprises a number of discrete, independently describable components (phonology, morphology, syntax, lexicon, semantics). Grammar (and syntax in particular) constitutes an autonomous structural domain distinct from both lexicon and semantics.
 - b. Linguistic semantics is best characterized by a system of rules akin to formal logic and based on truth conditions.
- (2) Cognitive Grammar
 - a. Unification of linguistic description. Linguistic systems are described in terms of a structured inventory of conventional units. Syntax is inseparable from semantics, and forms a continuum with morphology. Grammatical structure resides in the lexicon, as it is specified by the conventional units of the language.
 - b. Cognitive semantics: meaning is equated with conceptualization.
- (3) Conceptual Unification
 - a. Only semantic, phonological, and symbolic units are posited.
 - b. A symbolic unit consists of a semantic unit paired with a phonological unit.
 - c. Lexicon, morphology and syntax are fully describable by means of symbolic units.
- (4) Cognitive Abilities
 - a. to form structured conceptualizations
 - b. to perceive and articulate phonological sequences
 - c. to establish symbolic associations between conceptual and phonological structures
 - d. to use one structure as a basis for categorizing another
 - e. to conceive of a situation at varying levels of abstraction (schematization)
 - f. to detect similarities between different structures
 - g. to establish correspondences between facets of different structures or elements in different domains
 - h. to combine simpler structures into more complex ones
 - i. to impose figure/ground organization on a scene
 - j. to construe a conceived situation in alternate ways ("imagery")
- (5) Content Requirement: The only units ascribable to a linguistic system are (i) overtly occurring expressions (or components thereof); (ii) schematizations of the elements permitted by (i); and (iii) categorizing relationships between permitted elements.
- (6) Schematization: A is schematic for B if B is fully compatible with the specifications of A but is characterized with greater precision and detail.
- (7)
 - a. Components of overtly occurring expressions: [kɪt], [tɪp]
 - b. Schematization: [CVC]
 - c. categorizing relationships: [[CVC]---> [kɪt]], [[CVC]---> [tɪp]]
- (8) The content requirement rules out any "purely grammatical" constructs, elements which have neither semantic nor phonological value (e.g. empty diacritics, syntactic tree structures, syntactic coindexing, most grammatical constraints and filters).
- (9) Common assumption: The autonomy of grammar is established if any aspect of grammatical structure is less than fully predictable on the basis of meaning or other independent factors.
- (10) Type/Predictability Fallacy: (9) confuses two issues that are in principle distinct: (i) what kinds of linguistic units there are; and (ii) the predictability of their behavior.
- (11) Grammatical form is not predictable on the basis of meaning--rather it symbolizes meaning. Language is conventional symbolization.

(12) Basic Tenets of Cognitive Semantics

- (a) Meaning is based on conceptualization
- (b) A fixed expression is often polysemous, i.e. it has a variety of related established senses that form a complex category representable as a network.
- (c) The meaning of an expression is characterized with respect to one or more cognitive domains (or "frames", "idealized cognitive models", "scripts", "folk models", etc.) Any kind of experience, concept, conceptual complex, or knowledge system can serve as the cognitive domain for an expression.
- (d) An expression's meaning embodies conventional imagery or construal; i.e. it incorporates a particular way of structuring or construing the conceptual content provided by its particular domains.

(13) Network model of complex categories

- (14) (a) Categorization by schema $A \rightarrow B$
A is schematic for B; B elaborates or instantiates A.
B is fully compatible with the specifications of A but is characterized with greater precision and detail (e.g. DOG \rightarrow POODLE)
- (b) Categorization by prototype $A \dashrightarrow B$
A is a prototypical value; B represents an extension from the prototype. (e.g. ROBIN \dashrightarrow OSTRICH)
B conflicts in some way with the basic specifications of A but is assimilated to the category on the basis of some perceived similarity to A.



(15) COLOR SPACE: yellow ARM: elbow BASEBALL: shortstop

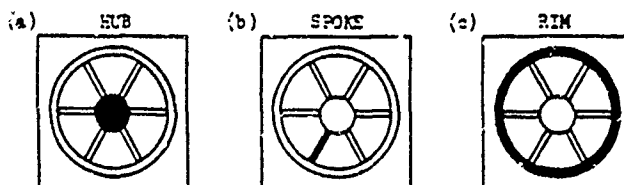
(16) Meaning = Content X Construal

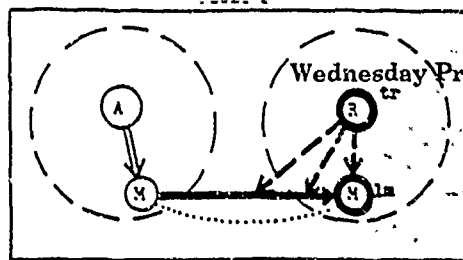
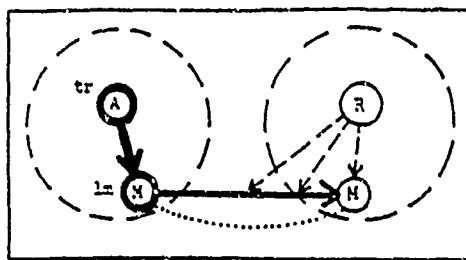
(17) Some Dimensions of Construal

- (a) level of specificity (schematicity) at which a situation is characterized
- (b) construal relative to different background assumptions and expectations
- (c) perspective (e.g. vantage point)
- (d) relative prominence of substructures (e.g. profiling; figure/ground)

- (18) a. The hill gently rises from the bank of the river.
- b. The hill gently falls to the bank of the river.

(19) The semantic pole of any linguistic expression is referred to as a predication. The base (or scope) of a predication is the extent of its coverage in relevant cognitive domains (i.e. how much of those domains it necessarily evokes and relies on for its characterization). A predication's profile is that substructure within its base that the expression designates.



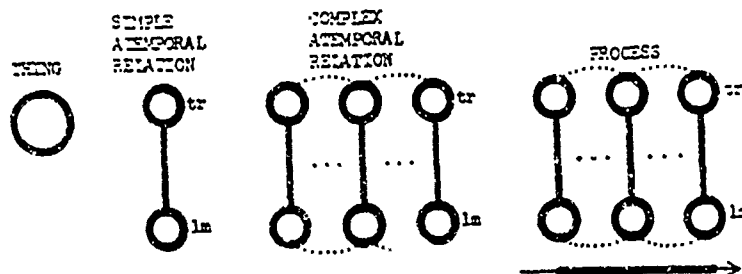


A = agent M = mover R = recipient \longrightarrow = energy transfer \dashrightarrow = motion
 $\cdots\cdots\rightarrow$ = perception/possession $\cdots\cdots$ = correspondence/identity \bigcirc = sphere of control

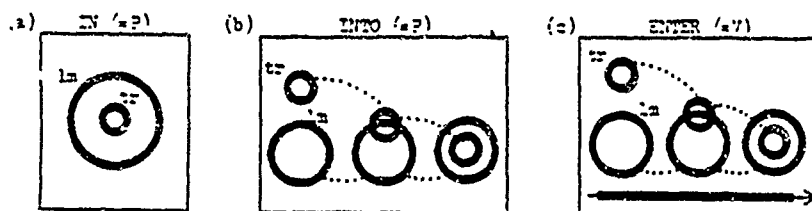
(20) (a) A nominal predication profiles a region in some domain. (Region defined abstractly—a set of interconnected entities, which need not be discrete, salient, or individually recognized.)

(b) A relational predication profiles interconnections among conceived entities. (Interconnections can be thought of as cognitive operations that register the relative positions of entities in a domain.)

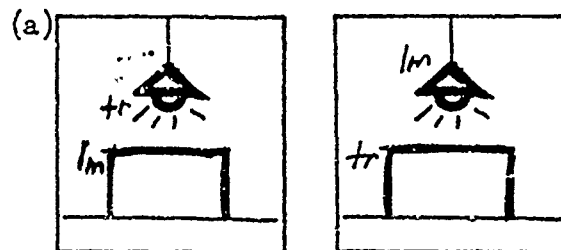
(c) Processes profile a series of relational states, conceived as arranged in time, and scanned sequentially.



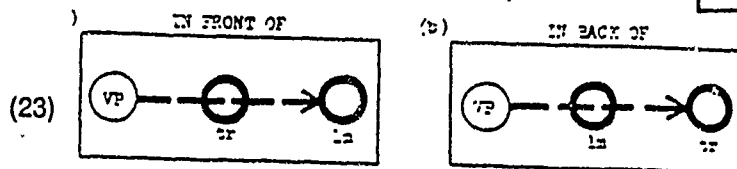
Subj. = 2
 tr - trajectory (figure)
 lm - landmark
 l = D.O.



(21) Relations involve figure/ground asymmetry; this is one aspect of construal.

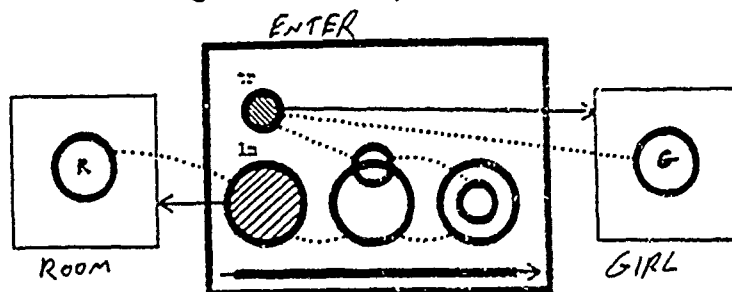


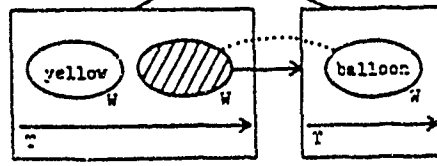
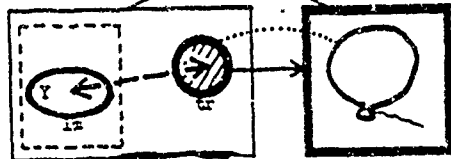
(22) a. The lamp is above the table.
 b. The table is below the lamp.



(24) a. Mary hit John.
 b. John was hit by Mary.

(25) Elements of grammatical composition: correspondences and elaboration





References

Langacker, R. (1987) Foundations of Cognitive Grammar, Volume 1: Theoretical Prerequisites
Stanford University Press

Langacker, R. (in press) Foundations of Cognitive Grammar, Volume 2: Descriptive Application
Stanford University Press

Connectionism

Why an appealing formalism?

Notational systems:

You can't do science without them,
but why do they always outstay their
welcome?

Case study: Controversy in Linguistics

Formalisms -- gain rigor, lose coverage

Folk labels -- make contact with the
diversity of language data

Why were Chomsky's formalizations
appealing?

What were the conceptual by-products?

Examples of unfelicitous analyses

Cul de sac: "Language acquisition in
the absence of experience"

Advantages, conceptual by-products of
an alternative idealization,
computational framework
(connectionism)

How do we escape serial monogamy?

Models or Metaphors --

Source Of Ideas, Not
Statement About The Natural
World

What Every Cognitive Theory Needs

Notational system

Descriptive constructs, (equivalence classes, taxonomic labels), part-whole relationships generalizations about allowed form	Noun, verb, object phoneme, morpheme, word, clause, sentence S → NP VP VP → (aux) V (NP)
---	--

Realizations of the world or of a particular
phenomenon

Explanatory system

Descriptive constructs are related to phenomena
outside the notational system

Mechanism

Conception of how it all might "work" -- either
computationally or neurally

The Pre-Chomskyan Era

From Bloomfield, Language 1933

Unclear how to idealize language

What is a grammar?

Pre-1700's: Latin is the true language,
grammarians tried to shape French, English
grammars into the Latin model

Intuitive understanding of linguistic
regularities

• morphological paradigms (e.g. Latin case)

• basic phonological

important impossible

No explanation for linguistic productivity

• Creativity in 'loan words' attributed to
the psychological process "analogy"

• new utterances are formed "on analogy"
to known "grammatical patterns"

What grammar is NOT:

Wednesday Presentations 46

Destroyed vague notion that language could be mediated by a stimulus-response system.

chain of words -- each word is stimulus for following word

Formalization of intuitive notions with finite-state automata, can't handle regularities that span gaps of arbitrary length

Subject-verb agreement across an embedded clause.

People's knowledge of grammar can't be:

- identified with knowledge of a particular corpus -- **productivity**
- knowledge of the meaning or significance of sentences
"Colorless green ideas sleep furiously"
- equated with any notions of statistical approximation *hat.*
a whale was wearing a green where.

Chomsky's Successes

Abstract, precise definitions must replace intuitive categories

"To attribute the creative aspects of language to 'analogy' or 'grammatical patterns' is to use the terms in a completely metaphorical way, with no clear sense and with no relation to the technical usage of linguistic theory."

Chomsky, 1966

Example of rigor: The 'explicitness' condition

Grammars (statements of the regularities in a set of sentences) must be described in a way that does not rely upon the intelligence of the understanding reader.

that grammar is autonomous and independent.

What had previously been a methodological principle was now the bedrock of linguistic theory --

the Autonomy of Syntax Hypothesis

6

Idealizations for language and grammar

A language is a set of strings (sentences)

A grammar is a device that fully and explicitly describes a language.

Notion of a mechanical procedure for generating all the sentences of a language

An answer to the question of linguistic productivity:

the grammar is in the head

Operations on strings of symbols: parsing, composition, table-look up

Transformations:

Explains the relationship between form and meaning

Why is it possible to say the same thing in different ways?

A bug is under the rug.

There is a bug under the rug.

7

Consequences of the Generative/Transformational Conceptualizations

Division between rote and rule

↓
Lexicon rewrite rules
Frozen phrases: "take advantage of"
Competence-performance distinction
Rules generate strings of arbitrary length
Assume processing limit: restriction on output buffer

Lack of explanatory framework

Can't appeal to communicative goals
because of A. of. Syntax
Notational arbitrariness -->
Theoretical isolationism

Mandatory modularity, nativism as the
best solution

Cognitive grammar approach:
Grammatical devices reflect different
conceptions of the event being
described.

construction
Passive makes the object or recipient the
sentence subject.

FINE:

John was kissed by Mary.

If the direct object of the verb isn't affected
by the action, passivizations "sounds odd."

Mary slept in that bed.

?? That bed was slept in by Mary.

That flea-bitten dog slept in that bed.

That bed was slept in by your
flea-bitten dog.

Infelicitous Analyses Wednesday Presentations 47

The active-passive relationship

Mary hit Bill. Bill was hit by Mary.

switch object and subject,
change verb to past participial form, add "by"

Mary weighed 120 pounds.

* 120 pounds were weighed by Mary.

Lexicalist solution: only some verbs
passivize, note this in verbs' lexical entries.

John left the auditorium.

? The auditorium was left by John.

John left the auditorium unguarded.

The auditorium was left unguarded by
John.

11

Connectionism "neural networks"

Learning algorithms for associating a set of
input-output pairs.

Output 0 0 0 0 0 0 0 "hidden layer"
input 0 0 0 0 0 0 0 constructs categories
to solve non-linear
mappings.

Extract regularities in the set of i-o
mappings.

mechanism for generalization

Generalization, capture the "rule-analogy
continuum"

language has sub-regularities with
partial productivity

An appealing formalism for informal processes:

Minimal assumptions about the data.

Anti-establishment sociology, aspiration
to capture the whole organism: rules,
minor regularities, and exceptions

I. Connectionism Is Representationally Inadequate

The data structures of Main Stream Linguistic Theory lack "neurally plausible" implementations.

II. Main Stream Linguistic Theory Is Inadequate

Good at describing:

form-form regularities

the most dominant and/or stable regularities in English

Bad at describing:

the nature of the sound-meaning association

partial regularities

between form and meaning.

natural to think of language as a conventionalized coding system for communication

<utterance, meaning> pairs

Mechanism:

Suppositional storage,

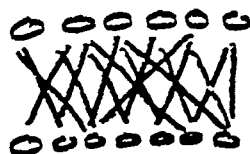
so instead of "mere associationism" we have the emergence of abstract structures

15

How are connectionist ideas and mechanisms helpful

- I. A mechanism for associating variable-sized units of meaning and sound

[MEANING]



[FORM]

- II. Extraction of regularities in sound-meaning mapping

Prototypes emerge

Irregular patterns maintained if favored by frequency

- III. Helps avoid the rule/list exclusionary fallacy (Langacker, 1986)

- IV. Redefines Autonomy of Syntax Hypothesis

AGREE:

Grammatical categories can not be equated with conceptual categories or with communicative function.

Subject \neq Agent

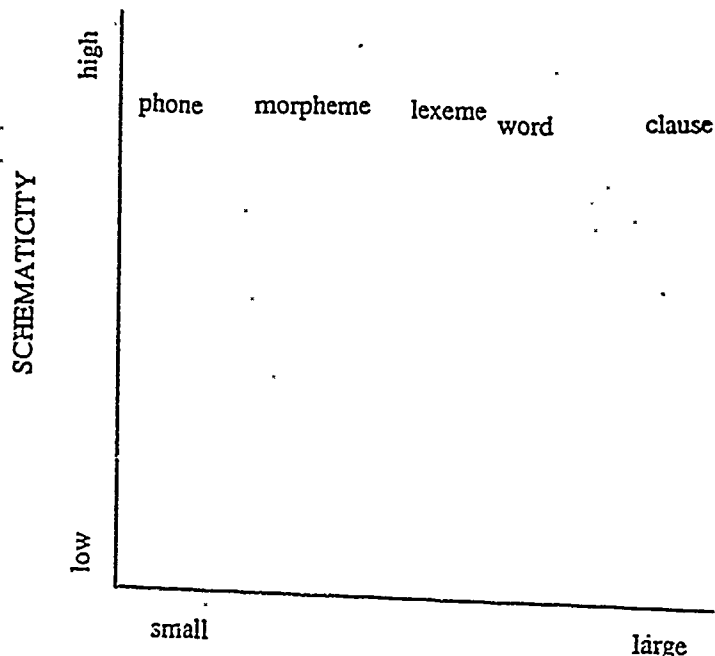
Subject \neq Topic

DISAGREE: Subject = Primary Clausal Figure

They are independent of cognition, communicative function.

INSTEAD:

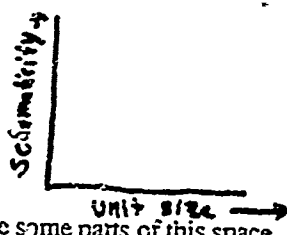
They are representations that emerge as a language community's solution to the problem of associating sound with meaning.



What is the highest level of schematicity at which a generalization can be stated?

2-3 same idea from Langacker, 1986

Other Questions



Languages only colonize some parts of this space.

Why do different languages differ in the grammatical devices they employ?

Different solutions to same problem

Where does phrase structure fit into the sound-meaning mapping?

It's Part of the solution to the problem of conveying multi-dimensional information through a linear channel (Bates, & MacWhinney 1992).

Conclusion:

Investigate sound-meaning regularities
Investigate the emergence of representation in networks that learn a large number of mappings.

Regularities that defy conventional labels:

Wednesday Presentations 49

sound-symbolism fumble, mumble
sniff, snout, sneeze

inflectional and derivational morphology: What is a word?

one word or two
noun compounds, contractions, clitics, "gimme"

tightness/looseness of clause combinations

idioms, conventional expressions

I wouldn't VP if you gave me NP
The more [Sentence 1], the more [Sentence 2]

father
fathers

father
paternity

form similarity ↑

meaning similarity →

19 Discussion Topics

• Is the notational system the theory?

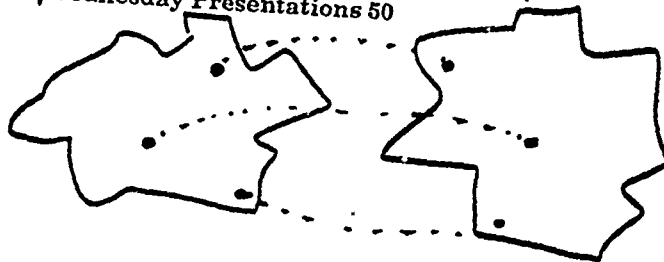
↳ Over-adherence

Little exploration of alternative explanations, confirmation bias

• If your formalism captures half of a truth, endless experiments will support its validity.

Hypothesize Z. If Z, then (logically) A.

Test. Find A.



What can you conclude about the World
from examining the behavior of your
PDP simulation?

Absolutely nothing.

Can we even say that simulation results are
"suggestive" of the real world?

Only with extreme caution.

"Experiments with model" can only:

- describe a state of affairs
- help us understand the computational
principles underlying the model --
use this as analogy to the World

Wednesday Afternoon Discussion

Shultis. I've been sitting around frantically trying to integrate over this day. Today was language day. Tomorrow is more general. Tomorrow is knowledge day. Unfortunately Cordell Green had to go back to his offices. He'll be back tomorrow morning, but he has to help people get ready for a site visit they're having. I said I was sorry to hear that he was going because I would have liked to hear what he had to say about the day. He replied that he was shell-shocked, that there was a lot of stuff going on and he was feeling overwhelmed. And I have to confess that even though I had some idea of what was going to happen today, I have been really impressed by the diversity and the number of ideas that have come up; but if I can at least make a stab at it, we were all talking about language today. One thing that I see as a common theme is that we're all concerned in one way or another with the problem of meaning and the problem of conceptualization and how they influence or affect the linguistic interface, that is, linguistic performance and capabilities.

Instead of sitting up here and pontificating on the subject, I'd like to see if I can solicit from you questions or issues, and write them up here on the board, to see if we can come up with a list of points that people feel it would be important to discuss at this point. As I said, I can supply some views of my own, but...

Standish. The question I have that seems to be a thread running through this proceeding and which may touch on something everybody deals with, is this: what do people mean when they talk about "informalism" or "informality"? What's the meaning of "informality"? In particular, is there any meaning of it that contravenes the finite symbol system hypothesis which is something that, as a CMU person, means roughly that any intelligent organism or machine can only exhibit intelligent behavior by means that deal with finite systems of symbols and their transformation by discrete rules through time. Something like that. If somebody else has heard another version of that maybe they can give it more articulately. Is there any meaning of "informality" that transcends finite systems of symbols and their transformations?

Shultis. By which I assume you mean finitely-based?

Standish. Yeah. Finitude is, I think, part of it; although others may have heard variants of it.

van Hoek. What is the import of finitude? Finite in what dimension?

I'm not familiar with this.

Reeker. In terms of the number of basic elements you (can manipulate); you can compose infinite...

Standish. I couldn't, for example, compute with continuous functions that are infinite in their domains and ranges. Like adding holograms, color holograms.

Harnad. That was almost the same question I was going to ask, but could you put down "What is formality?"

Shultis. OK. Yeah, that was a question you raised this morning.

Harnad. And could I have a subquestion. What did *you* mean when you said "computers can be described as formal symbol manipulators"? I mean, of course, according to Putnam, chairs can be described as formal symbol manipulators, too, but I thought in the case of computers, we were a little bit closer to the truth.

Shultis. Well, let me hazard a quick answer to that and then people can chew on it and get back to it. The fact is the computer is no such thing as a formal symbol manipulator. It's a bunch of silicon and wire that makes electrical pulses jiggle and move up and down and that's all it is. There's another kind of answer; the answer, if you like, is that the only way in which you see a computer as a formal symbol manipulator is because you interpret that physical behavior as being manipulation of formal symbols and it is an interpretation. Given that what you're doing is interpreting one phenomenon in a particular way, there is nothing in the physical device which requires that you give it that interpretation or that says that's the only interpretation you can give it.

Reeker. But isn't there a sort of a grain built in at some level into any computer? Not inherent in the silicon, but inherent in the design of it?

Littman. Its purpose.

Shultis. So: computers are intentional systems?

Littman. Yes, they're designed for a reason.

Standish. Well, they exist at many different levels. At one level you might push electrons in the current technology, but at an old level, it might have been sound waves and mercury delay lines or magnetic spots on a drum. In a future one, it might be optical pipes, or interactions, or macromolecular protein folding.

Shultis. Or even adding holograms.

Standish. Adding holograms, whatever. At a level above the basic material mechanics are certain logical symbol manipulation rules that are usually pretty faithfully implemented by the underpinnings, although not always when there are electronic glitches and whatnot. And from that point up, from just the pure logical equations, an awful lot of the power of abstraction that is built in a representation is built on top. So, at some level, if I cut it off at the physical device level, and just talk about the rules of logic that define the instruction sets, whatever they happen to be, it is really mathematics and logic, and has nothing to do with the underpinnings. They are just

material artifacts that usually fairly perfectly, and sometimes imperfectly, implement these finite symbol systems.

Shultis. Let me give another example of that and then I want to get off this and on to other questions. Larry showed an example earlier today. Suppose you take your X-windows display and there's a picture of Mickey Mouse on it, and you ask "What is the computer doing?" The answer is: "The computer is showing me a picture of Mickey Mouse". What I want to know is what partial recursive function it's computing. There is a construal of that bit pattern which has absolutely nothing to do with it as a formal system. So, just think about that. Other questions, issues?

Harnad. Well, there's a related one in your talk; in fact closely related. What role does the semantic interpretability play in the meaning of formality, or the meaning of physical formal systems?

Standish. Yeah. Good question.

Shultis. Could we formulate that in something I could write down?

Harnad. What role does semantic interpretability play in the meaning of "formality"?

Shultis. So this harks back to your earlier question about whether or not Post's game of Tag is a formal system or not. OK. There's no semantic interpretation there; it's just a game you play with scratches on a piece of paper, but there *are* rules that tell you which scratches to write down when.

Littman. I can't help it; I'm an empiricist. I'd like to see some examples of formality and some examples of informality. I feel like I'm up in the stratosphere here, which people accuse me of being in most of the time. But I think it would be interesting to try and make an extensional induction, pardon the expression, on those two concepts by identifying some really clear cases of informality and some really clear cases of formality. Then we could see whether the issues are process issues, or representational issues... So those are two questions.

Shultis. What you're suggesting, maybe, is a method for approaching these questions: "What is formality?" "What is informality?" by trying to find some central examples of the categories of what we mean by those terms.

Littman. Yeah. I assume that we want an intensional definition, but maybe... I don't know how to use the words right. Stephen maybe you can help me... Maybe the best place to start is try to enumerate some examples of formality and informality.

Harnad. ...an extensional definition of...

Littman. I'm suggesting that a methodology might be to develop an extensional definition; but presumably we'd also want a necessary and sufficient set of features, which is intensional definition. So it's "bottom up"—but it's methodological.

Fisher. It strikes me that the characterization of informality and formality is really a central theme of the whole workshop and needs to be discussed every day. I'm sure we're going to have new contributions to these questions every day, but at least three people that I noticed today, namely

Dave, Alan and Cathy did attempt to give at least partial definitions of these concepts during their talks today. We might want to go back and look at what they said.

Littman. I still want to see examples. I'll be a pain in the butt about this for three days.

Wight. I'm a little bit concerned that with a group of persons such as ourselves, who are all, if we admit it to ourselves, basically formalists, deep-down, because what we're trying to do is to organize the world around us...

Harris. I disagree.

Lethbridge. I disagree.

Littman. No, that's a good point he made... say what you said again. He said "We're formalists because we're trying to organize the world around us."

Wight. We try to make sense of the chaos that we're wading through

Littman. My question is: does it follow that, because we're trying to make sense of the world around us, we are formalists?

Several. No, not at all.

Wight. I should finish my thought. I'm a little bit concerned that a lot of the examples that have come up informally here are actually issues having to do with complexity which could otherwise be explained with fully "formal" existing systems of beliefs, symbol manipulation, and whatnot.

van Hoek. What do you mean?

Wight. A lot of the issues that we have been talking about, and I am thinking in particular of Larry's examples, have had to do with the imperfect user of a query database system. I mean, it was a more sophisticated application, but really it seems to me that, at a superficial level, we're dealing with this random event generator of a person interfacing with a complex application and cracking the hierarchy explicit or implicit in the design of the application by what should be, theoretically, foreseeable questions or foreseeable demands on the system. But it turns out to be very simple to create demands that are inconsistent or unreasonable for the systems that many of us here try to operate every day. To me, the engineer-on-the-street answer to that is that if I heard that the system that I'm supporting broke on a user, I would blame myself and say I have to go back to the drawing board, that there must be an inconsistency in the design. You know the famous incomplete program or inconsistent program: an incorrect program must be behind it. In other words, I haven't pursued the formalism to the level of complexity that I should have when in fact today, to me, it looks like I actually should say, "Hey! this is an informal interface! It's not broken; that's just the way it is. This is what happens, this is what it spirals towards, when you push the outside of that particular envelope." I'm not expressing this very clearly...

Reeker. It seems to me that there are probably different types of informality. We're talking about using informality; why are we interested in

informality in computing? One of the reasons might be that we think that there are some things that are in principle unformalizable, and we need to deal with that. Another might be that it is just too much trouble to try to formalize them because in fact there are so many of them that can occur, like all the different activities that a user might think of...

Harnad. What are the examples you have in mind? Surely not the truths of arithmetic, nor the facts of many-body mechanics, but what do you have in mind?

Reeker. Well, I'm thinking of natural language. In principal we might formalize some very large body of natural language, but in fact there are a lot of things that militate against doing so, a lot of computational difficulties, so we would be very content to work with some non-formalized version of it.

van Hoek. Or do a formal version that leaves out a lot of facts.

Reeker. Yes, one that's incomplete. And then keep building on that.

Wight. So informality is a precursor to formality?

van Hoek. It can be. That's the cognitive grammarian's position.

Reeker. That's one meaning of it. Historically, I think that's been true. People have dealt with things informally first, then formally. There are many cases where people have done things heuristically first and then come up with algorithms.

Harnad. Okay, so now you're starting to answer the questions, but we're still getting questions. We haven't gotten any questions on your half of the day yet.

Shultis. So, Steve, can you articulate a question?

Wight. Oh, Jeez...

van Hoek. How about: are we looking for formality in all the wrong places?

Wight. That says it as well as I could.

Littman. Can I try it? Maybe its something like this, but tell me if I'm not right.

Wight. Well, we can keep doing this 'til you get it right.

Littman. There's this thing about the processes people engage in which require informal reasoning. That's me using the computer. Then there's this thing about the underlying program—not the code, and stuff, but the design of the program. And those are two different things. And it seems to me that it's like a two by two table; you've got formal and informal, and you've got the processes that the system is supporting and the reasoning that the system does, and it seems quite plausible to me that you could have systems which reason about the formal processes that people want to do formally and about the informal processes that people want to do informally. You see what I'm saying? There's this distinction that you might want to make

	Formal	Informal
User's reasoning		✓
Computer's reasoning	✓	

between the processes the person that's using the system is doing and the way in which the system goes about its reasoning. Now what I was hearing you talk about was the clash between the two.

Shultis. So it's a distinction between what the system is doing...

Littman. ...how it's doing it...

Shultis. ... and how it produces that behaviour, or the difference between that behavior and the production of that behavior. Is that it?

Littman. Yeah, that's even better. Forget about what the person is doing. Just look at the behavior that's emitted from the computer system that the person relies on to perform their activities. Is that behavior construed as informal reasoning? It looks like informal reasoning could be generated from underlying mechanisms which are formal, or informal, or a combination of both. So, I guess the question is...

Wight. That is a very interesting point, but it's a little bit divergent from what I pictured in my head. I think I have a simple example; I'll just try this one more time. When we're talking about natural language processing, about processing "arbitrary" input from a user, the number of cases and frames and environments and empty dead branches that we have to waddle all the way to the end of and climb all the way back from indicate a large amount of complexity which we are navigating through using our formal systems and subsystems and sub-subsystems. Although such interpretation of natural language is formal, we have been referring to it as an informal computational interface, if I'm not mistaken. That's the usage of the NLP paradigm here so far.

Littman. Is there a question mark at the end of that?

Wight. That's a question mark.

??. It sounded like a statement.

Littman. Now I think it's coming closer.

Wight. I'm saying more what I wanted to say but I'm not helping phrase this question.

Shultis. OK, well, maybe you can think about that some more and we can formulate it later.

Here are some questions that I would like to ask, just to throw out here

maybe for some votes and we'll come back to this. Is there informality?

?? Absolutely.

van Hoek. I think we have to know what it is first.

Shultis. Yes, yes!

Fisher. If we have a definition, we can probably decide whether it exists or not!

Shultis. That's why I'm putting this on page two. If we answer those first questions...

Standish. We might want to answer to settle the second question first without knowing the answer to the first, first. And then we'll know...
[Simultaneous speech; jokes.]

Shultis. Is informality a good thing? Do we want it?

Carberry. If it exists. [Echoed by others.]

Standish. And if we can define it.

Shultis. Lots of good things don't exist. But is it a desirable thing, and why?

Fisher. It strikes me that the questions: "Can everything be expressed formally?" and "Are there limitations?" i.e. the questions you have on the previous page, are highly relevant. But the question: "Is there informality?" must either be answered "yes" by definition, or "no" by concluding that everything is formal. I don't understand the point of the question, I guess.

Shultis. The point of the question is to be provocative, and get other questions asked. [Laughter.]

Standish. You succeeded.

Lethbridge. I'd like to add a question to the list. It is: Assuming informality exists, how can we handle it pragmatically? In other words, what computational techniques do there exist for handling informality?

Littman. I'm going to claim that that falls under the distinction I was trying to make. Good question.

Shultis. So there's a question of: what is it, and how can it be done?

Littman. It is what it looks like. I mean, I might have a system that supports heuristic reasoning and the person that's using it is reasoning extremely informally, whatever the heck that means. Underlying such a system I might have the predicate calculus, Woody Bledsoe's theorem prover, all that kind of stuff. I might have this beautiful formal representation of heuristic problem solving, and so on. I've got a formal theory of that kind of problem solving, but the system sure doesn't look formal to the user. But I don't even know what this means yet. I'm still trying to understand these two places that informality might arise.

Shultis. So far there haven't been any questions specifically pertaining to informality and language. And since we spent a day talking about language, and informality in language, maybe people have some questions that are specifically focused on things that were said today in the talks.

Carberry. I'm not sure it's a question really of what informality is, because obviously we don't seem to know what it is. Everybody's got some

questions about it, but they come down to this: "What are the serious issues that we can't address with formalisms?" Out of that falls what we want informality to be.

Standish. Not clear. If I say here are items x, y, and z that current computers can't do, one faith that I could have is: do more of the same. Produce more formal systems that try to answer... fill the gaps.

Mundie. Yeah, until the point it feels like you're just pounding your head against the wall.

Carberry. But it's not what current computers can't do, so much as what we don't think formal theories are *ever* going to be able to do.

Shultis. That's good.

Lethbridge. Or better than that, what is it pragmatically more useful to do now? We might be able to in 20 years make it formal. But what can we do *now*, informally?

Standish. What is it we expect from this non-formal dimension, if we're able to define what it is?

Littman. Again, that's a technology question. Right?

Standish. It might just be an inflated-expectation question. What are the unrealistic, inflated expectations that we're trying to satisfy?

Littman. About the implementation of reasoning systems?

Standish. About what current systems can't do that we think they ought to be able to do, that we think informality might be a means to an end to accomplish.

Littman. In the implementation of the reasoning device, you mean.

Standish. If reasoning is needed to get whatever it is that we expect, yes.

Littman. Problem solving.

van Hoek. But also explaining language. I mean, your question applies to language, exactly. What is it about language that you can explain informally that you can't, at this point, at least, formalize?

Carberry. And that you don't think you'll be able to handle in a precise system.

Lethbridge. With reasonable effort.

Shultis. This sounds like it goes back to a discussion that we were having earlier about incremental incrementality as opposed to subductive incrementality.

??. Opposed to what?

Fisher. Yeah, that's a question, too.

Littman. Yeah, that's good. I agree.

Shultis. Yeah, the discussion of whether we can get there by just doing more of the same, making the systems more and more complex, and just building formalism upon formalism upon formalism, or whether there is something fundamentally wrong with that, something that says we're not ever going to converge on something that is truly intelligent.

Fisher. That question of incrementality I think should be on the list

somewhere.

Harnad. Should incrementalism be incremental? Tough argument. [Laughter.] Should incrementalism be subsumptive or can it be just incremental?

Fisher. Yes, good question.

Standish. Should the increments be small or large chunks? [Simultaneous comments.]

Standish. Replace the whole system or replace just pieces?

van Hoek. That, of course, is obviously a domain-specific question. In the field of linguistics the answer to whether we should or shouldn't do more of the same might be different than in, say, software engineering...

Shultis. If you'll excuse me for putting words into the mouths of those who have spoken today, I think that we've had people here today, like Karen, who basically argued for a kind of subductive incrementalism where you want to revolutionize things, and a major shift of framework is required, and then I think we've had people like Alan...

Harnad. Jon, it's "subsumptive".

Shultis. Sorry, I kept saying "subductive". I'm sorry.

Standish. Minor technical question. What is the difference between subduction, subsumption, abduction, and pure terrorism?

Shultis. Right. I was coming up with a portmanteau for "subsumption" and "abduction", for some reason.

[What was going on in Jon's head was probably something like this. "Abduction" was C. S. Pierce's term for the kind of reasoning step that leads to the invention of a theory, as distinguished from deduction and induction, which occur within an existing theory. In type theory, subsumption is the rule that any example of a type is also an example of any more general type, i.e., the special case is subsumed by the general. But it seemed to me that Steve was using the word "subsumption" to suggest something more, a kind of creative leap to a more comprehensive theory, but not by empirical induction (which is, I think, the accepted term for what we were calling "incremental incrementalism"). - Eds.]

Harnad. I would correct you to "subsumptive" in order to reflect the intended interpretation of the coinage of the expression.

Littman. Yes, formalize the concept.

Standish. Could you give us an informal definition of subsumption?

Harnad. I could give you an example.

Standish. OK

Harnad. In fact I did give you one. If you do a formal classical mechanics for a particular pool game with particular sized balls and a particular initial position, and a particular whack on one of them and whatever happens and if you do a theory for that, and then your next increment is to take another pool game and to do a theory for that and more of the same, your increments are going to be incremental 'til doomsday. You're not going to end up with general principles that account for all of these pool games. Once you come

up with Newtonian mechanics, you now have subsumed all of these; its not going to be case one plus case two plus case three, done more or less the same way, it's going to be subsumed by something that accounts for all of them by finding some kind of invariance that they share, and it's not going to be incremental in spirit any more, although it will be incremental in data because it will have incremented the set of data that you can explain.

Standish. When I heard that this morning, the line of questioning that I wanted to pursue was: what if I had a different set of increments, not just confined to different variations of the pool game. What if I give you kinematic experiments with projectiles in a vacuum and then projectiles in friction and then planetary orbits or celestial body mechanics and you keep on putting stress on the system and you have to keep building theories that account for more and more of these kinematic things. And finally you come up with Newtonian or Einsteinian kinematics. In one sense the original pool games converged on something that was very local and never stressed you to go beyond the boundaries and so you didn't create a subsumptive thing. In the other case, a non-convergent sequence of examples forced you to enlarge the theory, forced you to subsume a lot more. They were both incremental.

Harnad. I think if you're enlarging it just incrementally, then you're just adding more. You're saying, I've accounted for this, now I've got to account for this and this and this...

Standish. Yeah, but I'm going to keep popping orthogonal dimensions on you, forcing you to do what you would have done by subsumption and I'm going to call it incremental. Now what's the distinction?

Harnad. Well, if you increase the data domain in orthogonal directions, you're still just increasing the data domain.

Standish. Then I don't understand what subsumption is.

Shultis. Is subsumption different from generalization?

Littman. If you add new dimensions, then it's not clear to me that that's incremental in the straight sense of incremental that you had before. If you gain new insight into the old phenomena because you add a new dimension, then there's something subsumptive about that, but I'm not sure that it's the same kind of subsumption. I think the classic case of subsumption that you were talking about might be variabilization of a plan. So generating a script would be subsumptive because now you can handle all cases of fast food restaurants.

Standish. I know what it means in logic as a law. I can give you the formal law. It's something that includes all the other cases and rolls it into one neat summary and you don't need to mention all the others because the summary encompasses it all, roughly speaking.

Harnad. Maybe a complexity theoretic distinction would be the right one. Namely, the size of the putative algorithm that you're coming up with shouldn't be increasing with the order of magnitude of the number of incremental cases.

Fisher. I took your original definition earlier today, at least intuitively,

to mean that to be subsumptive you have to be constantly generating new theories about the data as opposed to just increasing the size of stored data.

Harnad. Well, let's say that your algorithm is not growing as fast as your data set.

Reeker. I like that descriptive complexity distinction.

Fisher. Can I go back to an earlier point? Dave had brought up the question about behavior, and I interpreted him to be taking the impersonation machine approach to the distinction between formal and informal. Is that what you meant, or is it not?

Littman. I was confused about how to say it, and Jon came up with the term "behavior". No, I don't mean... well, impersonation actually may have something to do with it. I'm not sure I understand.

Fisher. When this came up, it sounded like you were saying we could distinguish between formal and informal by virtue of whether our observations of its behavior were consistent with what we expected from machines versus what we expected from humans, along the lines of Turing's imitation-game approach.

Littman. No, I wasn't intending that, at all.

Fisher. Then what was the intent of that question?

Littman. Here's what I'm trying to get at. One thing we might talk about is how to build systems that are useful for people that are trying to solve problems. OK? And we find that the kinds of problems people solve fall into a whole bunch of different... well, I don't want to call them categories. But some of them require what we typically think of as informal reasoning. Heuristic reasoning might be an example. And others require reasoning which is much more formal. So if you're writing a specification document for a functional program, that's pretty formal. If you're trying to decide where to take your vacation, you might be doing some fairly informal kind of reasoning. That's one layer at which formality and informality appear. It's the kind of problem-solving process that the person or others, the computer, is engaging in. So that's the problem solving mode. Then there's the question of how we implement a computer, a support system, that will help the person who is trying to solve that problem to solve that problem. And here again it seems that there are some methods that you might say are clearly formal, like the predicate calculus, while others are not so formal.

So there can be very formal representations which are used to create, now I can say it better, systems which reason... no, which support problem solving activities that are either formal or informal. Then there might be technologies that we could use which we agree are *informal*, which *also* support that kind of problem solving. I'm just trying to get candidate technologies. At the layer where we're trying to build the system, you might imagine predicate calculus vs. neural nets, maybe. I don't know, I'm just trying to get a handle on this.

Harnad. What did you say before predicate calculus? What was the subject of that sentence?

Littman. I'm trying to get a handle on what this formality/informality dimension might be at the layer... no, I don't want to say layer... at the place where we're talking about the system that is supporting that problem-solving process, given that we're trying to develop a way to build technologies which will support various kinds of problem-solving.

Harnad. OK. I have a specific question about that. Are neural nets, interpreted as cognitive systems, formal?

Littman. OK. That's a reasonably good question.

Harnad. Not interpreted as neural nets, mind you, but interpreted as cognitive systems.

Littman. On the second level down, on the implementation level, can we say they are a clear example of a technology of informality? And is predicate calculus an example of a technology of formality? We may or may not agree that it is, but at least we should think about it...

Shultis. Well, I think that that gets back to your original point: let's get some examples.

Standish. Could we raise more attributes of possible informal reasoning, for example: probabilistic reasoning; Bayesian reasoning; satisficing; nondeterminism; deliberate use of incompleteness; reasoning with fallible or buggy plans; progressive debugging; beliefs; causality; buggy causality; superstition; things like that.

Littman. And these, you're suggesting, could be at either level, or is this at the problem solving level?

Standish. No, you were just talking about enumerating instances of technologies that might be considered informal or formal reasoning, if we agreed that they were, and we might not.

Littman. For each of these examples that you gave, I could imagine suggesting to a graduate student that they try to build a program which would do that kind of reasoning in predicate calculus. Do you agree?

Harris. And that would make them formal?

Littman. That would make them formal. Conversely, I could imagine...

Standish. The trouble is, these things are *reducible to one another*, and you can have things at different layers. At one layer, it's Bayesian probabilistic, or it's indeterminate, non-deterministic. On another layer, it uses Boolean algebra and it's completely reducible to predicate calculus.

Fisher. That was part of Steve's point. If you're going to talk about neural nets, you've got to say which level or view you're talking about.

Standish. Which layer.

Fisher. The level is absolutely critical.

Standish. Representational layers may differ in their mechanics and yet aid each others' gross behavior.

Kozma. Isn't informal reasoning an oxymoron?

van Hoek. I thought formal reasoning was an oxymoron! [Laughter.]

Shultis. Well, let's get some examples of each of those then. What is an example of something that's informal, an informal idea, or...

Kozma. I thought that people referred to heuristic systems as informal reasoning... I guess in one sense they are, but in another sense, in order to implement the system itself, you have to have some formally defined rules for it.

Fisher. That reminds me of the old question: "Are heuristics algorithms?" And it's pretty hard to argue that they're not, or at least that their implementations are not. And of course if they're algorithms, they must be formal. Again, it's the issue of levels.

Wight. Incorrectness does not invalidate an informal system.

Lethbridge. I think informality is a relative thing. When you say something's informal, or say something's formal, you've got to say what you're referring to, what it's relative to. I mean, you implement an algorithm: that's obviously a formal thing. You're going to get the thing running on the computer. Relative to the computer, it's formal, but relative to some higher-level process, or to the user, it may be very *informal*. Heuristic algorithms would be informal in that case.

Shultis. What makes the algorithms formal relative to the computer, and informal with respect to the user?

Lethbridge. It's got a precisely-defined semantics. I like that way of looking at things.

Biermann. Or another issue is proof of correctness. Maybe proof of correctness has something to do with formality.

Lethbridge. If it's got a precisely defined semantics, you can prove stuff with it.

??. Optimality, completeness. Any other things you'd like a formal system to have?

Lethbridge. Optimality? No, it doesn't have to be optimal.

Fisher. I don't know if this is redundant or not, but Dave Mundie's talk actually went over a number of characteristics that we normally associate with formal systems. And certainly in the DARPA Prism effort at Incremental Systems we have been operating on a principle that if you violate these characteristics of formal systems, then you have an informal system. A review of the points that Dave was making might be useful.

Shultis. Maybe we could put those up as proposed characteristics for what it is to be a formal system, and proposed characteristics of an informal system, and talk about them. Just to have something concrete in front of us.

Reeker. There's another question. What about creativity? Nobody's really mentioned it...

Shultis. Oh, golly...

Reeker. That seems to be something that's pretty hard to formalize. I mean Gödel himself suggested that... it's a famous memorandum...

Shultis. It's interesting that you bring that up... *[Brief tape gap.]* ...and so far we don't have any empirical evidence that would lead me to believe that either analysis is correct.

Reeker. The example that Gödel gave was coming up with more and

more powerful axioms of infinity. People that have done that over a period of time. Could a machine do it?

Shultis. The import of my originally putting up Gödel's incompleteness theorem is that what it says is that in any formal closed system, you're doomed either to inconsistency or incompleteness.

Harnad. Not in *any*...

Shultis. And...

Harnad. Not in *any*...

Shultis. Right.

Harnad. And certainly not in the propositional logic.

?? It has to be sufficiently rich.

Shultis. Excuse me.

Reeker. No, but this is a different thing I'm talking about...

Harnad. But we're talking about formality. And as a matter of fact, propositional logic is a formal system. So this can't be something wrong with formality.

Shultis. I think Larry's point is that there's something rather open-ended about what people do, something that is not captured by any formal system. Therefore you have to conclude that whatever it is that people do, it's not consistent or it's not complete, or it's both. So something else is going on. What is that something else?

Maybe we can structure a discussion around that question. Dave put these up primarily as things that people often cite as being characteristics of formalism and at this one extreme here, we've got a situation where we would include things like Post's Tag. Does everybody here know Post's tag—if I talk about that?

Standish. I know the correspondence problem. But what about Tag?

Shultis. The point of Tag is... OK. You write down a string of ones and zeroes—any string of ones and zeroes that you like—and you start at the left hand end of the string. And then there are two rules. And the rules are that if you see the first two characters are zero zero you cross them out. And of the first two characters... if the first character is... I don't remember exactly...

Harnad. It doesn't matter, you can invent any rule you want...

Shultis. It doesn't matter. It's something like this. If the first character is a one, then you cross out three characters and you write 1101 at the end of the string, the other end of the string. And so you just follow these two rules, *ad infinitum*. And now the question is, does the game ever terminate?

van Hoek. How long is the string?

Shultis. Well, the string eventually reduces to—if the string reduces to zero zero you cross it out and the game is over. And so starting with an arbitrary string of ones and zeroes the question is: does the computation terminate? In some cases it does and in some cases it doesn't. In fact, it's undecidable which, for an arbitrary string—it's in general undecidable whether or not the game will terminate. I've probably got the rules wrong.

But it's a fun thing to unleash undergraduates on in teaching them data structures. Build a program that will play Post's tag and make them do it for things where they have to come up with such complex encodings that they can't just use ones and zeroes because there isn't enough memory in the machine.

Harnad. The point of that...

Shultis. There's a system there, which is used, as Tim said earlier, as a theoretical device to investigate certain classes of mathematical systems where there's no interpretation intended at all. Where you're only interested in the properties of the reductive rule system. So that's one extreme of things. And then there are other characterizations here of informal systems.

Harnad. I think lots of games fall into this category. But the question is: are games in any interesting sense a formal system when there's no semantic interpretation other than the middle motions that you're going through in themselves? I think it's important... somebody over here raised the question about interpretability constraints. And I think they have a lot to do with what a formal system is. I mean, what makes formal systems interesting is that they will bear another weight. A weight that's outside of them. Namely the weight of the systematic semantic interpretation. If they are simply what they are defined as being, namely, do a squiggle then do a squaggle, then do two squiggles, then they're nothing! Nothing! They're just...

Reeker. They may be fun!

Harnad. They may be fun, but they're neither formal nor informal...

Standish. Suppose you're trying to build a theory. Just say you're a professional mathematician and you may come up with a puzzle. Are there an infinite number of primes, or is there just a finite number of primes? Now, at first that's just a puzzle, OK? But if I settle it one way or another, certain very dramatic things happen in the rest of the theory.

Harnad. But the whole point is that numbers have interpretations, namely numbers. Prime numbers are not just squiggles and squaggles. They're not just things that you say about scratches on paper...

Mundie. Yes, they are.

Harnad. To a formalist they are, indeed, things you do to scratches on paper, but all of those scratches on paper have this remarkable property, and are systematically interpretable as truths about numbers.

Shultis. What's a number?

Standish. Why should I care whether a number is prime or not, whether it has this funny property of being divisible only by itself and one?

Harnad. That's another issue. Why do we care about capturing human cognition?

Standish. At first that's really an irrelevant property unless I can demonstrate its utility by some other application.

Harnad. I think there are two different motivations that are being conflated over here. One of them is the one that was raised initially, which

is the notion of the intended interpretation, which clearly prime numbers have. And the other question is why we care about prime numbers, but that's a different motivation of course.

Shultis. But when you say that mathematicians' squiggles and squaggles, or whatever, are things that are about numbers...

Harnad. Umhuh.

Shultis. ... there's a serious question there, which is: what are numbers other than squiggles and squaggles...

Harnad. Yes, but we're not doing metaphysics here. I happen to be a Platonist, but what difference does it make? That's not what we're talking about here, is it?

Kozma. But you can't define what numbers are either.

Harnad. Whether I can define them isn't the issue, and whether I'm right that they're Platonic abstractions as opposed to the invariants that all extensions of them share, doesn't matter because we're not doing metaphysics here. God, let's abstain from metaphysics! But the fact is that the intended interpretation is *not* that they are squiggles and squaggles. That's all. Never mind which foundational preference we have about what they are. Let's just agree that they're not just squiggles and squaggles.

van Hoek. That's true.

Harnad. Even the formalist admits that the formal system has an intended interpretation. Even the formalist.

Kozma. An *intended* interpretation.

Harnad. Yeah. So in other words something outside the system that...

Kozma. But the interpretation itself can't be formalized.

Harnad. It doesn't matter. The formalist does not claim that what I'm doing is playing around with uninterpretable squiggles and squaggles. That's not what he's doing. He's not playing the bead game. He's not playing Post's Tag and that's relevant, I think, to what a formal system is.

van Hoek. Any formal system or just an *interesting* formal system?

Lethbridge. That's a good distinction.

Littman. I agree.

Harnad. No, I don't think so. I don't think so.

van Hoek. Well, which one is it then? You're talking about *a* formal system.

Harnad. Well, I think we wouldn't be talking about formal systems in the context that we are now, if we meant anything but the interesting ones, the ones that are semantically interpretable.

Shultis. Well, let's talk about interesting formal systems just for a minute. Let's take a little chapter out of model theory and formal logic. Quite some time ago, Church, as we all know, invented lambda calculus. And his intended interpretation for lambda calculus was that lambda expressions were supposed to denote functions over some domain. And of course you get into this minor problem that, in the untyped lambda calculus, there are no nontrivial models. Or so it appeared at first, because there was

this problem of having basically an isomorphism between the domain and its own function space. And so for some time, there was really a serious question about whether or not the formal system of lambda calculus had any meaning at all, of whether or not it was an interpretable formal system. Nevertheless, it turned out to be quite useful and an interesting thing and so on and so forth. It was only, of course, in the late sixties that Dana Scott did his work on reflexive domains...

Harnad. And so what conclusion do you draw from that?

Shultis. ... and came up with an interpretation, where, in fact, you can have a domain semantics for it. Now on your characterization of formal systems, what would you say about all the stuff that was done in lambda calculus...?

Harnad. If you had a formal system, if you took Post's tag, and you said to yourself: I'm looking for a non-trivial interpretation of Post's tag, by which I mean that the interpretation will not just be the game itself, you're in a legitimate epistemic game, right? It's a system looking for its interpretation. If you find an interpretation, then you've got a formal system worth talking about. If not, you've got a bead game.

Standish. No, I'd say, if they *are* bead games, then sometimes bead games can settle important issues in the evolution of a mathematical theory. Example: in the theory of context-free grammars, there's a question about whether deterministic push-down can recognize certain languages. And I can give you a language of unmarked palindromes that no such push-down machine can recognize. Now, is it a bead game? Because the grammar for it goes: x goes to xa , or x goes to a . And you'll never know with a push-down automaton how to recognize that. That's a bead game, but it settles something very important about the class of mechanisms that can deal with context-free grammar and cognition. So bead games can sometimes settle important foundational branch questions in the evolution of a theory. Why should we deny them that pragmatic utility if they don't have interpretations other than that?

Reeker. But here we're talking about informality in computing and I think that probably they don't play that sort of role. I think in the case of Jon's example, for instance, the lambda calculus was interpreted in the sense of Church's intended interpretation of it. Now, if it turned out not to have any actual models, I think that's probably a good example of the sort of thing that people reason with all the time. They have systems and although the systems aren't maybe totally formal, they have an interpretation in the person's mind and yet they may have inconsistencies or...

Standish. How about the paradoxes, like Russell's paradox? Set of all sets that aren't members of themselves. That's a bead game, isn't it?

Harnad. No. Wait. Let me take it one at a time. Post Tag: if Post Tag turned out to be a terrific weather predictor that wouldn't make it a formal system for me. It would just make it useful. It would turn out to be that the bead game had some other uses. The Russell thing is getting far afield, but I

don't think... I mean sets have intended interpretations, so you're not dealing with squiggles and squaggles. You're again going into something where it looks like an abstruse, uninteresting question but actually is an interesting question, right?... about whether sets, you know the question about those sets. But it's not a question about the status, formal or nonformal, of a particular formal system. It's a question about motivations. Why are some people interested in crazy problems like...

D'Ambrosio. It seems to me that the question of whether or not a formal system has to have an interpretation in order to be worthy to be considered formal is connected with something that was brought up earlier about the intended use of the system in the following way. If we allow a formal system to not need an interpretation, then it seems to me that anything we can write and implement in a computer is going to be a formal system, since according to that rule, we'll be able to interpret it that way. But more importantly, if we require that there be an intended interpretation, then we can ask, do all the manipulations that we perform in that formal system correspond—generate, in fact—valid interpretations?

Harnad. Exactly. That's the systematicity criterion.

D'Ambrosio. And if they don't, then we can talk about in fact being an informal system because it doesn't fully satisfy this mapping property. And that's the only case in which we can create interesting informal systems.

Standish. Could you do that again? That went by way too fast.

D'Ambrosio. OK. Let's try it again. If we allow formal systems to include purely syntactic ones that have no intended interpretation, then every program that we write is trivially a formal system. If we require that a formal system have associated with it an intended interpretation, or as a part of it, I should say, an intended interpretation, then the issue that was brought up earlier comes up. That is: do the manipulations in the formal system in fact preserve this intended interpretation or not? And if they don't, then we can say, that, well syntactically it's a formal system, which is the only thing we can implement. In fact it's an informal system in the sense that it is not guaranteed to always preserve the intended mapping.

Harnad. This property, which Fodor calls systematicity, was packed into the subquestion that I raised about systematic semantic interpretability because it's not just hermeneutic interpretability we're talking about, like horoscopes and the positions of the stars. When a formal system is semantically interpretable in the systematic sense, it's got to have this property... I mean it's a model-theoretic interpretability, right? The model has to be a model of the syntax. As a matter of fact, that's a good way of finding a model. Well, that's a good way of showing that a formal system is inconsistent, is to find... what... now I'm getting lost in my own syntax!

D'Ambrosio. ... is to generate some transformation under the rules of the model, under the rules of the system, that is not...

Standish. You appear to be trying to set up something that dismisses as not formal something that is merely frivolous but formal, like a computer

program that does nothing important.

D'Ambrosio. Well I'm saying that that's not interesting to us in the sense of trying to understand the dichotomy between formal and informal because everything is interpretable as formal under that interpretation.

Standish. But what if what is meaningful to me is frivolous to you and vice-versa. For example, I might build this computer program that choreographs dances for Skinner's pigeons and uses a Chomskian generative grammar to produce LeBas [?] notation for a pigeon dance. And you think that's frivolous, so to you it's not formal...

Reeker. But not in the sense that we're talking about.

??You're still producing the intended...

Shultis. Can I summarize your question, Bruce, in the following way? Is an informalism a broken formalism, in the sense that there is an intended interpretation, but that the formal manipulations which are supposed to correspond to something don't actually preserve or mirror the way in which it is supposed to work?

D'Ambrosio. Or *may* not.

Shultis. Right. And so it's an attempt to capture this, but it's not completely accurate and so it's broken in some way.

Mundie. I'm confused. I would have said that it was in exactly the case where it didn't correspond to the intended interpretation that it was a *formal* system.

Standish. Can I try another cut. We had this debate in the programming language community about whether there should be a formal definition for Ada, a new programming language. And some said, you haven't really defined the language until you've given a machine that works by discrete symbol rules that always decides an answer, that always decides the behavior and acts as a reference so that any question about how does it behave or what does it mean can be settled by running the formal model, the exact, discrete, completely defined, crisp, convergent thingamajig. Others said, nah, nah, let's do it in English, because we can understand the implications of the English. But we may not be able to understand the implications of this model. And now, what we get is an English definition of Ada and then a squadron of Ada language lawyers. And it looks like you may be able, *ad infinitum* and indefinitely, to come up with Talmudic variations, and unsetttable questions, and volumes could be written, and test sets grow, to decide what the language is. Is *that* an informal thing, if it has a non-convergent set of interpretations and appears to burgeon beyond any reasonable constraint? And is the U.S. constitution therefore similarly informal?

Fisher. I'm not sure that's the way it is in the Ada community. Yes, you have the language lawyers, but they act as a judicial body that makes interpretations. Those interpretations supposedly reflect the "good" of the community, but often are at odds with any normal interpretation of the English in the Ada Language Reference Manual. Ultimately the definition of Ada is provided neither by the L.R.M. nor by the language lawyers, but

rather by the validation suite.

Standish. Yes, in the case of Ada there's a test set, and there's a defined procedure for defining whether a compiler is or isn't one of these: whether it passes the test set. So it's operationally defined. Now I would call that a formal filter. It decides issues. For Algol there was none such. And for the U.S. Constitution, I suppose you have the Supreme Court to the degree that you can interpret precedent. But then again, that's elastic.

Fisher. It changes over time.

Standish. Yes, it changes over time. What do the founding fathers mean?

Kozma. That's more decidability than formality.

Standish. Yes, but decidability might be connected with informality.

van Hoek. I though formality was decidability.

Reeker. Not really. No?

Harris. It's not really?

Kozma. Well, let's go back to number theory. Can Fermat's conjecture be proved?

Standish. Nobody knows. We don't know yet.

Fisher. Yeah, we do. Two months ago. It's been disproved. It was in the Times.

Standish. Meaning what? There's a counterexample? I never heard of that.

Kozma. How about Ramsey theory?

Shultis. Can I try to summarize all the views on informalism? What I'm trying to do right now is come up with some ideas on what informalism is. Tim, would you accept this summarization of your position on informality? On your proposal? Something is informal if the interpretation is open. And it's something that is open and discussed. It's under discussion. It's under creation. Is that a fair characterization of what you were saying?

?? Hm. I like that.

?? I don't.

Standish. It pushes the problem back one level, onto what we mean by interpretation.

Lethbridge. That's why I said it's relative. It's relative to how you interpret the system...

?? Exactly.

Lethbridge. Either by machine or by the human. Informally in one sense and formally in another.

Fisher. I would like to propose an alternative. I have certain intuitions about what formal systems are, about certain characteristics that I observe in them, and certain classes of problems that I want to solve; problems whose solution is precluded by the characteristics of formal systems. And so what I would take to be an informal system is one which in fact has properties different from those of the formal systems that I observe—properties such as incompleteness, tolerance for inconsistency, intensionality, imprecision,

analogicality, and prototypicality.

Harnad. Can I try a rival? A hand at informal and formal? It's not a pun on the word formal that formal also means form, it doesn't just mean conventional and efficient in notation. It also means form. And, I forget, when you were going through it you mentioned that in a sense, you said in a sense, but I think it's literally true, that what we're doing in the case of the formal system is manipulating the shapes of the symbol tokens on the basis of rules that operate only on their shapes. And those shapes are arbitrary in the sense that they're not related in any way to the things that that symbol system can be interpreted as meaning. So if I have—let's just pick natural language, even though it's a vexed case—the shape of the symbol string “cat” is interpretable in English as referring to that hairy creature. If English were in fact just an interpretable but uninterpreted formal system (which it isn't), “cat” would be interpretable systematically as meaning cat in expressions like: “The cat is on the mat.” “The mat is on the cat.” “The cat ran.” etc. But its shape would not be related in any non-arbitrary way to the thing that it stands for. That's what formal system means. It means a syntactic system consisting of objects, physical objects, symbol tokens, that are manipulated on the basis of rules that operate only on their shapes. But that has the second remarkable property that all of those systematic goings on can be interpreted as meaning something like: the cat is on the mat.

Shultis. So is this your definition of informalism? I just want to make sure I capture these things. What makes something informal is that it's a non-arbitrary symbolism?

Harnad. That's what I would propose. Yes.

Standish. But by shape, you don't mean a continuum of elastically, reformable, possible shapes? You mean a system to discriminate among a finite number of them, don't you? The letter “e” can't occur in infinitely many variations which play a role in your theory.

Harnad. In practice, the symbol tokens in formal symbol systems, whether they're scratches on paper that notate arithmetic or computer programs or the graphemic systems of natural language, etc., the symbol tokens tend to be finite, discrete objects. You know, the issue about using... I mean, analog computing as an interesting thing to talk about as a separate side issue, and *relevant* here, and it gets into the domain of role of arbitrariness of all of this. But I think the model for a formal system is discrete, a finite set of discrete symbol tokens manipulated according to a finite set of rules.

Standish. Right. That's all I meant by the finite symbol system hypothesis.

Shultis. Something that I'd just like to say, by the way, since something you said reminded me of it, is that the informal/formal dichotomy is a false thing. There's really a gradation here. There are things that are more formal, and things that are less formal. Of course, when we talk about informal and formal, we're really talking about the extreme poles of this continuum. And so there's this gradation. Cathy?

Harris. Before Steve Harnad spoke I was going to offer up a try at defining formalism and informalism. And I completely agree with how he described it. Absolutely. I think he summed it up very well. However, I was going to phrase it a little bit differently...

Shultis. Steve is on your committee and you haven't defended yet?
[Laughter.]

Harris. No, I just think he's right. I was going to refer to a sentence or two from my talk, when I quoted Chomsky. I think that when Chomsky was trying to urge a certain rigor in the linguistic community, he wanted a formal description of the grammar. And the way he described it was explicit, so that it wouldn't require any of the intelligence of the understanding reader. The system could be described without relying upon an inference process. So an example of an explanation or a theory that is informal is a type of explanation that uses intuitive folk terms, such as: how do people produce novel utterances? Well, they do it on analogy to other forms they have already heard. That's an informal statement. And the formalization of that might then be a specific mechanism like a set of recursive rewrite rules, or a PDP system that learns to associate a mapping and can get a type of analogy out of that. So that brings me to the final thing I'll say, which is that for the top question of whether neural nets, considered as cognitive systems, are formal, I definitely see neural networks as formalisms, because they are symbol manipulation systems of the type Steve Harnad described.

Harnad. No, I introduced that just as a question. And I think that the answer to the question is no!

Littman. I'm uncomfortable... I think that if you say formal equals arbitrary symbolism, that's OK. Did you mean that?

Harnad. I missed that. I take back what I said. I didn't realize that the question was formed in that way. In that case, the answer is yes.

Littman. What's your question? I don't like the informal equals non-arbitrary symbolism. Formal equals arbitrary symbolism. OK. I really don't know what the concept of informalism is, except by negation, and if that's what we mean, then informal systems are those which aren't formal and if that's what we agree upon as a useful distinction, OK. I just don't know about that.

van Hoek. We bring a lot more to them than just...

Littman. It's a concept we're trying to make more understandable.

Harnad. There's a request for examples, and there are actually good examples. Johnson Laird, does anybody know Johnson Laird's book *Mental Models*? He gives good examples of informal systems, when he has his people trying to verify the truth of inferences by building up models in which they simply just imagine arbitrary shapes, and then they put arbitrary constraints on these shapes in such a way as to rule out certain conditions and rule in others. And they're definitely not formal... you can't deduce from these little informal models that they make for themselves the truth of all possible propositions, and in fact these informal methods are demonstrably incomplete

and inadequate and often wrong. Yet they are what people often use to solve long syllogisms and things like that. I found them very analogous to the kinds of things you're describing in Ron Langacker's theory—the kinds of little structures that are being used inside the head in order to...

Littman. So it seems to me that what you're getting at, if I understand what you just said, is that formality and informality are properties of the device which is interpreted in the sense of a computer program being interpreted by the machine, because you're saying it's a formal system if it's interpreted by a computer. And I think Cathy was just saying that a second ago. And that's because it doesn't understand what it's doing.

Harnad. No, but it's more specific than that. I was trying to give it as a positive example for the non-arbitrariness of the informal system. That's right, for the non-arbitrariness, because the reason these mental models work is because they try to take... You know, the syllogisms are really just formal syllogisms. They're just really a set of propositions which if you could do the calculation, you could figure out from the truth tables whether they're true or false. That's the formal way to do it, although as a matter of fact, truth tables are a vexed case. But the way Laird's subjects do it is completely informally, that is, they start constructing a case that matches a model, literally in the model-theoretic sense. They create an informal model, and based on what's true of the informal model, they make inferences about what's true about the problem that's given to them.

Littman. The model they built is clearly arbitrary.

Harnad. No, it's non-arbitrary, because what they're doing is: Oh, OK, this can be there, so I'll put a pebble into this...

Littman. That's what a computer can't do. And it wouldn't matter, if the symbols they were manipulating in fact were being manipulated only by virtue of their shape. It's the fact the people are interpreting them as they manipulate them and are making a mapping to the problem that they're trying to solve.

Harnad. No, I don't think so, because as a matter of fact Johnson Laird is agnostic about arbitrariness and non-arbitrariness and he gives an example of a mental model that a computer could perfectly well...

Littman. I'm just talking about the example that you gave... but that would be a formal model, because there would be arbitrary symbolism.

Harnad. Well, I don't know.

Littman. With respect to the computer. With respect to the computer, which is what I was suggesting before formality might require...

Harnad. I'm committed to some really bizarre consequences. For example, a dedicated computer in my sense is not a computer. A dedicated computer—that is, a computer that is irrevocably wedded to its peripherals—is not a computer and it's no longer just a pure formal system because some of its interpretation are fixed.

Shultis. Is it fair to say that what you're trying to get at is that arbitrariness is in the eye of the beholder?

Littman. Yeah. In two words or less.

Shultis. So whether or not something is arbitrary is...

Littman. It doesn't matter what the system is, it's what the computation is being performed on. The fact that the computer has no understanding of...

Shultis. Let me give an example from Ron Langacker's book. He talks about onomatopoeic expressions and in fact just the raw use of noises. I say, "He went 'Ah!'" That "Ah" is a perfectly decent piece of my language, and it's not an arbitrary symbol. It's itself. It's a noise that I make, and that I incorporate into the sentence to represent itself. And it is nonarbitrary to me and to you, because we're all people. But if you're a mosquito, and I don't know whether or not mosquitoes hear anything, but some animal, say, that has a very different way of interpreting sound waves, that, in fact, maybe a dog or something—at any rate, the noise I made may bear no resemblance whatsoever to that animal's auditory system, to the noise that was actually made, and he may say OK, that's an arbitrary symbol. He means by that something else. And he's just using that strange arbitrary symbol to mean this noise. So to some extent whether something is arbitrary or not is a property of interpretation.

Littman. It may in general be a property of, an issue of interpretation. I was thinking only of the relationship between the symbol system and the device which is performing operations on those symbols, which I think is close to what you're talking about as the symbol grounding problem.

Harnad. Are you agreeing with Jon? Because I would violently disagree.

Littman. Yeah. I don't think so.

Harnad. I don't think the interpretation is in the mind of the beholder. If you throw aside all mentalism and simply talk about the system itself, some circumscribed gadget, whether it's a pure symbol cruncher or has all kinds of analog peripherals, and you ask, is the symbol—the state—(because it will be a state of the device, that I'm calling symbol x) arbitrarily related to what x can be systematically interpretable as meaning? I think the answer to that is not in the mind of the beholder at all, but in the causal interactions of that system with whatever it is, that it allegedly can be interpreted as meaning. In other words, if that thing goes around calling cats cats, stroking the things that it tokens as cats and all the rest of the stuff systematically interpretable til doomsday, then it's not parasitic upon my interpretation at all. It's intrinsic in the causal relations of the device with the objects and states of affairs that its symbols could be interpreted as being, as standing for. Do you understand?

Littman. It looks like a duck. It walks like a duck. It talks like a duck. It's a duck!

Standish. The duck test.

Harnad. No. It's not that.

Shultis. Are you saying that there's something about cats which...

Harnad. I'm saying the question about arbitrariness is about how it is

that the symbol token "cat" is being tokened by that system and we have to be specific about what the system is. We can't partition it into interpreter and what have you the way that you did. There's a thing there that has a symbol token in it, "cat". And my question now is: is that symbol token arbitrary in relation to what it can be interpreted as standing for? The arbitrariness is not in my head.

Shultis. But, but... Let me just try to respond to this. Are you saying that a non-arbitrary symbol would be one in which there was some causal relationship between the symbol and the thing itself?

Harnad. Yes, whereas a pure formal symbol system simply has the property that it has states that can be systematically interpreted as corresponding to...

Shultis. Just to clarify something. I would consider that causal relationship to be part of the process of interpretation that goes on in that device.

Harnad. Maybe there's nobody home in that device. All it's doing is squeaking around in the world. Why do mentalistic interpretations have to come into it at all?

Shultis. Interpretation to me means the use of one thing to represent another and the process of doing that, if it's a causal link, then it's a causal link.

Harnad. Under special conditions I might be inclined to agree with you. But as a general principle I think that that's saying too much.

Shultis. So there's a proposal up there. Are there other proposals? Dave.

Fisher. At the risk of putting words in Alan's mouth, in his talk he took your view that there's a spectrum of formal and informal, that it isn't just a hard boundary, and that it has to do with the degree of specification...

Harnad. Specification of what?

Biermann. This actually is compatible with the business of interpretation. If you specify completely, then the interpretation is complete. If you specify rather weakly then that would be rather informal and the interpretation is lacking. So I think there's a certain compatibility with the point of view that you were giving and the point of view that I was giving.

Littman. Wait, wait. He's got the non-arbitrary symbol.

Harnad. Mine is non-arbitrary. You're agreeing with the middle one.

Littman. The middle two, right. Yeah, Steve is non-arbitrary symbolism which is...

Harnad. I don't agree with the middle one because it's too epistemic. It really does make the formality just in the eye of the beholder. If you understand the system completely. The very same system, according to that definition, the very same symbol system can be informal if I don't know what the correspondence is or only know partly what the correspondence is, and formal if I do.

Littman. Yes, yes.

Fisher. But Alan doesn't address that. Alan doesn't address the issue of interpretation.

Littman. Why don't you like that idea? The idea that if it's running on a stupid computer it's a formal system and if it's running on us and we know what these things are it's an informal one. Sorry to bring mentalism into this.

Fisher. The distinction isn't between people and machines, but between implementation and higher levels of interpretation. I would claim that the implementation level is formal, or at least mechanistic, for both people and computers; but that at higher levels at least people think and act *informally*. I am interested in how that informality can be exploited at the higher levels in computers.

Shultis. If you define formal to be what can be done with a computer then in fact you've reduced the title of the workshop to an oxymoron. And I think that Steve's point about when a computer is not a computer is a relevant question there. If it becomes fully embedded and engaged in the world and its functioning is defined by its engagement in the world, and its involvement of the world in its processing and so forth, is it no longer a computer?

Littman. But that doesn't mean that it's formal or informal. I just don't see how it speaks to that issue. But I'll shut up now.

Shultis. Well, I don't think that we're going to arrive at a conclusion here.

van Hoek. I just feel like speaking up as one of the few linguists here. I'm not used to arguing about these terms, and I wanted to say how I understand formal and informal, just from my background, and I want to know how this fits with some of the discussion going on. To me, it's very strange to talk about a machine or a system or anything else as being formal or informal in an abstract sense. I'm used to using the terms formal and informal as descriptions of theories or descriptions of analyses. That is: what my background has taught me is that a formal theory or a formal analysis is one in which all the symbols are defined and everything is laid out with explicit rules so that in a sense an automaton can do it, or as Chomsky said it doesn't require human interaction or understanding. To put it more bluntly, it's an analysis with no fudge factors. Where every single bit is laid out very explicitly. That's my understanding of what formalism is. And informalism is when you can't lay out every single factor—the term that has come up several times is non-specificity. And then one question that arises in my area is: when is it appropriate to be concerned with getting a formalism? Of course the cognitive grammar answer is: not too soon, if it precludes developing intuitively satisfying analyses for which you don't quite know yet how you're going to spell out every single bit. You wouldn't want to forgo pursuing these analyses just because you don't know how to spell out every single bit. And then the other question is: will we ever have a complete formalism? Will we ever spell out every last little bit? And the answer, I

believe, is: probably not for language, in the sense that spelling out every last little bit might mean spelling out every last neural connection in a native speaker's head. So, in a certain sense we'll never spell out every last little bit.

Shultis. But even if you do that, though, there's the fact that (and I don't think this is anything you'd argue with) even if you spelled out every last bit and every last connection in a native speaker's head, you still have the problem that there's a social phenomenon of language, which is not captured by that.

van Hoek. You don't know what's going to happen to them tomorrow and what they'll say and how that will affect the neural connections, exactly, yeah.

Shultis. And so what do you do? You go off and do that detailed description of the physical setup for every creature on the planet and do you still have it? Probably not. And the thing is that you may have a description of a way to reproduce the mechanism, possibly, but is that a description of language? No, because the categories are wrong. It's a description of a mechanism that implements language, but it's not a description of the language.

van Hoek. I don't think you can have a fully formal description of language. With natural language, I do not believe you can, at all.

Shultis. That's an interesting question...

van Hoek. The question is how much formalization is useful for our purposes.

Shultis. That's an interesting question. I'd just like to take a real quick straw vote on it. How many people here believe that some things are in principle unformalizable?

Littman. Well, wait.

Carberry. What do you mean by formalizable? [*Pandemonium.*]

Littman. This is an informal straw poll.

Carberry. I want to know what you mean by formalizable first.

Shultis. If we take it seriously, one of the questions is: there is a notion that people have that says that the world is reducible to formal theory.

Harnad. It's called Church's thesis.

Shultis. Yes. Everything is reducible to some formal description.

Littman. That one can describe everything...

Harnad. So you asked us whether we believe in Church's thesis?

Shultis. Right.

Reeker. Well Church's thesis doesn't really say that. It only says that for recursive functions...

Harnad. Yeah. You can interpret it as referring to just abstract objects. It's been interpreted as referring to physical systems as well, and certainly in the sense of Turing equivalence it has.

Littman. But now that we've constrained this question down to a

manageable one, how many people believe Church's thesis?

Fisher. That's an important question. I think it was an assumption in what Jon said that Church's thesis addresses the entire world. And I don't agree with that assumption. In particular, I believe that incompleteness is essential and that we can never have a complete description of any part of the physical world.

Shultis. Obviously there's a question of how you interpret Church's thesis, of what its scope of applicability is.

Reeker. Yeah, I think that's it. For instance if you talk about the sort of traditional heuristic problem solving program. That works in some cases. In some cases it doesn't. Now, you can say it's a totally recursive function, that it's going to halt, that it's going to say "no" or "I don't know". But from the standpoint of its usefulness, in fact in certain cases it might as well not halt. It's not giving you an answer to your problem.

van Hoek. I'm just curious. How does Church's thesis deal with the Heisenberg uncertainty principle? I really don't know.

Shultis. You need to read Roger Penrose's book.

Standish. Heisenberg says that you can't really measure the state of the world because the act of measuring it may disturb the thing being measured and you can't get within a certain envelope of uncertainty about the thing you're trying to measure because of the disturbance created by your measuring instruments. So you never could get a symbolic formal state and a state transformer in the symbol system sense sufficiently well described that they could capture the full determinism of the universe.

van Hoek. Right.

Harnad. An uncollapsed wave packet is not equivalent to a Turing machine. But there's strong motivation for saying you should only talk about collapsed wave packets if you're talking physics. And the collapsed wave packet has no problem with being Turing equivalent.

Shultis. The collapsed wave packet is a point of observation. But the fact is, in order to project the unfolding of physical events, you have to deal with the mathematics of the uncollapsed ones. And, you know, quantum mechanics is, after all, a nice, formal, or at least a mathematical, theory of sorts, but what it does, it imposes constraints upon the universe.

Harnad. Right, but it's an empirical theory and so what it really has to account for is the data points and not what happens between the data points. Between the data points you can have complete continuity which also is a formal notion and that really does violate Church's thesis. Such a system, in a superimposed state...

Carberry. I think what we want to do, I mean everybody seems to be agreeing, is get beyond what formal systems are able to accomplish, however we define formal systems. But I feel uncomfortable with the term informal computing because I do think it's an oxymoron, or even the term informalism because I don't think, to the outside world, it'll get across what we want to do. I think there will be different views of this and you're going to spend a

lot of time trying to communicate what you mean. I really think you need a better term to capture it.

Littman. Essentially, I have just the opposite intuition. I think we could do ourselves a lot of damage by trying to drop down into the philosophy of it. If our goal is to establish a science of informal computing, we could do a lot of damage by trying to discuss the philosophical issues and we could actually make a lot more progress by saying: there's this stuff and it's informal and it's like heuristic reasoning and when I... It has very much for me to do with the degree of specification, and I think *that* is something everybody will understand. But if you start saying it's non-arbitrary symbolism, or broken formalism, I think it's a mistake. [*Changing tapes.*]

Shultis. We're beating around a radial category of some kind. There are more central examples, there are more central ideas. There are different notions, but there is something that ties it together. And maybe the pursuit of trying to nail down exactly what it is, is in fact a futile one. Given that there is something in common that we're all trying to wrestle with, and I think there is, the real question, as Sandra said, is: where do we go from here? What do we do? How do we overcome the limitations, the brittleness, if you like, of our formal systems? How do we deal with the natural phenomenon—and I think it is a natural phenomenon—of informality in the world? We need a science of some kind. How do we make progress on trying to put together some research, some things that we could do, some experiments. How do we build some of these things? Try to think about how are we actually going to do some of them. I think that your suggestion, that we get on with it, instead of trying to define things, to nail it down, might be a good one to think about for the discussion tomorrow.

Carberry. I think that to the outside world, the term "informal" isn't going to conjure up the ideas that we're discussing.

Standish. It may be a PR mistake.

Littman. I hope not, because I think that these ideas are what will be very confusing to people.

Standish. I have no idea what people's ideas are myself.

Carberry. Well, the idea of lack of specificity, which I really think is one of the core ideas here.

Littman. That's something we try to do in fact in software engineering—there's an audience that you can tap into right away, the informal specification community. You just have to say it the right way. Or somebody's trying to build a house and is first trying to get an idea of how much lumber and how many bricks they will need, and is going about it informally.

Shultis. If it turns out that there's a better way to describe what enterprise we're involved in, that's fine. But I think that most people would agree that there are informal, unformal, or nonformal processes and things in the world and that we're trying to understand them. And if people get the idea that that's what this is about, I don't see that that's a bad thing... But

one of the things that I don't want to focus on too much is our PR, how we look to the rest of the world. I want us to cohere as a body and see how we can start to make some progress.

Fisher. If we're going to cohere, though, we have to have some common understanding of what we're talking about. And what I thought you just said was let's throw all of these out because we aren't going to be able to come up with a formal definition. But I think, at least I heard four or five people now really endorsing this degree of specificity aspect of things. I think it is something that a lot of us share as a key characteristic of informal systems.

Shultis. Let me say this. I guess what I was suggesting, Dave, was not that we throw them all out, but that we accept them all.

Fisher. But what if I personally disagree with several of them?

Shultis. That's OK. It's not necessary that everybody in the group endorse every aspect of it to have a common understanding.

Fisher. You're suggesting that having any of these properties would suffice?

Shultis. We're all talking about things that are related. There are family relationships among these concepts, and that's what we're down to. Some of us will think that some these are more central to the category than others and that's fine. The enterprise is still this family of ideas.

It's six o'clock and we could probably go on forever and forever. What I would like to suggest is that we wrap it up in five minutes and all go off and stay up all night and talk to each other and come in bleary-eyed in the morning and solve all the problems.

Reeker. People might also think about what specificity means.

Shultis. What I'd like to think about for tomorrow is: how are we going to build these things? And what kinds of things are we hoping to achieve by doing informal computing?

Standish. I would like to put one more stimulus in the pot. Is there such a thing, such a natural phenomenon, as intuition—let's say human intuition? If so, what are its characteristics? Is it at all important in deciding what we mean by informality, and is it suitably important to want to seek mechanisms to implement it? For example, Gary Kasparov plays Deep Thought and he beats it. Is Gary doing pre-cognitive things about planning where things are going to happen in the long-range future that are not achieved by finite symbol system searching down search trees, according to the rules of chess? And does he know that if you bottle the end game up over in the corner and get it to temporize so that you can't do anything you can sort of pick away at the corners? Does he have an intuition about how to beat it and is that what makes him world champion and enables him to beat Deep Thought?

van Hoek. Sounds like you're saying he's drawing generalizations that the computer may not have acquired.

Standish. Some of this can be mechanized in limited forms, like Arthur

Samuel's checkers-playing program, which became the Texas state champion. And its mechanism for implementing that kind of intuition was (God forbid!) polynomials with learning coefficients. So I don't care what the mechanism is, but does it exist? and is it important? It's an interesting thing to me.

Littman. Well sure it is, and sure, and of course, to answer your questions one, two, three.

Harnad: It's interesting that there's formally an analogous debate at the foundation of mathematics between intuitionists and formalists. And some of these issues are very much the same. The intuitionists who are questioning... but it's turned upside down, because of course constructive proof is *better* than a purely formal proof, not worse... In a way, the way Kripke settled this in the interpretation of foundations is relevant here, too. The idea is this: you can prove something to be true formally by showing that it leads to contradiction if it's false. That's a formalist proof. You can prove something to be true constructively. That is to say, you can construct a model of it, perhaps even a physical model. That's an intuitionistic proof. What Kripke asked was, "What is the status of the truth of something that we know to be formally true but we don't yet know to be intuitionistically true: is it true or is it false or what have you?" And in the end I think that's the down side of your notion of informality: it's based too much on ignorance. Kripke's solution was to think of epistemic time for a mathematician. There's a time line, and on the time line when the constructive proof has not yet materialized the truth of the thing is sort of in a three valued logic limbo and it collapses into the two valued logic world that we're in when you come up with a constructive proof. But you see why it's upside down here. Here we're talking about informality as a form of ignorance. Right?

Shultis. A couple of things, I'd like to...

Standish. Oh no. It makes really good performance in the case of intuition. If Gary Kasparov is using it and he's world champion then it may have very strong problem solving consequences.

van Hoek. But we can't formally specify how he did it.

Harnad. But we don't know how or why and we haven't formalized it.

Shultis. Let me say something. A couple things I'd like to say. There is formal constructivity and there is intuitionism the way that Brouwer saw it and they are in very great conflict with one another. The second thing is that Kripke's construal of the semantics of constructivism is incorrect.

Harnad. It will be interesting to hear that.

Shultis. The question of truth, the question of truth of the proposition is meaningless to the constructivist.

Harnad. The bivalent truth is meaningless. The classical bivalent truth is meaningless.

Shultis. Even trivalent truth. It's not a meaningful question. What you know when you have proved a proposition, to a constructivist, is identified with knowledge of the method of that proof. That is to say, knowledge of the proof. And there is no question of truth or falsity...

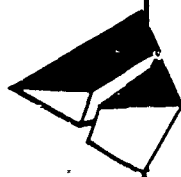
Harnad. But that's just semantics, because you're redefining truth as what I can construct. You're saying, no, don't ask whether this is true, ask whether I can construct the proof of its truth or I can construct the model in which it is true.

Shultis. The right answer is that if you ask: is a proposition true or false, independent of knowledge of the proof or disproof of it?, the correct answer is not "yes", "no", or "maybe", but rather "muh!", and... another time. Shall we break and go off and have these discussions at dinner?

What is Informalism ?

David A. Fisher

Workshop on Informalism
Santa Cruz, California
May 28-30, 1991



Fisher @ IncSys.com
412-621-8888
412-621-0259 (fax)

Incremental
SYSTEMS CORPORATION

What is Informalism ?

- Perspective: language design and compiler constructions
- Background: observations and assumptions
- Mechanized Informal systems: problems and solution approaches
- Exploiting Informalism to computational advantage

2

Perspective

Program Development =
Language Design + Compiler Construction

- Language Design
- Compiler Construction: Interpretation and efficiency
- Real solutions relate to real problems
- Computer systems must be able to represent and reason about the real world

3

Background — observations and assumptions

- People and machine roles
- Abstract and physical worlds
- Models and learning
- Linguistic mechanisms
- Persistent database
- Where formal methods succeed
- Where formal methods fail

4

Roles of People and Machines

- People and machines are problem solving devices
- Role of machines is to assist people in problem solving

5

Two separate worlds

- Two worlds: physical and abstract
- Abstractions do not exist in the physical world
- Abstractions exist in minds of people / machines
- Objects of the physical world do not exist in the minds of people / machines
- Only abstractions exist in the minds of people / machines

6

Sensors, Activators, Models and Learning

- People / machines have *sensors* — their only means to access the physical world
- People / machines have *activators* — their only means to change the physical world
- People / machines build *models* (representations) of the physical and abstract worlds
- Models must be *learned* based on sensory input and interaction with the physical world
- Process must be *incremental* because it is always incomplete
- Goal in learning: build predictive models consistent with observations (evidential semantics)

7

Linguistic Mechanisms

- People / machines do not share models; each must independently model the world based on interaction with physical world
- Such interaction among people or between people and machines is called *communication*
- *Language* is set of conventions about the communication process
- Language too must be learned, modeled and cannot be shared

8

Persistent Database

- People / machines must have persistent databases
 - models of physical and abstract worlds
 - models of language
 - programs (i.e. Interpreters and compilers)
 - database of reusable application information
 - shared context and culture
- Physical world is own best database

9

Where Formal Methods Succeed

- Formal methods: assume finite descriptions; often require completeness, consistency and precision
- Abstractions are complete, consistent and precise.
- Abstractions can be modeled completely
- Physical world is complete, consistent and precise.
- Models of physical objects are finite and can themselves be complete, consistent and precise
- Formal methods are adequate for describing and reasoning about abstractions and models, including models of the physical world.

10

Where Formal Methods Fail

- Formal methods: assume finite descriptions; often require completeness, consistency and precision
- Physical objects do not have complete finite descriptions
- Physical objects cannot be modeled completely
- Formal methods are inadequate for describing and reasoning about the physical world

11

Characteristics of Mechanized Informal Systems (problems and solution approaches)

- Encompass formal methods
- Intensional
- Incomplete
- Intensional language
- Inconsistent
- Nonaxiomatic or prototypical
- Imprecise

12

Informal Systems must Encompass and Exploit Formal Methods

- Must reason about abstractions and models, as well as the physical world
- There is tremendous investment and existing body of knowledge in formal methods
- Most efficiency issues involve reasoning about the models (not real world)

13

Informal Systems must be Intensional

- Cannot anticipate all concepts to be represented
- Cannot build-in all compositional primitives
- Must build models of relationships among objects (attribute system)
- Use property-based types
- When composition is used, what is primitive should be context dependent.
- Example: English language dictionary
- Example: Iris definition system

14

Informal Systems must be Incomplete

- Cannot have a complete description of any physical object
- Use property based type system
- Use logic that assumes and accounts for incompleteness
- Use lazy evaluation to avoid unnecessary information requirements
- Use linguistic mechanisms that do not require overspecification

15

Informal Systems must have Intensional language

- There must be a linguistic component
- Cannot anticipate all interpretation contexts
- Cannot uniquely label all intentions
- Intensional models require intensional language
- Must build intensional linguistic model (prism)
- Use overload resolution for lexical disambiguation
- Use property based types and anaphoric reference
- Use property based inheritance as guide in selecting context

16

Informal Systems must manage *Inconsistency*

- Input is dependent of conditional and unknown contexts (preconditions, chronology)
- Inconsistency in models arises from incorrect assumptions about intension and intention
- Reasoning system must admits to errors in models
- Detect inconsistency through conflicting predictions
- Detect inconsistency through sensor feedback
- Detect inconsistency through dialogue
- Correct inconsistency through dialogue and changes to models (requirement)
- Strive for consistency; no correctness only belief

17

Informal Systems must be *Nonaxiomatic and Prototypical*

- Integers are closed under addition vs. birds fly
- Processing must distinguish definitions from observations and prototypes
- Use logic that exploits prototypical properties, but not when it creates inconsistency
- Prototypical properties probably not statistical, but absolute in absence of inconsistency

18

Informal System will be *Imprecise*

- Combination of incompleteness, inconsistency and intensionality
- Provides impression of imprecision, inexactness, vagueness and sometimes ambiguity
- Alternative view: are dealing with summary information
- Not an independent problem

19

Exploiting the Characteristics of Informal Systems to *Computational Advantage*

- Intensionality: interoperability, primitiveness, and open-ended architecture
- Incompleteness: linear cost, scalability, and rapid prototyping
- Imprecision: representation selection, and reduced cost of proof
- Dialogue: experience, culture, and efficiency
- Abstraction and generalization

20

Informal Systems can exploit *Intensionality*

- Common Interpreter and learning mechanism provides interoperability not achievable in extant computing systems
- Context dependent grounding allows specialization of implementation without loss of interoperability (iris primitiveness)
- Primitiveness and attribution provide a true open-ended architecture

21

Informal Systems can exploit *Incompleteness*

- Completeness => proof and compilation processes combinatorial in amount of unconstrained detail (case analysis)
- Incompleteness => proof and compilation processes combinatorial in number of properties used (constraint propagation)
- Can partition on independent properties to approximate linear cost and to enable scaling
- Can suppress properties for rapid prototyping
- Can suppress detail (via properties) to scale or generalize applications

22

Informal Systems can exploit *Imprecision*

- Representation selection key to problem solving
- Imprecision allows optional properties to be assumed or ignored as is convenient to proof line
- Enables multiple trial representations with criteria for selection

23

Informal Systems can exploit *Dialogue*

- Have a narrow window on real world (experience)
- Dialogue provides access to experience of others
- Dialogue provides access to conclusions of others (culture and civilization)
- Dialogue reduces amount of information that must be saved and processed

24

Informal Systems can exploit *Abstraction and Generalization*

- Successful reasoning with property-based types distinguishes critical and noncritical properties
- Type can ignore noncritical properties (abstraction)
- Can then apply operations in new domains that share critical properties (generalization)
- Example: property based inheritance

88

World of Practical Informal Systems

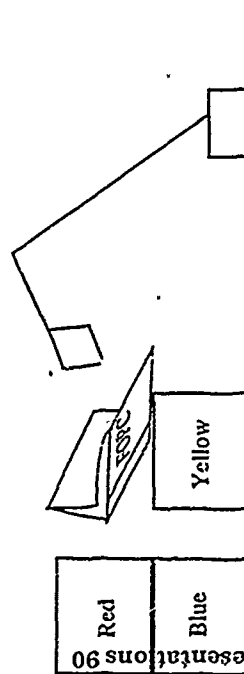
- Property based types and inheritance
- Intensional models of relations among objects
- Intensional language with variableless quantifiers, anaphoric reference and incomplete specification
- Universal interoperability through primitiveless interpretation and open-ended attribution
- Formal methods for reasoning about models and abstractions
- Informal methods for reasoning about the physical world

26

Reaction in Real-Time Decision Making

Bruce D. D'Ambrosio
Department of Computer Science
Oregon State University

Once upon a time...



Problem solving was thought to be equivalent to proof.

What's the problem?

Consider:

1. Driving slowly out of an empty parking lot and avoiding a speed bump.
2. Making a right turn in light traffic and avoiding a bottle in the road.
3. Avoiding a small box in the road while rounding a turn on a crowded freeway.
4. Noticing, while in situation 3, that another box just fell off a truck up ahead.

The world is not static:

- Solution value is a situated function of time. (1 - 2 - 3)
- Problem definition is changing. (4)

① I'm not on in forward.
② But I am interested in a lot of probs.
③ In attempting to formalize, probabilistic.
④ Early stage
⑤ Circuits as I currently understand them

Why is it a Problem?

- For Real-Time Computing:
 - Approach: guaranteed worst-case performance.
 - Problem: Inapplicable or suboptimal in the face of tradeoffs.
- For AI:
 - Approach: Satisficing, Static Domain, Design-time Tradeoffs.
 - Problems:
 1. Inapplicable or suboptimal in the face of tradeoffs.
 2. Domain isn't static.
 3. Situations vary widely.
- For Software Engineering in high-risk domains:
 - Requirements:
 1. publicly inspectable values.
 2. predictable performance.

Toward a Normative View of IRTPS

- What *OUGHT* an embedded system do?
- Chapman and Agre:
 - Before and beneath any deliberation is the *continuous decision* of what to do NOW.
- Decision in the face of tradeoffs and under uncertainty:
 1. Utilities (von Neuman and Morgenstern)
 2. + Beliefs (Savage)
 3. = Statistical Decision Theory

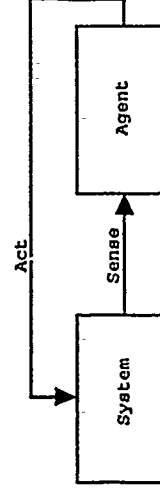
Outline

- Real-time embedded systems face tradeoffs.
- Decision Theory:
 - Rational decision making in the face of tradeoffs.
- But: Finite agent capabilities?

Remainder of talk:

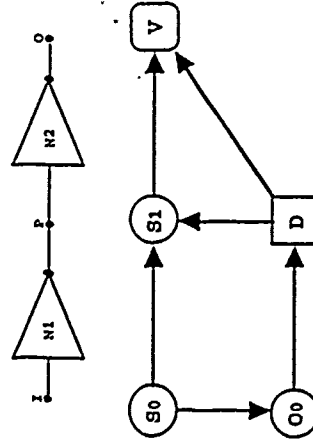
- The OLMA domain.
- Minimal decision making.
- Reactions and meta-level decision processes in RTDM.
- Loose ends: Decision basis construction, Planning, *Reaction Adaptation*

On-Line Maintenance

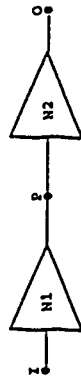


- System is in operation.
- Faults can occur at any time.
- Goal is action, not diagnosis:
 - minimize total operational cost.

Example - On-Line Maintenance



- S0, S1: State(N1), State(N2)
- O0: I, P, O
- D: Replace(N1), Replace(N2), Nothing
- V: Utility(D, S1)



Given:

1. a specification of the decision basis:

- $\{P(N1_{i0} = Ok) = .9, P(N1_{i0} = Faulted) = .1\}$
- $P(P_{i0} = 1 | I_{i0} = 0 \& N1_{i0} = Ok) = 1.0$
- $D : \text{Replace}(N1), \text{Replace}(N2), \text{Nothing}$
- $P(N1_{i1} = Ok | N1_{i0} = Ok \& D_{i0} = \text{Nothing}) = .9$
- $V(PCorrect_{i1} = t \& OCorrect_{i1} = t \& D_{i0} = \text{Replace}(N1)) = -\1000

2. And a set of observations (eg, $I_{i0} = 0, O_{i0} = 1$),

Compute the optimal decision according to MSEU:

- $SEU(\text{Replace}(N1)) = -\$1248.$
- $SEU(\text{Replace}(N2)) = -\$2736.$
- $SEU(\text{Nothing}) = -\$1856.$

MSEU Computation for Decision

Caso	P	Nothing	N1	N2
		P O	P O	P O
DirectCost	1.0	0	-1000	-1000
$N1_{i0} = Fl, I_{i1} = 0, N2_{i1} = Ok$.36	-360 -360	0 0	-360 -360
$N1_{i0} = Fl, I_{i1} = 0, N2_{i1} = Fl$.04	-40 -40	0 -40	-40 -0
$N1_{i0} = Fl, I_{i1} = 1, N2_{i1} = Ok$.36	-360 -360	0 0	-360 -360
$N1_{i0} = Fl, I_{i1} = 1, N2_{i1} = Fl$.04	-40 -40	0 -40	-40 -40
$N1_{i0} = Fl, I_{i1} = 0, N1_{i1} = Ok$.04	-0 -40	0 -40	-0 -0
$N1_{i0} = Fl, I_{i1} = 0, N1_{i1} = Fl$.004	-4 -4	0 -4	-4 -4
$N1_{i0} = Fl, I_{i1} = 1, N1_{i1} = Ok$.04	-0 -40	0 -40	-0 -0
$N1_{i0} = Fl, I_{i1} = 1, N1_{i1} = Fl$.004	-4 -4	0 -4	-4 -4
$N1_{i0} = Fl, N2_{i0} = Fl, I_{i1} = 0$.04	-40 -40	0 -40	-40 -40
$N1_{i0} = Fl, N2_{i0} = Fl, I_{i1} = 1$.04	-40 -40	0 -40	-40 -40
Total		-1856	-1248	-2736

- $SEU = \sum_{d,o,h} E(U(d, o, h))$

Is It Really That Simple?

Problems:

- What about effects of finite computational capability?
- How did we decide we needed to decide?
- Where did ID come from?

Opportunities:

- Minimal decision making.
- Reactive decision making.
- Meta-level decision making.
- Dynamic decision model construction.

MSEU Computation for Decision

Case	P	Nothing	N1	N2
	P	O	P	O
Direct Cost	1.0	0	-1000	-1000
$N_{1,0} = Fl, I_{1,1} = 0, N_{2,1} = Ok$.36	-360	0	-360
$N_{1,0} = Fl, I_{1,1} = 0, N_{2,1} = Fl$.04	-40	0	-40
$N_{1,0} = Fl, I_{1,1} = 1, N_{2,1} = Ok$.36	-360	0	-360
$N_{1,0} = Fl, I_{1,1} = 1, N_{2,1} = Fl$.04	-40	0	-40
$N_{2,0} = Fl, I_{1,1} = 0, N_{1,1} = Ok$.04	-0	-40	-0
$N_{2,0} = Fl, I_{1,1} = 0, N_{1,1} = Fl$.004	-4	0	-4
$N_{2,0} = Fl, I_{1,1} = 1, N_{1,1} = Ok$.04	-0	-40	-0
$N_{2,0} = Fl, I_{1,1} = 1, N_{1,1} = Fl$.004	-4	0	-4
$N_{1,0} = Fl, N_{2,0} = Fl, I_{1,1} = 0$.04	-40	0	-40
$N_{1,0} = Fl, N_{2,0} = Fl, I_{1,1} = 1$.04	-40	0	-40
Total		-1856	-1248	-2736

• $SEU = \sum_{o,h} E(U(d, o, h))$

• Few terms dominate.

Minimal Decision Making

Normative decision basis evaluation:

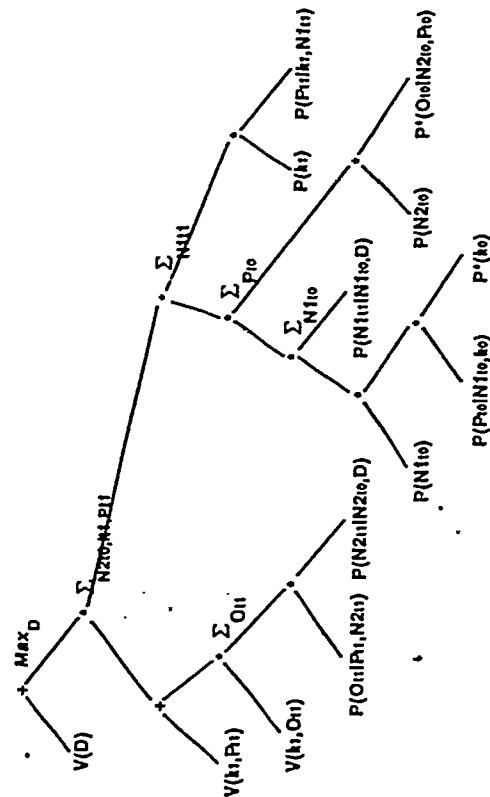
Evaluate ALL decisions wrt ALL outcomes under ALL state hypotheses.

A minimal decision maker:

Evaluate SOME decision wrt SOME outcome under SOME state hypothesis.

Which decision, outcome, hypothesis?

Search Example



Decision Making as Search

- Search space: Partial and complete instantiations of variables.
- Start state: No variables instantiated.
- Operators: Propose set of variables to instantiate.
 1. depth-first left-to-right in eval tree (null heuristic).
 2. (1) but select value with largest mass.
 3. Largest probability mass in a distribution.
- Goal state: Complete instantiation along a path.
- State evaluation heuristics:
 1. Value so far,
 2. Value so far + mean-value of uninstantiated variables,
 - Issue: Space management (caching and search fringe size).

Decision Making as Search: Summary

- Exact evaluation of a decision basis is NP-hard.
- Incremental (Anytime) approximate evaluation can be seen as search.
- Individual term time-complexity is linear in problem size.
- Status: prototype search code in final debugging now.

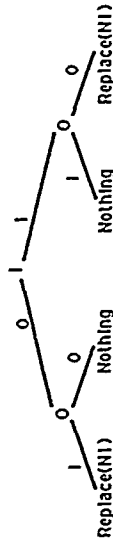
• Next: Reactions.

• In formal:

- because probabilistic?
- because approximate?

Reaction

- Can compile DB into Policy, only four reactions:
 - If $I = 0$ and $O = 0$ then do nothing.
 - If $I = 0$ and $O = 1$ then replace(N1).
 - If $I = 1$ and $O = 0$ then replace(N1).
 - If $I = 1$ and $O = 1$ then do nothing.
- Build decision tree, allocate processor to each node.



*Reaction: decision conditioned on observable.
no internal state (beliefs, utilities).*

Reactive Decision Making

- Only correct for first decision!
- DB evaluation uses updated probabilities.
- In general, need entire input history \rightarrow

*dim: 6ary
v. 3
S = log p + 1.0*

$$|Reaction\ set| = O(n^{History\ length})$$

But: see later!

• What role for reaction?

1. If $I = 0$ and $P = 0$ replace(N1).

Reaction Revisited

- What role for reaction?
 1. If $I = 0$ and $P = 0$ replace(N1).
 2. If $I = 0$ and $O = 1$ and not evaluating DB then evaluate DB.
 3. If $I = 0$ and $P = 0$ and evaluating DB then replace(N1), stop evaluating DB.
 4. If evaluating DB and accumulating-terms and $|term\ set| = 10$ then switch to selecting-optimal-alternative.
- But reactions can only look at observables?
- Reactions 2,3,4 are meta-level decision processes.

Meta-Level Decision Making

Meta-level decision process: (process: DB eval on reaction)

A decision process which takes as its situation the state of some other ongoing decision process.

Examples:

- Initiate a deliberation?
- Find another term?

When?

Initiate regress?

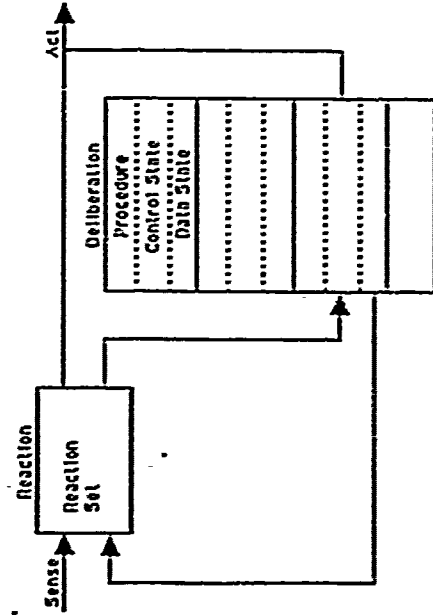
Situation state and action alternatives?

Meta-Level Decision Making

1. Situation State: Data state, Procedure, Control state.
2. Action Alternatives:
 - (a) Initiate evaluation.
 - (b) Terminate evaluation.
 - (c) Modify
 - i. Data state (decision basis construction/reformulation).
 - ii. Procedure (select appropriate state heuristic).
 - iii. Control State (continue or terminate execution, modify agenda, etc).

3. Regress grounds in reaction.

Reaction and Meta-Level Decision Making



Reaction and Meta-Level Decision Making

- DB evaluation = a set of meta-level reactions?
- Yes, but...
 - Production system model is space inefficient.
 - Wasto of processing resources, typical execution is sequential.
- Minimal reaction set:
 - Minimal set of decision processes that can evoke full range of behavior.
- Optimal reaction set:
 - Minimal + others as space available.

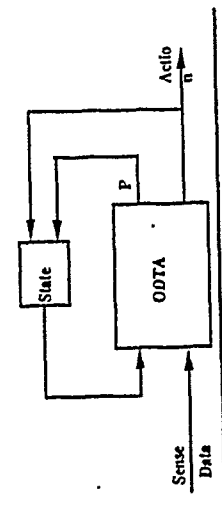
Optimal is agent relative.

Conjecture: CPT... is not really a conjecture, but a derivative for interactivity agent-based, but complete.

Work in Progress

- Refinement of Model.
- Second generation agent architecture:
 - under construction,
 - experiments begin this summer.
- Experimental study of Reaction Size:
 1. Simulation environment with Type I DT Agent.
 2. Trace behavior.
 3. Trace → inductive learning program.
 4. Measure space/performance.

A Finite State Machine Agent



How might we model this? How about table lookup?
Number of entries?

7 Action Alternatives
32 Probabilities, 8 bits each(?)
5 sense lines, 1 bit each
3 bits
128 bits
5 bits
136 bit address!

A FSM Model of Reflex

$$ODTA(P \times A \times O) \Rightarrow P \times A$$

The Optimal Decision Theoretic Agent can be viewed as a function from the cross product of:

- A joint probability distribution across component states,
- An action (the previous one),
- The space of observations.

To the cross product of:

- A joint probability distribution across component states,
- An action.

Reaction Size Experiments

- Model of DT agent as FSA:
 1. Input:
 - (a) Sense data (5 bits),
 - (b) Previous action (3 bits),
 - (c) Probability distribution over states (32 floats, ~128 bits).
 2. Output: Action, updated probability distribution.
- 136 bits input → 2¹³⁶ reactions!
- Five steps:
 1. Learn action using actual prior probability + sense.
 2. Learn 1 posterior probability.
 3. Learn 1 posterior probability from discrete input.
 4. Learn action from discrete probability.
 5. *In vivo* experiments.
- So far:
 - Steps 1 - 3/4.
 - Best alternatives: C4.5, Cascade Correlation.

Reaction Size Experiments

Number of Objects		Error Rates	
Trained on	Tested on	Total	Acts only
560 ¹	560 ¹	18/560 (3.2%)	15/480 (3.1%)
640 ¹	640 ¹	17/640 (2.7%)	16/480 (3.3%)
800 ¹	800 ¹	14/800 (1.8%)	14/480 (2.9%)
3498	3498	18/3498 (0.51%)	17/99 (17.2%)
6592	6592	15/6592 (0.23%)	13/99 (7.2%)
560 ¹	3498	660/3498 (18.9%)	1/99 (1.0%)
640 ¹	3498	191/3498 (5.5%)	1/99 (1.0%)
720 ¹	3498	99/3498 (2.8%)	1/99 (1.0%)
800 ¹	3498	123/3498 (3.5%)	1/99 (1.0%)
3498	1084	6/1084 (0.55%)	6/17 (35.3%)
3498	3094	28/3094 (0.9%)	25/69 (36.2%)
3498	6592	84/6592 (1.3%)	74/180 (41.1%)
6592	3498	18/3498 (0.51%)	18/99 (18.2%)

¹ Selected data weighted with "real" acts

Reaction and Meta-Level: Summary

- Reaction is not just a front end to deliberation:
 - Initiates and terminates deliberation.
 - Maintains coordination between deliberation and world.
- Meta-level is essential, can be reactive or deliberative.
- Space requirement for purely reactive still open.

Reaction as front end:
P. Cohen, T. Linday, ...
FLAT "commitment" system: B. E. P.
SOAR

Reaction Size Experiments

Set Size	Inputs	Classes	Nodes in Tree	Tree Size	Variable
560	38	7	83	3518	Acts
640	38	7	81	3427	Acts
720	38	7	81	3427	Acts
800	38	7	83	3530	Acts
880	38	7	79	3336	Acts
920	38	7	79	3336	Acts
3498	38	7	37	1569	Acts
6592	38	7	57	2382	Acts
3498	38	13	3	202	P(N1& N2 Ok)
3498	38	13	55	3708	P(N1 Ok, N2 S1)
3498	38	30	87	11788	P(N1 Ok, N2 S1)
3498	38	47	113	23035	P(N1 Ok, N2 S1)
3498	38	73	159	48964	P(N1 Ok, N2 S1)
3498	38	94	169	66219	P(N1 Ok, N2 S1)
3498	38	115	189	90001	P(N1 Ok, N2 S1)

Reactive System Size (in nodes): $32 * 55 + 79 = 1839$

Deliberative (probability & utility values + DB nodes): $532 + 8 + 12 = 552$

Where next?

- Constructing world models:
 - Forbus (QP theory):
 - Explore ALL possible combinations of ALL possible instantiations of first-order descriptions (processes and views).
 - Charniak and Goldman (Wimp):
 - Explore A FEW combinations of A FEW possible instantiations of first-order descriptions (events and equality statements).
 - Can't build complete graph!
 - Status: One student up on Wimp and QP theory, looking at AUV.
- Planning:
 - Local evaluation may not be robust.
 - Lookahead caches ("commitments?") save time in future decisions.
- Performance Prediction?

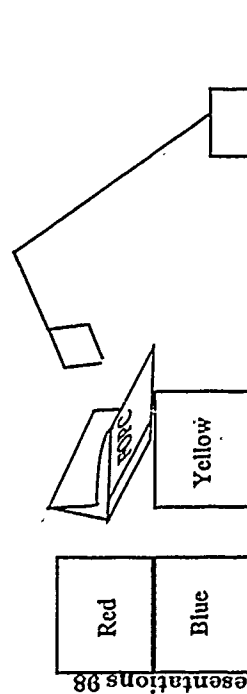
Review

- **Situated quality/time tradeoff:**
 - construction and evaluation guided by mix of reactions and meta-level DBs.
- **Dynamic environment:**
 - potential for action (base or meta-level) at every world/system state change.
- **Inspectable values:**
 - values (utilities) explicitly stated and incorporated into decision process.
- **Predictable performance:**
 - rigorous statements about expected utility.

Summary

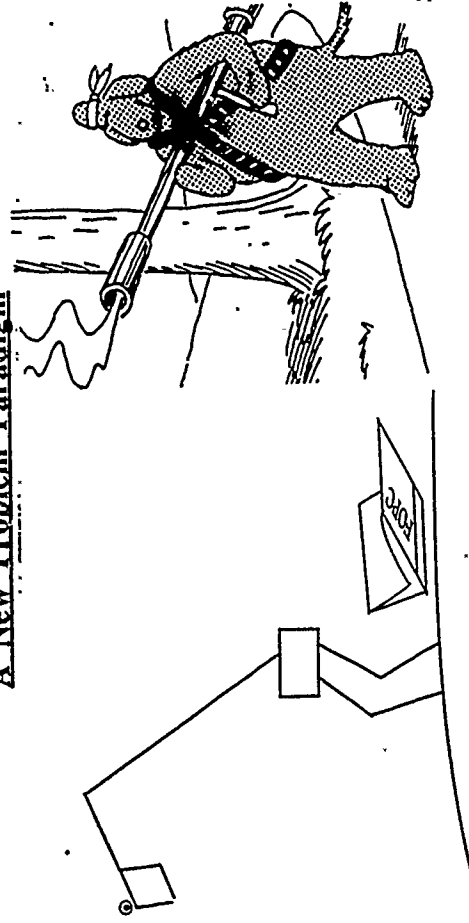
- IRTPS involves *Tradeoffs*
- **Decision Theory:** a normative theory of decision making in the face of uncertainty and tradeoffs.
- But, incomplete.
- Incremental, search-based, decision algorithms.
- Reaction and meta-reasoning.
- Space/performance of purely reactive systems.
- Knowledge based construction.

Once upon a time...



Problem solving was thought to be equivalent to proof.

A New Problem Paradigm



- The fundamental need is to Act!
- We know a lot about the world.
- The world won't wait.

Reaction in Real-Time Decision Making Extended Abstract

Bruce D'Ambrosio
Tony Fountain

*Department of Computer Science
Oregon State University
Corvallis, Oregon, USA 97331-3202
503-737-5563
dambrosio@cs.orst.edu*

Abstract

We derive a model of the role of reaction in real-time decision making from a consideration of the computational dilemma facing finite agents acting in the world. This dilemma is simply that more time spent in computation will generally provide a better solution, but more time also means more delay, which may have costs of its own. Our model provides a more fundamental role for reaction than is generally assumed. We illustrate this model with examples from a prototype real-time decision maker we are constructing, and briefly survey several current proposals for real-time problem solving architectures in light of the role they assume for reaction.

Introduction

This paper is an extended abstract of a longer paper submitted to IJCAI-91. In it we investigate the role of reaction in real-time decision making. The problem of real-time decision making, or acting in time, presents a fundamental challenge to AI approaches to problem solving in general, and to decision making in particular [1], [6]. Two predominant responses to this challenge are reaction [2], [10] and meta-level reasoning [11]. We find most discussion of the roles of reaction and meta-level reasoning in real-time problem solving confusing and confused. The primary goal of the full paper is to present our understanding of the roles of and interactions among potential decision-making elements, and especially the role that reaction plays in decision making and in problem solving in general.

Our interest is in how finite agents cope with a complex and dynamic environment in the pursuit of their goals. The fundamental requirement of an autonomous agent is that it be able to *act* in pursuit of its goals. Therefore, we take the ability to choose an act, or *decide* as primitive, and study decision making, rather than planning. We will organize our discussion of decision making around three characteristics of any decision situation: the process by which the decision will be made, and the domain

of action for the decision, and the way in which a decision situations arises and is recognized.

Decision-making processes

Traditional theories of decision making or planning in AI model the process as the application of a general-purpose reasoning procedure to a problem representation stated in some language. These models generally presume the knowledge is organized in a form convenient and compact for expression, and the reasoning complexity is unbounded if the language is at least as expressive as FOPC. Such decision processes are typically termed *reflective* or *deliberative*. By the use of the term *reflective* we do not mean to necessarily imply any form of self reasoning, as in [13]. Rather, we simply use the term in its dictionary sense of "to think deeply." By contrast, the term *reaction* is typically used to denote a decision-making process in which the decision is made based on simple, direct sensor-effector connections, and in which either the depth of computation is bounded, or the total computation time is bounded.

We find this characterization of reaction and reflection unsatisfactory. By the above definitions, processes of the complexity of finding the optimal decision given a decision basis, which are known to be NP-hard, are "reactive." This seems counterproductive. In the following we build up a model of real-time decision making processes from which we derive a very different characterization of reaction and reflection.

Models of decision processes There is an essential element missing in the above definitions: they do not take into account either the nature of the environment, the task, or the agent [4]. Given a probability distribution over domain decision situations an agent might face, and a set of utilities over outcomes, we begin by defining a *decision process* to be any computational process which recognizes a subclass of situations and selects a single action more or less appropriate for that entire subclass. An *Optimal real-time decision process* is then a bounded-space bounded-time decision process that computes that action for which the expected utility is the maximum over all possible actions, where the expected utility is evaluated at the time the computation will complete when executed on the agent for which the process is designed. Optimal real-time decision processes for a given environment, task set, agent combination can vary in two ways: The class of situations to which they respond, and the computation time required. An *optimal real-time decision-process set* is a set of optimal real-time decision processes that together do not exceed the resource limits of the target agent, and which as a set provide the maximum expected utility of any such set.

Decision domains In general, decision-making can take as its domain the environment, the agent, or both. When the domain is the environment, input is the state of the environment, as evidenced by sensors, and output is an action, to be performed by effectors. When the domain is the agent (that is, meta-level decision making), input is (some aspect of) an ongoing decision process, and output is a modification to that process. We conjecture that an optimal real-time decision process set for highly resource constrained agents and/or agents operating in highly dynamic environments will often include meta-level decision processes.

Meta-level decision making requires modeling ongoing decision processes. Any computational process can be modeled as consisting of three aspects, data state, control state, and procedure. At this level of abstraction we can say little more. Data state is simply the declarative input/output interface to a process, and its form is determined by the language the decision-making process requires. Similarly, control state cannot be specified separately from procedure. We chose to commit to decision theory [12] as our theoretical model of decision making, and the *decision basis* [7] as our representation of a decision problem. This defines a data state as consisting of five sets:

- Parameters representing aspects of the state of the domain,
- Beliefs about the values of these parameters and their interrelationships,
- Action alternatives,
- Beliefs about the effects of actions on parameters, and
- Preferences over possible outcomes.

We will have more to say about control-state and procedure later, when we describe the prototype agent we are constructing.

Decision Situations

We are almost ready to define a reactive decision process. One question remains: how does a decision situation arise? We assume a symbolic, event-based interface to the external world, which makes the first question simple for base-level decision situations: a new decision situation arises whenever new symbolic data arrives from the world. At the meta level the same principle applies: a new decision situation arises whenever the state of the active decision process or the external world changes.

Now we face a second problem: how is an agent to recognize that a new decision situation has arisen? All decision processes could be active simultaneously,

monitoring for situations to which they can respond. However, this approach is in general infeasible: for arbitrary forms of meta-level reasoning the number of potential decision processes could be unbounded. An alternative is to seek a minimal set of decision processes which must be active at all times. This minimal decision process set must be sufficient to generate all possible decision processes in the full set under appropriate circumstances, and will in general contain three kinds of decision process:

1. decision processes that yield an action directly.
2. meta-level decision processes that initiate new decision processes.
3. meta-level decision processes that modify existing decision processes.

While not required by our model, the easiest way to ensure that this set is closed and finite is to require that all decision processes in our minimal set be stateless. This permits them all to be executed by a computational element which need not monitor its own state. We are finally in a position to define a reaction: any decision process in this set is a *reaction*, and the entire set the *minimal reaction set* for the larger decision process set. It may be that there exist stateless decision processes not in this set which, if added to the set, would improve the expected performance of the agent (by reducing response time to the decision situation responded to by those processes). The minimal reaction set, supplemented by the subset of such stateless decision processes which maximizes expected performance of the agent, is termed the *optimal real-time reaction set*. For convenience, we term all decision processes not in the reaction set *reflective*.

Reaction and Reflection in Real-Time Decision Making

We can now present a general computational model of real-time decision making. First, we posit a decision maker consisting of at least two computational elements, a reactive processor and a reflective processor. The reaction set will be located in the reactive processor. Since all decision processes in the reaction set are stateless, the reactive processor can be implemented as a combinational circuit. Input to the reactive processor must include all state change information, both external to the agent and internal. The output of the reactive processor can be any one of the following:

1. An action to be performed in the external world. In this case the reactive element is responding to an external decision situation by choosing to make the decision reactively, and producing the decision.

2. A change to an ongoing reflection. In this case the reactive element is responding to a meta-level decision situation by choosing to make the decision reactively, and again producing the decision.
3. A new reflection to be begun. In this case the reactive element is responding to a base or meta-level decision situation by choosing to make the decision reflectively, and initiating the reflection.

Of course, the reactive element can always choose to ignore a decision situation entirely. Note that in this model all reflection is initiated by reaction. When initiating a reflection, the reactive element must provide initial values for the data state, control state, and procedure. If the data state scope includes only the external environment, base level agent action, and a problem statement, then the reflection spawned is base-level. However, if the scope includes one or more elements of other on-going reflections (data, control, or procedure), then by definition the new reflection is meta-level. We assume that only the highest level reflection currently outstanding is active, and that all lower level reflections are suspended until it completes or is terminated by the reactive element. Variables in reactive decision processes are bound to the currently active decision process in the reflective element.

We have introduced a variety of definitions, and claimed to provide a general model of real-time decision making in terms of those definitions. Clearly these definitions are useful only to the extent that we can do something with them. Ideally, we would like to provide a procedure for solving the design problem implicit in the model: given an ETA description, derive the optimal real-time decision process set and corresponding optimal real-time reaction set. We have not yet produced such a result, and doubt it is achievable in the near future (if ever) for non-trivial problems. Rather, in the full paper we present two illustrations of the utility of this framework. First we illustrate this model with some examples from an agent we are constructing. Second, we survey, from the perspective of our model, several current proposals for real-time problem solving architectures.

Conclusions

Reaction is typically assigned the role of serving as a "quick and dirty" front end to a more "intelligent" deliberative (reflective) process. We have argued that reaction plays a much more fundamental role in real-time decision making: that it is the ultimate ground of all reflective processing, and that the essence of a reaction is that it is a stateless decision process which is always active. We have introduced the notion of an optimal real-time reaction set as a complete specification of the optimal solution to a real-time decision problem, where the problem specifications include

environment, task, and agent characteristics. Finally, we have found that our model of reaction is adequate to compactly describe the kinds of decision making dynamics we have found useful in a study domain, the on-line maintenance agent.

Acknowledgments

This work owes much to the Schemer architecture developed by M. Fehling, as well as to many conversations with him. An earlier version of some of these ideas appears in [4]

Bibliography

- [1] P. Agre. *The Dynamic Structure of Everyday Life*. PhD thesis, MIT, 1989.
- [2] R. Brooks. A layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14-23, 1986.
- [3] P. Cohen and *et al.* Trial by fire: Understanding the design requirements for agents in complex environments. *AI Magazine*, 10(3):32-48, 1989.
- [4] B. D'Ambrosio and M. Fehling. Constrained rational agency. In *Proceedings of the International Conference on Systems, Man, and Cybernetics*. IEEE, November 1990.
- [5] Barbara. Hayes-Roth. A blackboard architecture for control. *Artificial Intelligence*, 26(3):251-323, July 1985.
- [6] E. Horvitz, G. Cooper, and D. Heckerman. Reflection and action under scarce resources: Theoretical principles and empirical study. In *Proceedings of IJCAI89*. IJCAI, August 1989.
- [7] R. Howard and J. Matheson. Influence diagrams. In Howard and Matheson, editors, *The Principles and Applications of Decision Analysis*. Addison-Wesley Publishing, Menlo Park, Calif., 1984.
- [8] J. Laird, A. Newell, and P. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33(1):1-64, 1987.
- [9] M. Pollack and M. Ringuette. Introducing the tileworld: Experimentally evaluating agent architectures. In *Proceedings Eighth National Conference on AI*, pages 183-189. AAAI, August 1990.
- [10] S. Rosenschein. Synthesizing information tracking automata from environment descriptions. In *Proc. First Intl Conf on Principles of Knowledge Representation and Metareasoning*, 1989.

- [11] S. Russell. Principles of metareasoning. In *First International Conference on Principles of Knowledge Representation and Reasoning*. AAAI, 1989.
- [12] L. Savage. *The Foundations of Statistics*. Dover Publications, 1972.
- [13] B. Smith. *Reflection and Semantics in a Procedural Language*. PhD thesis, MIT, 1984.

Supporting

Informal Reasoning

David Littman

Background
possibilities

Thursday Presentations

Common sense
Constraints

Problem Specific

Goals

Formalizations
plans

Tenets

- 1) Informal Reasoning useful when no Stock plan
- 2) Informal Reasoning can lead to Stock plans
- 3) Informal Reasoning can lead to learning
- 4) Computers should support Informal Reasoning

Today's Comments

- > Program Design
- > Robot Design
- > TP+

NLD: What's The Problem?

NLP: Some Process & Some Data

GDAC: Goal-Data-Action-Control

PP: Indexed or Created from GDAC.

Littman

NLD: Keep up with new customers at health club

NLP: info about each client to database

GDAC: Goal: Update DB w/ new member info
Data: Multi-info about each client
Action: Add multi-info to DB
Control: Do for all new members

PP: Loop (fill client-info-rec DB) } Loop-Add-new-DB
(Add client-info-rec DB) }
Until no-more client
Plan Notebook
Code

Littman

Aspects of Informality

- > Problem Description: Goal statement
- > Creating NLP: Subgoal expansions
Some process
- > Building GDAC: Id NLP components
Mapping to known
Leaving unspecified

Robot Design

- > Underspecified: Build a toy-picker-upper
- > Assembly of Components
- > Requires Exploration
 - Requirements
 - Materials
 - Compositionality
 - Evaluation

Littman

Aspects of Informality

> Requirements

- "Toys aren't too big"
- "House is pretty large"

> Assembly of Components

"I've gotta get the power to the hand."

- Formulation & Exploration of options

> "Son of TP" Thursday Presentations 109

> Prepares Tutorial Plans Dynamically

> Reasons in Space of Cognitive Acts

> Cognitive Act:

- Cognitive Operation
- Cognitive Object

> TP+ causes student's thought train

Litman

Litman

While num-vals < 100: ~~Print~~
Read (new-vals)
Add (sum new-vals)
End

Fact: num-vals not modified \rightarrow infinite loop

Goal: student build accurate mental model
of behavior of loop

Resultant: student detect [infinite loop] num-vals

Plan: Simulate with Focus

Focus Action: Loop condition

Focus Outcome: when ≥ 100 ?

Produce Evidence: sum new-vals num-vals

Weigh Evidence: num-vals?? num-vals ≥ 100

Find Cause: num-vals = num-vals all times

Generate Cause Hypothesis:

...

Questions

> What are informal representations?

> How are they (naturally) manipulated?

> How are they composed?

> How are they modified?

> How can they be affected?

- new functional requirements?

- optimizations?

> How can we represent \uparrow in computer?

Litman

Decision Making With Informal, Plausible Reasoning

David Littman

Department of Computer Science

George Mason University

Fairfax, VA

dlittman@gmuvmx2.gmu.edu

Debbie Boehm-Davis

Department of Psychology

George Mason University

Fairfax, VA

dbdavis@gmuvmx.gmu.edu

Introduction: Motivation and Goals

The purpose of the research described in this report is to explore the possibility of developing a computational model of the informal, plausible reasoning that occurs when people try to solve decision making tasks that arise in the course of everyday human activity, such as using a computer. We believe that very little, if any, of this kind of reasoning is "formal" in the traditional sense of that term. Because the type of decision making that we intend to model is based primarily on methods of plausible inference and not on methods of logically sound inference, we call this type of decision making Plausible Decision Making (PDM).

Because PDM is based on rules of plausible inference (cf. Collins and Michalski, 1989), a hallmark of PDM is that it typically generates new knowledge. That is, after an episode of PDM, the decision maker has learned something. What the decision maker has learned may or may not be correct, of course, but learning has occurred and it will affect future problem solving.

For example, suppose one is trying to decide what kind of compact car to buy and therefore must decide whether Hyundais are better than Fords. At the beginning of the PDM episode, the decision maker may have no opinion on this issue and knows only that Hyundais are made in Korea. During the PDM episode the decision maker may reason as follows: Because Korea is "close to" Japan, the two countries probably have close economic and technical ties. Through these ties, Korea probably benefits from Japanese expertise in automobile design and engineering. Because all Japanese cars are better than all American cars (the decision maker reasons) Korean cars are probably better than American cars. Notice that this conclusion implies assertions that the decision maker believes after the PDM episode but that did not exist before it. Therefore, PDM typically generates new knowledge. Plausible decision making, such as deciding which make of car to purchase when one has only fragmentary

knowledge about car makes, models, countries of origin, reliability, value for the money, and so forth, has been studied extensively by psychologists (cf. Kahneman and Tversky 1982). Such studies have yielded valuable insight into both the heuristics that people use to attack naturalistic decision making tasks, as well as some of the factual knowledge to which they appeal.

The psychological work, however, has not produced an account of PDM that is sufficiently detailed to support the specification of a computational theory of this ubiquitous kind of human cognition. That is, we do not yet have a precise, detailed description of 1) the structural properties of the informal knowledge that is the basis of naturalistic decision making, 2) the computational form of the rules that carry out reasoning in naturalistic decision making tasks, or 3) the control structure that governs the application of the rules during decision making in this type of informal reasoning. In short psychological studies of informal, naturalistic, decision making have produced descriptive theories rather than process i. e., computational, theories. As well, we do not believe that current "formalistic" models of reasoning can account for Plausible Decision Making. Because our primary goal is to develop a computational theory of informal, naturalistic decision making that can serve both as an explanation of PDM and as the basis for the construction of artificially intelligent support systems for informal reasoning that behave reasonably in extremely underspecified tasks, our research focuses on three facets of plausible decision making that have been largely neglected. In this project, we therefore address the following three research questions:

Research Question 1: What are the properties of the declarative knowledge structures (representations) of beliefs and facts that comprise the knowledge base for informal, naturalistic decision making?

Research Question 2: What are the forms of the rules of plausible inference that perform the inferential reasoning in naturalistic decision making tasks?

Research Question 3: What control structure governs the application of the rules of plausible inference during an episode of informal reasoning for decision making?

Although we intend to study PDM in several types of naturalistic decision making tasks, our goal is not simply to describe patterns of plausible decision making as they arise in the many separate tasks. Rather, we intend to develop a general process model of PDM which describes the characteristics that govern the structure of the informal knowledge used in plausible decision making.

Tasks and Domains for Plausible Decision Making

As we have suggested, plausible decision making is knowledge-intensive and inference-intensive. PDM occurs when a reasoner must make a decision but does not have sufficient knowledge to generate a logically justified decision. In these circumstances, the reasoner generates a "good as necessary" decision by performing plausible inference operations (Collins and Michalski, 1989) on structured knowledge that is potentially relevant to the decision.

We have identified for study three common types of decision tasks that typically require the decider to rely on imperfect information and therefore require the decider to perform plausible inferences. The three decision tasks are:

Selection Tasks: In selection tasks a person must either 1) choose from among several alternatives that are provided or 2) must both generate alternatives and choose from among them.

Advice Giving Tasks: In advice giving tasks, a person listens to a specified problem e.g., of another person and either 1) reasons to select from among alternative courses of action or 2) generates alternative courses of action and then reasons to select from among them.

Resource Investment Tasks: In resource investment tasks, a person must decide whether to 1) begin investing a resource, 2) continue investing a resource, or 3) reallocate resources with the goal of achieving some desirable, but potentially underspecified, outcome or 4) discontinue pursuit of the goal.

We have identified several domains to which we are applying the theory of plausible decision making. Two domains which are relevant to the interaction of computers and humans, and in which PDM play an essential role, are:

Adaptive User Modelling: Effective user modelling appears to be essential to good interactions between humans and computers in complex problems solving situations. A key, unsolved problem in the area of user modelling is giving computers learning algorithms so that they can acquire new models of users when models which the system already possesses are found to be inappropriate. It seems quite clear that when human consultants, or tutors, are helping someone solve a problem – and are therefore maintaining a "user model" of the person they are helping – they are very good at adapting their user models to new kinds of problem solvers. Our initial studies of human tutors shows that they do not reason "logically" when they are developing new, appropriate models. Rather, they use a combination of informal, plausible reasoning strategies

to develop new user models. This reasoning, and the knowledge behind it, is 1) very powerful, 2) poorly understood, and 3) potentially very useful in smart computer systems.

Everyday Problem Solving: It is clear that people do not reason "logically" when they are making everyday decisions such as which car to buy; whether to use leftover prescription medication; how to help someone solve a problem, and so forth. Rather, they use informal, plausible reasoning methods to arrive at a decision. We are currently investigating the potential power of models of plausible reasoning in explaining this phenomenon. We hope that the research will lay the groundwork for developing computer based assistants that can help people make such decisions as well as to learn to improve their decision making. This work is expected to feed into the work on user modelling, described above.

In the full report, we will describe our methodology, results, and tentative analyses.

Informality and The Epistemology of Computer Programming

Tim Standish

DEFINITION

e.pis'te.mol'o.gy (ĕ.pĭs'tē.mōl'ō.jē)
[Greek epistēmē knowledge, + -ology]

The study or theory of the
origin, nature, methods, and
limits of knowledge.

WHAT IS IT?

Ans: It is a theory of the different kinds of
knowledge used in computer programming, and
of the relationships between different kinds
of knowledge.

WHY BOTHER WITH SUCH AN UNDERTAKING?

- better system understanding techniques
needed for:
 - documentation
 - maintenance & upgrade
 - diagnosis & repair
 - training
- deep understanding of types of knowledge
programmers use needed if we are to
find powerful new ways of helping
programmers perform activities of the
software lifecycle.

There is a dream that we can
make a computer understand what we
want it to do for us in a much easier,
more natural fashion than is possible
at present.

This dream keeps recurring.

And it takes many forms.

It propels repeated explorations.

Question: Can we make the computer a
much less mismatched partner in
the process of system implementation and
upgrade?

one form of
exploration

use natural
English

another form of
exploration

concise, clear
readily
comprehensible
specification

detailed
efficient
(less legible)
implementation

... BUT THE DREAM KEEPS BREEDING NIGHTMARES

FORMAL SPEC LANGUAGES SEEM LIKE
"JUST MORE ASSEMBLY CODE"



function Last_Leaf (T: Tree_Node) return Tree_Node is
begin

if T.Left_Child = null and then
T.Right_Child = null

then return T;

elsif depth(T.Left_Child) > depth(T.Right_Child)

then return Last_Leaf(T.Left_Child);

else return Last_Leaf(T.Right_Child);

end if;

end;

function depth (T: Tree_Node) return integer is

begin if T.Left_Child = null and then T.Right_Child = null

then return 1;

else return 1 + max(depth(T.Left_Child), depth(T.Right_Child));

end if;

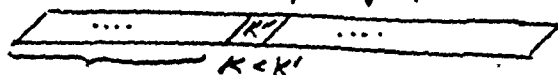
end;

Thursday Presentation
Suppose we start listening to programmers as they explain to each other how programs work.
Hypothesis: When they convey meaning to each other & they are not communicating as mismatched partners — we can get samples of higher cognitive programming techniques in action.

What might we hear?

Examples

"If it finds that the search key K is less than the key at the midpoint of the interval, then it knows K must be to the left of the midpoint and that it cannot be to the right of the midpoint, so it tries to find K in the left half of the interval."



"the rightmost leaf on the bottom row"

36 chars.

VS
115 chars.

What motivates me to think about this stuff?

- program understanding
50-90% of maintenance, 35-50% of lifecycle cost
- better debugging & diagnosis
- the old dream again
- improved methods for teaching C.S.
- enhanced software reuse systems.
if we're going to do software construction from components, we'd better have improved ways of knowing what the components do and are good for.

? FORMALISM VS INFORMALISM ?

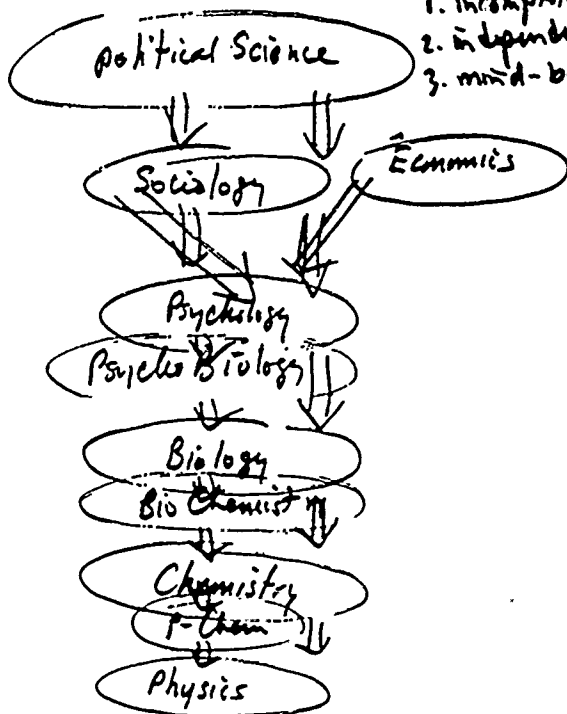
What's the temperature in Celsius?

Examples of Informality:

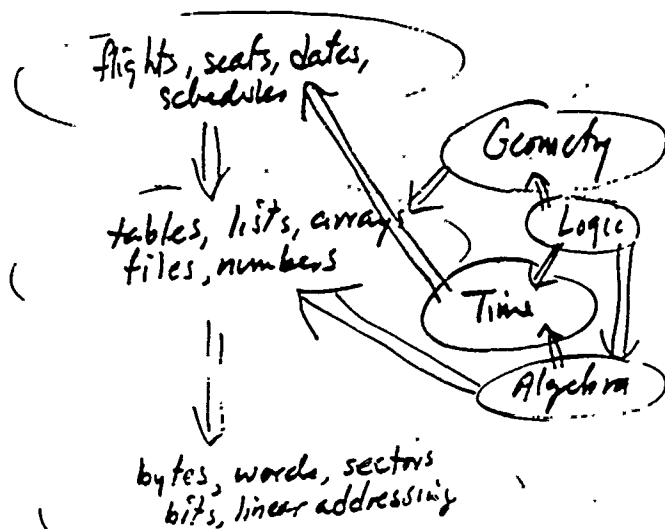
- Use of hunches & heuristics
- Use of buggy causality (superstition)
- intuition
- risk avoidance behavior
- probabilistic reasoning
 - satisficing
 - bayesian probabilities
- non-determinism (coin-flipping decisions)
- progressive debugging
- using (fallible) plans (2 modal logic)
- Poincaré phenomenon
 - incubation & sudden illumination
- incompleteness in descriptions
- vagueness - open to interpretation

Problems:

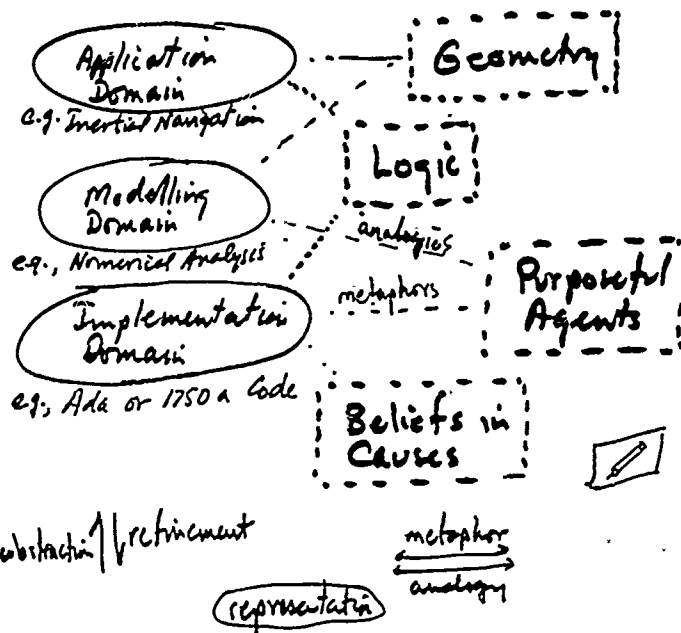
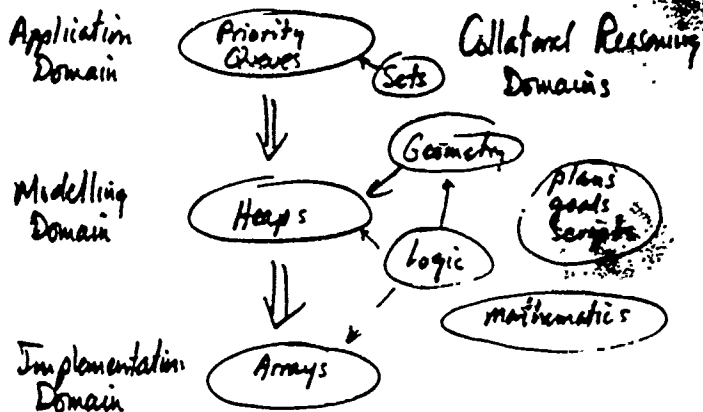
1. incompleteness
2. independence
3. mind-body



Airline Reservation System



Reductionism and Representation



Things we find in Domains

Scripts & Plans
 Transformations
 Definitions
 Basic Mechanics
 Relationships
 Laws
 Facts & Theorems
 Objects
 Operations
 Properties
 Beliefs & Expectations
 Second Order Knowledge

General Purpose Domains

PROLOG WORLD	GEOMETRY WORLD	LOGIC WORLD
MATH WORLD	COMMON SENSE WORLD	

RELATIONSHIPS BETWEEN DOMAINS

- Representation -- e.g. p-t sort in geo-world
- Script Instantiation (2 Plan Application) -- e.g. find max or min of diff. fn.
 -- e.g. conv. by repeated halving of problem size
 • Newton's SQRT
 • Binary Search
- Chronological Jumps -- e.g. geo-jumps to make reasoning easier or explanation clearer

$$a < b \Rightarrow a < \frac{a+b}{2}$$

mid-pt

easier than:
 $a < b \Leftrightarrow 2a - a < b$
 $\Leftrightarrow 2a < a + b \Leftrightarrow a < \frac{a+b}{2}$

ANOTHER VIEW OF THE VARIOUS DOMAINS

MATH WORLD

- Geometry
- Logic
- Algebra
- Axiomatic Data Str.
- Axiomatic Goal Collection

COMMON SENSE WORLD

- goal mechanics
- script application
- 2nd order knowl.
- empirical regularity

PRECISION

IMPRECISION

scripts, plans, goals

[NL] John went to a restaurant. He ordered cog and vin. He left a large tip;
 [mathematics]

THEOREM $\text{Sum}(1..n) = n * (n+1) / 2$

PROOF

"We proceed by induction. The base case follows from the equation $1+2/2 = 1$. Assume the result is true for n . Now note:
 $\text{Sum}(1..n+1) = (n+1) + \text{Sum}(1..n) = (n+1) + n(n+1)/2$
 $= (n+1)(n+2)/2$ "

LOGIC

Assume P is false; get a contradiction; therefore P must be true

PROGRAMMING

{some initialization to get things started};

while {not close enough to the goal} loop

{take a step that goes halfway to the goal};

end loop;

binary search, newton's square root, AVL-tree search

non-stereotypy: "out of menus"; interpolation

John went to a restaurant.
The waiter came over and said they
were out of money. John asked the
waiter what they recommended.

STORY

John wanted some money.
He decided to rob a liquor store.
He bought a gun, and went to
"Al's Package Store" on the corner
of Fifth and Vine.
He escaped with \$300.

Why did the owner give John the money?

-- exchange contents of x and y
temp := x ; -- <Your comment goes here>
 x := y ;
 y := temp;

- o Wilensky's Theory of Intentional Explanation
- o Sussman's Theory of Progressive Debugging
"bugs are necessary for the intellectual
economy of general problem solvers"

<u>item to explain</u>	<u>explanation</u>
Event	Plan of which event is part
Plan	Goal at which plan is directed
Goal	Theme that gave rise to goal, or plan to which goal is instrumental

Theory: your comment will say is effected

- * -- save value of x for later use
OR
- * -- prevent destruction of value of x

Gerald J. Sussman
A Computational Model of Skill Acquisition

progressive debugging

"bugs are necessary, you can't build big systems without them."

e.g. radio

exchange x, y

contents(x) = x0 & contents(y) = y0

↓
 contents(x) = y0 & contents(y) = x0

divide & conquer

achieve(A & B) => achieve(A); achieve(B)

↓ ↓
 x := y ; y := x

note interaction bug, remove bug - unsatisfied prerequisite bug

Temp := x; -- prevent destruction of value of x
 x := y;
 y := Temp;

Binary Search
 Thursday Presentations 119
 e.g., telephone book

procedure Binary-Search (K: Key) is

-- let T be an array of keys
 -- numbered from 1 to n
 -- T = [K₁, K₂, K₃, ..., K_n]
 -- where K_i < K_{i+1} for each i in 1..n-1.
 -- let L, R and i be integers

Begin

L := 0;
 R := n+1;

Loop

i := Integer-Part-Of ((L+R)/2);

exit when i = 0; -- exit when {search interval is empty}

if K = T(i) then
 return i;

elsif K < T(i) then
 R := i;

else
 L := i;

end if;

End Loop;

End Procedure;

"If it finds that the search key K is less than the key at the midpoint, then it ~~also~~ knows K must be to the left of the midpoint, so it tries to find K in the left half of the search interval."

Here Binary Search is portrayed as an agent that:

(1) is pursuing a goal - trying to find the search key in the table

(2) can know facts - MTrans (K must be left of the midpoint)

(3) can act on facts to direct its goal pursuit activities.

Purposeful Agents

Purposeful Agents model agents that can plan, reason, know, remember, move, manipulate objects, store and retrieve objects, use instruments, pursue goals, solve problems, decide, want, try, accept obligations, obtain and grant permission, believe in causes, reason about time and causality, use logic and facts to make deductions, act on the basis of reasoned conclusions, possess, lend and borrow, schedule and prioritize, postpone goals and activities, navigate in scenes, execute procedures, apply scripts, and recognize scripts.

Paps model our common sense about rational beings acting in pursuit of purposes.

John wanted to find his leather jacket. He knew it was either in the hall closet or in the bedroom closet. He asked Mary where it was, and she told him it wasn't in the hall closet. So he looked for it in the bedroom closet.

↓ abstraction

If agent A is trying to find X in B or C and discovers X is not in C, then it knows X must be in B, so it tries to find X in B.

```

function G(x:Real; n:integer): Real;
var p: Real;
begin
  if n = 0 then
    return(1.0)
  else begin
    p := G(x, n div 2);
    if (n mod 2) = 0 then
      return(p * p)
    else begin
      return(x * p * p)
    end
  end
end
end(G);
    
```

```

function Power(x:Real; n:integer): Real;
var p: Real;
begin
  if n = 0 then
    return(1.0)
  else begin
    p := Power(x, n div 2);
    if (n mod 2) = 0 then
      return(p * p)
    else begin
      return(x * p * p)
    end
  end
end
end(Power);
    
```

(base case, $x^0 = 1.0$)

(let $p = x$ to the half(n) power)

(if n was even, return Square(p))

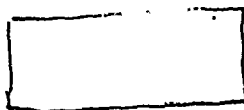
(if n was odd, return x*Square(p))

The proof of the theorem goes something like this:
 proof:

① Assume $f(x) < c$.
 ... get a contradiction

Now,
 ② Assume $f(x) > c$
 ... get another contradiction

③ Therefore,



what?

strange gratuitous inferring:

Garden Path Sentences

"The old man's glasses ...

Picture a whale with a
 top hat and a cigar.

Theorem 4.19 Weierstrass Representation 12 [a, b].

If $f(a) < f(b)$, and if c is any number such that $f(a) < c < f(b)$ then there exists a point x in (a, b) such that $f(x) = c$.

Proof Let A be the set of points in $[a, b]$ such that $f(t) < c$. Since $a \in A$, A is nonvacuous, and A is clearly bounded. Let x be the lub of A . Then $x \in [a, b]$.

Suppose $f(x) < c$. Since f is continuous at x there exists a point $y \in (x, b)$ such that $f(y) < c$, so that $y \in A$. This contradicts the fact that x is an upper bound of A .

Suppose $f(x) > c$. Since f is continuous at x , there exists a point $y \in (a, x)$ such that $f(t) > c$ if $y \leq t \leq x$. Also, $f(t) \leq c$ if $a \leq t \leq y$, since x is an upper bound of A . Thus $f(t) \leq c$ if $a \leq t \leq b$, so that y is an upper bound of A . This contradicts the choice of x as the least upper bound. Hence $f(x) = c$.

SCRIPTS USED

construction
 suppose $f(x) < c$

two reductio ad absurdum

⊙

$\neg(f(x) < c)$

and a modus tollendo ponens

suppose $f(x) > c$

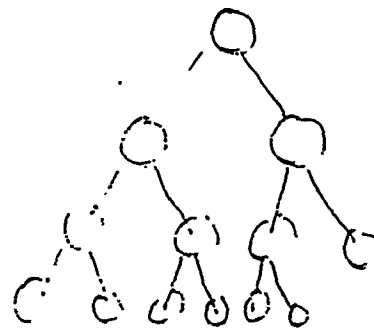
⊙

$\neg(f(x) > c)$

$f(x) < c \vee f(x) = c \vee f(x) > c$
 $f(x) = c$ m.t.p.

look at what you have to interpolate to fill in the scripts

Geometry Domain



- right most leaf on bottom row
- last leaf in level order

the drawings in Figure 15-5 and asked to select the most plausible among them. We were sure this task would be quite easy. The subjects would have all 15 versions before them. If the correct version did not just pop out at them, they could compare and contrast the alternatives in any way they pleased.

100

of 36 female students from Lesley College (all U.S. citizens) was given the 15 drawings from Experiment IV, each printed on a separate card. The subjects were told that one of the drawings was correct, and that each of the others had one or more things wrong with it. The

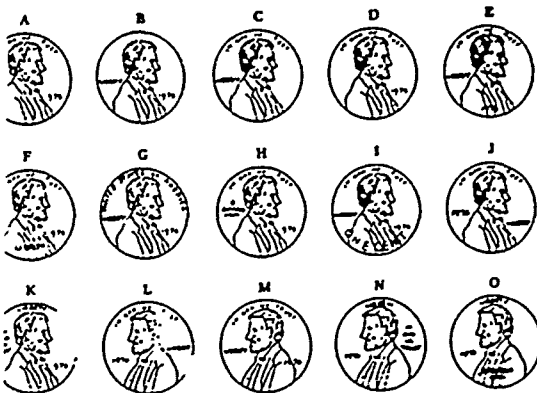
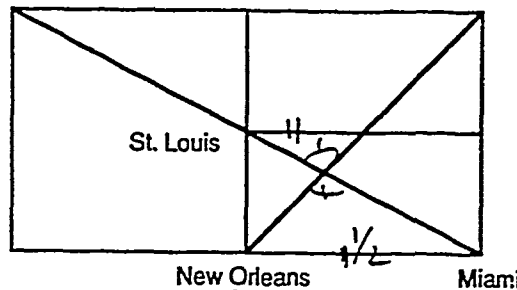


FIGURE 15-5
Fifteen drawings of the top side of a penny that were used in experiments IV and V. A brief characterization of each drawing is given in Table 15-3.

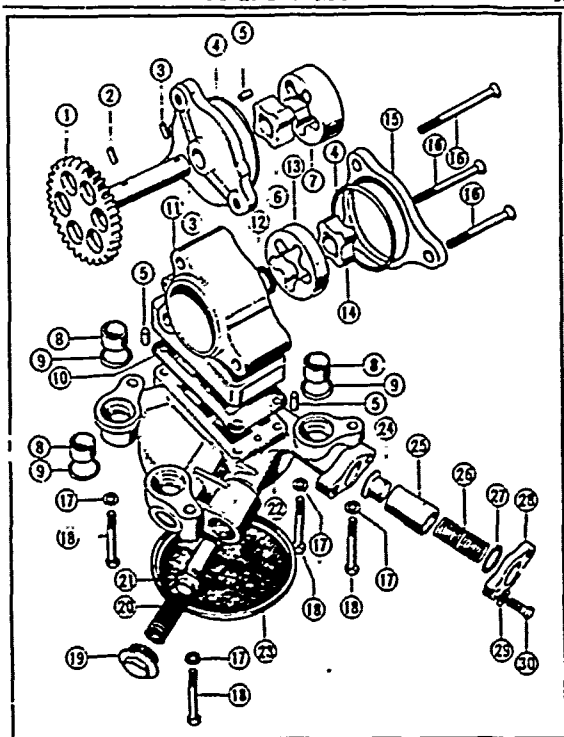
Seattle

New York



3-2 LUBRICATION SYSTEM

31



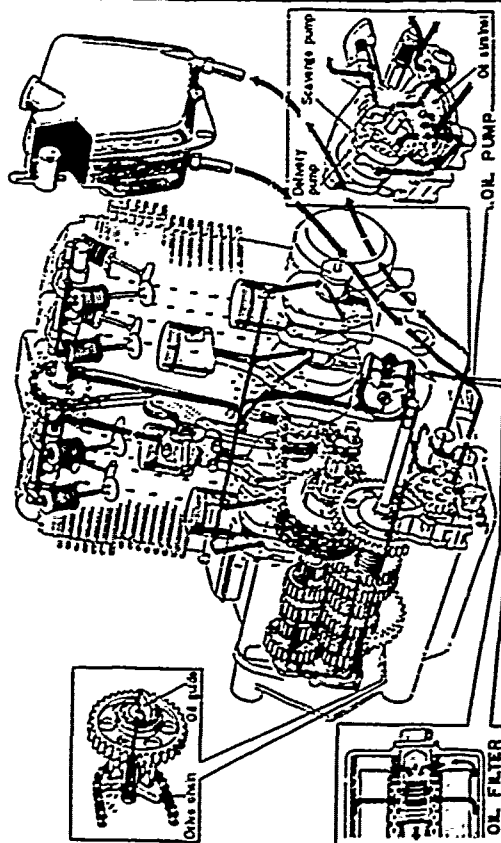
- | | | |
|------------------------|------------------------|----------------------------|
| 1 Oil pump drive gear | 11 Oil pump body | 21 Relief valve |
| 2 4x14mm pin | 12 11x15x3 O-ring seal | 22 Oil pump base |
| 3 Oil pump right cover | 13 Outer rotor A | 23 Oil strainer screen |
| 4 4x2 "O" ring | 14 Inner rotor A | 24 Oil test stopper seal |
| 5 4x8 dowel pin | 15 Oil pump left cover | 25 Oil test stopper valve |
| 6 Inner rotor B | 16 2x29 flat screw | 26 Oil test stopper spring |
| 7 Outer rotor B | 17 6mm flat washer | 27 11x23 "O" ring |
| 8 "O" ring cover | 18 6x12 hex bolt | 28 Oil test stopper cap |
| 9 11x23 "O" ring | 19 Relief spring cap | 29 6mm flat washer |
| 10 Oil pump gasket | 20 Relief valve spring | 30 6mm flat bolt |

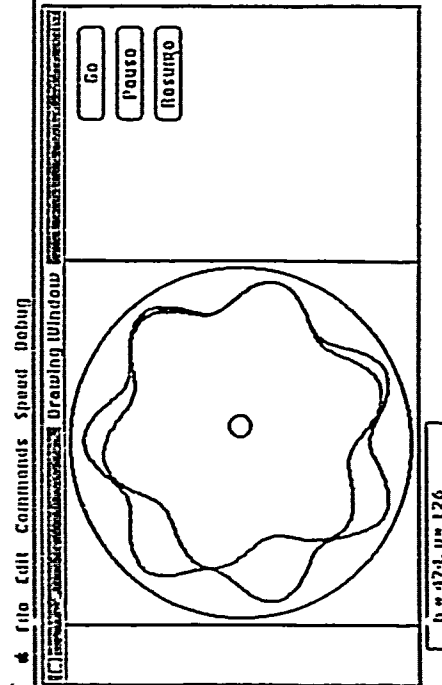
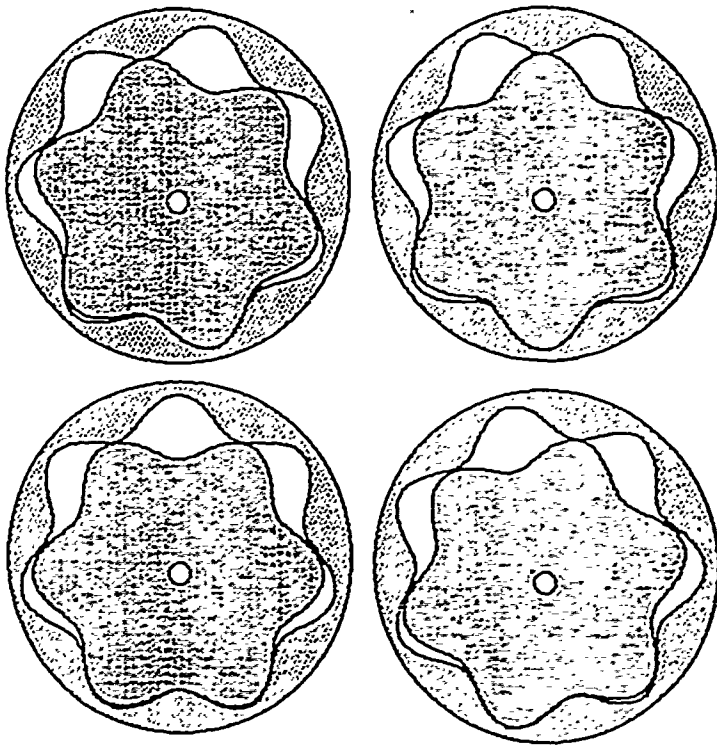
Fig. 3-17

26

3. ENGINE MECHANICAL

3-2 LUBRICATION SYSTEM





Conclusions

1. If we investigate epistemology and cognition, it is likely that:
 - a. We can develop (and apply profitably) - better explanation techniques
 - b. We can develop limited forms of enriched linguistic mechanisms (and the formal inferencing needed to support them) to use in our system description languages and our programming languages, so that the computer becomes a less mismatched partner at the task of system description.

Intensional Logic and the Metaphysics of Intentionality

Edward N. Zalta

1 Four Apparently Unrelated Problems

Properties, Relations, and Propositions

Should they be taken as primitive or defined?
Under what conditions are PRPs identical?
What is the theory?

Situations and Possible Worlds

Can they be incorporated into a single theory?
What is the theory?

The Problem of Fiction

According to *The Tempest*, Prospero had a daughter.
Prospero is a character of *The Tempest*.
The events and characters of *The Tempest* are fictional.
So, Prospero is a fictional character who, according to *The Tempest*, had a daughter.

Ponce de Leon searched for the fountain of youth.
So, Ponce de Leon searched for something.

The Loch Ness Monster doesn't exist, which is why no one has found it.

Fregean Senses and Substitution in Intensional Contexts

Frege's Puzzle: Cognitive Significance of $a = a$ vs. $a = b$.
Susie believes Mark Twain wrote *Life on the Mississippi*.
Mark Twain = Samuel Clemens.
Susie doesn't believe that Samuel Clemens wrote *Life on the Mississippi*.

Susie believes that Alan is my brother.
Being a brother is the same thing as being a male sibling.
Susie doesn't believe that Alan is my male sibling.

2 The Theory

Language

One minor modification of the standard logic of exemplification—add a new atomic formula of the form ' xF ' to express the statement that object x encodes property F . Distinction between ' $Fx_1 \dots x_n$ ' and ' xF '. Complex formulas and terms: $\neg\phi$, $\phi \rightarrow \psi$, $\forall\alpha\phi$, $\Box\phi$, $[\lambda x_1 \dots x_n \phi^*]$. Distinguished predicate: E !

Interpretation of Language

Primitive Domains: Objects, Relations, and Worlds. Each relation receives an exemplification at every world. Each property receives an encoding extension (that stays fixed from world to world). ' $Fx_1 \dots x_n$ ' is true (relative to an assignment) just in case the n -tuple of objects assigned to x_1, \dots, x_n is an element of the exemplification extension of the relation assigned to ' F '. ' xF ' is true (relative to an assignment) just in case the object assigned to x is an element of the encoding extension of the property assigned to F .

Logic

Standard propositional logic. Standard quantificational logic. Standard $S5$ modal logic (possibilist quantifier ranging over a single, fixed domain). The logic of encoding: $\Diamond xF \rightarrow \Box xF$.

Non-logical Principles

(A) Ordinary individuals necessarily fail to encode properties.

$$(\forall x)(O!x \rightarrow \Box(\exists F)xF)$$

(B) For every condition on properties, it is necessarily the case that there is an abstract individual that encodes just the properties satisfying the condition.

For every formula ϕ in which x doesn't occur free, the following is an axiom:

$$\Box(\exists x)(A!x \ \& \ (\forall F)(xF \equiv \phi))$$

- (C) Two individuals are identical if and only if one of the following conditions holds: (a) they are both ordinary individuals and they necessarily exemplify the same properties, or (b) they are both abstract individuals and they necessarily encode the same properties.

$$x=y =_d$$

$$(O!x \& O!y \& \Box(\forall F)(Fx \equiv Fy)) \vee (A!x \& A!y \& \Box(\forall F)(xF \equiv yF))$$

- (D) If two individuals are identical (or two properties are identical), then anything true about the one is also true about the other.

$$\alpha = \beta \rightarrow (\phi(\alpha, \alpha) \equiv \phi(\alpha, \beta))$$

3 The Explanation

Properties, Relations, and Propositions

Principle: For every exemplification condition on individuals, there is a relation which is such that, necessarily, all and only the individuals satisfying the condition exemplify it.

For every formula ϕ in which there are no free F's and no encoding subformulas, the following is an axiom:

$$(\exists F)\Box(\forall x)(Fx_1 \dots x_n \equiv \phi)$$

Principle: Two properties are identical just in case it is necessarily the case that they are encoded by the same individuals.

$$F=G =_d \Box(\forall x)(xF \equiv xG)$$

So, there are two notions of necessary equivalence: (a) same exemplification extension at each world, and (b) same encoding extension at each world. Properties that are necessarily equivalent in the sense of (a) need not be necessarily equivalent in the sense of (b): it does not follow from ' $\Box(\forall x)(Fx \equiv Gx)$ ' that ' $\Box(\forall x)(xF \equiv xG)$ '. So the former doesn't entail ' $F=G$ '.

Situations and Worlds

Situation(x) =_d $\forall F(xF \rightarrow \exists p(F = \{\lambda y p\}))$, i.e., a situation is an object that encodes just properties of the form being such that p , for some state of affairs (proposition) p

$s \models p =_d s\{\lambda y p\}$, i.e., state of affairs p is *factual* in situation s iff s encodes the state-of-affairs property being such that p .

Theorem: $s = s' \leftrightarrow \forall p(s \models p \leftrightarrow s' \models p)$; i.e., two situations are identical just in case the same state of affairs are factual in them.

$x \trianglelefteq y =_d \forall F(xF \rightarrow yF)$; i.e., x is a part of y iff y encodes every property x encodes.

Theorem: Part-of (\trianglelefteq) is reflexive, anti-symmetric, and transitive on the situations; i.e., \trianglelefteq partially orders the domain of situations

$World(s) =_d \Diamond \forall p(s \models p \leftrightarrow p)$; i.e., a situation s is a *world* iff it is possible that all and only factual states of affairs are factual in s .

Theorems: Worlds are maximal, consistent, and closed. There is a unique actual world. A state of affairs p is necessary iff it is factual at all worlds.

The Problem of Fictions

Let c be a character of story s . Then,

$$cF \leftrightarrow \text{In the story } s, Fc$$

Fregean Senses and Substitution in Intensional Contexts

Cognitive Significance of $a = a$ vs. $a = b$

$$B(s, Wt) \quad (re)$$

$$t = c$$

$$B(s, Wc) \quad (\text{Substitution permissible})$$

$$B(s, Wt_s) \quad (dicto)$$

$$t = c$$

Rule of Substitution inapplicable

$$B(s, Ba) \quad (re)$$

$$B = [\lambda x Mx \& Sx]$$

$$B(s, [\lambda x Mx \& Sx]a) \quad (\text{Substitution permissible})$$

$$B(s, B, a) \quad (dicto)$$

$$B = [\lambda x Mx \& Sx] \quad (\text{Rule of Substitution inapplicable}).$$

(MIT/Bradford Books, 1988)

(1) a set of PHYSICAL TOKENS (scratches on paper, holes on a tape, events in a digital computer, etc.) that are

- (1) a set of PHYSICAL TOKENS (scratches on paper, holes on a tape, events in a digital computer, etc.) that are
- (2) manipulated on the basis of EXPLICIT RULES that are
- (3) likewise physical tokens and STRINGS of tokens. The rule-governed symbol-token manipulation is based
- (4) purely on the SHAPE of the symbol tokens (not their "meaning"), i.e., it is purely SYNTACTIC, and consists of
- (5) RULEFULLY COMBINING and recombining symbol-tokens. There are
- (6) primitive ATOMIC symbol-tokens and
- (7) COMPOSITE symbol-token strings. The atomic tokens, the composite tokens, the syntactic manipulations and the rules are all
- (8) SEMANTICALLY INTERPRETABLE: The syntax can be assigned a systematic meaning (e.g., as standing for objects, as describing states of affairs).

There are a few tricky points associated with the concept of a symbol system that people keep misunderstanding. All eight of the properties listed above are critical to this definition of symbolic. Many phenomena have some of the properties, but that does not entail that they are symbolic in this formal, explicit, technical sense. A sense of "symbolic" that is merely unexplicated metaphor.

(1) Searle's Chinese Room Argument:

One can perform all the symbol-manipulative functions of a pure symbol-cruncher without understanding the symbols' meanings (e.g., one could pass the Turing Test for Chinese by manipulating Chinese Symbols without understanding Chinese). The composer's "symbols" are ungrounded; their meanings are parasitic on the meanings of human symbols, whereas the meanings of human symbols are intrinsically grounded.

(2) **The Chinese/Chinese Dictionary-Go-Round:**

Case 1 (hard):

五五五五五

Suppose you had to learn Chinese as a second language, and the only source of information you had was a Chinese/Chinese dictionary.

The trip through the dictionary would amount to a merry-go-round, passing from one meaningless symbol or symbol-string to another, never coming to a halt on what anything means. The only reason cryptologists can do this is because their efforts are grounded in a first language and real world experience and knowledge.

Case 2 (impossible?):

Suppose you had to learn Chinese as a first language, and the only source of information you had was a Chinese/Chinese dictionary.

(3) Symbolic AI and its Toy Models

Symbolic AI's very limited problem-to-problem, generality and its failure to produce nontrivial cognitive principles may be a consequence of the ungroundedness of pure symbol manipulation. Just as in the case of the chimpanzees (or pigeons) trained in "Yerkish," the meaning of AI's symbols may be in our heads only.

【斑点】 在一种颜色的物体表面上显出来的别种颜色的点子。

【证明】 〈书〉雄壮多彩，五色~。

【斑蟊】 昆虫，触角呈棒状，腿细长，鞘翅上有黄黑色斑纹，吃小虫。中医用来治疥癣和腐肉等病。

【斑纹】 在一种颜色的物体表面上显
露出来的别种颜色的条纹。老虎身上有美丽
的~。

【斑竹】竹子的一种，茎上有紫褐色的斑点。茎可以制装饰品、手杖、笔杆等。也叫湘妃竹。

●不須測具體情況，把現成的制皮、縫線、方法、圖句等拿來使用。～收條／生～硬套。

【策定】全家离开原居居住的地方

【搬弄是非】把別人背後說的話傳來傳去，蓄意挑撥，或在別人背後亂加議論，引起糾紛。

【策反】把旧军策反出来，~故事。
【搬运】把大量的东西从一个地方运

CONJECTURE 1: Grounded higher-level categories can be generated by simply stringing together the symbols for grounded lower-order categories in the form of propositions about category membership.

Example of Groundings:

- (1) Suppose the symbol "horse" is grounded by iconic and categorical representations, learned from experience, that reliably discriminate and identify horses on the basis of their sensory projections.

Now consider that the following category can be constituted out of these elementary categories by symbolic description of category membership alone:

(3) A "zebra" = "horse" & "stripes"

Conjecture 2: 漢王

- Connectionism is a natural candidate for the pattern learning mechanism that underlies the categorical representations.

If these conjectures are correct, then rather than being viewed as symbolic AI's rival for cognitive hegemony, connectionism should be viewed as a potential component in a nonsymbolic system in which symbolic representations emerge as an intrinsically dedicated functional level grounded in nonsymbolic representations rather than the autonomous symbolic module envisioned by symbolic functionalists.

Strengths:

- (1) Nonsymbolic Function (no symbol grounding problem)
- (2) Generality (same algorithm applies to many problems)
- (3) "Neurosimilitude" (looks brain-like)
- (4) Pattern Learning

Weaknesses:

- (1) Nonsymbolic Function (no systematicity)
- (2) Generality (not every problem amenable to pattern learning; and there may be pattern learning problems that C cannot handle)
- (3) "Neurosimilitude" (brain-likeness may be superficial and may mask performance limitations)

SYMBOL-MANIPULATION: ITS STRENGTHS AND WEAKNESSES

Strengths:

- (1) Symbolic Function (Turing power, systematicity, semantic interpretability)
- (2) Generality (Turing equivalence)
- (3) Practical Successes (Intelligent machine performance)

Weaknesses:

- (1) Symbolic Function (Symbol grounding problem)
- (2) Generality (Toy problems, ad-hocness)

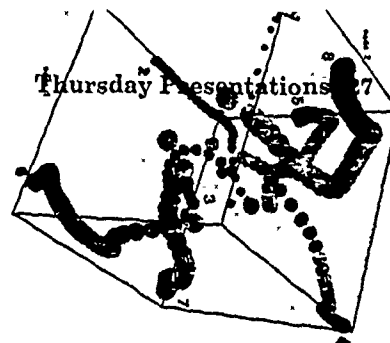


fig 2c-1

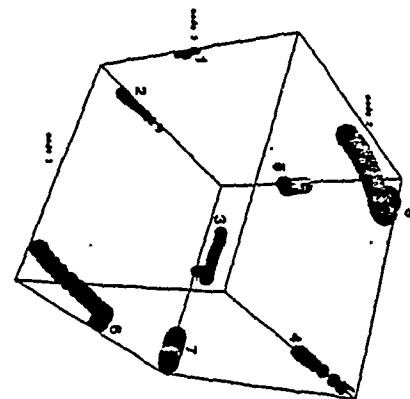


fig 2c-2

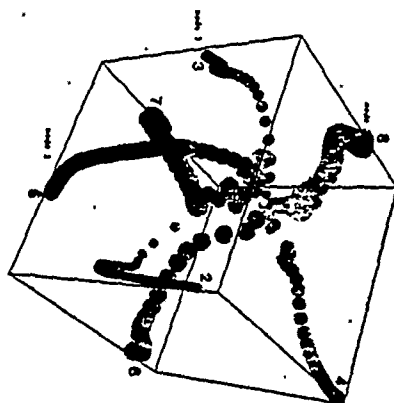
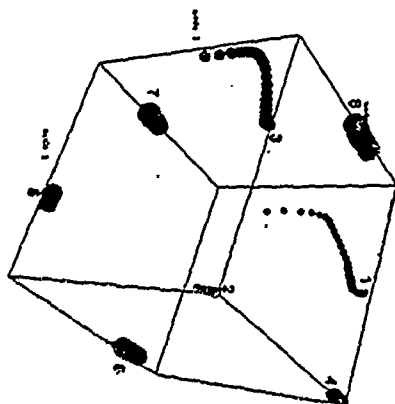


fig 2d-1



2d-2

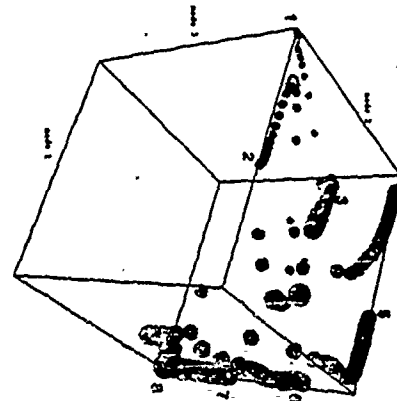


fig 2c-1

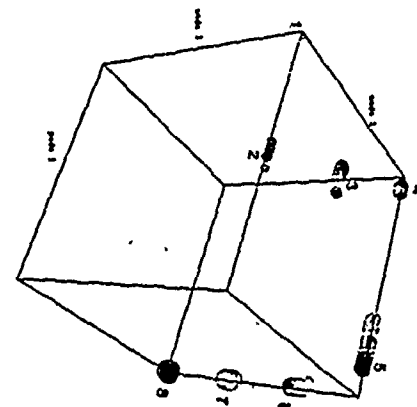
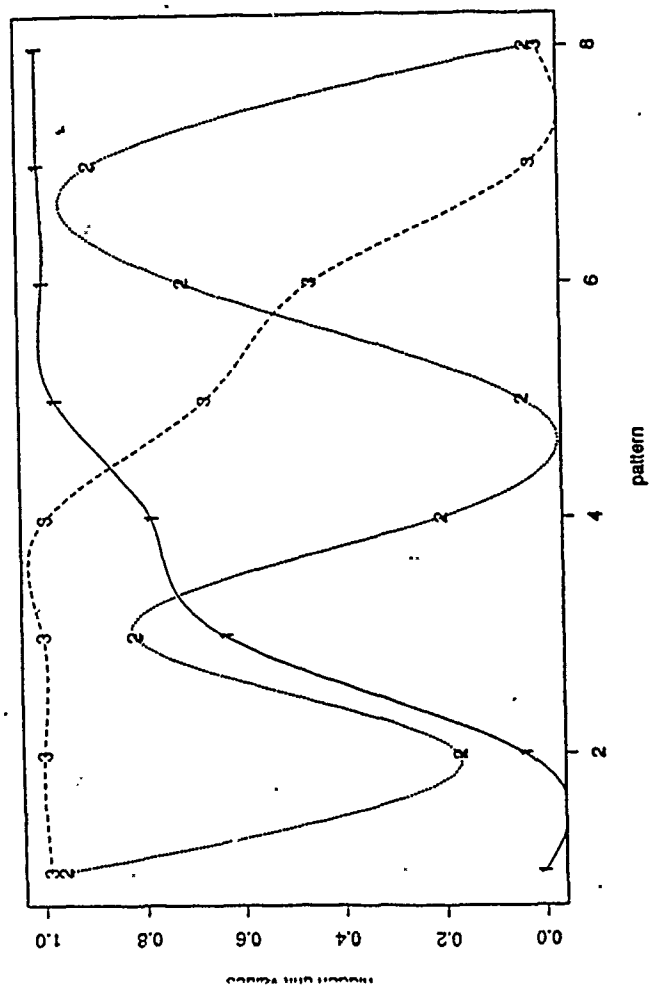
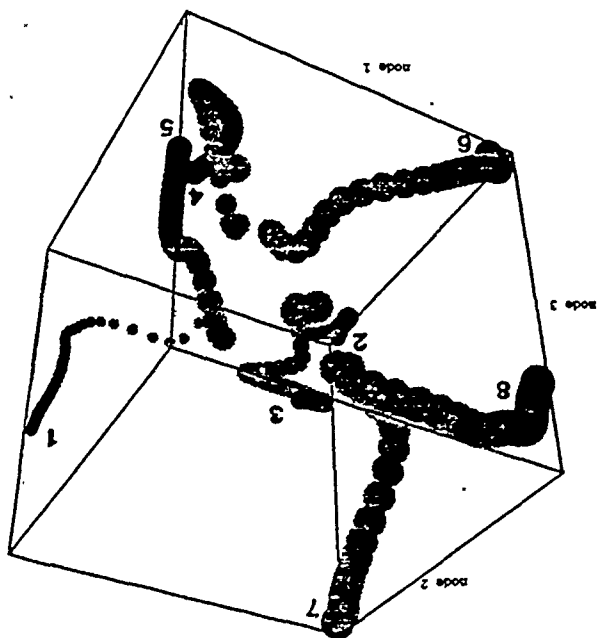
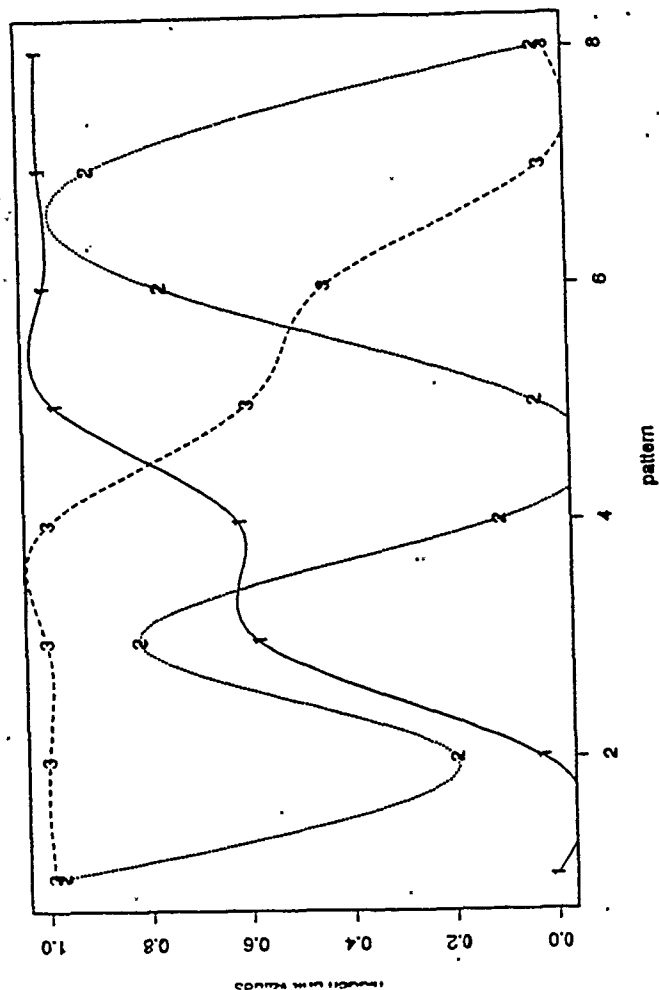


fig 2c-2

Series 7 -- Discrete Line (3 hid) -- Network a0

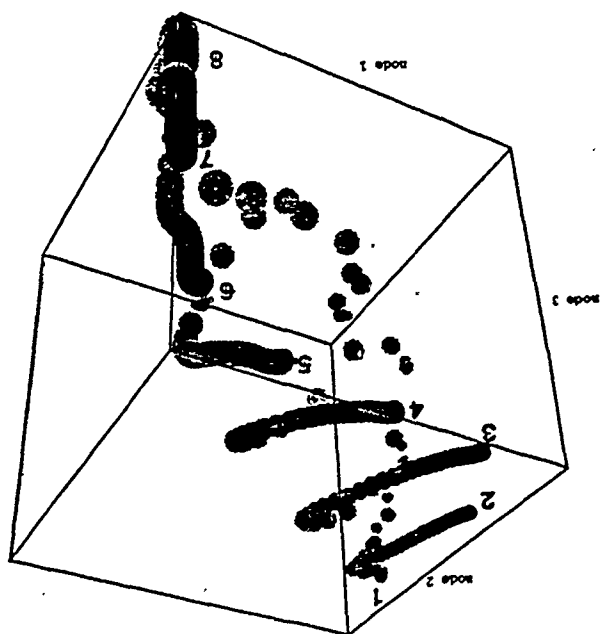


Series 7 -- Discrete Line (3 hid) -- Network.c0.0



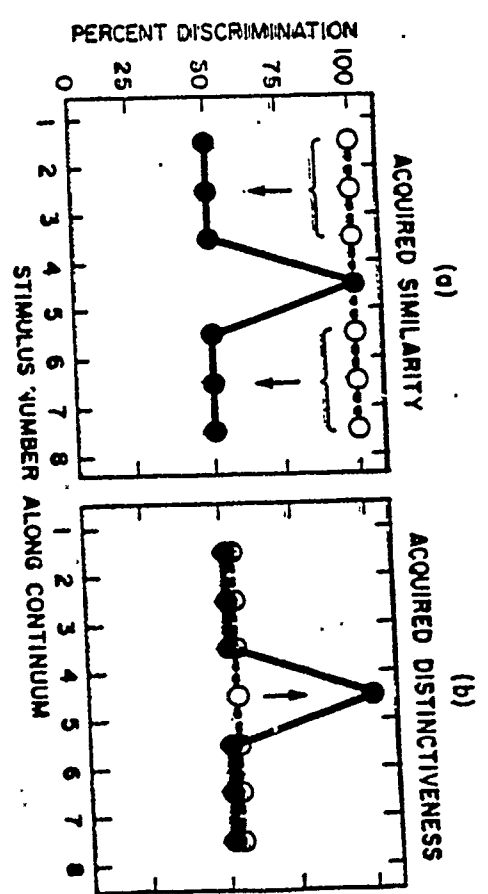
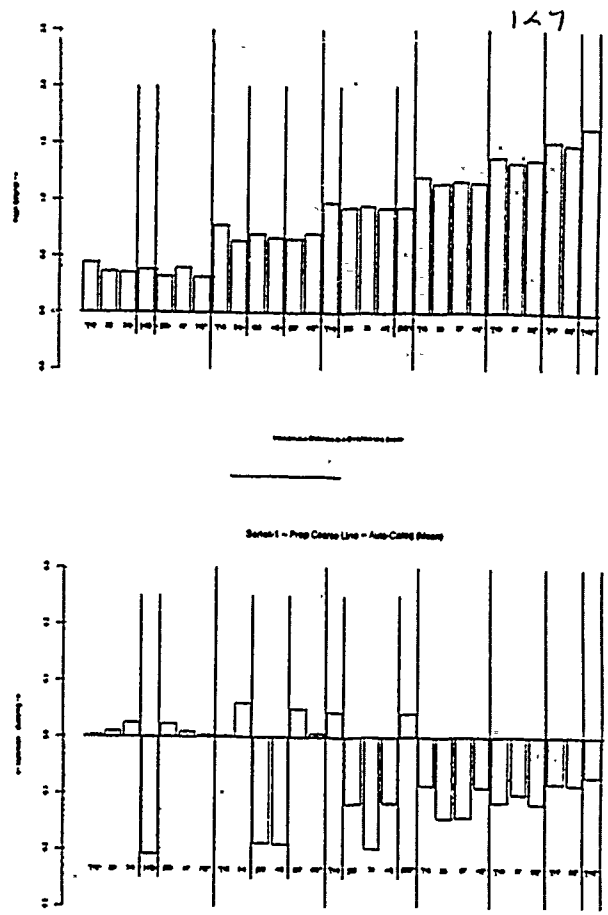
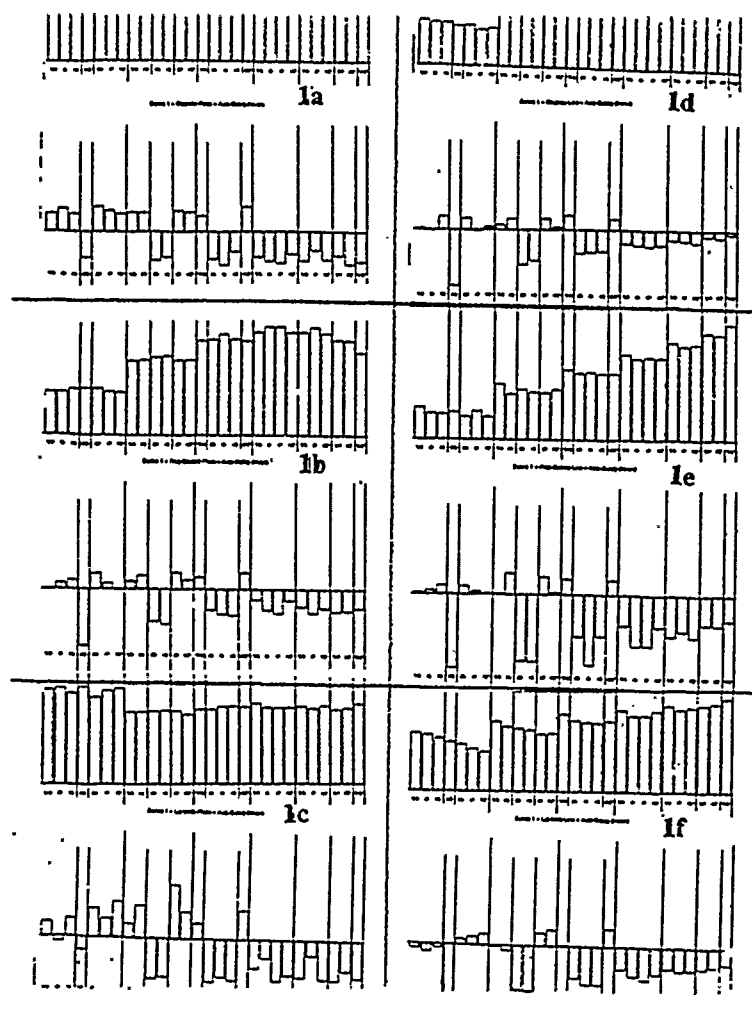
Series 7 -- Discrete Line (3 hid) -- Network a0 -- Passes 0-1000 (BY 20)

fig 2a

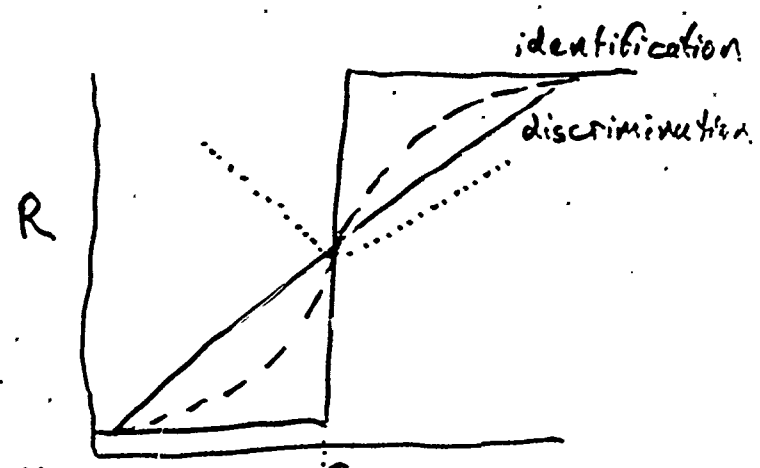




Series 7 -- Discrete Line (3 hid) -- Network.c0.0 -- Passes 0-1000 (BY 20)

fig 2b



Categorical Perception (CP)



Ind's:
isotropic: 
anisotropic: 
(warped)

Color CP and the
Whorf Hypothesis's
Phoneme CP and the
Motor Theory

DISCRIMINATION:

Judging whether two inputs are the same or different and how similar they are (a relative judgment).

Requisite representation: Iconic Representation (analog of sensory projection).

IDENTIFICATION:

Assigning a unique arbitrary response (a label) to a class of inputs (an absolute judgment). Also called "categorization."

Requisite representation:

Categorical Representation (preserves only the invariant features of the sensory projections of the members of the category that reliably distinguish them from nonmembers with which they can be confused).

Note: Both iconic and categorical representations are NONSYMBOLIC

Modeling Human Performance Capacity

The data are in: Human beings are able to

- (1) discriminate,
- (2) manipulate,
- (3) identify and
- (4) describe

the objects, events and states of affairs in the world they live in, and they can

- (5) produce descriptions and
- (6) respond to descriptions

of the objects, events and states of affairs in the world they live in.

Now cognitive science's burden is to explain **HOW** human beings (or anything) manages to do all this.

A candidate model that would do all this would pass the Total Turing Test, exhibiting all of our performance capacities, linguistic (5-6) and robotic (1-3) (i.e., sensorimotor). (The standard Turing Test requires linguistic performance capacity only; it is equivocal about the status, scope and limits of pure symbol manipulation.)

Manipulation (2) will not be discussed here, but (1) discrimination and (3) identification will be examined in the context of "categorical perception" as bases for grounding symbolic description (4-6).

Thursday Afternoon Discussion

Shultis: Some of the things that started to come up at the end of yesterday's discussion, and that have come up again today are, "Well, what do we do? What sorts of projects can we outline?" I think that one way of looking at the problems that we all confront is that the fundamental fact is our unease, our feeling that there must be something else, that there must be more to life than formalisms. And perhaps a way to start thinking about what we might do to start drawing up some plans, or conclusions, or projects, for ourselves as a community is to start thinking about what it is that we want. I think we've seen a lot of wishes, or a lot of feelings about what sorts of things we might like. An example of that that came up earlier today was the problem of code explanation, for doing maintenance. Another project that came up today, rather prominently, and it's sort of been sitting there all the time in the background, is how we build our successors. *[Laughter.]* If you take seriously the idea of building an autonomous cognitive agent, then you can say, well, look, we can probably build it so its spiking frequencies are a lot higher than human spiking frequencies, and maybe it can do things a lot faster and maybe just better than us, anyway, and we eventually just become instruments of this device, and it's just the next stage along evolution, or something. Somebody said something like that—I guess you did, David.

Littman: But it's only half. I mean, when I set out to establish a methodology for supporting guided robots, it came from an idea I've had for a long time, which is to develop a support environment for building minds.

Harris: I'd like to make a comment on that. Terry Sejnowski has an example of why we needn't fear a computer takeover...

Littman: I'm not afraid of it!

Harris: ...and it's just that we would need something like 10^5 times as many Cray computers as we currently have on the planet to have the same number of pieces of hardware as there are in a human brain.

Littman (laughing): Well, I think, not to digress, the IRS already has more information about me than I'd like them to have, so I think they've already taken over. *[Laughter.]*

Shultis: Well, just to get a focus, or a bead on where we're going, we might say that building robots is a long-term project, and in terms of thinking about the kinds of capacities and abilities that we're trying to achieve here, maybe we get to a point where what you're talking about is something that has a good deal of robotic capability. Whether we compete with them or not

for the same resources and end up in trouble I'll leave for the future to decide, because it's pretty moot right now.

Littman: It seems to me, and I'm serious about this, that a long-term goal might be to build robots that Steve Harnad would not eat.

Shultis: Well, I think he could never really be sure. There's always a nagging doubt that maybe that carrot really does have qualia. But a more serious question is: can we identify small projects that might help to focus us in terms of what we can do in the medium term. There are two set of things that have come up that Bruce [D'Ambrosio] and I were talking about this morning, and that was to look at decision-making in engineering design. If you think about the sorts of things Bruce was talking about this morning—real-time decision-making, decision-making under constraints (that fact that you have to do something now, that you can't wait)—in engineering, you're also in a space where you need to make decisions, and you need to make them within certain limitations of time and other resources. And, even though it's a different kind of application domain that doing on-line maintenance and repair of machines—it's not all that different. Somebody said, I can't remember who, that programming is just debugging a blank page; there's some repairing to be done. And so, that's one kind of project that I've seen come up. Correct me if I'm wrong, Bruce, but is that a medium-term kind of project, not something we could expect to accomplish in the next 2-3 years?

D'Ambrosio: Medium-term meaning not short-term, yes. There's a whole separate range of issues on the human-computer side. I've been looking at mechanical engineering design, at least, and there's a lot of issues there just in terms of the human interface and problem acquisition.

Standish: Somewhere floating around Washington was a document created, maybe by Feigenbaum and some others, on an engineering assistant, and they wanted to pose grand challenge problems that would plant money-suckers on the government and flow large amounts of cash to their coffers and do something big and useful, and they felt that making an engineering design assistant, as an artificially intelligent creature was a worthy goal that has a large, programmatic description. Has anyone ever seen that?

D'Ambrosio: I haven't seen the document. I have heard Steve Crocker's talk about assistants and associates.

Standish: Yeah, that's because it probably went to him and he's floating that. Is that going to go anywhere? Because if it is, that's the train to hitch a ride on.

D'Ambrosio: My impression is that they're trying to sell it inside and outside. Inside to get the funding and outside to get the outside support for the funding.

Shultis: Another sort of project that came up, before I forget about it, is: what can we do about program explanation, or doing code archaeology? For purposes of maintenance, can we do program explanation in a way that would take some of the abstractions one might build on something like the

KIDS system, say, and do some analogical or metaphorical reasoning, and help people to understand what is going on in a program in terms of collateral reasoning domains and the epistemology of computer programming that Tim [Standish] was talking about? Those are the sorts of projects that I see people talking about. Another is Sandra [Carberry]'s cooperative dialogue systems for certain kinds of applications. Maybe there are other applications than advising students where there is a serious demand for that, and a serious challenge that would bring out some of the issues that we've been talking about. I don't mean to be saying anything against your system, Sandra, but I think you would agree that advising students in a college is just a small-scale application.

Carberry: Yes, but you can use it for any kind of consultation dialogue. For instance, the IRS wants to build a taxpayer's assistant.

Shultis: Exactly. But when you start talking to the IRS about doing tax consultation, then the legal issues become so serious that you can't do it by half measures. You've got to get it right. There are legal consequences.

Standish: Even though in the current system your chance of getting a wrong answer is 33%.

Shultis: But that doesn't matter, because if you get it wrong because the computer screwed up, then there are real liability problems, whereas people are expected, and allowed, to make mistakes.

Fisher: I don't think there is any legal difference. The principle is that you must do at least as well as accepted practice in the given engineering discipline.

Shultis: No, there's a serious legal issue there, and I'll describe it in another situation altogether. A friend of mine was designing building analysis tools for doing passive solar designs for office buildings and things like that. The company he was working for collapsed, because they couldn't sell it to anybody. The engineers said, "If we use this stuff, and the building fails, we're liable. If we follow standard, approved, and accepted engineering practices, it's OK, we can blow it. As long as we followed the right practices we're not legally liable."

D'Ambrosio: I'll give you, perhaps, a counterexample. There's a system called Pathfinder that's been developed to aid pathologists in identifying pathologies in tissue samples, and it's being backed by the American Pathological Association, or whatever that group is, and the claim by the developers is: they're not so much concerned about the suits for the system malfunctioning, as they're looking forward to the first suit against a pathologist for *not* using the system.

Shultis: Well, there are those kinds of issues, and whichever way it goes I don't really care. But the reason I'm bringing up these projects is so that we can start asking: what is it about these kinds of tasks that requires or would benefit from informal processing, or whatever you want to call it—pragmatic computing, maybe grounded robotics.

D'Ambrosio: I'd like to just raise one other point, that the long-term

task, potentially, is one that requires grounding, and the medium- and short-term aren't. I think a potentially valuable place for informalism is in systems that clearly require grounding, simply because formal systems have been such failures in areas of dealing with real sense information. I'll turn to Steve: do you have any recommendations for medium- or short-term useful products, if you will?

Harnad: What?

D'Ambrosio: Well, products in the sense of engineering design aids, or program explanation, or... prototypes of products?

Harnad: I've neglected to say that I'm just a cognitive modeler. I don't think I'll ever make, or have anything to do with, a product. I have projects, and I consider the general project of pattern recognition and categorization to be one that's burning for a breakthrough. All I have done is to look at a few generic approaches—I mean, Minsky almost did in this field; it's time to revive it.

Green: Incidentally, he tried very hard to do in the whole field of logic and theorem proving as well.

Harris: What did he do?

Green: Oh, mostly tirades. Ridicules.

Littman: I wanted to make a suggestion; I'm trying to generalize. These are specific projects, but what if we state the long-term goal to be the production of knowledge-acquiring autonomous agents which use informal reasoning in order to achieve their goals. Medium-range projects would be something like knowledge-based support tools that support tasks that require informal reasoning. Short-term projects might be to hit particular domains, and identify the representations for informal knowledge, the informal reasoning strategies, and so on, so that it would be cumulative. The point is that we should try to build in the idea of reasoning informally at each level. The philosophical issues about grounding, and so on, are an orthogonal dimension to that.

Shultis: What you're suggesting, it sounds like, is, as a first stage, something more along the lines of trying to make a program simply of identifying, cataloging, and characterizing...

Littman: What is informal reasoning?

Shultis: ... the limits of formality. In trying to clarify some of these things we've been talking a lot about one sort of issue, which is the construal of the behavior of the system, or the interpretation of it. Steve [Harnad]'s question yesterday: is a neural net, interpreted as a cognitive system, informal? And there's that point of interpretation, that if you interpret it as just a bunch of hardware computing squashing functions, doing backprop, and things like that, then of course it's just a bunch of code in a simulator, and of course it's a formal system, if it's treated that way. It's just doing a bunch of computation. But if you think of it in terms of what it is doing cognitively, what it's doing may be inferencing, reasoning, categorizing, and that is not necessarily formally describable.

Harnad: Could I make a comment on Bruce's question? It took me by surprise; I've never been asked to propose a product before. There is definitely a split in the agenda, and there always has been, between cognitive modeling, which is trying to figure out what goes on in organisms' heads, and modeling it, and making useful products. Right? I mean, sometimes there's a spinoff—it can happen. The cautions I've been urging are *anything* but philosophical; they are empirical and methodological questions...

Littman: Issues of grounding?

Harnad: Yeah.

Littman: ??

Harnad: This is not a philosophical question: what should you do?

Littman: I agree with that—it's an implementation issue...

Harnad: I'm inclined to say it's not a philosophical point, either, in that for cognitive modelers, unlike for artificial intelligence researchers and creators of products, there's an entry point problem, which is: there's this big cognitive domain of which the human mind is capable. Is it modular? Can you just pick out a little piece of it and handle it on its own terms? There are reasons to believe that even a big chunk like syntax is not an autonomous module that can be done on its own terms. I don't know how it affects on-the-shelf products, but as far as cognitive modeling is concerned it's a big empirical and methodological question whether you've carved out a natural piece when you've decided to do one of these short-range projects.

Littman: That's why I think that the only way I can understand this issue of informality, or informalism, is to look at it in terms of the kinds of activity that people do and which we typically say is informal reasoning. I'm just trying to understand, at the first step, what those activities are. I gave a couple of examples in my talk, and I think Tim did as well. The short term I see as partly interacting with the PR issue; you've got to tell people what you are doing, and why you are doing it. Now as to the question about whether you have to solve all the problems in cognitive modeling to pick out a piece that's interesting: I think syntax is a really stupid piece to pick out. I never would have picked that out. I might have picked out representations that are used in developing programs that solve the following kinds of functionality because there's some sort of fairly natural traceback to the kinds of reasoning people do when they're trying to solve these problems naturally. But again, I'm just being stupid about this, and trying to avoid philosophical problems, and maintaining this view... If you prove to me that I can't have a knowledge-acquiring autonomous agent that doesn't have a grounded symbol system, I would be very happy for you to prove that...

Harnad: ?? acquiring what?

Littman: This thing that's walking around like a kid. Walks around this room and it learns things, all that kind of stuff. I would probably be willing to believe that. I would be very interested if you could show me that I couldn't have a knowledge-based support tool that supports the informal reasoning that people do, without having a grounded symbol system.

Harnad: I don't expect to have anything to say about that. I mean, who knows what it takes to develop a good tool to help people? But to the extent that having a good tool depends on doing it in a people-like way...

Littman: That's the issue.

Harnad: What I would say would be that some of the short- and medium-term projects seem to me like they're hanging from skyhooks from the grounded point of view.

Littman: Well, it seems to me that a question is: What are the grounding issues? And if we agree that informalism requires grounding, again, I can't understand that very well; maybe I'm just being stupid. And that's one track of issues I think are very important. The other track of issues is: the kind of reasoning people engage in when they solve problems, using informal reasoning. The implementation of systems that support that may or may not require grounding.

Harnad: I agree.

Littman: Now, there's the further question of: if you have a system that helps a person in exactly the same way that a human consultant helps him: does that system then require grounding? I guess the answer to that would have to be 'yes', because presumably it would be having the same cognitive activity as the human consultant. But those are two different questions. I think it would be useful to distinguish the kinds of applications we're after from the development of the theory.

Lethbridge: As I see it, there are three kinds of things we can look at in the short term, perhaps. I've put stars by them in my notebook, here. One of them is informal knowledge, which I think we should distinguish from the second one, which is informal reasoning. The third one, which hasn't been spoken much of except in some of the talk about graphics today, is informal interfaces. I think those are three different directions: you can have an informal interface, for instance, with totally formal knowledge and formal reasoning; you can have informality on any of the three levels. I'm actually planning some experiments in which we want to see if users are willing to use a system which has an informal interface. They can specify things informally through the interface, but internally the knowledge becomes more formal. So, I certainly think there are some possibilities there.

Mundie: What is an informal interface?

Lethbridge: What I mean by an informal interface is one where the user doesn't see a formal language inside the thing, and he doesn't see a formal reasoning mechanism inside the thing: he interacts through the human interface, whether it be by buttons, or bits of language, or whatever, but that's only a passing thing in time. Once it gets into the system, it can take any form, whether it be informal or formal.

Mundie: Is X windows a formal interface?

Lethbridge: Is X windows a formal interface? Ummmm.

Mundie: Or Motif?

Lethbridge: I'm not prepared to answer that; I think that's up for

discussion. I would suggest that ...

Shultis: It's an implementation medium.

Lethbridge: Yes, it's an implementation medium. The thing is, I see people using AI systems, using knowledge representation systems, and they get hung up by formalism. We worked for a year in industry with a company, and people would say, "That's too formal. I can't get a grip on it." So we set about trying to take our tool to such a point where they could use it in ultimately an informal way. We turned what was originally a way of specifying complicated frame-based semantics into basically what would be an outline processor. They could manipulate statements just like Microsoft WORD outline processing mode. And that could gradually be transformed as they became more expert into a more formal way of inputting things. But it was a question of usability.

Fisher: Is this what Dave Mundie on the first day was calling non-ugly?

Lethbridge: Yeah, you could say things along those lines. I think this is very important if we're going to have good PR and want to make informalism respectable to the community.

Biermann: By the way, wouldn't you say communication rather than interface?

Lethbridge: It's two-way. I don't think it makes much of a difference whether it's communication or interface or not, but it is certainly two-way. I'm talking about the user to the computer and the computer back to the user.

Shultis: In setting this workshop up, on the first day I think I said that we had a huge list of themes that you could focus on, and we chose language as the opening wedge into things. Certainly you could broaden language to interface issues in general. We settled on language because there seemed to be enough people thinking about that kind of interface to run up against these barriers there. But I think that's a legitimate question, and some of the things that Tim was talking about earlier, these various representations, like simulation, raise the same question. Can you think about direct manipulation environments and so on as interfaces with some grounding to them? Is artificial reality grounded?

Harnad: No!

Lethbridge: I think a distinction that people perhaps don't make is that between languages and internal knowledge representation. I think there should be a distinction made between the language which is used to communicate and the internal representation, which can be considered a language, too: a language of logic, a language of frames, or whatever. But that is a distinct kind of thing from the language used to communicate into it.

Shultis: What I wrote down here [*on the easel*] is that we'd like to identify some problems that we think can use informalism, or exploit some things outside the boundaries of the formal. So we need to identify what those things are. We need to find some problems we can solve with them,

either problems that can be solved that can't be solved otherwise, or that can't be solved at reasonable cost. The cost issue is one of the things that Bruce's work is addressing because the costs there are. I guess your cartoon is the best example of the cost, because if you don't act fast enough, you lose.

Harris: My complaint about this discussion session and the one yesterday is that things are being pitched at too abstract a level. I think it would be useful to summarize more concretely some of the things that were said during the day. One theme that came up today that I thought was sort of glossed over was the issue of overspecification. I have some suggestions for some heuristics for overspecification that come from experiments on humans, experiments in word recognition, that I was flagging my head, saying, "Oh, I'd like to share these with people who are working on overspecification in contexts like lazy evaluation." But this hasn't come up yet, so this is why maybe I'm asking whether some time in the next hour we could return to a more concrete view, bringing up more specific issues.

Standish: That could be a short-term project: the kinds of experiments you could perform to settle certain issues that are easy to imagine, easy to conduct, and easy to get results from in a short-term period.

Shultis: What you're talking about is fleshing out this program of characterizing what it is, and we've had a number of things come up here. Maybe that's a good direction to move at this point. Can we identify some things that we think are outside the boundary of formality, and summarize them? We can return to these other issues as we go along.

Standish: I'd like to hear about Cathy's experiments on word recognition.

Harris: No, just put "overspecification" on the list, and maybe we can return to it.

Shultis: Does everyone want to do this, shall we try to draw up a list of things that are "beyond formality"?

Reeker: There are a number of things that have been mentioned that are evolutionary in nature, where the process of getting at a task is really a process moving toward something and we don't know where it's going to lead.

Littman: If we said we believe that informalism as a computational activity will be particularly useful in problem-solving tasks, for example design tasks, at the very early stages of development, that would be amazing, because then all these people working on requirements analysis, and so on and so forth would tune into it. So, yeah, underspecification, at the very early stages of problem solving, where you don't have formalisms yet, where you're in the process of developing them, for example.

Shultis: Basically, preformal notions that are ambiguous in some ways, and can be fleshed out in a number of different ways.

Reeker: Yeah, or where you don't know if you ever *will* have formalizations.

Littman: Better posing ill-posed problems. I keep having this intuition that informalism comes up when people don't understand stuff.

Standish: Buggy reasoning, muddling through, reasoning in the presence of superstition and bugs.

Littman: When you don't have a formal representation you can use to reason about and communicate to other people, then you are thrown back into using these kinds of heuristics and representations for solving problems, and nobody is working on that, because it's a real hard problem. People work on it, but everybody wants it and nobody can do it.

Shultis: Can you give me one word, or a slogan?

Wight: Organized religion? *[Laughter.]*

Littman: The problem-groping phase. No, that's not good.

Harris: I had a slogan for a certain related idea. It came up during David Fisher's talk this morning: the idea that we can't anticipate all the concepts that will need to be represented, so we must have a system that can do some type of dynamic concept formation.

Shultis: OK. Dynamic concept formation.

Harris: On the spot, as the need arises. Be able to...

Standish: Improvise.

Lethbridge: I'm not entirely convinced that that is necessarily informal, though. You can do that in a formal system, I would have thought.

Harris: OK, fine. The issue, as Fisher was saying, seems to be something that current computer AI technology can't do very well. We need to move towards a way to get a system that can create concepts on the fly.

Shultis: And, in these areas, these labels are going to be problematic, because if I put down things like underspecification, or dynamic concept formation, or creation, or something like that, are you doing anything more than composing old concepts? What is it that characterizes dynamic concept formation that isn't just writing down a definition in terms of old terms?

Harris: Yeah, one of your mechanisms might be composition...

Shultis: Because we already have definitional extensions.

Harris: Well, ask Fisher what he wants, then.

Fisher: Maybe I didn't understand what you were saying, because I thought you just changed your position in the last half sentence, but...

Shultis: What I was saying was that there are ways of doing things like underspecification in formal systems, as well. There are formal analogues to these things.

Fisher: Sure—or maybe there are just formal implementations of these things.

Shultis: So these slogans don't completely capture things, that's all I was saying.

Fisher: The topic I brought up was not underspecification, but rather forced overspecification, as for example in programming languages and almost all other software systems. It is the point that formal systems require completeness and thus specification of irrelevant detail and "don't care" cases. It is why formal systems can only describe and reason about models, never about what is being modeled. Underspecification is an important but

different property, namely, the ability to deal with the absence of *crucial* information...

Shultis: OK, well, I was trying to put up the positive form of that...

Fisher: But they're not the same thing. They're both valid points, I think.

Littman: You asked me for a phrase that captures that, and I think I've got one: representational development processes. People are all screaming about that. Software engineers want it, and that's the thing that I see. It's those processes that lead to some easily sharable representation that seem to carry the weight of informal reasoning.

Schwartz: Well, that sounds like concept formation, something very similar.

Littman: I agree.

Fisher: I never responded to Cathy, of course, on concept formation. Certainly the point I was making was that we cannot anticipate, when we're building the system, all the concepts we're going to have to represent. We can't pre-ground everything; we can't build in a set of primitives and have everything else we're ever going to have be composed from them; instead we must have mechanisms for grounding concepts that occur *after* we've built the system. And I think that drives us in two ways. One is the robotic sort of grounding, but the other one is the linguistic one, and I find it quite acceptable, and quite desirable, to have ungrounded symbols over the linguistic interface. It's not that we're not going to ground them, it's rather that we can do a lot of reasoning about their relationships before we get around to grounding them, so it's kind of a lazy evaluation of the grounding.

Harris: Maybe it's my naïveté, but I don't know anything about the literature on how you can do this, about AI programs that do this dynamic concept formation on the fly, about how they're able to develop a representation that...

?: There's a whole learning literature in that...

Harris: ... so I gather that's not a problem, anymore.

Littman: Oh, yeah, it's a big problem. Are you kidding?

Lethbridge: Big problem. [*Pandemonium.*]

Schwartz: The problem is that, at one level or another, all the attempts, or virtually all the attempts, get down to some primitive representation. Whether they're continuous or discrete, it doesn't really matter. The problem is to convince yourself that you're doing something more than just stringing together a definition, and creating something in terms of some base vocabulary.

Fisher: Just an example on this grounding thing. If I say, "Glitzes are winkel" and then I say "There's a glitz", I can conclude that it's probably winkel. I don't really have to have grounding of those terms yet. Now, there's an assumption, and I think we do this in human conversation, that when we hear words and sentences, we don't have to know their complete interpretation immediately in order to reason and communicate. Interpretations can be delayed until they are needed for reasoning, and then

they can be grounded through more communication.

Littman: Once you point and say, that's a glitz, that's kind of chiouuuuu! éclat, I guess. That's the grounding.

Fisher: Yes, that's the grounding of glitz, but it doesn't say anything about wintzel.

Littman: Well, that's grounding, too, because I can reason about it. I get it for free, I guess.

Shultis: Let me try to summarize these issues, then. What you want is control over the degree of specification.

Littman: How about form *and* degree? If you say form as well as degree, I'd be happier. Degree begs the question of what the representation is going to be like.

Mundie: He said "specification".

Littman: What did I say?

Mundie: "Representation".

Littman: I know. Form. I mean...

Shultis: OK, so there are two issues. One is form, representational form. The other is...

Lethbridge: Degree of specification.

Shultis: Degree of specification.

Littman: Yeah, I'm just being paranoid, because...

Shultis: Maybe one way of putting it is that we want to have more control or freedom over what kinds of representations we use, or what we interpret them as being about, or how they're grounded, or how much we're committed to them, or how much we infuse them with any meaning at all. And I think that Dave's point is that we should be able to have symbols that do float around a bit and only have, if you like, functional roles. In other instances you can fill out their details incrementally, as you go along, as you need them, and you have the ability to fill things out partially, and then backtrack, maybe to change those specifications. Is that it?

Fisher: Yes. One can reason precisely and correctly without complete information or grounding, and that information need not be obtained until it is actually to be used. Humans do this all the time. If reasoning only uses a limited, finite amount of information, then I only need the information crucial to that reasoning, never complete information.

Reeker: So, you're talking about a partial specification...

Standish: Delaying the arrival time until you need it, or something.

Littman: And what's so interesting is that you can do very, very powerful heuristic evaluations of whether you're going in the right direction, based on these extremely ambiguous and underspecified specifications...

Fisher: Because the minute you have any relationships at all, you have a lot of constraints on the system.

Littman: And the more you know about the constraints...

Fisher: Yes.

Shultis: I'm going to let you guys fight it out, but tell me when you're

done, come to a consensus, and you tell me what to write up here.

Lethbridge: We should certainly have something about grounding up there, since it seems to be a topic under discussion.

Shultis: What I'm looking for now I guess is something to replace this line [underspecification].

Fisher: I don't have any trouble with that line.

Shultis: What?

Fisher: I don't have any trouble with that line. I suspect Dave doesn't have any trouble with that line!

Lethbridge: No, neither do I.

Littman: No. Again, I say I was being paranoid. This is personal, but the thing that I keep coming back to is that human beings in general, but especially we, here, strive to develop representations of situations that we don't understand very well. And I think that is one of the key things that we use informalism and informal reasoning for. And so I take it one of the things that we try to get is representations for things.

Shultis: OK, let's try to focus on something a little bit. We've talked about things like informal representations. What's an example, in your mind, of an informal representation?

Littman: Well, in the programming domain there are a lot of things that won't run on a computer, but which we can use to communicate with other people about what we want a program to do. They come from a lot of places. I think Tim talked about several of those places this morning. For me, that's a reasonably, pardon me, prototypical example of informal representations. The kinds of reasoning strategies that you use on them, I think, are different from the ones you use when you have a more formal representation. When, for example, you actually start proving loop invariants, you forget about the meaning and just start treating the symbols. So, an example of a set of informal representations would be the kinds of descriptions of the activity of a program that you find people using when they're trying to develop some complex program.

Schwartz: What would they look like? Are they natural language, or...

Standish: Take the blocks world, at the low level of the Piagetian thing that was being talked about by Larry Reeker yesterday. You have a collection of blocks, and they're different heights, and you tell a kid, "pick out the biggest one, and then the next biggest, and the next biggest, and arrange them in a decreasing order column". And they can usually do that if they can discriminate between the sizes, at a certain age, when they can do the discrimination, and take the goal, and understand what the goal is. It's pretty much muscular: you're not giving them a terribly formal spec, and that really is prototypical for priority queue sorting, or selection sorting, or heap sorting. You can take that basic template, and throw it into...

Schwartz: You refer to it as a template, but I'm curious to know how you specify it other than just calling it object number 49, or template number

49. How do you specify it in a way that the computer can make some use of it?

Standish: First you asked me what was an example of an informal thing, now you're asking me to give you a formalization of it. I could do that, but I'm not sure that...

Schwartz: No, I'm not saying that... well, maybe that is what I'm getting at. It seems to me there's an oxymoron floating around...

Standish: I tend to think that preformal things are stimuli to develop the formal things, but that they can both exist, and one can lead to the other. I think that's what Dave was getting at...

Schwartz: It just seems uninformative to me to posit an object. I agree with your intuition that there are such things, but the question is...*[Recording gap.]*

Reeker: ... using that representation within the computer and maybe having the computer operate with that representation at some other level that helps you work with it. Maybe the interface would be a good way to think of it, in the sense that you can use interfaces with a lot of graphics and stuff, and the graphics can help you to conceptualize problems of one sort or another, which can then, once that conceptualization is put into the computer, be used to create a program or some formal object. So the interface representation means something to you that it doesn't mean to the computer at all.

Schwartz: OK, well, that's a key aspect.

Reeker: Maybe it isn't non-formal, but it's formalized in a different way in the computer; it's a different formal system in the computer than it is to you.

Wight: That seems allegorical. It sounds as though we're talking about programming by allegory. Tell a story, then flesh in the meanings, bit by bit.

Zalta: What category does that fall under? Filling in the missing elements of a story? It seems to be a general task that is performed all the time, and which seems to be a classic case of informal reasoning. Which one of these categories, so far, does it fall in? It's like the story Tim gave us of a guy who needed some money, and he robbed a bank—bought a gun and robbed a bank—and then he asked the question, why did the teller give him the money?

?: Schema completion.

Schwartz: In AI, it's referred to as analogical reasoning.

Lethbridge: Exactly.

Littman: No! No, no, no, it's exactly what Steven [Wight] said. What you've got is an underspecified representation of what happened, and you try to figure out what probably happened, so you're building a more complete representation.

Standish: It's plausible inferencing.

Littman: It's plausible inferencing.

Shultis: Or, interpolation?

Standish: Script-based interpolation, yeah.

Zalta: ... interpolation. That seems to be the task that you perform. So that would be the informal reasoning task... *[Everyone talks at once.]*

Harnad: I suggest you call it "pattern completion".

All: Yeah. Sure.

Wight: But we're not really talking about pattern completion initially, we're talking about generating the pattern with the gaps in it, and manipulating that before we interpret it. Stories can be arbitrary, and can be of arbitrary kinds, so I don't see why it's a pattern. Stories can have arbitrary subject matter. There's no pattern that's being completed there, it's just a question...

Littman: It's not a *single* pattern. I think that this is a semantic problem, perhaps. Stories, it seems reasonable to imagine, are composed of many patterns with interconnections between them, and the way you fill them in to figure out that he pointed the gun at the clerk in Al's store at Fifth and Vine is to make the inference that he had the gun in his hand and so on and so on...

Carberry: It is underspecified, though.

Shultis: Pattern completion suggests something that is a specific kind of interpolation. Is that fair?

Littman: Yeah, that's interesting. Pattern completion might be a prototypical example of interpolation. Everybody will understand if you try to talk about it, though they may disagree that that's what happens.

Shultis: And let me suggest a couple of other things, since we're here. Along with interpolation, some other things that come to mind are approximation and projection.

Littman: Approximation and what?

Shultis: Approximation and projection.

Harnad: What's projection?

Mundie: Extrapolation.

Shultis: Extrapolation, if you like.

Littman: Ah! So, what is he going to do with the money? Probably buy drugs.

Kozma: What about the converse, where you in an informal system dealing with inconsistency? In trying to throw away information, there...

Standish: Exactly. We took some protocols of people developing programs and the really good ones first simplified the problem to something that was ridiculously oversimplified by stripping away information, and making unwarranted simplifying assumptions. That controlled the problem complexity. Then they were really good risk-takers. They would take risks only where there was very little risk to be exposed to. And very carefully controlled the complexity of what they were reasoning about, got the crystalline core of the solution, then went back and added the extra cases and the conditions to fill it out to the whole thing, sort of expanding the core. So

that information stripping was one of the first things they did. Simplifying; deliberate simplification; deliberate *oversimplification*; irrationally stripping away to core of the problem.

Shultis: So what you're talking about there is approximate representation and reasoning.

Standish: It's deliberately buggy reasoning, solving a non-problem that isn't the same as the real problem in order to get a core solution which then can be expanded to the whole...

Fisher: But it's not deliberately *buggy*, it's deliberately *simplified*.

Standish: Fine.

Littman: A bad representation is better than no representation at all.

Shultis: But the reason you do that, and this is the first time I've heard of a utility for any of this stuff...

Several: OK, yeah, agreed

Shultis: ... is that you do it in order to control...

Standish: ... the mental complexity of the search space.

Carberry: You're doing a kind of inexact reasoning. It's inexact reasoning, and it's like something somebody was talking about the other day, I think it was Dave Fisher: moving to different models, where you might try a number of different models in trying to solve the problem, and some might lead you down a path that wasn't successful.

Fisher: If you underspecify you can afford to try different representations. If your reasoning does not assume completeness, it need never be buggy.

Carberry: Well, it's inexact reasoning, though, because if it were exact, you would know that it was going to work out.

Fisher: No. It may be incomplete, but it is exact.

Standish: Then there's even the kind Alan Biermann was talking about that is incorrect without the person knowing it is incorrect. We had an interesting example of that. There's that anecdote where Hardy went to see Ramanujan when Ramanujan was on his deathbed, and Hardy couldn't think of anything to say, so he recited the number of the Taxi cab license, which I think was 1739 or something, and asked, "Do you find anything interesting about that number?"

?: No, he said it was a boring number.

Standish: Boring number, and Ramanujan said, "Oh, no, that's the smallest number that's the sum of two cubes in at least two different ways!" [Laughter.] So, the problem then was to write a program to find this, and the guy who did it thought that the sequence in which he was doing it, which was to go line by line in the subdiagonal matrix generated monotonically increasing values. That was false; it's a sawtooth function—as you go to the end of the row it drops as you come to the next line. But it doesn't matter, because the next one beyond that is, um..., so the assumption of monotonicity was wrong, but it still leads to the right conclusion. You need a different proof, one that fixes the bug. But anyway, it illustrates the point that

sometimes you use completely fallacious reasoning to get to the solution that later you have to go back and say, "Oh, that's the right solution, but I need a valid justification".

Wight: This might not be directly related, but these things all fit into, in my mind, the science, or pseudoscience, of the creation of operating system shells. Basically, we're talking about the shell around the computational core of this theoretical computer application. I can fit all these into shell attributes, categorically. I don't know if that's relevant, at all...

??: Could you...

Wight: It might just be wrong. Maybe I'm thinking too much in terms of interface questions, but we are attempting to describe a flexible, dynamic interface...

Rogers: But I thought at this stage what was being discussed was the informal reasoning which humans and designers go through, so maybe we should say something like "construction and manipulation of mental models" You are exploring all these different possible models, but not in an exhaustive search...

Standish: That's kind of neat. In algebra there is word problem solving, which seems to bring forth the use of mental models, and even their explicit representations on paper. An example is: two trains are heading toward the same railroad station, one going 80 kilometres per hour and the other going 40, and they cross at the railroad station and head in opposite directions. How long will it be before they are 480 kilometers apart? It's remarkable. Today's college students will often times draw them heading in the same direction, or draw them in different directions and subtract the velocities rather than add them. And there are all sorts of bad mappings of the words of that problem onto diagrams and diagrams into equations. We're not sure why the new generation of birdbrains is doing this wrong, but it's certainly a prevalent phenomenon. *[Laughter.]*

Littman: I just wrote down three things. One, shifting grain size. That seems to me to be something that has to do with informalism. You think of your problem at different levels of, I don't even know how to say it, maybe complexity.

Lethbridge: Granularity.

Littman: A second one is shifting focus. That led me to the third point: a lot of informal reasoning goes on in the domain of the problem, where you're reasoning about the causality in the domain and the issues of agency in the domain. A lot of the reasoning you do has the goal of trying to make some kind of abstraction about the structure of some given behavior. I think that that would count as informal reasoning. Insofar as a problem-solver reasons about the processes in the domain with the goal of developing a more formal representation of them, that would be informal reasoning. So, shifting grain size, shifting focus, and reasoning in the domain might be three attributes of informal things.

Reeker: That's something I've never been able to see any way to do

formally.

Littman: Are you talking about the latter, or the first two?

Reeker: Shifting grain size. I referred to that yesterday. There was a little memorandum that Kurt Gödel wrote. When people asked him about machines thinking, and whether he thought that Gödel's theorem precluded it, he said no, he didn't think it did, but that in fact there might be one phenomenon that he saw, one thing that humans seem to be able to do but he didn't see how machines could do it, and that was that people can keep coming up with sharper and sharper axiom systems, in other words they make finer and finer distinctions. This is in fact one of the main attributes, I've always felt, of high intelligence, as opposed to mediocre intelligence: coming up with finer and finer distinctions.

Harnad: Until you get the geniuses, who just split hairs.

Shultis: Grain size of what? Would it be fair to say that you take these representations and these reasoning rules that are rough-hewn, crude, mechanisms, and say, "Well, I think XYZ because of W, and that's typically true", and you correct that later, or...

Reeker: I'm thinking of systems based on constructs, particular constructs, and so your constructs are all set up there, and you can do all sorts of manipulations on them, and then someone comes along and says, "Yeah, but..." Those constructs bring us very close to concept formation. Then you form new concepts by splitting the old concepts up, and then you can create a new formal system which works with these new concepts, and you create different ideas, basically. Maybe this is a sort of paradigm shift...

Fisher: It goes the other way, too.

Mundie: Yes, I'm a bad piano player because I think in terms of the individual notes, rather than in terms of higher-level chunks like chords and harmonies. The same with my chess playing—I think in terms of the individual moves of the pieces, rather than in larger patterns.

Littman: Yeah. You were asking me for another example of shifting grain size. Here is one from social psychology. You're trying to explain why families do what they do. Psychologists used to say, "This person says this to that person": there were very small, dyadic explanations. Then it got to be, "Well, it's sort of this configuration, so there are three or four people", and then it got to be, "Well, it's sort of the family", and then the extended family. Today the good models of social process I'm familiar with in the area of dysfunctional families take into account the social system that they're in, the whole social service system—what the schools are, and the whole thing, so that understanding a family and why it does what it does requires reasoning on a number of different levels of granularity about the things that impinge on that family. The idea is that you can shift very fluidly across these different construct levels which take into account larger and larger but perhaps weaker and weaker effects. Again, that seems to be a fairly informal activity.

Schwartz: I don't know that that example is really an example of granularity. It seems to me more one of going from a local reasoning process to a more global one. What I think of when I think of shift of granularity is some of the work that was done in hierarchical planning. Say you want a plan for going to Russia. It would be very complicated if you actually had to give a specification of all the actions you were going to take. Say you have a vocabulary of actions you can take, and you order their preconditions so that some number are very important and some you can ignore for the moment. If you're going to take a flight, then you take the availability of a ticket to be very important, but the way to get to the airport is somewhat less important. Then you can make one pass through the plan by only looking at these very important facets of the action you're going to take, and then you successively narrow down to include more and more details.

Shultis: So there's an aspect of holoprasting there, of ignoring certain detail. But I wonder if all the things that are being talked about here are examples of granularity. What you're really trying to do is come in on a situation and impose some kind of conceptual organization on it, and it may be one that is larger grained, or smaller grained. You view a family as a collection of individuals, or you view it as a unit in a community, and you bring in certain theoretical terms, or abstractions, if you like, that you use to describe it. You fit the situation into the description. Sometimes you *force* it in, use it as a procrustean conceptualization. That ability to take the material you're talking about, and fit it into a number of different conceptual frameworks, for different purposes of reasoning, is crucial. What we tend to do now with programming is to build hierarchical structures, and they're *fixed*. Once you box something up and put it in a package and define the interfaces, that's the conceptual organization. Whereas here you want is to tear all those pieces apart and reorganize them and put them in a different conceptual organization. Is that fair? Is what we're talking about having an ability to reconceptualize somehow?

Standish: Cognitive transmutation! [*Laughter, assent.*]

Shultis: I'll write it up here if everybody likes it! [*Dissent.*]

Littman: Yeah, it's the Gestalt idea, I guess, being able to see things in terms of different perspectives. But perspectives isn't quite the right idea—it's being able to see things in terms of different wholes.

Shultis: Schematization.

Harris: Levels of schematization.

Littman: It's not just level of, but it's level and kind. So it's which schema, and how much detail you're bringing into the schema.

Shultis: So, flexible schematicity, or schema flexibility? What do we want to throw on this list?

Littman: What Cathy said is really neat. You might pick a piece of one schema which, if you went to a lower level of specificity wouldn't work at all, but that's OK, because the reason that you're using it is that there's some kind of metaphor in it at a very high level of generality that you find useful.

So, which one and how much specificity.

Wight: Meta-metaphor?

Shultis: What do I call this?

Biermann: Why are you avoiding the word "granularity"? Why do you need so many other words to avoid this one?

Shultis: The only reason that I didn't write up "granularity" is that I wasn't sure that it really captured all that was going into it. There was a lot being loaded into that term, and I want terms whose meaning we'll remember tomorrow.

Littman: The thing that has come out of this discussion, for me, anyway, is: which grains, and how big they are. Before we just had how big they were. But now this idea of schematicity brings in "Which grains?", and then "How far down do we specify?" brings in the size.

Reeker: I'm glad schematicity means that to you... *[Mayhem.]*

Fisher: I want to speak in favor of granularity, too, because it does convey an important idea that has not been stated explicitly. Granularity not only implies levels, but also that at each level you must treat a different set of entities as atomic. It is only by viewing and processing entities at higher levels as atomic rather than as compositions of lower-level entities that we can avoid multiplying the processing costs as we move to higher levels.

Littman: And the rules of inference that you get at each level are of the same order of complexity, and you don't have to, say, unpack everything all the way down...

Reeker: That influences the schema that I might apply.

Shultis: The thing that I want to get away from, though, and the thing I'm afraid of, with "granularity" (let me express my discomfort with the term) is that it lends itself to an interpretation which is: hierarchical.

Many: No, it doesn't.

Harris: And what's wrong with hierarchical?

Littman: Well, it's a commitment we may not want to make. There are people who claim that the knowledge you've got in your head is hierarchical, and the problem is, they're probably wrong.

Harris: *[Impatiently.]* I think we should get our ideas out there.

Fisher: Yes.

Shultis: OK, that's fine, that's fine, that's fine.

Standish: Granulophobia. *[Laughter.]*

Shultis: OK, I'll write granularity up here. Bruce, you've had your hand up for a while.

D'Ambrosio: Yes, I want to raise a concern about what this endeavor is. It seems to me that the traditional AI community would very very happily endorse most of these points as wonderful problems to be working on. Granularity, for example. I know that Forbus has recently directed a great deal of attention to the theory-selection problem, both in terms of the size at which you should discretize the domain, and the theory that you

should apply (thermodynamics or whatever). I want to question whether, methodologically, what we should be doing is focusing on mechanisms or focusing on tasks. And perhaps a main contribution informalism can make is being extremely faithful to the task domain, and not saying, "Well, we're going to carve it up into this set of mechanisms and somehow force the domain into these mechanisms."

Standish: OK, interesting. What are examples of the things you think might satisfy that desideratum? Or, at least one?

D'Ambrosio: I'm not exactly sure where to go from there, other than, what it points to is an agenda somewhat like the agenda I understand, from a distance, the ethnomethodologists are taking in human-computer interaction. The work at Xerox PARC, for example, which would involve a very careful examination of the actual human performance in the domain, without preconceptions about the mechanisms that are supporting it.

Standish: OK, behavioral investigations.

Littman: Can you get a Ph. D. in that?

Harnad: How about "thick reasoning"?

?: What kind of reasoning?

Harnad: "Thick". That's the anthropologists' term for immersing yourself in the local problem. Local reasoning.

Lethbridge: Explain.

Standish: Like, hey, man, let's go native?

Harnad: No, no, no. You come into a culture, and you immerse yourself in the myths and symbols of that culture, and start to spin out an explanation in *those* terms, rather than in some larger terms in which you try to invent those terms. [*General muttering.*]

I'm trying to capture what it is that Bruce meant, and I assume that if you go to a problem and you start thinking in terms of the particulars of the problem, rather than in general formal principles, you're doing something like thick explanation...

Littman: This morning I suggested that in informal reasoning questions are looked at in a very domain-specific way: what are the representations that are used in the domain, which ones are particularly useful for informal reasoning, what are the rules of composition, and so on and so forth, exactly in the spirit of what you're talking about.

Rogers: They call it situated activity, so why not situated reasoning?

D'Ambrosio: Unfortunately, I'm not sure that's something we can do as a group, without a particular task to focus on, so I'm not sure what that leaves for us to do right here.

Shultis: Well, there are two sorts of things. I would put your suggestion more at the programmatic level: what are the sorts of tasks that we need to do?

D'Ambrosio: Right.

Shultis: Can you give me a term? What phrase do I use?

D'Ambrosio: Steve suggested "thick"...

Harnad: "Thick description". I think "thick description" is what they call it.

Lethbridge: Situated reasoning.

??: Too opaque. But I don't have a better one.

D'Ambrosio: Task focus. As opposed to method focus. Or mechanism focus.

Shultis: So your suggestion is...

D'Ambrosio: ...a methodological suggestion, that the focus of informalism should be extremely task-directed, instead of mechanism-directed, instead of focused on generic methods.

Shultis: So you're suggesting that there are specific tasks, or problems, that we can enumerate...

D'Ambrosio: Tasks of the kind we were identifying earlier: an engineering assistant, or program explanation, or whatever.

Shultis: OK, things like that where we think that informal methods have a part to play, in performing those tasks, in some way.

D'Ambrosio: Right, and we should come without previous commitment as to what kinds of mechanisms must be used there.

Shultis: Sure. Because I think your point is well taken, that in the process of trying to perform those tasks, you start to reveal what really is important there, and what the essential ideas are. It is currently ten to six, and we were supposed to break up five minutes ago, but I'm not going to cut the discussion off entirely...

Littman: Let me just say something. I think it integrates what Bruce was saying about this, and it's the thing he said about how most people in AI would endorse this. In AI we tend to work on one of those methods, and not a bunch of them. My suggestion would be that when we look at these domain-specific problems, the thing we might focus on is a control structure that governs the integration of all of these different kinds of reasoning to solve a problem.

Lethbridge: Exactly. Yes, precisely.

Fisher: I don't buy into the task view. If I ask, "How would we distinguish ourselves from AI?", I'm hard pressed to believe that we want to say, "We're going to solve problems in this application area, and they're not", or vice-versa. I believe the differences will be in the mechanisms. And I don't think it's a matter of our using a specific mechanism, but rather there must be characteristics of mechanisms that we are anticipating and the AI community is not. Certainly a lot of the distinctions that I see—this list, in fact [*Pointing*—enumerates things that I think would not generally be used in the AI community.

Littman: They would, but I think they would try to solve the whole problem using only one or maybe two of the mechanisms, so it's the issue of the control structure that, in part, governs the integration...

Fisher: You just reminded me of a point I was going to bring up earlier. That is, one thing I wouldn't have thought of before I came to this

workshop, but something I've seen among so many of the speakers here, has been this issue of incrementality. And I don't mean incrementality just in the interface, I mean in the whole philosophy of the way you attack these problems. Everything we're talking about is incremental. Larry just a few minutes ago was talking about something I view as very incremental. Certainly I was advocating incrementality—I see that from Steve's talk today. It just strikes me that there is an incrementality in the approach that I don't see in traditional AI. I think it comes about because of this view of incompleteness that we think is inherent, and you can't deal with incompleteness without incrementality.

Littman: In fact, unpacking that, what is it to solve a problem incrementally? You were talking about this earlier,...

Wight: Are you guys setting us up?

Lethbridge: Incremental systems!

Mundie: Yeah, yeah! *[Laughter.]*

Reeker: There is another thing I thought I'd just mention that relates to incrementality. I was thinking of Alan Perlis in his last years. He would talk about organisms, what he called organisms, and his idea was that any time you have a system, algorithmic or otherwise, that is beyond a certain size or complexity, then it tends to take on a life of its own and tends to evolve. He felt that it was sort of inevitable that these systems were going to evolve, all the time, and for that reason he didn't see that proving invariants on a system was that useful, because the next thing you know it changes *[Laughter.]* and you have to keep just keeping up with it. So you have these systems which inherently—I mean, this is a sense in which they may inherently be beyond formality—they just keep moving on you. What's formalized at one point has to be changed at another point. So you sort of have to take a leap forward.

Green: It seems like the algebraic notion of parameterized theories is a formal notion that informal people should probably take a look at, as a stepping-off point, just so you don't recreate it. It seems like the coming trend: I see it in the way software knowledge is getting organized. The CPL (Common Prototyping Language) effort is more or less standardizing on it, the British are standardizing on it, and I see it emerging as a standard way to organize your knowledge to deal with some of these problems.

Reeker: Some of what David was saying this morning sounded a lot like that, too.

Green: Unfortunately I don't know of a good reference for it, though.

Lethbridge: One point I haven't seen on the list there, which I think perhaps deserves to be there, is adaptive representation. What I mean is that you pick the kind of representation, the kind of language which is suitable for this particular aspect of the problem, and for another aspect of the problem you pick a different kind of representation. Totally different syntax, totally different way of representing the things.

Harris: That's very nice. I agree with that, too.

Lethbridge: I think that's pretty important, and I don't see that it fits into any of those categories.

Reeker: At some points it might be a pictorial representation, some points might be a linear representation...

Lethbridge: Exactly. Some parts could be natural language, other parts some logical formalism,...

Reeker: We saw some examples of that about fifteen minutes ago.

Harris: The system knows how to pick the right representation for the task.

Lethbridge: Or the user decides, and the system can handle whichever you put in.

Carberry: Or the system changes its representation as it goes along.

Lethbridge: Sure, exactly.

Littman: So it's heterogeneous, too.

Lethbridge: Heterogeneous and adaptive, both.

Littman: Which are separate.

Lethbridge: Yeah, they are separate—you're right.

Shultis: I think that the intuition I have gotten from this discussion is very closely related to that of parameterized theories. The notion of parameterized theories entails adaptation of a general schema or conceptual framework to a particular situation, and there's a notion of instantiating something, or adapting it or unifying it with some of the information that you have, in some way.

Green: Yeah, and there's ways to match it to the problem, to fit it to the problem. And it allows pretty ornate mappings between theories and problems.

Fisher: I guess I would take adaptability to be a more generic term than parameterized theories. In particular there need not be any preconceived notion of which aspects constitute parameters.

Littman: The idea behind adaptability is more *mutability*.

Shultis: Well, I guess we should stop here...

Chorus: Heterogeneity! Add heterogeneity to the list!

Shultis: I'll do that.

Mathematical Modeling
of
Digital Systems

Donald I. Good

Computational Logic, Inc.
1717 West South #290
Austin, Tx 78703

512-322-9951

good@cli.com

Some Personal Interests
Friday Presentations 155

No: Make machines behave more
like us.

Yes: Help digital systems* engineers
do their jobs better.

* "Digital systems" encompasses both
hardware and software.

Engineering

Classic Def: "Engineering is the art
of directing the great sources of power
in nature for the use and convenience
of man." - Thomas Tredgold, 1828.

Engineer use physical materials to
build products, under time and cost
constraints, for human consumption.

"Engineering is doing for one dollar
what any darn fool can do for two."

S. Florman. The Civilized Engineer. 1987.
St. Martin's Press.

Doing Digital System Engineering Better

- Improve Product Quality
(includes safety)
- Reduce Product and Engineering
Cost
- Reduce Time to Market

The Role of Models in Engineering*

Models predict the behavior of a system without physically building or operating it

Benefits: early error detection

- Saves time
- Saves money
- Saves operational disruption
- Saves operational mishaps

Risks: model misrepresents the system

- Inaccurate
- Incomplete

Kinds of Models: physical, analog, schematic, mathematical

Blanchard + Fabrycky, Systems Engineering and Analysis, Prentice Hall, 1990

*There is much more to engineering than modeling!

Why Mathematical Models?

Can provide very accurate, precise descriptions of physical systems.

Can describe a very large (even infinite) number of system states with a very small number of symbols.

Can provide low cost system simulation by manipulation the symbols that describe it.

Less costly to build and less risky to operate (simulate) than the physical system.

A Classic Model

Near Earth Free Fall:

$$d = 16 \cdot t^2 \quad \begin{array}{l} d \text{ in feet} \\ t \text{ in seconds} \end{array}$$

-- due to Copernicus, Galileo, Kepler, Newton, others...

Simulate some specific cases:

$$\begin{array}{rcl} \text{For } t = .2, & d = 16 \cdot (.2)^2 & \begin{array}{r} .2 \\ \times .2 \\ \hline .04 \end{array} \\ & & \begin{array}{r} 16 \\ \times .04 \\ \hline .64 \end{array} \\ & = 16 \cdot (.04) & \\ & = .64 & \end{array}$$

.....

d	t
.64	.2
2.56	.4
5.76	.6
10.24	.8
16.00	1.0

Why does this symbol manipulation describe the physical world so well?

Accuracy Preserving Transformations

Distance Model $d = 16 \cdot t^2$

Theorem: If $d = 16 \cdot t^2$, then $t = \sqrt{\frac{d}{16}}$

Proof: $d = 16 \cdot t^2$

$$\frac{d}{16} = t^2$$

Sound deduction

$$\sqrt{\frac{d}{16}} = t$$

preserves accuracy.

$$t = \sqrt{\frac{d}{16}}$$

QED.

Time Model $t = \sqrt{\frac{d}{16}}$

Simulate time from distance

For $d = 6$, $t = \sqrt{\frac{6}{16}}$

$$= \sqrt{.375}$$

$$\begin{array}{r} 375 \\ 16 \overline{) 6.000} \\ \underline{48} \\ 120 \\ \underline{112} \\ 80 \\ \underline{80} \\ 0 \end{array}$$

Validating the Accuracy of a Model

- Show that the model follows by sound mathematical deduction from another model with known accuracy.
- Test the model against measured observations of the physical system.
 - state a model
 - test it
 - adjust it
- One cannot construct a mathematical proof that a model is an accurate representation of a physical system

Friday Presentations 157 Hardware Model Observables

A hardware system
is composed
of physical switches.

Nancy Stern. From ENIAC to UNIVAC: An Appraisal of the Eckert-Mauchly Computers. Digital Equipment Corporation, 1981.

Next page.



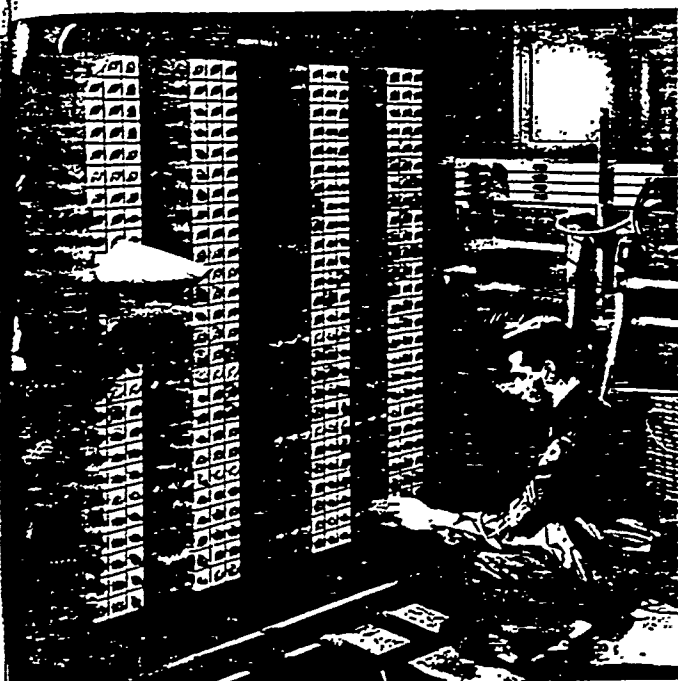
Copyright © 1991 Computational Logic Inc.

note-109.msc /
10

20 Feb 91

9

ENIAC RESEARCH AND DEVELOPMENT 35



The switches of Function Table A are being set by Mr. Thomas M. Smith and Ed. from a Catalogue. Three manuals will function tables are set in the room where Gordon M. is a member of Electrical Engineering, University of Pennsylvania.

11

Physical Law of Computation.

Use a physical interpretation of a variant of Church's Thesis as follows:

Recursive functions are a very accurate model of the physical, computational behavior of a digital system.

12

Use Discrete Mathematics to Model Hardware

- Switches by binary digits
- Operation by recursive functions

s0 | 0 1 1 0 0 0 0 1 1 1 1 |

s1 | 1 0 1 0 0 1 1 0 0 0 0 |

s2 | 1 1 1 0 0 0 1 0 1 0 1 |

o o o



An MC68020 Machine Model

```
MC68020(s,n) =
  if haltp(s) or n=0
  then s
  else MC68020(NEXT(s), n-1)

NEXT(s) =
  if evenp(pc(s))
  then if pc_readp(mem(s),pc(s))
        then EXECUTE(FETCH(pc(s),s),
                      update_pc(s,...))
        else halt(s, pc_signal)
  else halt(s, pc_odd_signal)
```

```
EXECUTE(ins,s) =
... [50 pages for 90% user ins.] ...
```

Provides a mathematically precise and consistent machine language reference manual.

Yuan /u. PhD Thesis (in progress). University of Texas.



The VIPER Machine

A 32-bit microprocessor "whose functions are totally predictable."

- Accumulator
- 2 index registers
- Program counter
- Comparison register
- 16 instructions

Avra Cohn. A Proof of Correctness of the VIPER Microprocessor: The First Level. Technical Report 104, University of Cambridge Computer Laboratory, January, 1987.

W. J. Cullyer. Implementing High Integrity Systems: The VIPER Microprocessor. In Computer Assurance, COMPASS 88. IEEE, June, 1988.



A VIPER Machine Model

```
NEXT(ram,p,a,x,y,b,stop) =
  if stop
  then (ram,p,a,x,y,b,stop)
  else (noinc \ / illegaladdr) \ /
        if (illegalcl \ / illegalsp)
          \ / (illegalonp \ / illegalwr)
        then (ram,newp,a,x,y,b,T)
  else
    ... [about 7 pages] ...
```

where

ram - a memory of 32-bit words
 p - 20-bit program counter
 a - 32-bit accumulator
 x,y - 32-bit index registers
 b - 1 bit compare result register
 stop - stop flag



The FM8502 Machine

A 32-bit microprocessor.

- 2 address architecture
- 4 addressing modes
- 8 general purpose registers
- 2^{19} 20-bit instructions

Warren A. Hunt, Jr. FM8501: A Verified Microprocessor, Ph.D. Thesis, The University of Texas at Austin, 1985.

---, Microprocessor Design Verification. Journal of Automated Reasoning. Vol. 5, No. 4, Dec 1989.



Copyright © 1991 Computational Logic Inc.

note-109.msc 76

20 Feb 91

17

Friday Presentations 159

An FM8502 Machine Model

```
FM8502(ms, mn) =
  if not(listp(mn))
  then ms
  else FM8502(NEXT(ms),
               rest(mn))
```

```
NEXT(ms) =
  list(next_memory      (ms),
        next_register_file(ms),
        next_carry_flag  (ms),
        next_overflow_flag(ms),
        next_zero_flag   (ms),
        next_negative_flag(ms))
```

... [about 10 pages] ...



Copyright © 1991 Computational Logic Inc.

note-109.msc 76

20 Feb 91

18

An FM8502 Register Transfer Model

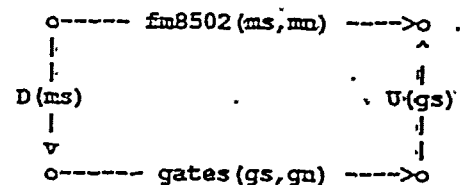
```
GATES(gs, gn) =
  if not(listp(gn))
  then gs
  else GATES(COMB_LOGIC(gs, car(gn)),
             cdr(gn))
```

```
COMB_LOGIC(gs, gn) =
  ... [on bit operators, e.g., b_xor] ...
```

where

gs - [regs, flags, mem, int-regs]
 regs - 8 32-bit vectors
 flags - 4 Booleans
 mem - 2^{22} 32-bit vectors
 int-regs - 32-bit vectors for internal registers, flags, latches

Connecting the Models



Theorem: $H(ms, mn) \rightarrow$
 $fm8502(ms, mn) =$
 $U(gates(D(ms), Kg(ms, mn, md)))$

Under the conditions \exists ,

- the fm8502 model is just as accurate as gates
- but with some details suppressed by U.



Copyright © 1991 Computational Logic Inc.

note-109.msc 76

20 Feb 91



Copyright © 1991 Computational Logic Inc.

note-109.msc 76

20 Feb 91

The Kit Separation Kernel

- Uses a modified FM8501 (ms, mn) machine
- Interrupts for timer and I/O
- Process management
 - fixed number of processes
 - process scheduling (round-robin)
 - process communication (message passing)
 - response to error conditions
- Device management for character I/O to asynchronous devices
- Memory management uses hardware protection

William R. Bevier. Kit: A Study in Operating System Verification. IEEE Transactions on Software Engineering. November 1989.



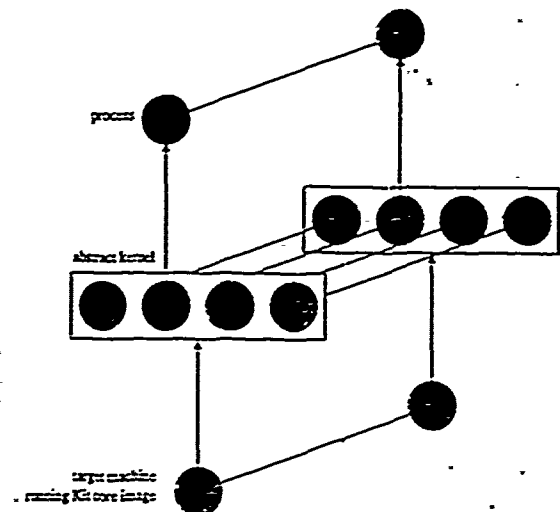
Copyright © 1991 Computational Logic Inc.

none-109.msc 74

20 Feb 91

25

Friday Presentations 161 Kit Operating Requirement, R



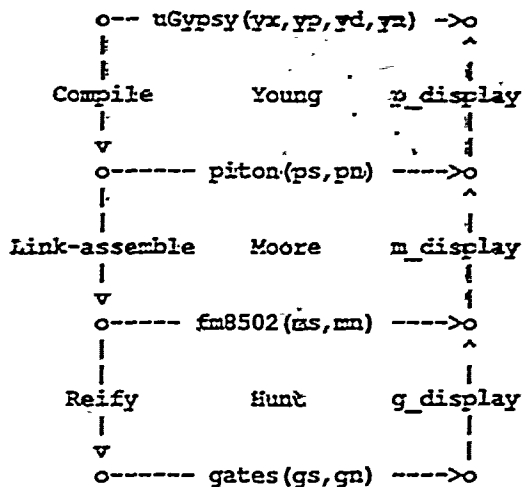
Copyright © 1991 Computational Logic Inc.

none-109.msc 74

20 Feb 91

26

The CLInc Stack



Warren A. Hunt, J Strother Moore II, William D. Young. Journal of Automated Reasoning. Vol. 5, No. 4, Dec 1989.



Copyright © 1991 Computational Logic Inc.

none-109.msc 74

20 Feb 91

27

The Piton Language

The Piton language has

- execute-only program space
- read/write global arrays
- recursive subroutine calls
- formal parameters
- user-visible stack
- stack-based instructions
- flow-of-control instructions

The cross assembler produces an FM8502 binary core image.



Copyright © 1991 Computational Logic Inc.

none-109.msc 74

20 Feb 91

28

The Micro Gypsy Language

The Micro Gypsy subset of Gypsy has

- types integer, boolean, character
- one dimensional arrays
- procedure calls with pass by reference parameters
- sequential control structures if, loop,
- condition handling signal when.

The compiler produces Piton.

The Stack Theorem

Theorem: $E' (yx, yp, yd, ya) \rightarrow$
 $uGypsy (yx, yp, yd, ya) =$
 $U' (gates (\exists' (yx, yp, yd),$
 $Kg' (yx, yp, yd, ya, md)))$

Proof: Mechanically checked.

Under the conditions E' ,

- the uGypsy model is just as accurate as gates
- but with many details suppressed by U' .

Boyer-Moore Logic

Robert S. Boyer, J Strother Moore II. A Computational Logic Handbook, Academic Press, 1988.

Matt Kaufmann. A User's Manual for an Interactive Enhancement to the Boyer-Moore Theorem Prover. TR 19, Computational Logic, Inc., 1988.



A Hierarchy of Models of a Programmed Machine

$R(yx0, yp0, yd0, ydk)$

$uGypsy(yx0, yp0, yd0, yk(yx0, yp0, yd0))$

$piton(ps0, pk(ps0))$

$fm8502(ms0, mk(ms0))$

$gates(gs0, gk(gs0))$

Corresponding to these is a hierarchy of program descriptions....

Gypsy Program Description

```
procedure mult(var ans:fm8502_int;
               i,j:fm8502_int) =
begin
  ENTRY j ge 0;
  EXIT ans = NTIMES(i,j);
  var k:fm8502_int := 0;
  k := j;
  ans := 0;
  loop
    ASSERT j ge 0 & k in [0..j]
      & ans = NTIMES(i,j-k);
    if k le 0 then leave end;
    ans := ans + i;
    k := k - 1;
  end;
end;
```



Piton Program Description

```

PG-MULT
(X ZERO ONE B ANS I J)      ;format%
NML                          ;locals
(PUSE-LOCAL ANS)             ;ans := 0;
(PUSE-CONSTANT (INT 0))
(CALL MG-SINGLE-CONSTANT-ASSIGNMENT),
(PUSE-LOCAL K)                ;k := j;
(PUSE-LOCAL J)
(CALL MG-SINGLE-VARIABLE-ASSIGNMENT)
(DL L-1 NML (NO-CF))         ;loop
(PUSE-LOCAL B)                ; b := k la 0
(PUSE-LOCAL K)
(PUSE-LOCAL ZERO)
(CALL MG-INTEGER-LE)
(PUSE-LOCAL B)                ; .if b then leave
(FETCH-TEMP-SIX)
(TEST-BOOL-AND-JUMP FALSE L-3)
(PUSE-CONSTANT (RAC 0))
(PCF-GLOBAL C-C)
(JUMP L-2)
(JUMP L-4)
(DL L-3 NML (NO-CF))
(DL L-4 NML (NO-CF))
(PUSE-LOCAL ANS)              ; ans := ans + i;
(PUSE-LOCAL ANS)
(PUSE-LOCAL C)
(CALL MG-INTEGER-ADD)
(PCF-GLOBAL C-C)
... [14 more support routines] ...

```



23 Feb 51

33

Friday Presentations.163
FM8502 Program Description

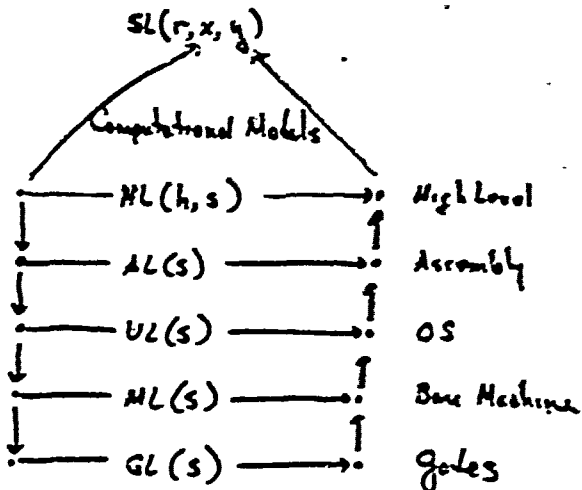
[illegible]

22 Feb 67

34

Requirements Models

Requirements Model



Computational Models: Receptive fields are an accurate model.

Requirements Model: Any kind of mathematics is acceptable

Operating Requirement

```

procedure mult (var ans:←8502_int;
                i,j:←8502_int) =
begin
  ENTRY j ge 0;
  EXIT  ans = KTRVES(i,j);
    pending;
end;

```

```
type f8502_int =  
  integer[ -(2**31) .. (2**31)-1];
```

{A Simple Problem Domain Theory}

```
function NTIMES(x,y:integer):integer =
begin
exit (assume result =
      if y = 0 then 0
      else if y = 1 then x
            else x + NTIMES(x,y-1)
      fi fi);
end;
```



25 Feb 73

36

Mathematical Requirements

- Unambiguous: Requirements have a well-defined interpretation that tells exactly what they do say.
- Analyzable: Do the requirements say the "right" thing?

$R(x, y) \rightarrow \text{good_thing}(x, y)$

- Consistency: Requirements contain no contradictions.
- Enable modeling a program component before building it (and thereby save the time and cost of designing a poor program.)

To get these benefits, the requirements notation must have a rigorous mathematical foundation (semantics).



Design >> Requirements

- There is more to designing a digital system than just stating and refining mathematical requirements.
- One must still construct a program for some machine.
- Mathematical models of commonly used languages and machines are still very scarce.



Summary

For either design of a new system or operation of an old one, mathematical modeling of digital hardware and software systems offers

Benefits: early error detection

- Saves time
- Saves money
- Saves operational disruption
- Saves operational mishaps

Risks: model misrepresents system

- Inaccurate
- Incomplete



Conventional Non-Wisdom

Use "formal methods" (mathematical modeling)

- only after a system is built to certify it
- only before a system is built to design it
- to guarantee perfect system behavior
- to eliminate the need for testing



Aspects of "Formality" in Engineering

- design automation
- engineering standards
- use of mathematical modeling
- formal mathematics (automatic deduction)

41

Recommendations Friday Presentations 165

Abandon the "____ formal" terminology
in favor of something more useful.

My Area (for example)

Math Modeling of Digital Systems ["Formal Methods"]

Research Agenda: Increase the scope &
effectiveness of those modeling capabilities

This Workshop

Digital Modeling of Human Behavior ["Informal Computing"]

Research Agenda: Increase the scope &
effectiveness of those modeling capabilities

A good terminology should help
our thinking progress.

42

Friday Presentations 166
Ideographs
Epistemic Types
and
Interpretive Semantics

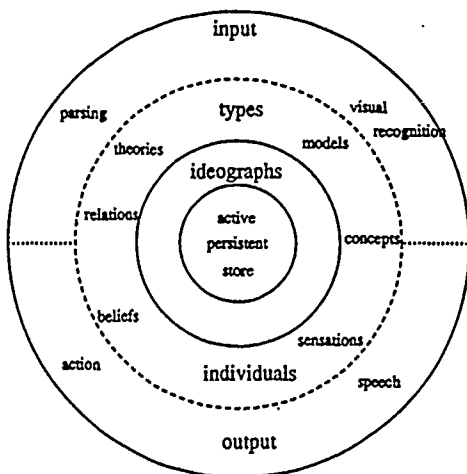
Jon Shultis
Incremental Systems Corporation

31 May 1991

Explanation of the Title

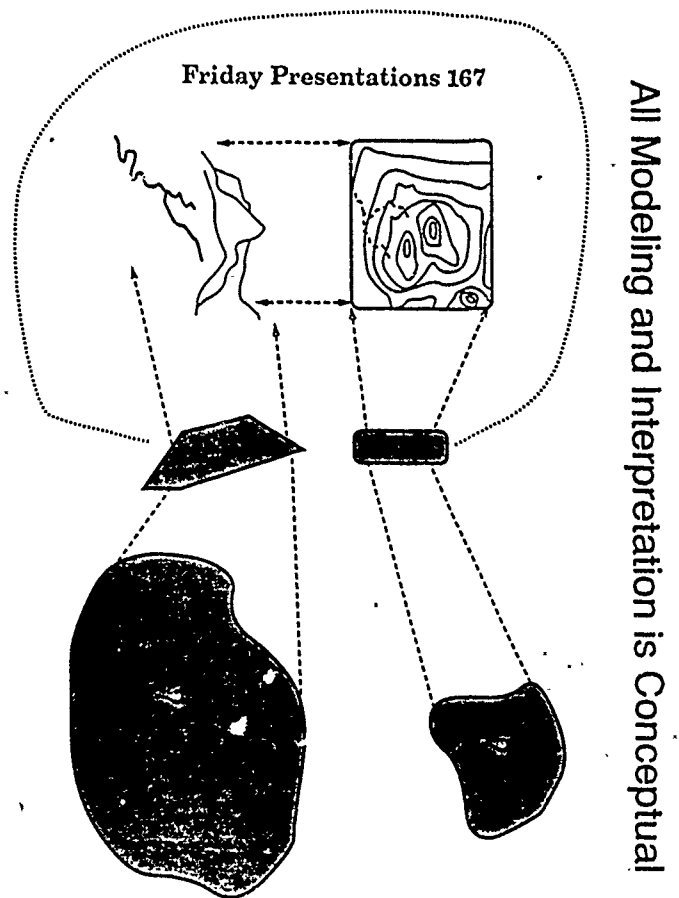
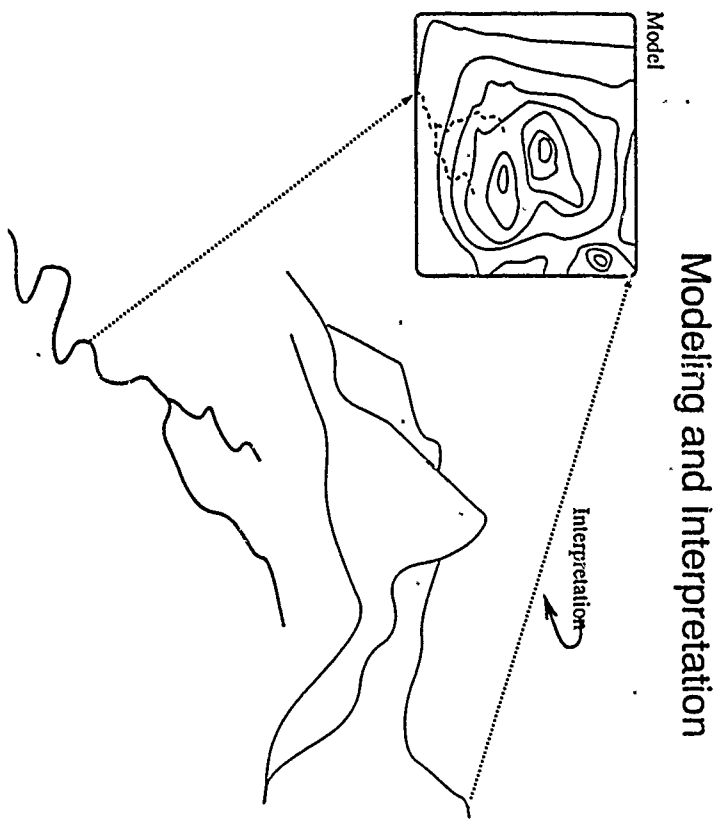
- * Ideographs = *structure for conceptual modeling*
- * Epistemic Types = *constructively grounded categories*
- * Interpretive Semantics = *"meaning is use"*
 - ✓ Formalist: *The world is a model of my language*
 - ✓ Informalist: *My language is a model of the world*

Cognitive Agent



Outline

- * Modeling
 - ✓ Modeling and Interpretation
 - ✓ Conceptual Modeling and Interpretation
 - ✓ Formal Modeling
 - ✓ Informal Modeling
- * Categorization
 - ✓ Formal and Informal Classifiers
 - ✓ Formal Types
 - ✓ Informal Types
- * Implementation (*structure + process*)
 - ✓ Abstract Syntax
 - ✓ Iris
 - ✓ Attributes
 - ✓ Ideographs
 - ✓ Epistemic Ideographs
 - ✓ Connectionist Implementation
- * Conclusions



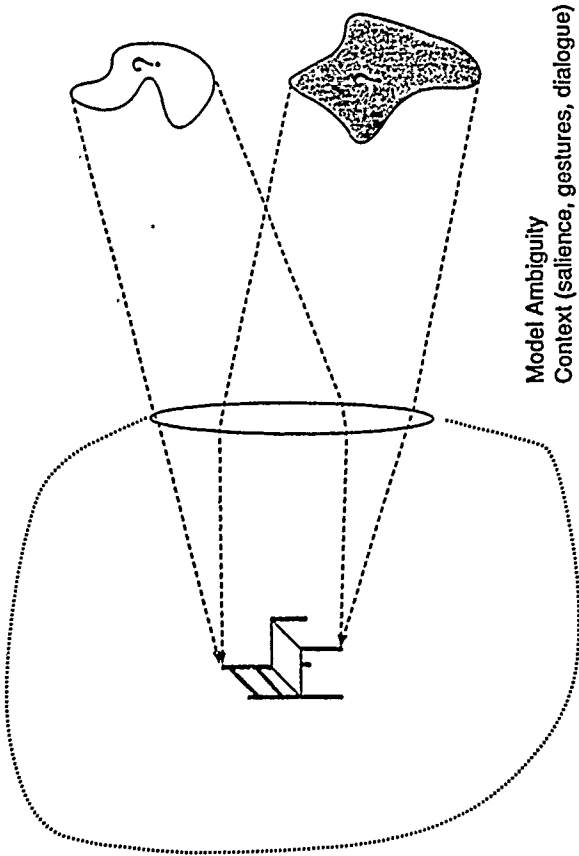
Formal Modeling

- * central dogma: dissociation of form and content . . .
- * representing object manipulated w/out reference to, or influence from, modeled.
- * implication: formal system is a constraint on the world; hence model theory
- * $\llbracket \rrbracket : \mathcal{A} \rightarrow \mathcal{M}$; i.e., make world conform to language
- * ex: digital airframe in a digital wind tunnel.

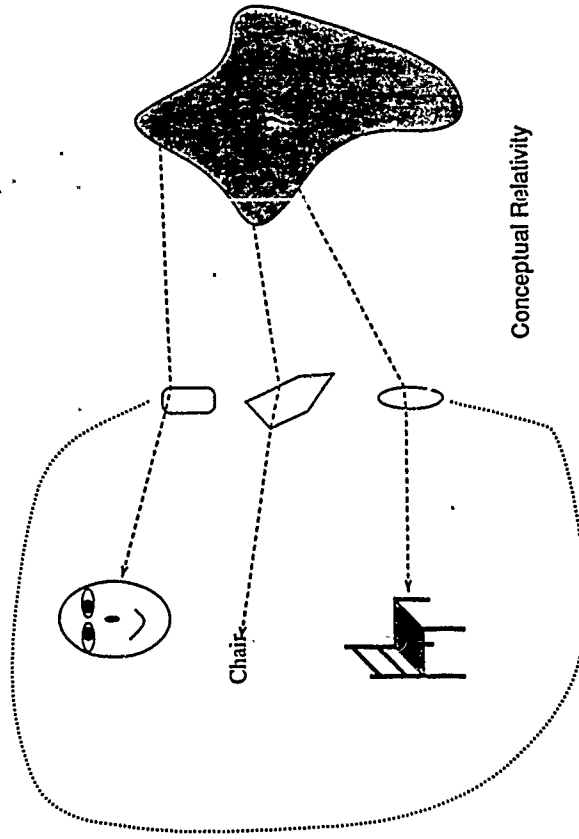
Informal Modeling

- * central dogma: groundedness (context-involvement) is essential
- * Exploits partial access to modeled domain, which in turn
- * constrains form of the model. Involvement of modeled domain leads to greater completeness of model + interpretation.
- * $\llbracket \rrbracket : \mathcal{M} \rightarrow \mathcal{A}$; i.e., try to make language conform to world
- * ex: balsa airframe in physical wind tunnel.

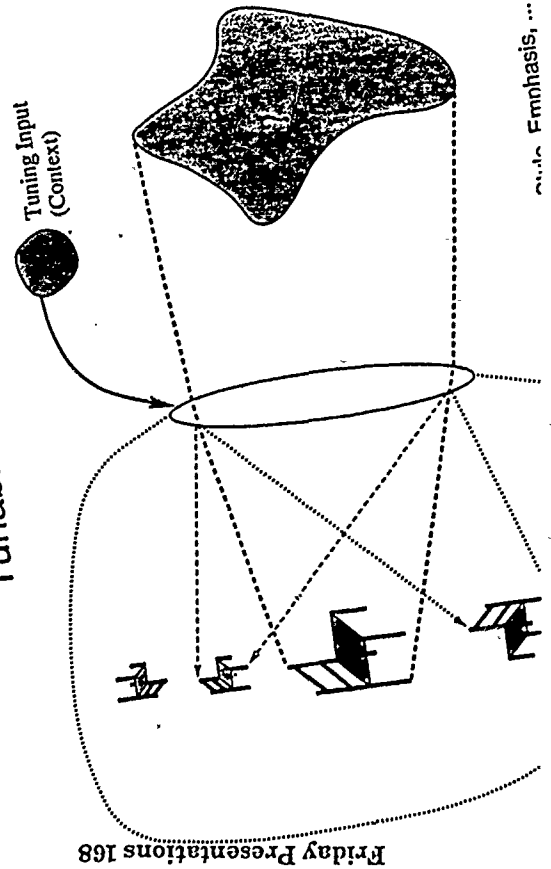
Simplifying Interpretation



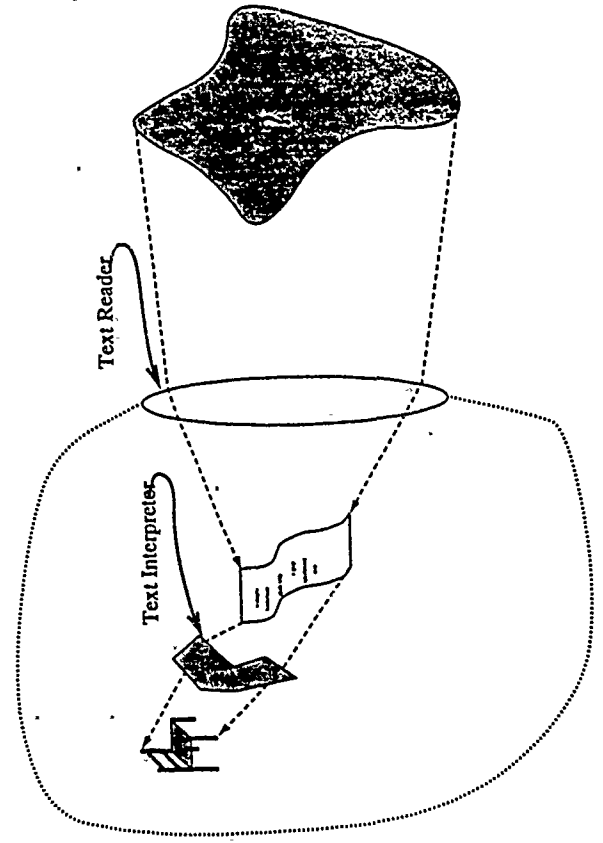
Multiple Interpretation



Tunable Interpretation



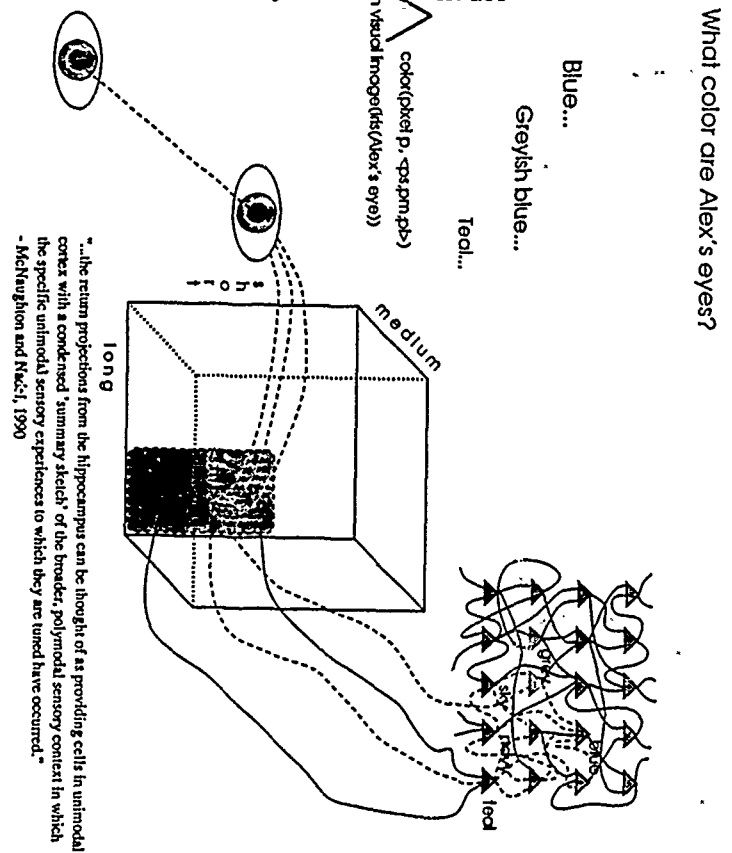
Text Interpretation



Formal Types

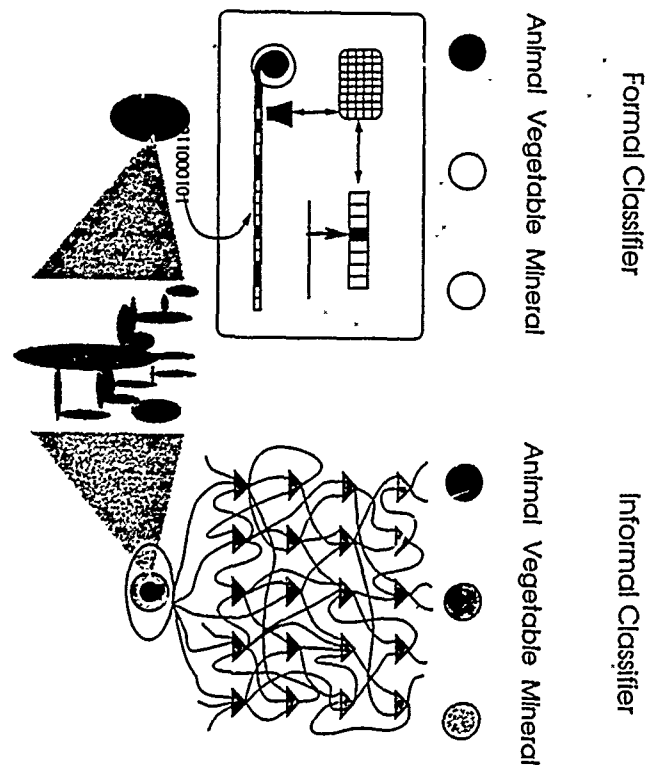
- * Extensional: classes
- * Intensional: set of properties
- * Members v. examples
- * Formal constructive types
 - ✓ Evidence = Process (construction)
 - ✓ $\llbracket P \rrbracket$ = Type of constructions
 - ✓ Epistemic: know-what = know-how

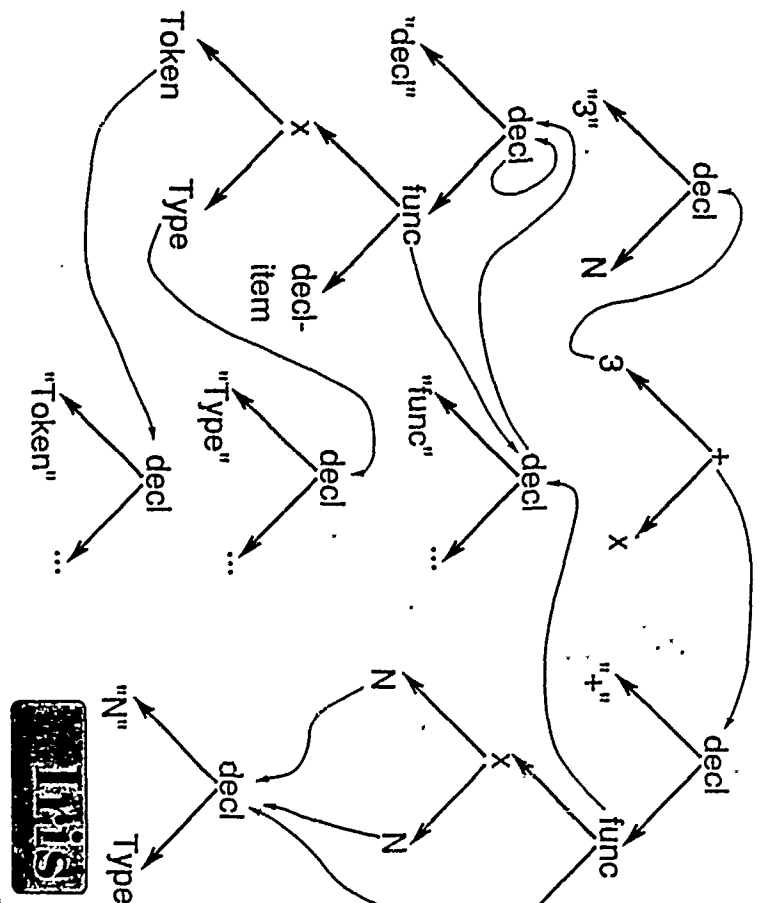
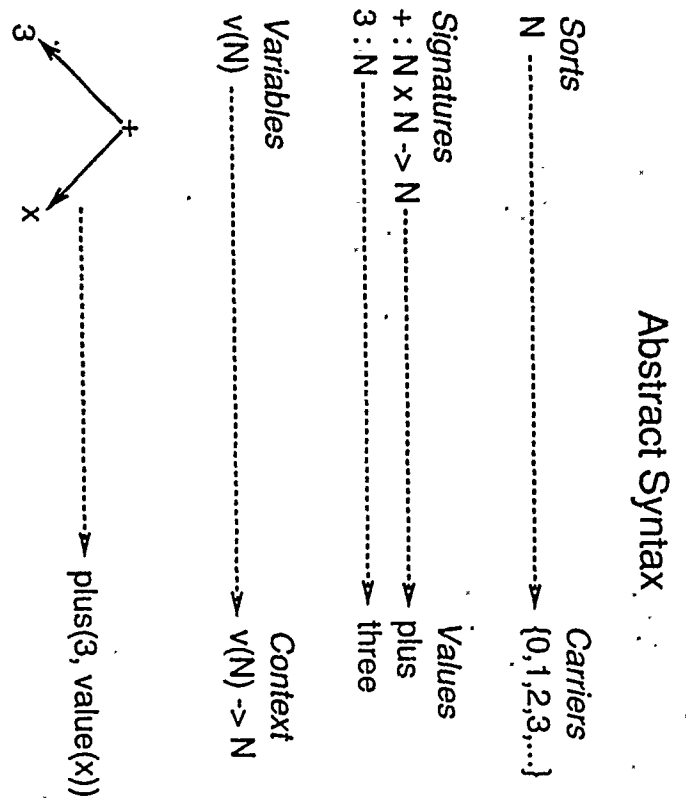
Friday Presentations 169



Informal Types

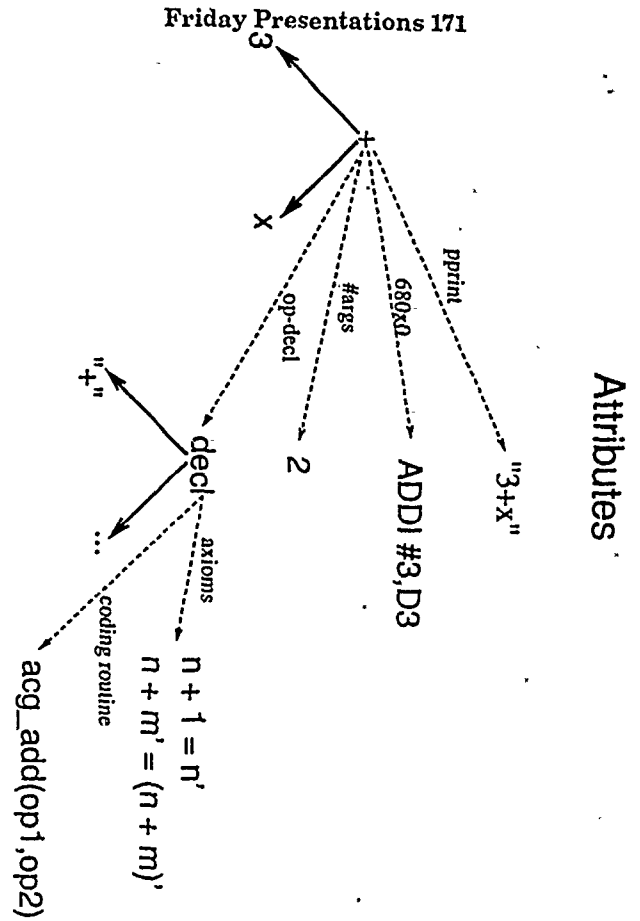
- * Graded
- * Theory-dependent (conceptually relative)
- * Epistemic, graded theories
- * Radical empiricism, i.e. pragmatism
- * Intuitionism in Brouwer's sense





Iris

- ★ Structure:
 - ✓ Dictionary (directed graph)
 - ✓ Internalizes Sorts, Signatures (lexical information)
- ★ Process:
 - ✓ Applicative Evaluation
 - ✓ Overload Resolution (lexical disambiguation from context)
- ★ Issues:
 - ✓ Interpretation: Structure \rightarrow Meaning
 - ✓ Grounding: none
 - ✓ Epistemology: fiat
- ★ Recurrent
- ★ Structural constraints on interpretation



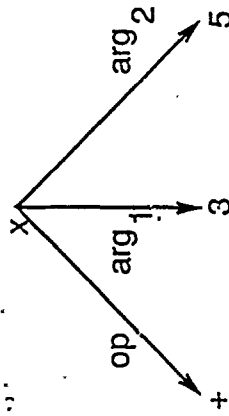
Attributes

- ★ Structure: Labelled Graphs + Externals
- ★ Process: AE + call-out/call-in
- ★ Issues:
 - ✓ Interpretation: Interactive, composite
 - ✓ Grounding: insulated contact
 - ✓ Epistemology: fiat
- ★ Open-ended, flexible
- ★ Equivalent to feature structures
- ★ Attribute relations are external

Ideographs

- ★ Δ : Internalize attribute descriptions
- ★ "Encyclopedic"
- ★ Self-referential Encodings

Trees, Types, and Evaluation

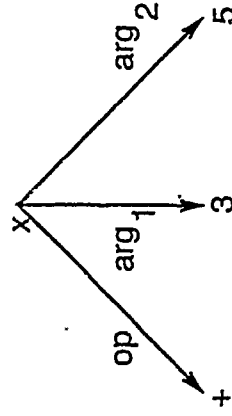


Constituents contribute information about x

$x : \{op=+, arg_1=3, arg_2=5\} \vdash$

$\therefore x'val = 8$

Belief



Constituents contribute beliefs about x

$\vdash x'op = +, \vdash x'arg_1 = 3 \quad \vdash x'arg_2 = 5$

$\therefore \vdash x'val = 8$

Constructive Types

$[[P]] = \{ \pi \mid \pi \vdash P \}$

π witness evidence process proof construction

ex: $\lambda x.x \vdash p \rightarrow p$

Euclid's algorithm $\vdash \text{gcd}$

where $\text{gcd} = \{ \text{For all pairs } \langle x, y \rangle \text{ of natural numbers there exists a unique } \text{least } z \text{ such that } z \text{ divides both } x \text{ and } y. \}$ ^{greatest}

Epistemic Types

$[[P]] = \{ \pi \mid \pi \sim \vdash P \}$

π witness evidence process proof construction

Not Quite...

$[[P]]\Gamma = \{ \pi \mid \Gamma \sim \pi \vdash P \}$

The meaning of P in context Γ is the set of evidence that would lead, in context Γ , to belief in P.

But...

Belief Strength

μ_P

Constructive inspiration: identify proposition with evidence! e., believing P with strength σ is equated with what causes the associated node to be stimulated to level σ .

Contributions to stimulation levels are just the stimulation levels of other nodes, adjusted by weighting functions. So...

$$b_\sigma = \frac{1}{1 + e^{-\sum_{i:1} \omega_{i,b} (i_\sigma)}}$$

Propositional content may be inarticulable.

Symbolic expressions summarize (model) activation patterns.

Friday Presentations 173

Epistemic Ideographs

* Structure:

- ✓ Internally labelled graphs
- ✓ Spontaneous nodes (sensorimotor periphery)

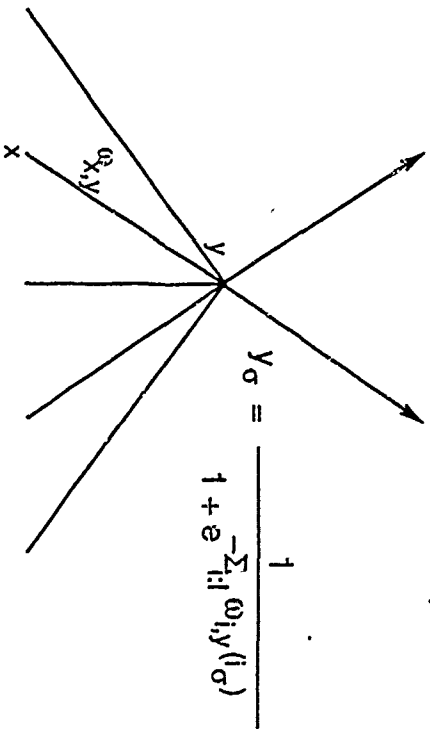
* Process: parallel, distributed

- ✓ equilibrium
- ✓ spreading activation
- ✓ ergodic/chaotic

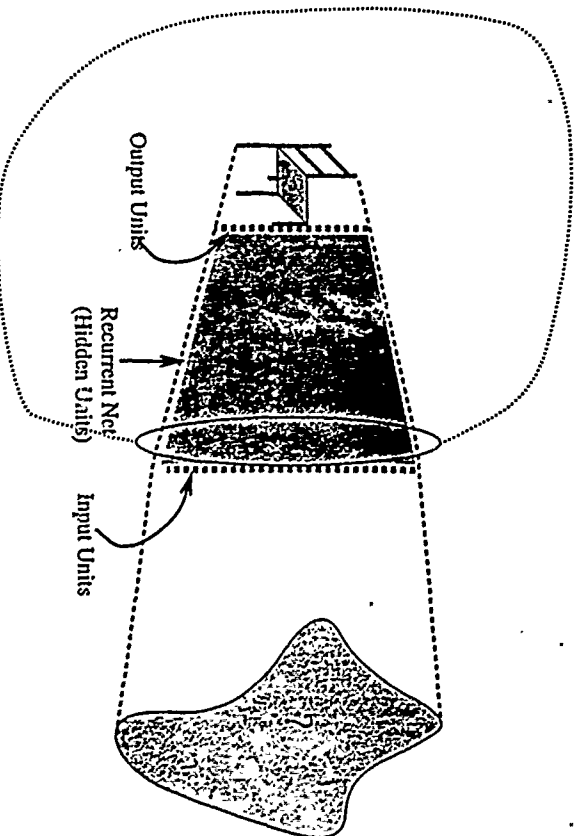
* Issues:

- ✓ Interpretation: activity patterns are models
- ✓ Grounding: Constructive; spontaneous base
- ✓ Epistemology: Evidential

Epistemic Ideographs

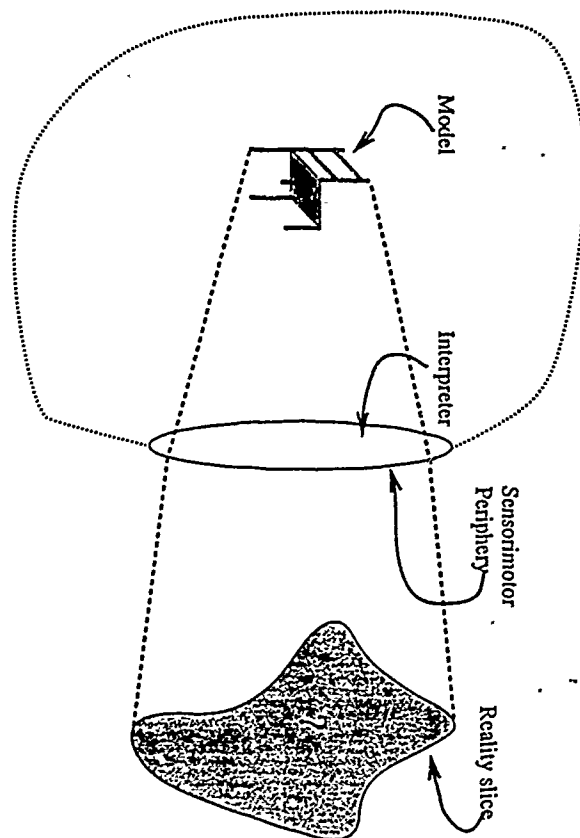


y_σ is a generalized constructive "proposition"



Connectionist Implementation

Conceptual Modeling and Interpretation



Conclusions

- * Reality is unformalizable.
- * Cognition is grounded (hence informal).
- * Grounding is an epistemic/causal link between reality and mental representations.
- * Interpretation is the mapping of conceptual structures (input units) to conceptual structures (output units) through other conceptual structures (hidden units).
- * Analogy and metaphor are the normal case: modeling and interpretation.
- * Epistemic ideographs a plausible structure for cognitive modeling.
- * Formalism a limiting case. Paradigmatic in a radial category.

A Model for Informality in Knowledge Representation and Acquisition (an extended abstract)

Timothy C. Lethbridge

Department of Computer Science
University of Ottawa
Ottawa, Ontario, Canada K1N 6N5
(613) 564-8155 FAX: (613) 564-9486
tcl@csi.uottawa.ca

Introduction

This extended abstract summarizes how we are handling informality as a fundamental aspect of our paradigm for knowledge representation (KR) and acquisition (KA). We have developed this paradigm as a result of several years of experience with industrial application of our research KA system CODE [SKUC 89]. One of the most striking observations from this experience is that people need to be able to work at any level of formality (or informality), and to freely mix such levels. Among various things, our paradigm attempts to systemize the formality spectrum in knowledge based systems.

Concepts and the formality spectrum

In our paradigm, all knowledge is represented in computer memory objects called *concepts*, intended to correspond to concepts in the human mind. We have additional constructs called *knowledge maps* and *knowledge masks* which allow us to interpret networks of concepts in terms of such traditional models as frames or semantic nets, and to restrict the focus of the knowledge to useful contexts.

As in most KR paradigms, concepts are classified ontologically, i.e. on the basis of their knowledge content. Of equal importance though, is an orthogonal classification of concepts in terms of how they are to be interpreted: what processing is performed or performable on them.

The hypothetical extremes of the formality spectrum are defined as follows: The representation of a completely informal concept in a knowledgeable agent would contain only strings or bit maps that are not immediately interpretable by the agent. Uninterpretable strings typically contain natural language fragments. The representation of completely formal concepts would contain only links (which must also be completely formal concepts) to other completely formal concepts.

In practice, neither extreme of the formality spectrum is found (our definition of a completely formal concept is infinitely recursive). Concepts have links to other concepts (if nothing else they may be placed in an isa hierarchy under, say, *entity*) which makes them not totally informal in our paradigm. Similarly they are not totally formal because they either have some uninterpreted content or have direct or indirect links to concepts with uninterpreted content.

Concepts can be partially ordered in terms of their formality level. To compare two concepts one compares their uninterpreted content and the formality of linked concepts.

At present we use a heuristic algorithm for this. Note that one does not assign a formality 'measure' to a concept, concepts are merely *comparable* in terms of formality.

Knowledge acquisition as the increasing of concept formality

We consider knowledge acquisition to be the process of increasing a concept's level of formality (by supplying it with more links and hence with more structure or form). The process is recursive in that the primary way to increase a concept's formality is to increase the formality of its linked concepts. The secondary way is to add new links; often this is done as a by-product of increasing the interpretability of the concept's uninterpreted content (by virtue of the acquisition of other concepts, the potential for a system to make automatic interpretations increases).

In our KA model, the early phases of a user's efforts are devoted to entering highly informal concepts (perhaps using an outline-processor style of interface) to capture the user's stream of consciousness [LETH 91a]. As KA progresses the user increasingly enters links that give concepts more formality. By comparing the formality of concepts, a system can help the user decide on a productive direction in his or her knowledge acquisition efforts.

A word about our use of the terms 'formal' and 'informal'

We use the term 'formal' in the ordinary English sense of 'has a form or a structure that follows rules'. A concept that has less such structure than another is more informal.

It is important not to confuse this idea with the sense of 'formal' as used in 'formal language'. The latter sense typically is 'has an unambiguous syntax and semantics'. Our use of 'formal' is more general than the latter: We would prefer to call the latter 'representationally formal', where concepts are only compared with consideration given to representation relations, not all relations.

Conclusion

We have developed a knowledge paradigm that integrates the idea of a formality spectrum. We believe that most existing knowledge based systems stress formality, ignoring the continuum that could exist between them and informal natural language. Integrating the notion of the formality spectrum allows a more accurate representation of what is in a person's mind (which typically has low formality). This can greatly facilitate the knowledge acquisition process by reducing difficulties caused by the user prematurely having to formalize ideas.

We are implementing our ideas in a completely new version of our knowledge based system called CODE [LETH 91b].

Acknowledgements

This research is funded by Bell-Northern Research and the Natural Sciences and Engineering Research Council of Canada. The author thanks Douglas Skuce, Ingrid Meyer and Bruce Porter for their ideas and inspiration.

References

[LETH 91a] Lethbridge, Timothy C., "Promoting the Free Flow of Thinking During Creative Knowledge Acquisition", in preparation for *4th Knowledge Acquisition for Knowledge-Based Systems Workshop*.

[LETH 91b] Lethbridge, Timothy C., "A Detailed Conceptual Description of CODE Version 4", *technical report* in preparation, Dept. of CS., Univ. Ottawa, 1991.

[SKUC 89] Skuce, Douglas, Wang, S., and Beauvillé, Y., "A Generic Knowledge Acquisition Environment for Conceptual and Ontological Analysis," *Proc. 4th Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff (Canada), 1989.

Friday Morning Discussion

Shultis: What I'm going to propose is that we try an exercise here, which I hope that people will be able to carry on elsewhere. It's the kind of exercise that has been done lots of times, in lots of places, but with different goals. The problem is basically this. I want people to break into groups of three. One of the three will be an observer, a recorder. The other two of you are going to engage in a dialogue in which you're going to try to solve a problem which I don't think you've ever solved before, but which you have a lot of background for. I'll give you the problem in a minute. But the goal is this: for ten minutes, two of you are going to try to solve this problem, and you're going to engage in a dialogue about it. One of you is going to watch, and write down what you see happening. What kinds of reasoning domains do you see people using, what kind of rules, what kinds of floundering; can we try to characterize the floundering?

Littman: Jon, let the observer give the problem, because if you give people 30 seconds, even 15 seconds, it's gone.

Shultis: Fine; I'll do that. You're absolutely right. Anyway, the premise of this whole workshop is that informal computing, whatever it is (and I still don't want to call it mathematical modeling of human behavior, though there's a grain of truth in what Don said) ...whatever it is that extends those capabilities we now have, it would make computers more useful and helpful. That's the premise here. So the question is: what is it that we're looking for, and what makes it helpful? So, I want observers to note things like: what domains people are reasoning in, what representations they're using, what kinds of inferences, what sorts of methods, what kind of control structures are going on, or what kinds of passing control, granularity issues, anything that you can think of that's appropriate. A better way to do this might be to videotape this and analyze it later, but we've only got an hour, and we don't have the equipment.

So, ten minutes of problem solving, and then we're going to take 20 minutes to sit around as a group and analyze the data that was collected, and try to identify some things about the data. I'll tell you now what we're going to look for. We're going to look for some useful phenomena that we label as "informal", identify its utility...

Littman: Why don't you not tell us now? Otherwise, people will...

?: ... you'll alter our behavior...

Littman: It's an interesting bootstrap, but watch out for Heisenberg.

Shultis: All right. Look, I understand... this is experimental design on very short notice; you're right. I wish that we had more time to do this. OK, I need observers. OK, you, you, and you.

Littman: It's got to be a really hard problem. With Standish and D'Ambrosio here, it's not fair!

Shultis: Oh, it is hard. I'll buy any group that actually solves this problem a six-pack of beer.

??: I don't drink alcohol.

Shultis: OK, look, beer is a prototype. Category good-thing. Solve the problem and I'll do something good for you.

JS to observers, privately: The problem is: prove the Pythagorean theorem. You can assume you know the formula for the area of a square, and that you know polynomial algebra.

[10 minutes pass]

Shultis: Within each group, now, what I'd like you to do with your observer is the following: try to identify, out of what went on, one thing that you can identify and characterize as an informal phenomenon. If you can't come up with any such thing, say so, but try to identify one thing that was going on that you can say is informal, where, if you like, "informal" is defined here operationally as the kind of good thing we'd like to have. Try to articulate what its utility is, why would it be a useful thing, why it is a good thing. And try to articulate what about it is informal. I imagine that will take some time, talking about what happened and analyzing it, so let's do that for about 15-20 minutes, and then I'll have each of the observers report out on their group.

[20 minutes pass]

Shultis: OK, let's get a rundown on what you found out.

[First group: David Fisher, John Kozma, David Mundie (observer; absent to pack his bag)]

Kozma: We used pictures. We drew, and used transformations. That has been referred to as "informal", although I'm not so sure I would call it that. I think it's possible to conceive of some kind of formal system of graphical, pictorial, proofs for geometric theorems, especially in this instance. We tried to restate the problem, and spent a lot of time trying to figure out whether the instruction to use our knowledge of polynomial algebra was significant, and how we were supposed to do that, or whether it was just a red herring, or what. We tried to draw a picture of one special case, that of an isosceles triangle, and to solve that, and then move on to the more general case of any right triangle. Then time got called.

Talking afterwards, I have the impression that my mental processes, trying to solve the problem, were as follows: I have seen a logo for somebody,

or something—a picture of a 3-4-5 right triangle with squares on it. They're all marked off, gridded off, so I somehow have the idea that that's the kernel of the proof. If you can somehow figure out how to manipulate those squares, you can solve the problem. So we drew a picture of a triangle (though not a 3-4-5 triangle). We were trying to work with the diagonals, and trying to match them up. Again, I don't know whether you would call this formal or not...

Shultis: What you did there was this: you saw something that stimulated your thinking...

Kozma: *That* aspect of it was certainly informal.

Shultis: ... some association to the problem, and it seemed that there was something analogical there that you remembered from before that you might be able to apply. Then you tried to use that pattern.

Fisher: The one thing that I found myself doing that I certainly don't see in formal systems, was that I basically started looking at almost random aspects of it. You know: let's square the sides, let's draw some things and look for correlations there that might have some relevance, but it was all done without any guide to what that relevance would be. I just felt that if I could identify a pattern then I might be able to relate it to something else I knew to generate a solution.

Shultis: So you were just generating things in the space and seeing what you could do.

Kozma: But we were guided by the idea of taking the edges of one triangle and drawing squares on them...

Fisher: It was all geometrical reasoning, based on a diagram that we thought represented the problem.

Shultis: Well, you can't solve the problem that way without the real calculus.

Standish: Pythagoras did it! His original proof was actually in terms of...

Shultis: But that's a limitation. It's a limit proof, that way.

Standish: No, it has to do with congruent triangles. You carve up the squares on the three sides of the right triangle.

Fisher: I'm not sure that's relevant to our discussion

Standish: But the constraint was to do it in terms of polynomial algebra. That wouldn't have given an algebraic proof, that would have given a proof by direct geometry.

Reeker: But it would have been one you could have algebraicized, I suppose.

Standish: Not without the Pythagorean theorem.

Kozma: One more thing. The fact that the problem statement included a reference to polynomial algebra hindered me in a way, because I thought

there was some simple application of High School algebra that I should be trying to recall. It got in the way of trying to work with the squares and coming up with some other solution.

Shultis: I think that this use of associations of things that you knew before, and trying to fit them onto the problem, is one kind of phenomenon we've talked about a number of times—trying to use analogical reasoning in this inexact way. So you saw that happening, and it seems useful in that sometimes that kind of thing works. Another group?

[Second group: Tim Lethbridge, Steve Wight, Larry Reeker (observer)]

Reeker: Tim and Steve tended to be pretty geometrically oriented, too. They tried to use drawings to come up with some method, different types of drawings, including the one you mentioned, which I think most of us remember vaguely from our geometry books, and other drawings. They changed things around, and redrew, sometimes abandoning the drawings altogether. They took slightly different tacks on this. Tim was trying to look at a specific case, like you said, for the isosceles case, or the 30-60, and Steve was trying to do the general case, trying to keep it clear of constraints. That was the basic approach: labeling the sides, in terms of thinking about putting it into an algebraic form. I was interested to note that nobody really tried to do it using analytic geometry, basically with drawing a vector out there and giving it coordinates. I don't know whether that was prohibited by the restriction to polynomial algebra.

Shultis: Well, the answer is that, as far as the experiment is concerned, the important thing was that you felt there were some constraints on the problem solution—otherwise it's too easy. But it had to be a problem where you had enough background knowledge so you could make some progress on it, so whatever you construed the constraint to be is fine with me.

Reeker: They observed that by using the pictures they were trying to keep track of things, so that they could manipulate things in their minds without getting too much information that they had to keep in their mind at once. They could manipulate one part of the picture at a time. Sometimes there were two things that they would use as markings in their picture. They would use some of them to keep track of what they were doing as they tried to cut up these figures, and so on, some markings just to analyze what was going on. Also, the redrawing sometimes had to do with attaching new semantics. They might come up with the same drawing again, but it was because a new idea had come along, and they were attaching some new semantics to the drawing.

Shultis: So there's that point of interpretation. You've got this representation or model, and you're interpreting it in different ways, depending on how you're trying to construe it. It would be nice to tease out what was going on with that conceptual shifting.

Lethbridge: I found that, during the conversation afterwards, I had enough of the diagram written down that my mental processes could continue.

I just needed this to act as some kind of a crutch. I didn't have enough short-term memory to remember exactly the configuration of lines, but once I had it down on the paper I had the ability to continue the analysis completely in my mind.

Reeker: As we were talking, he suddenly started writing furiously...

Lethbridge: That's because I'd been thinking for a minute or so.

Mundie: I don't if this has already been talked about, but thinking about this while I was packing my bag, what struck me was how close the process they went through came to Hofstadter's model of bottom-up, essentially random observations at the bottom, combined with unification with the top-down specification. They were making random observations and suggestions like: "Oh, this line is the same length as that line over there." "This is z, too." "Draw another bigger one around the whole thing."

Wight: Generating random patterns...

Littman: It's *not* random, though.

Lethbridge: It's *not* random.

Fisher: It's certainly not *goal-oriented*.

Littman: Yes, it is, yes, it is. And the reason it's not random—actually, I've got a student working on this—is that there are heuristics for exploring that space. We may not know what they are, but connecting lines together, and looking, deciding what to focus on...

Fisher: What I'm suggesting is that the goals we were applying there were *not* ones related to the theorem we were trying to prove. They were in fact goals having just to do with our understanding more about the space.

Littman: But it's not random, though, in the sense that my student is enumerating heuristics for doing that kind of exploration. What's going to happen is that he will build a control structure, and it's going to do that kind of searching, and that control structure will *not* be a random number generator.

Fisher: What I think is important is that it was not a hierarchically guided search; we didn't take this top goal and divide it into smaller goals. We really were operating without knowledge of what that higher goal was.

Littman: I don't think that's right, because given another problem you might have looked at that thing differently.

Shultis: But, David, it's an empirical question what kind of searching that is. We need to build models of it, and we need to test them. And that's an excellent topic for further case studies.

Littman: I was just reacting to "random". [*Digression about randomness.*]

Shultis: OK, third report.

[**Third group:** Tim Standish, Bruce D'Ambrosio, David Littman (observer)]

Littman: We had some informality in the process, and some informality in the data. Tim came up with one kind of informality in the process. Bruce came up with another, and then we have the sketches, which we all consider to be informal...

Standish: I can try to recreate what my exploration sequence was. I first tried, historically, to dredge up what I knew about the problem. I recalled that Pythagoras had solved it by actually putting the squares on the sides of the right triangle and cutting them up into pieces. I think I knew that I couldn't reconstruct that in the time given. So there was always risk analysis going on: will I be able to complete a memory dredge plus bringing it up to completable and correct form in the time given? So I cut off that exploration instantly, because it didn't seem to lead to the solution within the time limits. Then, the next thing was to pursue a trig formula. You start with $\sin^2\alpha + \cos^2\alpha = 1$, and then plug it in and it yields the solution. But that's disallowed, because you're supposed to use polynomial algebra. Since it was disallowed, could you get rid of the trig by substituting algebraic formulas for \sin^2 and \cos^2 . And we thought, well, maybe we could substitute the Taylor series, but then that screwed up because it wasn't quite polynomial. Then we thought maybe we could dredge up the formula for the area of a triangle in terms of the three sides. That has to do with the semidiameter, and the radius of the inscribed circle. So I made a lame attempt at dredging that up. And then time was really pressing on, so we thought before I could dredge that up and verify it, we'd better go back to the trig thing, and see if we could eliminate the trig again. We did that by taking area is one half of the product of the two sides times the cosine of the included angle, and got rid of the cosine by solving that for it and plugging the area in, and that just about worked, within one quarter, until Bruce said, "Well, you can't really do that, because it depends on the trig formula $\sin^2 + \cos^2 = 1$, and doesn't really eliminate the trig from which you're plugging it in". It'll look like it did, but it really won't. So he had one that was based on tiling the square, where you duplicate the triangle along its hypotenuse and you get a rectangle. Take the triangle and reflect it across the hypotenuse, and you get a rectangle. And then, you take the longer side and make a square out of it, and then take the remaining part of the square and divide that. And you try to get an algebraic relation out of that, and then we thought at the end if you take the formula for the area in terms of the three sides only, and then the buzzer sounded, and we had to quit. So we're not sure if that leads to a solution or not.

The meta-comment is: we were using risk analysis to know whether to pursue certain avenues, and the time limit was so severe that we couldn't explore very long before saying we'd better abandon this and progress and deepen elsewhere. So we started one path, started another, decided that wouldn't fit within the time limit, came back and reprogressed and deepened another. To be truthful, there wasn't all that much time to keep coordinating

about what we were doing, so we were both on independent exploration tracks, and only a little time was allocated to communicating to see if pieces would fit together, and the thing we were pursuing when the buzzer sounded was a piece of his thing with a piece of my thing trying to fit it together, the area in terms of the three sides only.

Littman: There were two processes we counted as informal. Tim just talked about one of them, and that is the heuristic evaluation of risk, heuristic evaluation of whether you're going to succeed... it's not A-star, but it's that kind of idea, just that we have weak heuristics, I guess, except that he has good ones, since he's a mathematician. And that's interesting. It's not fair for this problem, anyway.

And the other process was one that Bruce mentioned. I guess you called it "deliberate oversimplification for the purpose of scrounging", where you try to just get anything to happen. This was very much like what you all were talking about.

Fisher: Yes, in fact, I was very fascinated by that, because this timing thing, I think, had a *tremendous* impact on my approach to it. This randomness—what I was claiming is randomness—was a consequent of saying "There's not enough time to really look at the alternatives and decide which one is worth progressing, so I'd just better pick things rapidly or I'll never get through within the time limit." So I think if you had said we had three times that amount of time, we would have seen a very different response.

Littman: The interesting thing that comes out of this is that Tim is trained as a mathematician, so if you look at it in terms of heuristic evaluation search functions, he's got real good ones.

Standish: I didn't get the solution!

Littman: It doesn't matter. You knew the kinds of things that you might want to look at, and you had a very far horizon for how hard it was going to be to combine things. You quickly saw, for example, that eliminating the trig would still be just covering it up.

Standish: I thought that was really slick. We got one solution, and all we had to do to get from one solution to the other was to sweep the trig under the rug and not let it appear.

Littman: Just for my own curiosity, would that have been a fair solution, if he had done that? If he got rid of the trig and expressed it in terms of...

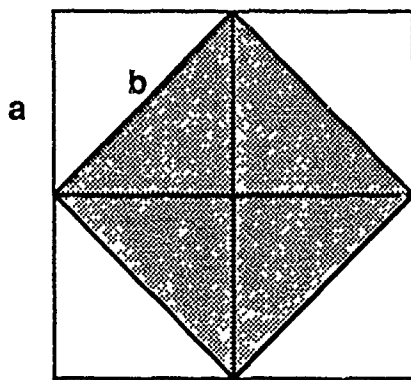
Reeker: Usually you derive the trig using the Pythagorean theorem.

Littman: What I was getting from Tim was interesting because it relates to his other interests. Tim is very interested in models of proof, not in the mathematical sense but in the cognitive sense. So what I saw Tim doing was searching the space of models for proofs. He used the idea that "If I could just eliminate this then..." and that was one of the key points. And then when you two combined your solutions again it was trying to eliminate stuff. But the idea was that he has very good heuristic search of

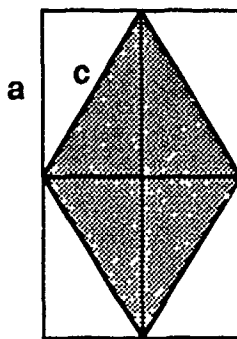
this space of models of proofs.

Shultis: If somebody wants to do an experiment on this, I'm going to spend just a couple of minutes telling you all what the answer is, and noting something funny about it. This is a variation on a puzzle that has plagued me for years. The proof for the isosceles right triangle you can do fairly easily off of this diagram from the *Meno*. You take the isosceles right triangle diagram out of the *Meno*, and take the argument out of the *Meno*, and you show it to students, show it to random people, and you say, great, this is a nice argument...

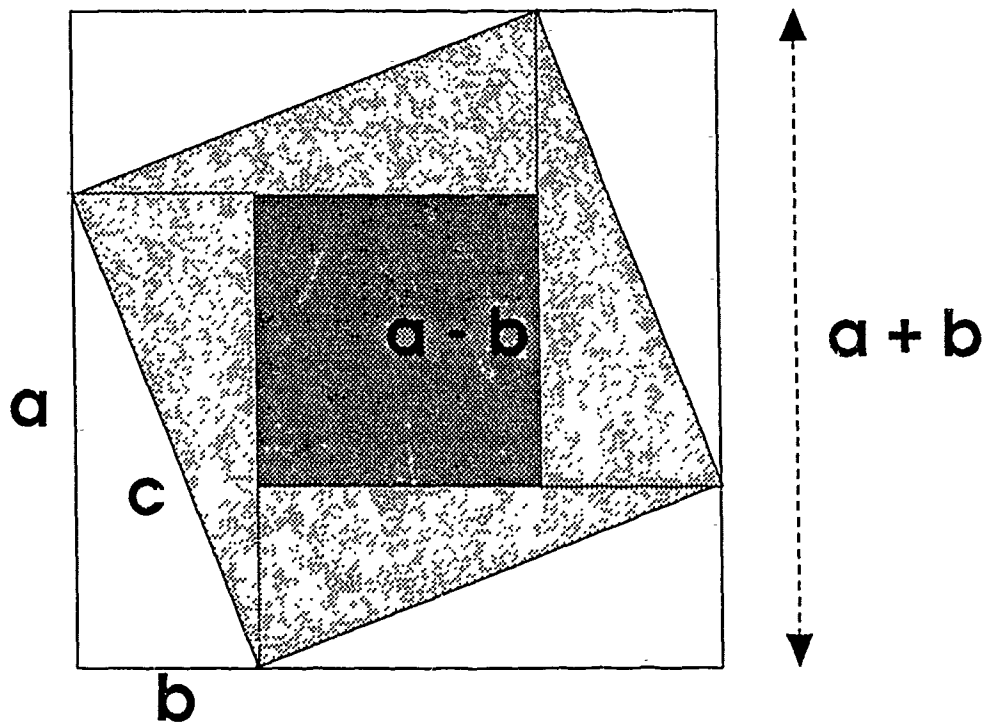
Standish: Jon, I'm not on my Plato. What is the argument? All I see there is a few lines in the middle of a rectangle. I see a lozenge, with a cross, inside a rectangle. What is it?



Shultis: Sorry. You've got two a's and a b. In the special case here, the argument is that you can see that this line here bisects this area. And you can see that all these things are just rotations of the same thing, yes? And so you can see that this area here is half of this area out here, right? And so that b^2 , in this case, is equal to $(2a)^2/2$, so that b^2 is equal to $2a^2$. The funny thing is that if you ask people to generalize this argument, I've observed that almost everyone does it wrong, like this:



The right generalization is this:



$$\text{[light gray square]} = \frac{(a+b)^2 - (a-b)^2}{2}$$

$$\text{[dark gray square]} = (a+b)^2$$

$$\begin{aligned} c^2 &= \text{[light gray square]} + \text{[dark gray square]} = \frac{(a+b)^2 + (a-b)^2}{2} \\ &= a^2 + b^2 \end{aligned}$$

OK, Just to summarize some things. My purpose in doing this little experiment was to see whether or not we could establish some kind of a methodology for doing two sorts of things, over a period of time. The first question is: is this kind of experiment a good way to try to tease out, and identify, the issues and phenomena we've been talking about at this workshop? what utility do we think there would be for them, and what about them do we think is different from the formal? That's a question, and we can discuss that in a minute. But on the assumption that it is, there are two lines of short-term research that I can see. I'm going to put these forward as possible conclusions, and then you can tell me, no, these shouldn't be conclusions of the workshop at all, but... Basically, I think we need to do some empirical research, and get some evidence in. This is one way to collect it, and there are two sorts of things you can do with studies like this. One is to try to build a taxonomy of the phenomena: what are they, what's informal about them (whatever "informal" means there; maybe we need a better terminology; I'm all for it), and what do we think would be useful about them if we could do this kind of thing in computer systems?

The other thing you can do with this kind of study is to make some case studies: look for the same kind of techniques in many different situations, try to see what is common about them, try to characterize a particular kind of phenomenon in depth. We're looking at problem solving. I was talking to Don for a minute, and he had a suggestion for at least one aspect of what's going on. This doesn't cover your thing, David, I don't think.

Littman: Which thing?

Shultis: Well, maybe it does. Um, sure it does! Computer-assisted problem-solving, is one of the things we're talking about. We want to extend the computer to help us solve problems, and help us with the activities that we do when we're solving problems, whatever they are.

Littman: Wait, that's my thing, or that's what this doesn't cover?

Shultis: I retract that statement altogether. I think we can include all of the things we've been talking about under that one roof. In a sense, what we're complaining about is that in order to use computers, we have to make our problem-solving methods conform to those of formal description, specification, and so forth. What we'd like to do is make the computer's support conform to our problem-solving, so that computers help us to solve problems. And David Fisher, in characteristic form, has done something he constantly accuses me of doing—of saying, after everything is all said and done, that that's what I was saying all along. But here [*putting up DAF slide stating that "Humans and Computers are Problem-Solving Devices"*] he has the evidence. [*Laughter.*]

Good: An invariant!

Shultis: And maybe that's a better title for a follow-on: Computer-assisted problem-solving. But there are these two sorts of things we can do

in the short term: one is to try to develop a taxonomy and the other is case studies of specific phenomena.

Standish: I think that coming to a conclusion is probably one of the hardest things to do in one of these workshop exercises. I have a feeling that people came here wanting to share and find commonality, and were interested in what the others had to offer, but they all came with agendas of their own life and work, partly wanting to share and offer that to others, and curious about what others are doing. But, as you go around the room, and without naming names, you could find philosophers who want to do beautiful new logic systems that were parsimonious and explained all the conundrums in that profession, and did well at it; and cognitive scientists who want to explain the bases for human cognition, with some sensitivity to the philosophical problems of logic, and so forth; and then there are people who want to synthesize programs, sometimes explaining things in English to the machine, and sometimes choosing from a parameterized space; and there are people who want to verify that programs did things to satisfy constraints on engineering situations; and then I think that there are Edisonian inventors, who would like to do something neat and wonderful and powerful, like before there were movies invent movies, and before there were victrolas invent victrolas, and before there were light bulbs invent light bulbs, and make the world new and different in some very powerful way, and they wanted inspirations about where the Edisonian inventiveness should be channeled to get some new domain of technological power; and then there are the modelers, who... well, I don't know if I should keep going on, but a lot of people had these different agendas, and it isn't clear that you can take one thing, particularly this new and latest thing, and say, "Well, our real purpose was to get better machine assistants." Wait a minute! How come we didn't get Rich and Shrobe and Waters here to talk about the programmer's assistant, or something, if that was our real agenda? A cognitive human assistant in the computer wasn't even represented in our set of talks, and so how come all of a sudden that became the noble goal that would unify us? So I think it's very hard, under these different agendas, to find a common focus.

Shultis: Maybe what I can take from that is that we've come here as an event, to share our ideas and thoughts about these things, but we're going to pursue different courses and we've got different goals that motivate us, individually. There are sort of two models of a workshop like this. One says you bring people from lots of different areas, and you get them to come together and talk about something and then they go their own ways again. The other model says that there are some things that we're all trying to talk about together, and maybe we can continue to help each other over a period of time to wrestle with those things. Maybe it's not that we have some common goal that motivates us, but a common set of interests, or problems, that could cause us to gather again. I think that you're right, that it's probably more like the latter.

Standish: Yeah, it was disparate. I left out several things.

Fisher: It strikes me that you can come to a conclusion that there's a commonality, and you'd like to work toward some common goals without being able to formulate what those goals are, and even admit that probably there isn't any one property that you know about those goals that could be written down so that everyone would agree. That doesn't bother me. In fact, one of the questions I wanted to ask is: regardless of what your agenda was when you came or your views on things when you came, and regardless of what they are now, how many here feel that they've had some changes in their views of this subject matter? I certainly have, and to me that's the test of whether a workshop was useful. *[General agreement, assent.]*

Shultis: Well, it's past 12:30 and we're supposed to break up. I'd just like to remind everyone that there is a steering committee for a follow-on after lunch, and if enough people show up, then we've got a committee! Just to talk about what sorts of things we might do, who might do what, and see who's there, and start to get a group of people together to plan the next one, because one of the things that helps a meeting be a good success is all of the people who attend it and contribute to it. I'd like to personally thank all of you for coming, and for making this a very interesting and stimulating experience. *[Note: At the steering committee meeting it was decided that there should be a follow-on conference in about 18 months, and that David Littman would take on responsibility for organizing, with David Fisher acting as figurehead. —Eds.]*

Standish: Well, on behalf of us I'd like to thank you for organizing it, and coordinating all the different strands that brought us here.

Fisher: Yeah, I thought it was a terrific group of people, and I just thought that the interdisciplinary aspect of it was really quite refreshing.

?: ... one of the best ever.

**Contributions to
Arcadia Consortium
and other
DARPA Activities**

**sponsored under
Languages Beyond Ada and Lisp**

**Incremental Systems Corporation
319 South Craig Street
Pittsburgh, Pennsylvania**

IRIS as an Enabling Mechanism for the Use of Formal Methods

**Deborah A. Baker
David A. Fisher
Jonathan C. Shultis**

November, 1989

**Incremental Systems Corporation
319 South Craig Street
Pittsburgh, Pennsylvania 15213**

**This work was supported in part by the Department of Defense, Advanced Research
Projects Agency, Order No. 5057, monitored by the Department of the Navy, Space
and Naval Warfare Systems Command, under contract N00039-85-C-0126**

IRIS as an Enabling Mechanism for the Use of Formal Methods

Deborah A. Baker, David A. Fisher, Jonathan C. Shultis

Incremental Systems Corporation
319 South Craig Street
Pittsburgh Pennsylvania 15213

October 19, 1989

1 Introduction

IRIS¹ is a semantically based system for representing information that can be viewed as an utterance in some formal language [T⁺90, WCW90, BFS88]. It serves as a medium of information exchange among a variety of tools of a software development environment. It is an extensible and open-ended system with respect to the information it can capture and represent. This paper emphasizes the role of IRIS in providing reusable infrastructure for the use of formal methods in software development. In Sections 2 and 3, IRIS is viewed syntactically and semantically, respectively. Section 4 describes the IRIS attribute system, which captures and maintains arbitrary user-defined information. Other special characteristics of IRIS that make it suitable for the support of formal methods are presented in Section 5. In Section 6, the relevance of IRIS to the workshop themes is discussed explicitly, and some examples are given of its current and proposed use.

2 A Syntactic View of IRIS

At a syntactic level, an IRIS instance is a tree. An IRIS tree represents an utterance consisting of *references* and *applications*. Corresponding to this, an IRIS tree is composed of two kinds of nodes: *reference nodes* and *application nodes*. For example, the expression $f(x, g(y, z))$ consists of references to entities named f , x , g , y , and z and applications of f and g . The corresponding IRIS tree is shown in Figure 1. Circles depict application nodes and squares depict reference nodes.

The first child of an application node is its *operator*. The operator identifies an *operation* which is applied to the remaining children, which are called *arguments* (actual parameters). Frequently, the operator is a reference to the declaration of a named operation, but it can be any operation-valued expression represented as an IRIS tree.

To avoid clutter, IRIS trees are often drawn in the style shown in Figure 2 where the name of the operation is shown next to the application node. In this case it is understood that the operator is a single reference node referring to the named operation and is not shown.

¹IRIS is an acronym for Internal Representation Including Semantics. Iris was the Greek goddess of the rainbow, and messenger of the gods.

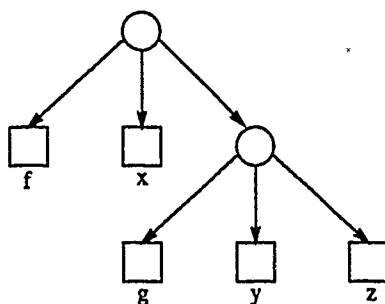


Figure 1: An IRIS Tree

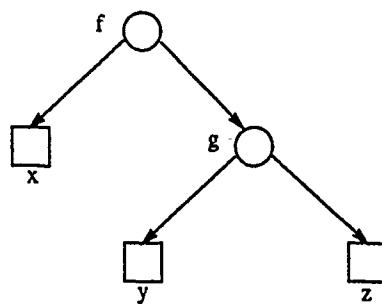


Figure 2: A Simplified IRIS Tree

In IRIS, an *entity* is anything that can be defined, computed, or named. Entities include such things as constants, variables, types, functions, packages, exceptions, statements, declarations, axiomatic specifications, operational descriptions and proofs. Some entities can be used as operations. Every operation has an associated *signature* which specifies the number and type of the arguments that an application of it requires and the type that the application yields.

Some operations, called *declarative operations*, associate identifiers with entities. An expression whose primary operator identifies a declarative operation is called a *declarative expression* or *declaration*. A declaration is an actual association between an identifier and an entity. The scope and visibility of a declaration is determined by other IRIS operations, consistent with the definition of the language of an utterance.

IRIS is similar to commonly used internal forms for (first-order) abstract syntax, but differs from them in that it demotes the operator from a nonterminal class to a distinguished subtree. This frees IRIS from any particular choice of operators, thus contributing to language independence.

The graphs used for representing terms in combinator reduction systems share with IRIS the use of a single nonterminal class representing application of the leftmost child to the remaining children. However, every operator in such a system is either a reference to a primitive combinator, or a composition of primitive combinators. In IRIS, such primitive combinators could be represented as unresolved reference nodes, but the recommended procedure in IRIS would be to define the combinators in terms of one another, and have each "primitive" combinator reference resolved to its description. For the formalist, perhaps the appropriate metaphor for IRIS is "pure combinatory calculus", where the "purity" is that of pure λ -calculus, which has no constants.

3 A Semantic View of IRIS

There are two semantic aspects of IRIS, depending on whether one is interested in IRIS as a means of representing semantic information, or as the subject of semantic investigation. This paper emphasizes the former, because we are interested in how IRIS can be used as an integration technology. The two issues are not completely independent, however, and at least a passing acquaintance with the semantic interpretation of IRIS is prerequisite to understanding how it can be used to represent semantic information.

IRIS semantics consists in interpreting each node as an object and each attribute as a relationship among the objects in a manner that is consistent with the description of those objects and

relationships given by the syntactic IRIS. The most important constraint given by the syntactic IRIS is that the object which interprets each node in fact be equivalent to the composition which interprets its description. Briefly, IRIS semantics generalizes the notion of algebraic homomorphism to the non-well-founded case.

Now, a tool which processes IRIS descriptions can glean semantic information about an object either by giving an interpretation to the object itself, or by computing it from the IRIS description of the object and its attributes. At some point, of course, most tools will have to give some external interpretation to at least some aspects of some of the operators in an IRIS tree, but the determination of which ones, and what information is required about them, is determined entirely by the tool, and not by anything in the IRIS itself.

Thus, IRIS simplifies tool building by allowing tool builders to focus on those operations and properties of operations that are inherently important to the functionality of the tool and to ignore other operations and properties of operations. For instance, a semantic analyzer that is part of a front end tool will be interested in, and give interpretation to, only a subset of the operations, typically those for scope and declaration.

Moreover, the fact that tools only depend on (partial) interpretation of a subset of operations means that they can be reused for processing IRIS descriptions in any application where the subset recognized by the tool is present. As an example, there are certain features shared by many languages. IRIS facilitates the development of a reusable infrastructure consisting of a set of tools based on operations common to the description of many languages. Actually finding a set of operations that can be shared by many language descriptions is a design problem that is facilitated by the use of IRIS.

Perhaps the most important semantic characteristic of IRIS is its primitivelessness. For instance, in representing static semantics, the signatures of all operations are represented in a uniform way in IRIS, instead of being represented in some other form which is peculiar to a tool or set of tools. The point is that semantic information about IRIS objects (in this case, types) can (and should) be represented in the IRIS itself.

In a sense, IRIS is to applicative systems what graphs are to categories. Instead of representing compositions in arbitrary categories, it represents applications in arbitrary applicative domains. (Incidentally, category theory itself makes inherent and necessary use of application; without it, the composition $f \circ g$ cannot be expressed, because it requires the application of composition to f and g . Thus application is prior to composition, despite some categorists' apparent distaste for it.)

As in category theory, any operation can be described as a composition which may reveal information about it, and every operation has a signature which constrains its composability. Unlike category theory, the form and content of signatures is not dictated by IRIS; in particular, they are not restricted to first-order ground terms, but can be arbitrary descriptions allowing unlimited degrees of polymorphism and subtyping.

To carry the analogy further, languages in IRIS correspond to categories, IRIS instances correspond to arrows (described by composition), and IRIS attributes correspond to arbitrary mappings from a category. When attribute values are themselves represented in IRIS, certain constraints are required to make the attribute mapping sensible, corresponding to the notion of functor. We hope that this sketch of a metatheory of IRIS helps to give some insight into its scope and nature, but we shall not pursue it further here.

4 The IRIS Attribute System

The IRIS attribute system is a mechanism for attaching user-defined information to IRIS nodes. The attribute system permits any number of attributes to be associated with each node.

The IRIS attribute system is unusual in that values for a given attribute and IRIS tree are allocated and managed separately from those of other trees and other attributes. This means that each tool can have local attributes without knowledge of other tools and without imposing space or time costs on other tools. Finally, by physically partitioning the attributes by attribute rather than by node, those that are shared among tools can be stored in the library and loaded only by tools that reference or modify them. It should be noted that, although IRIS attributes are stored independently in this fashion, attributes that are frequently used together may be colocated for efficiency.

Formally, the attribute system can be modeled as a set of triples

$$< IRIS\ node, attribute, attribute\ value >$$

where there is at most one attribute value (at a given time) for each IRIS node and attribute combination. The attribute system does not associate any intrinsic properties with nodes except their distinctness (i.e. the nodes constitute a set). Attribute values can be of any type appropriate to the tools using them. The IRIS system does not impose restrictions on the types of attribute values. Since tools can add attributes, the IRIS system has no inherent bias towards any particular kind of attributes, such as those useful for compilation.

5 Special Characteristics of IRIS

The process of *resolution* consists of replacing unresolved references by resolved references. An IRIS tree may have all unresolved references; such a tree is also called a *parse tree*. A tree in which all references are resolved (that is, the referent of every reference node is another IRIS node) is called a *resolved IRIS tree*. It is also possible to have a *partially-resolved IRIS tree* in which there are both kinds of references.

A fully resolved IRIS description is complete in the sense that everything that is used in the description is itself described; no part of the description depends on externally defined "primitives". A tree that is fully overload resolved is the quintessential IRIS tree; some tools may expect and require their input to be fully resolved IRIS trees. Inconsistent, incomplete and otherwise incorrect utterances are normally represented by IRIS trees in which all nodes that are complete, correct and unambiguous have been resolved as far as possible.

Another characteristic of IRIS is open-endedness. It is not possible, of course, to foresee the specific information needed by all tools, extant and future. Even if it were possible, it would be undesirable to impose the cost of such complexity and volume of information on all tools. While the actual IRIS tree form contains no tool specific information, the use of the attribute system allows inclusion of any arbitrary tool specific information at no cost to tools which do not use particular attributes.

An important feature of IRIS is language independence, a consequence of primitiveness. There are of necessity a variety of languages used during the software development process. Identical tools can be used to process utterances in any of the various languages when the languages

are represented in the same internal form. Furthermore, use of IRIS can promote experimentation on new languages and new language concepts in a research and development environment.

6 Relevance of IRIS to the Formal Methods Workshop Topics

The utility of formal methods for large-scale software development can be expected to increase steadily with improvements in engineering technology and foundations. Unfortunately, there is currently no clear path for the transfer of formal methods from research to industrial practice.

It is clear that no existing or foreseeable suite of formal methods and tools can cover the entire range of software development issues and tasks. Practitioners must therefore be able to apply formal methods selectively and partially. However, many existing and proposed systems based on formal methods assume a more global perspective, and can be applied only to entire projects.

Another aspect of the problem is that many existing systems do not support sharing of common technology, or the migration and integration of new or foreign technology. A few projects, like Edinburgh LF [HHP87], ERGO [LPRS88], and Larch [GHW85], and the somewhat less instantiated work on *institutions* [GB85] address some aspects of the integration problem, but only within limited domains. For example, institutions promote sharing and reuse by abstracting the common features and structure of logical systems and permitting specific logics to be composed from common components. The motivation behind ELF is similar. A similar kind of capability is needed that cuts across the boundaries between logics, languages, implementation regimes, paradigms, and lifecycle tasks, as well as the boundaries between formal and informal methods — in short, a genuine basis for integration.

IRIS is an internal form which provides a basis for the integration of formal systems, methods, and tools. At the core of IRIS is the notion, common to all formal systems, that there is some domain of objects (individuals) of interest, and these are represented by IRIS *nodes*. Another underlying assumption of IRIS is that any description of an object is either a direct reference to it, or a composition consisting in the application of an operation to some other objects.

Fully resolved IRIS descriptions have major advantages for any kind of processing, as a single, uniform algorithm can be used to process the entire description, with no need to relate things represented in IRIS to external, non-IRIS information. On the other hand, incomplete IRIS descriptions are allowed, and simply represent partial information. IRIS descriptions that cannot be completed consistently are also allowed, and these simply represent incorrect or inconsistent information. Such descriptions are required by formal tools that deal directly with intensional objects such as programs, specifications, and proofs.

The open-endedness of the IRIS attribute system allows IRIS attributes to be processed by IRIS-based tools. For example, if a predicate transformer attribute uses IRIS representations, then the predicate transformers can be checked, analyzed, transformed, translated, displayed, and so forth using the same algorithms that are used to process the programs they attribute.

As a direct consequence of the partitioning of attributes by attribute as well as by IRIS tree, any number of attributes, of arbitrary value type, can be contributed by tools processing an IRIS description. Similarly, tools can share and reuse attributes contributed and modified by other tools. If one looks at the sets of attributes which are used and produced by any tool as a kind of "signature" for the tool, it is easy to see that IRIS-based tool composition is far more flexible than ordinary pipeline or functional composition. The open-endedness and flexibility of this tool

composition and partitioning of attributes provide the crucial path for integrating new technology into existing systems. Adoption of IRIS as a *de facto* standard in the formal methods and tools community would greatly enhance sharing and reuse, and facilitate the gradual transfer of formal methods to practice.

IRIS has been used by Incremental Systems as the internal form for a compiler for the Ada programming language. The Arcadia Consortium has adopted IRIS as the common internal form for its tools, including front end tools, analyzers and interpreters. IRIS is being evaluated for use in the STARS (Software Technology for Adaptable, Reliable Systems) program. Computational Logic, Inc, is using IRIS in its study of a provable subset of Ada [CSS88]. There is some interest in using IRIS to support ANNA annotations and the ANNA toolset [LvH85].

Acknowledgements

This work was supported in part by the Defense Advanced Research Projects Agency (Arpa Order 5057) monitored by the Department of the Navy, Space and Naval Warfare Systems Command under contract N00039-85-C-0126 and by the Defense Advanced Research Projects Agency (Arpa Order 6487-1) under contract MDA972-88-C-0076.

We thank our Incremental Systems colleagues David Mundie and Frank Tadman, both of whom were instrumental in the development of IRIS.

We gratefully acknowledge the Arcadia Consortium whose members have encouraged this work with their reviews, comments and suggestions.

References

- [BFS88] D. A. Baker, D. A. Fisher, and J. C. Shultis. The Gardens of IRIS. Technical report, Incremental Systems Corporation, 1988.
- [CSS88] D. Craigen, M. Saaltink, and M. K. Smith. The nanoAVA Definition. Technical Report 21, Computational Logic, Inc., June 1988.
- [GB85] J. A. Goguen and R. M. Burstall. Institutions: Abstract Model Theory for Computer Science. Technical Report CSLI-85-30, Center for the Study of Language and Information, Stanford, CA, August 1985.
- [GHW85] J. V. Guttag, J. J. Horning, and J. M. Wing. Larch in Five Easy Pieces. Technical Report SRC Report No. 5, DEC SRC, Palo Alto, July 1985.
- [HHP87] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 194-204. IEEE, June 1987.
- [LPRS88] P. Lee, F. Pfenning, G. Rollins, and W. Scherlis. The Ergo Support System: An Integrated Set of Tools for Prototyping Integrated Environments. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 25-34. ACM Press, November 1988.

- [LvH85] D. C. Luckham and F. W. von Henke. An overview of ANNA, a specification language for Ada. *IEEE Software*, 2(9):9-22, March 1985.
- [T+90] F. P. Tadman et al. *IRIS Reference Manual*. Incremental Systems Corporation, May 1990.
- [WCW90] J. C. Wileden, L. A. Clarke, and A. L. Wolf. A Comparative Evaluation of Object Definition Techniques for Large Prototype Systems. *ACM Transactions on Programming Languages and Systems*, to appear 1990.

Reference Manual for Iris-Ada

Deborah A. Baker and Frank P. Tadman, editors

**Incremental Systems Corporation
319 South Craig Street
Pittsburgh, PA 15213
(412) 621-8888**

16 March 1990

Version 1.1

**Incremental Systems Corporation Technical Report 891001
incorporates Arcadia Documents IncSys-89-01 and IncSys-89-02 with revisions**

© 1990 Incremental Systems Corporation. All rights reserved.

Permission to copy all or part of these materials is granted, provided that the copies are not made or distributed for commercial advantage and that the Incremental Systems Corporation copyright notice set forth above appears on each copy.

Approved for Public Release. Distribution Unlimited

Acknowledgments

The concept of Iris originated with David A. Fisher of Incremental Systems Corporation. The first use of Iris was in portions of the Incremental Systems Ada compiler developed by David Fisher, David A. Mundie, and Richard M. Weatherly beginning in December 1984. The first presentation of Iris outside Incremental Systems was at an Arcadia consortium meeting held in December 1984. Iris concepts, and a grammar and set of specifications were further refined and used in an Ada compiler effort finished in December 1988. A standard Iris-Ada instantiation was agreed upon for the Arcadia consortium in October 1989.

The declarations contained in this technical report were developed primarily by the staff of Incremental Systems, including Deborah Baker, David Mundie, Jonathan Shultis and Frank Tadmán. The grammar contained in this report was developed primarily by Deborah Baker, David Fisher and Edward Pervin. In addition, we would like to acknowledge significant contributions made by William Easton (Peregrine Systems) and by Arcadia members Kari Forester (University of Massachusetts), Stephen Sykes (TRW), Peri Tarr (University of Massachusetts), Izhar Shy (University of California at Irvine), Richard Taylor (University of California at Irvine), Steven Zeil (Old Dominion University), and Alexander Wolf (ATT), who have pointed out errors and made many constructive suggestions for the improvement of the presentation. Arcadia members Lori Clarke (University of Massachusetts) and Richard Taylor (University of California at Irvine) have been instrumental in facilitating the standardization of the Iris instantiation for Ada. Arcadia member Frank Belz (TRW) has been a source of encouragement for our Iris work. We have learned much from users of Iris-Ada including Michael Smith (Computational Logic, Inc.) and Craig Snyder (University of California at Irvine). We thank Kari Forester, Peri Tarr and Alan Kaplan of the University of Massachusetts for their proposed naming conventions for IRIS-Ada; we plan to incorporate standard naming conventions in a subsequent version.

This work was supported in part by the Defense Advanced Research Projects Agency (Arpa Order 5057) monitored by the Department of the Navy, Space and Naval Warfare Systems Command under contract N00039-85-C-0126 and by the Defense Advanced Research Projects Agency (Arpa Order 6487-1) under contract MDA972-88-C-0076.

Contents

1	Introduction	1
2	Reading the Ada Augmented Declarations	2
2.1	Lexical Extensions	3
2.2	Visibility	3
2.3	Types	3
2.3.1	Type Computation	4
2.3.2	Formal Parameter Modes	6
2.3.3	The Type Hierarchy	6
2.3.4	Some Special Types, Constants and Operations	8
2.4	Conventions	8
2.4.1	Declarations	8
2.4.2	Units	9
2.4.3	Iris Tree Superstructure	9
3	Ada Augmented Declarations	11
3.1	Functions	11
3.2	Declarations	11
3.3	Types	12
3.4	Metatypes	13
3.5	Formal Parameters	13
3.6	Scalar	13
3.7	Discrete Types	14
3.8	Integer Types	17
3.9	Universal Real	17
3.10	Floating Point Types	18
3.11	Fixed Point Types	18
3.12	Lists	19
3.13	Arrays	19
3.14	Access Types	20
3.15	Records	21
3.16	Sequential Control	21
3.17	Exception Handling	22
3.18	Packages	22
3.19	Duration, Task, and Entry	23
3.20	Separate Compilation	24
3.21	Generics	24
3.22	Pragmas	24
3.23	Representation Specifications	25

A	Reference Manual for the Iris-Ada Grammar	27
A.1	Introduction	27
A.2	Reading Grammars	28
A.2.1	The Form of a Grammar	28
A.2.2	Metasymbols Used in Specifying a Grammar	28
A.2.3	Details of the Grammar	29
A.2.4	Contexts	30
A.2.5	Metagrammar Reserved Words	31
A.2.6	Nonterminal Names	31
A.2.7	Descriptions of Abstract Syntax	32
A.2.8	Lexical Issues and Literal Kinds	32
A.2.9	Actions	33
A.2.10	Precedence	35
A.3	The Metagrammar	37
A.4	Iris-Ada Grammar	38
A.5	Appendix B. Updated Iris-Ada Grammar	46
A.6	Appendix C. Updated Ada Declarations	53

References

- [Ada83] U.S. Department of Defense, Ada Joint Program Office, ANSI/MIL-STD-1815A-1983. *Ada Programming Language Reference Manual*, 1983.
- [Bak89] D. A. Baker, editor. IRIS Glossary. Technical Report 891003, Incremental Systems Corporation, 1989.

1 Introduction

Iris is a language and machine independent form for representing the sentences of any formal language. Iris simplifies the development of tool components for the analysis, query, and synthesis of programs. It is especially useful as an integrating mechanism for program development and maintenance environments that incrementally process large, long-lived, or continuously changing software applications. Iris was developed at Incremental Systems in cooperation with the Arcadia Consortium. This document provides the Iris specification for the Iris representation of the Ada programming language as used by the Arcadia Consortium, in the STARS program, and at Incremental Systems.

Ada is a modern programming language that requires compile-time analyses to discover programming errors that traditionally were detected only during execution. Additional analyses and transformations are required for optimization to generate code which satisfies the real-time constraints of its applications. The simple structure and open-endedness of Iris allows algorithms which do these analyses and transformations to be smaller and simpler and provides a framework in which their results can be shared by other environment tools.

The Iris representation is tree-structured with only two kinds of nodes: reference and application. Each Iris tree represents an expression composed of applications and references. Reference nodes are interpreted as references to declarations that appear elsewhere within the Iris structure. Application nodes are interpreted as the application of an operation to a sequence of arguments. The operation is always the value of the operator (which is the leftmost subtree) of the application node. The arguments are the values of the remaining subtrees. If the reference nodes of Iris are viewed as leaves (terminals), then the Iris representation can also be viewed as an abstract syntax tree with the application nodes acting as nonterminals. Each reference node, however, contains a reference to a declaration which is itself an application node appearing earlier in (a preorder walk of) the Iris structure.

Iris is unique in that all operations are described within its own structure. This means that individual tools need to recognize and provide special case processing for only those operations that relate directly to the functionality of the tool. For example, the overload resolution portion of a semantic analyzer needs to recognize only those operations that are declaration, scope, or type valued. It does not have to distinguish between control structures and arithmetic operations. This means that individual tools and tool components are often significantly smaller than with traditional representations. Also, because tools process most operations based on the internal definition of the operation rather than by explicit reference, the language being represented can often be modified or extended without modification to the tools.

Iris is also a higher order system in that it provides full support for computed operations at any level. A computed operation may appear either in place at the point of its application (i.e., as another application node which is the operator of the application) or as the value of a declaration which is referenced at the point of call (i.e., as a reference node which is the operator of the application). The combination of internal and higher order specification means Iris can be used to represent any formal language and that Iris based tools can be reconfigured for multiple languages and other changing requirements with little or no change to their components.

To specify the representation of any language L , two things are needed: a *grammar* and a set of *L -augmented declarations*. The grammar describes the correspondence between the concrete

also necessary to decide what the abstract syntax for each feature would be. A formal specification of the abstract syntax is given in Appendix A. Finally, certain conventions were adopted in the design of the abstract syntax to ensure that operations with similar functionality would have similar forms as an aid to understanding and to facilitate processing them uniformly. These conventions are discussed in Section 2.4.

2.1 Lexical Extensions

It was necessary to extend the set of legal lexical units to include identifiers for built-in features of the Ada language while insuring that their names could not be referenced in (non generalized) Ada programs. Thus, most of the declarations in the Ada augmented declarations package have names that begin with a tilde (~). Because the ~ names are illegal Ada identifiers, they accomplish two important goals. First, those declarations with ~ names are hidden from the Ada programmer who is not allowed to make use of them in writing Ada programs. Second, user-defined Ada subprogram, type, etc. declarations cannot mask or hide the declarations with ~ names. The remaining operations, type, and constants defined in the Ada augmented declarations package have legal Ada names. These entities are those defined in the Ada package *standard*. They must be visible for use in writing Ada programs.

2.2 Visibility

Two extensions to the Ada visibility rules must be made to understand or process the Ada augmented declarations package. Each operation, type and constant declared in that package is visible throughout the entire package, except in its own specification and sometimes in its body². This extension allows forward reference in the Ada augmented declarations package. This extension is required for the earlier declarations in the Ada augmented declarations package because every designator is declared as a composition that references other declarations, and no designator is predefined (i.e., defined outside the Ada augmented declarations package).

In Ada, formal parameters and function return types may not reference other parts of the specification. This Ada restriction has been lifted so that formal parameters and function return types may reference earlier formal parameters. This generalization is necessary to formalize the dependencies among parameters of Ada's built-in operations.

2.3 Types

The Ada type system has been generalized by removing several restrictions that Ada places on its users but not on itself, and by generalizing the Ada concept of type to include all concepts of the language. Although these changes are straightforward and involve only concepts already in Ada, they have a dramatic impact on the power of the language.

The first generalization is to view all Ada concepts as Ada types. Acceptance of this view means that record, array, access, exception, function, statement, package, type, entry, generic, universal integer, and so forth, are types.

²The restrictions on visibility within an entity's own specification or body are the normal restrictions imposed by Ada.

	<i>Operator</i>	<i>Operation</i>	<i>Operand type</i>	<i>Result type</i>
(1)	*	multiplication	any integer type	same integer type
(2)	*	multiplication	any floating point type	same floating point type

	<i>Operator</i>	<i>Operation</i>	<i>Left operand type</i>	<i>Right operand type</i>	<i>Result type</i>
(3)	*	multiplication	any fixed point type	INTEGER	same as left
(4)	*	multiplication	INTEGER	any fixed point type	same as right
(5)	*	multiplication	any fixed point type	any fixed point type	<i>universal_fixed</i>
(6)	*	multiplication	<i>universal_real</i>	<i>universal_integer</i>	<i>universal_real</i>
(7)	*	multiplication	<i>universal_integer</i>	<i>universal_real</i>	<i>universal_real</i>

Figure 1: * operators from the Ada Language Reference Manual Sections 4.5.5 and 4.10

```

function "*" (left, right: ~univ_integer) return ~univ_integer;
function "*" (left, right: ~univ_float) return ~univ_float;
function "*" (left: ~univ_fixed; right: ~nonderivable(integer)) return ~univ_fixed;
function "*" (left: ~nonderivable(integer); right: ~univ_fixed) return ~univ_fixed;
function "*" (left: ~independent(~univ_fixed); right: ~independent(~univ_fixed))
  return ~nonderivable(~univ_fixed);
function "*" (left: ~nonderivable(~univ_real); right: ~nonderivable(~univ_integer))
  return ~nonderivable(~univ_real);
function "*" (left: ~nonderivable(~univ_integer); right: ~nonderivable(~univ_real))
  return ~nonderivable(~univ_real);

```

Figure 2: * operators from the Iris-Ada declarations

used to mark the type of any formal parameter whose type cannot be changed even if the operation is inherited (with respect to other parameters). For instance, there are seven "*" operators defined in Sections 4.5.5 and 4.10 of the Ada Language Reference Manual [Ada83]; they are reproduced in Figure 1. In Figure 2, the Iris-Ada specification for each of these operators is given. Focus on the third and fourth operators in Figure 1. These are fixed point operators defined in Section 4.5.5 of [Ada83]. Each has a formal parameter that must be of the predefined type integer. In Figure 2 the corresponding formal parameters are defined to be ~*nonderivable(integer)*. The function ~*nonderivable* disallows the usually legal type derivation. The fifth, sixth and seventh "*" operators also require the use of ~*nonderivable* in their declarations.

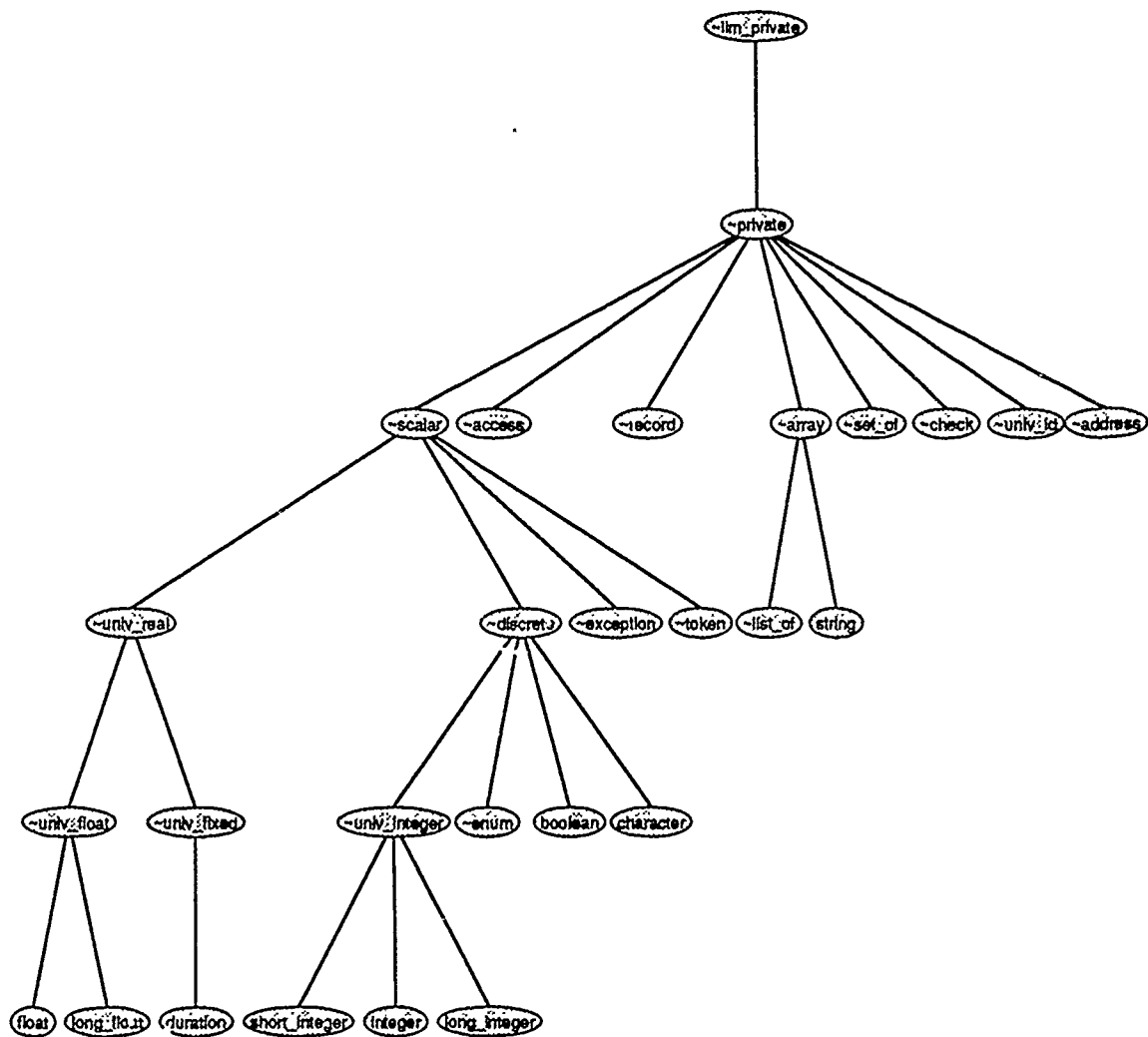


Figure 4: The type hierarchy of the Ada augmented declarations, Part 2.

`function ~incomplete_spec(name: ~token; t: ~metatype(~type)) return decl_item;` This is the standard Ada operation to enable recursive type definitions.

`function ~instantiation_decl(name: ~token; t: ~type; value: t) return ~decl_item;` This is a variation of `~vis_decl` that is used for generic instantiations in Ada.

2.4.2 Units

There are several kinds of library units, namely `~lib_spec`, `~lib_body` and `~lib_subunit`. Each of these has three parameters, the first of which is the parent context, the second of which is the Ada context and the third of which is the specification or body, as appropriate. See Figure~6.

2.4.3 Iris Tree Superstructure

Figure 6 shows the Iris tree superstructure for the sample Ada program shown in Figure 5. The IRIS-Ada parser produces a forest of trees, one for every subunit. A separate context constructor then inserts the interunit references, shown in Figure 6 as curved lines.

```
with Q; separate (P.S)
package body V is ...end V;
separate (P)
function S is ...
package body V is separate ; ...end S;
with C;
package body P is ...
function S is separate ; ...end P;
with A, B;
package P is ...end P;
```

Figure 5: Sample Ada program structure.

3 Ada Augmented Declarations

language *ada*

package *standard* is

```
-- Copyright © 1990 by Incremental Systems Corporation. All rights reserved.
-- Permission to copy all or part of these materials is granted, provided that
-- the copies are not made or distributed for commercial advantage and that the
-- Incremental Systems Corporation copyright notice set forth here appears on
-- each copy.
-- This work was supported in part by the Defense Advanced Research Projects
-- Agency (Arpa Order 5057) monitored by the Department of the Navy, Space and
-- Naval Warfare Systems Command under contract N00039-85-C-0126 and by the
-- Defense Advanced Research Projects Agency (Arpa Order 6487-1) under
-- contract MDA972-88-C-0076.
```

3.1 Functions

```
type ~function(
    fpl: ~quote ~list_of(~fp_item);
    rt: ~type) is private;

function ~body(
    dl: ~quote ~list_of(~decl_item);
    sl: ~quote ~list_of(~statement)) return ◇;
-- return type depends on the context in which called
-- return type can be any Ada return type, ~statement, ~package or ~generic

function ~return(x: ◇) return ~statement;
-- x is restricted to the return type of the function in which it is called.

function ~return return ~statement;
-- may be called only within procedures and accept statements
```

3.2 Declarations

```
type ~decl_item is private;
type ~token is new ~scalar;           -- a nonoverload resolved designator

function ~spec(
    name: ~token;
    t: ~type) return ~decl_item;

function ~preinit_decl(                -- initialized constant declaration including enumeration items
    name: ~token;
    t: ~type) return ~decl_item;

function ~decl(                        -- value evaluated at elaboration of declaration
    -- ~decl is a composition of ~define and ~qual_exp where ~define is:
    -- function ~define(name: ~token; value: ◇) return ~decl_item
    name: ~token;
```

```

function "="(left, right: ~private) return ~nonderivable(boolean);
-- function"/="(left, right: ~private) return ~nonderivable(boolean);
-- must be automatically inserted after every declaration of "="
function ":="(x: out ~private; y: ~type_of(x)) return ~statement;

```

3.4 Metatypes

```

type ~metatype(t: ~type) is private;
-- the metatype of t
function ~new_type(t: ~type) return ~type;
function ~qual_exp(t: ~type; x: t) return t;
function ~type_convert(t: ~type; x: ◇) return t;
-- type convert is implicit in Ada function form syntax
function ~type_of(x: ~quote ◇) return ~type;
function ~base_of(t: ~type) return ~type;
function ~independent(t: ~type) return ~type_of(t);
-- allows parameters of the same type name to be independently
-- derived, not available to Ada users
function ~nonderivable(t: ~type) return ~type_of(t);

```

3.5 Formal Parameters

```

type ~fp_item is private;
function ~in_mode(t: ~type) return ~type_of(t);
function ~in_out_mode(t: ~type) return ~type_of(t);
function ~out_mode(t: ~type) return ~type_of(t);
function ~value_mode(t: ~type) return ~type_of(~in_out_mode(t));
-- corresponding value of type t is copied to a local variable of that same type
function ~quote_mode(t: ~type) return ~type_of(t);
function ~fp_spec(name: ~token; t: ~type) return ~fp_item;
function ~default(t: ~type; value: ~quote t) return ~type_of(t);
-- used for default formal parameters, variable initial values, and
-- default field values
function ~named_exp(name: ~token; x: ◇) return ~type_of(x);
-- used for named actual parameters and aggregates

```

3.6 Scalar

```

type ~scalar is private;
function "<"(left, right: ~scalar) return ~nonderivable(boolean);
function "<="(left, right: ~scalar) return ~nonderivable(boolean);

```

type character is (

```
~nul, ~soh, ~stx, ~etx, ~eot, ~enq, ~ack, ~bel,
~bs, ~ht, ~lf, ~vt, ~ff, ~cr, ~so, ~si,
~dle, ~dc1, ~dc2, ~dc3, ~dc4, ~nak, ~syn, ~etb,
~can, ~em, ~sub, ~esc, ~fs, ~gs, ~rs, ~us,
' ', '!', '"', '#', '-', '%', '&', ''',
'(', ')', '*', '+', ',', '.', '/',
'0', '1', '2', '3', '4', '5', '6', '7',
'8', '9', ':', ';', '<', '=', '>', '?',
' ', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
'X', 'Y', 'Z', '[', '\', ']', '^', '_',
' ', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
'x', 'y', 'z', '{', '|', '}', '~', ~del );
```


3.8 Integer Types

```
type ~univ_integer is new ~discrete;
function "+"(right: ~univ_integer) return ~univ_integer;
function "-"(right: ~univ_integer) return ~univ_integer;
function ~abs(right: ~univ_integer) return ~univ_integer;
function "+"(left, right: ~univ_integer) return ~univ_integer;
function "-"(left, right: ~univ_integer) return ~univ_integer;
function "*" (left, right: ~univ_integer) return ~univ_integer;
function ~mod(left, right: ~univ_integer) return ~univ_integer;
function ~rem(left, right: ~univ_integer) return ~univ_integer;
function "**"(
    left: ~univ_integer;
    right: ~nonderivable (natural)) return ~univ_integer;
function "/"(left, right: ~univ_integer) return ~univ_integer;
-- ranges are implementation dependent
type short_integer is new ~univ_integer range -128.. 127;
type integer is new ~univ_integer range -32768 .. 32767;
type long_integer is new ~univ_integer range (-2147_483647) - 1 .. 2147_483647;
type positive is integer range 1 .. integer'last;
type natural is integer range 0 .. integer'last;
```

3.9 Universal Real

```
type ~univ_real is new ~scalar;
function "+"(right: ~univ_real) return ~univ_real;
function "-"(right: ~univ_real) return ~univ_real;
function ~abs(right: ~univ_real) return ~univ_real;
function "+"(left, right: ~univ_real) return ~univ_real;
function "-"(left, right: ~univ_real) return ~univ_real;
function "*" (
    left: ~nonderivable(~univ_real);
    right: ~nonderivable(~univ_integer)) return ~nonderivable(~univ_real);
function "**"(
    left: ~nonderivable(~univ_integer);
    right: ~nonderivable(~univ_real)) return ~nonderivable(~univ_real);
function "/" (
    left: ~nonderivable(~univ_real);
    right: ~nonderivable(~univ_integer)) return ~nonderivable(~univ_real);
function ~mantissa_of(t: ~metatype(~univ_real)) return ~univ_integer;
```

```

function "*" (
    left: ~nonderivable(integer);
    right: ~univ_fixed) return ~univ_fixed;
function "/" (
    left: ~independent(~univ_fixed);
    right: ~independent(~univ_fixed)) return ~nonderivable(~univ_fixed);
function "/" (
    left: ~univ_fixed;
    right: ~nonderivable(integer)) return ~univ_fixed;
function ~delta_constrain(
    t: ~metatype(~univ_fixed);
    del: ~static(~univ_real)) return ~metatype(~univ_fixed);
function ~delta_of(t: ~metatype(~univ_fixed)) return ~univ_fixed;
function ~aft_of(t: ~metatype(~univ_fixed)) return ~univ_integer;
function ~fore_of(t: ~metatype(~univ_fixed)) return ~univ_integer;

```

3.12 Lists

```

type ~list_of(field_type: ~list_of (~type)) is private;
function ~list(x: ~list_of (◇)) return ~type_of(x);
-- ~list is the only operation which can have a variable number of parameters.
-- It cannot be overloaded.
-- Unlike other operations, ~list must be capable of handling large numbers
-- of operands.

```

3.13 Arrays

```

type ~array(
    index_type: ~list_of (~independent(~metatype(~discrete)));
    field_type: ~type) is private;
function ~aggregate(x: ~list_of (◇)) return ~array;
-- Ada aggregates have special visibility rules and thus require special
-- case processing. The only special aggregate operations are
-- ~aggregate and ~named_exp. The arguments to an array aggregate
-- must be static if there are more than one.
function ~index_constrain(
    t: ~metatype(~array);
    constraint: ~list_of(~metatype(~discrete))) return ~type;
function ~subscript(
    x: ~array;
    i: ~type_of(x).index_type)
    return ~type_of(x).field_type;
-- ~subscript is implicit in Ada function form syntax

```

3.15 Records

```
type ~record(fpl, cl: ~quote ~list_of(~fp_item)) is private;
function ~aggregate(x: ~list_of(◇)) return ~record;
    -- Ada aggregates have special visibility rules and thus require special
    -- case processing. The only special aggregate operations are
    -- ~aggregate and ~named_exp.
function ~case(
    discrim: ~discrete;
    x: ~quote ~list_of (
        ~set_of(~static(~discrete)),
        ~list_of(~fp_item))) return ~fp_item;
```

3.16 Sequential Control

```
type ~statement is private;
function ~compound(sl: ~quote ~list_of (~statement)) return ~statement;
function ~scope(
    dl: ~quote ~list_of (~decl_item);
    sl: ~quote ~list_of (~statement)) return ~statement;
    -- Ada declare
function ~if(
    x: ~quote ~list_of (~independent(boolean),
        ~list_of (~statement))) return ~statement;
function ~case(
    discrim: ~discrete;
    x: ~quote ~list_of (
        ~set_of(~static(~discrete)),
        ~list_of (~statement))) return ~statement;
function ~loop(val: ~quote ~list_of (~statement)) return ~statement;
function ~while_loop(
    c: ~quote boolean;
    val: ~quote ~list_of (~statement)) return ~statement;
function ~for_loop(
    i: ~quote ~decl_item;
    val: ~quote ~list_of (~statement)) return ~statement;
function ~for_reverse_loop(
    i: ~quote ~decl_item;
    val: ~quote ~list_of (~statement)) return ~statement;
type ~block_label is new ~statement;
function ~exit(b: boolean := true) return ~statement;
function ~exit_with_label(bl: ~block_label; b: boolean := true) return ~statement;
type ~goto_label is new ~statement;
```

3.19 Duration, Task, and Entry

```
type duration is delta 0.000050 range 0.0..107374.1823;
-- actual values are implementation dependent
-- duration 'small corresponds to 50 microseconds,
-- duration units are seconds,
-- range is 1..24 days.

function ~delay(d: duration) return ~statement;
type ~task is new ~package;
function ~abort(x: ~list_of (~task)) return ~statement;
function ~callable_of(x: ~task) return boolean;
function ~terminated_of(x: ~task) return boolean;
type ~entry is new ~function(◇, ~statement);
type ~accept_timeout_terminate is new ~statement;
function ~count_of(e: ~entry) return ~univ_integer;
function ~accept(
    e: ~entry;
    t: ~metatype(~type_of(e));
    sl: ~quote ~list_of (~statement)) return ~statement;
function ~accept(
    e: ~entry;
    t: ~metatype(~type_of(e));
    sl: ~quote ~list_of (~statement)) return ~accept_timeout_terminate;
function ~delay(d: duration) return ~accept_timeout_terminate;
function ~terminate return ~accept_timeout_terminate;
function ~select(
    alternatives: ~quote ~list_of (
        ~independent(boolean),
        ~accept_timeout_terminate,
        ~list_of(~statement));
    else_part: ~quote ~list_of(~statement)) return ~statement;
function ~timed_entry(
    entry_call: ~quote ~statement;
    sl: ~quote ~list_of (~statement);
    d: duration;
    dsl: ~quote ~list_of (~statement)) return ~statement;
function ~cond_entry(
    entry_call: ~quote ~statement;
    sl: ~quote ~list_of (~statement);
    else_part: ~quote ~list_of (~statement)) return ~statement;
```

```

function ~call_pragma(
    p: ~function(◇, ~pragma); x: ~list_of(◇)) return ~statement;
    -- Ada requires that no errors be reported for incorrect pragma calls
function controlled(t: ~metatype(~access)) return ~pragma;
function elaborate(lu: ◇) return ~pragma;
function inline(f: ~list_of(~function)) return ~pragma;
function memory_size(x: ~univ_integer) return ~pragma;
function pack(x: ~metatype(~array)) return ~pragma;
function pack(x: ~metatype(~record)) return ~pragma;
function priority(x: ~static(integer) range 0..7) return ~pragma;
function shared(x: in out ◇) return ~pragma;
function storage_unit(x: ~univ_integer) return ~pragma;
function system_name(x: ~token) return ~pragma;
function interface(f: ~function) return ~pragma;
function list return ~pragma;
function optimize return ~pragma;

```

3.23 Representation Specifications

```

type ~address is private;
    -- the address type required in Ada package system is
    -- subtype address is ~address;
function ~address_rep_spec(x: ~quote ◇; a: ~address) return ~decl_item;
function ~length_rep_spec(
    t: ~type;
    f: ~function;
    val: ~static(~univ_real)) return ~decl_item;
function ~enum_rep_spec(t: ~type; val: ◇) return ~decl_item;
function ~record_rep_spec(
    t: ~type;
    alignment: ~static(positive);
    component: ~list_of(
        ~token,
        ~static(~univ_integer),
        ~set_of(~independent(~static(~univ_integer)))))) return ~decl_item;
function ~address_of(x: ~quote ◇) return ~address;
function ~first_bit_of(x: ~quote ◇) return ~univ_integer;
function ~last_bit_of(x: ~quote ◇) return ~univ_integer;
function ~position_of(x: ~quote ◇) return ~univ_integer;
function ~size_of(x: ~quote ◇) return ~univ_integer;

```

A Reference Manual for the Iris-Ada Grammar

A.1 Introduction

Section A.2 specifies the notation used to describe grammars. The notation is largely historical and was not intended for publication purposes. It is, however, the formal input language from which tables are generated for the Incremental Systems parser. It is also currently the only formal notation in which the correspondence between the shape of Iris trees and the concrete syntax of the Ada programming language is specified.

The grammars that we use are somewhat unusual in that they have a primary nonterminal, `exp`, from which almost every syntactic construct of the language can be derived. Two observations will help in the explanation of why this is desirable. Traditional context-free grammars specify both the syntactic constructs of the language and their syntactically-valid compositions (nestings). However, some compositions require semantic information such as type information to validate their composition (or even to overload resolve their operator) and so the composition rules cannot be completely applied until semantic analysis.

The second observation is that syntactic categories (i.e., the nonterminals of a grammar) can be reinterpreted as types, thus allowing legality of composition to be determined during semantic analysis as part of the overload resolution and type checking processes.

By eliminating composition checking from parsing, parsing becomes simpler and faster, and all checking can be performed in a uniform manner during semantic analysis. Error reporting and recovery from incorrect compositions can be delayed until semantic analysis when there is more (semantic) information available to assist the error reporting and recovery processes. A forgiving parser is also needed so that incomplete and incorrect Ada programs can be translated into an Iris representation and then be analyzed and revised directly in that form.

A grammar may contain a few other nonterminals. Most of these are nonrecursive with respect to `exp`, that is, all derivations of the nonterminal from itself involve an intermediate derivation to `exp`. These nonterminals could be eliminated by expanding each reference to them in the `exp` productions, but are used to avoid the repetition of identical subexpressions or to factor out subexpressions for readability. Other nonterminals which are recursive with respect to `exp` are also allowed. Two features of our grammar, iterative operations (see section A.2.2.1) and contexts (see section A.2.4), eliminate the need for them in most cases. Their use for other purposes unnecessarily restricts the input language, limiting the number of incorrect programs which can be parsed. Thus, we rarely use them³.

³The only such nonterminal in the Ada grammar is `if_tail`. While the concrete syntax could be written nonrecursively, our current abstract syntax notation could not describe the correct IRIS tree if we did this.

- { } Braces enclose syntax that can be repeated one or more times. The items within the braces always produce a list with the operator *~list* (though, if enclosed by square brackets, the list may have zero items).
- @ Used within braces. The braces enclose two expression lists, one before the @ and one after. The expression after the @ is used to separate multiple occurrences of the expression preceding the @.
- \$ Used within braces. The expression following the \$ is used either to separate or terminate the multiple occurrences of the expression preceding the \$.
- * **x* indicates that the context *x* (see section A.2.4) should be associated with a subexpression.
- ~ *~x* indicates that this production may be used only to match a subexpression associated with context *x* (see section A.2.4).

A.2.2.2 In the abstract syntax

- \$ Refers to the *S*ith item generated by the (productions invoked by the) concrete syntax. See Section *~A.2.3*.
- () The first item in the parentheses is the operator of the subtree and the remaining items are its arguments (i.e., standard S-expression notation).

A.2.2.3 In either the concrete or the abstract syntax

- ' Any lexical unit which is immediately preceded by a quote is interpreted as a terminal of the grammar even if it is a reserved word of the metagrammar or a metalanguage symbol.
- % Signals an action (see Section A.2.9).

A.2.3 Details of the Grammar

The grammar is applied to a source program using a standard recursive matching process. Each time a nonterminal of the grammar is encountered in a production, the parser applies one of the productions for that nonterminal to the input text. When the concrete syntax of the production is completely matched, the abstract syntax describes zero or more trees to be produced which are then passed back to the higher-level production. These trees are appended to the list of values produced so far. When the concrete syntax is completely matched, the result is a list of tree values. The abstract syntax of the higher-level production describes zero or more trees to be produced from the result of the concrete syntax.

Therefore, every item in a production can produce zero or more values. An item is a nonterminal, a keyword, a metagrammar reserved word (*id*, *id_list*, *any*, or *int_lit*), a repeated item, or an optional item. Nonterminals produce values as already described. Keywords never produce any values. Metagrammar reserved words produce a single value⁴. Repeated items always produce

⁴*Id_list* causes the replication of the value produced by the production containing it as a side effect. However, it contributes a single value to each of these replications.

```

exp ::= id_list : 15      #T (~preinit_decl $1 (~var ~2))
      | id_list : 15 ~fp  #E ~fp_spec
      | id_list : 15 ~fd  #T (~fp_spec $1 (~value $2))
      | id_list : 15 ~st  #T (~label_decl $1 %identifier ~block_label $2)

```

In a statement list, the last of these alternatives is chosen. If this construct appears in an illegal position, e.g., in

while *y*: integer loop ...

the first alternative is chosen.

A.2.5 Metagrammar Reserved Words

There are several metagrammar symbols for lexical categories.

id *id* matches any lexical unit that should be interpreted as an identifier (i.e., any token which is neither a literal nor a reserved word⁵).

id_list *id_list* matches any list (of length one or more) of identifiers separated by commas. *id_list* causes the subtree generated by the production in which it appears to be replicated for each item in the list (e.g. *x, y, z: T;* becomes *x: T; y: T; z: T;*).

any *any* matches any single token except end-of-file.

int_lit *Int_lit* matches a single integer literal.

A.2.6 Nonterminal Names

root This marks the root of the grammar being defined.

exp *Exp* matches any literal or identifier in addition to any construct specified by its productions. When *exp* matches an identifier, it is interpreted as reference to the identifier, not as a token literal. Exceptions to this rule can be specified, see the description of the *%identifier* action in Section A.2.9.

There are several ways to refer to the nonterminal *exp* in a grammar.

- *Exp* may appear explicitly, as in this production from the Ada grammar:

```
apl ::= { exp }, #N.
```

- *Exp* may be referred to with a non-zero positive integer. For example, in the Ada grammar there is a production:

```
exp ::= 70 + 75 #E +.
```

This usage also associates a precedence (see Section A.2.10) with the occurrence of *exp*.

⁵This is more general than the typical definition of an identifier. For example, in Ada operators and operator strings are both included.

identifier is a token literal and the literal kind assigned is *token_lit*⁷. If an identifier matches the implicit production

`exp ::= id`

then this use of the identifier is a reference and the literal kind assigned is *identifier*.

In Ada, certain builtin operators can be referred to by string designators, which the parser calls *keyword strings*. For example $a + b$ can also be written `"+"(a, b)`. The parser cannot always tell whether a string literal is being used as a string or as a designator. A string which is syntactically legal as a keyword string can match the metagrammar reserved word `id`; the literal kind assigned is *kw_str_lit*. A string literal can also match the implicit production

`exp ::= (string literal)`

If the string is syntactically legal as a keyword string, then the literal kind *kw_str* is assigned; otherwise, the *str_lit* is assigned. *kw_str* may later be replaced by *str_lit* when it is determined, using semantic information, that this use of the string is not as a keyword string.

As described in section A.2.5, the tree produced by any production containing *id_list* is replicated once for each identifier in the list. For example, `"x, y, z: T;"` becomes `"x: T; y: T; z: T;"`. In order to have the information needed to apply the Ada rules for matching specifications and bodies to Iris-Ada trees, it must be known whether such replication took place. Therefore, the token literals produced for all but the last replication are assigned a literal kind of *nonlast_token_lit*. *nonlast_token_lit* is identical to *token_lit* in all other respects.

A.2.9 Actions

Inserting an action in a grammar alerts the parser of the need for some special processing which is difficult or impossible to describe in BNF or regular expression form. The languages that can be recognized by our parser are approximately those that are LALR(1). In a few places this results in conflicts which cannot be resolved with single token lookahead and so an action is used to instruct the parser to look ahead further to resolve the conflict.

Actions are specified operationally. Most of the actions are used just once or twice in the Ada grammar. One appears only in the metagrammar, and it is the only action that appears in the metagrammar. The built-in names `id`, *id_list* and `any` are in fact predefined actions.

%aggregate Distinguishes between aggregates and parenthesized expressions and produces the tree (*~aggregate* (*~list* \$1 ... \$n)) in the former case and the tree (*~parentheses* !) in the latter.

%apply This action appears only in this production:

`exp ::= 100 '(apl ') #T $1 $2 %apply`

%apply replaces the list operator of its second argument with its first argument. A semantic analyzer for Ada may make additional tree transformations based on distinguishing whether the expression is a function call, an array reference or a type conversion.

⁷However, `id` may be followed by the action *%reference* (see section A.2.9) to change the literal kind to *identifier*, thereby changing the reference node produced by `id` from a literal reference to a named reference.

%label_match This action is similar to *%id_match*, but the matching occurs at a higher level according to the Ada rules for matching block and loop labels.

%of Appears in the following two productions:

```

exp ::= :
      | 100 " 110          #T ($2 $1) %of
      | 100 " 110 '( exp ' ) #T ($2 $1 $3) %of
      :

```

Ada attributes are built-in functions of the Ada language that are called with their own special syntax. On the one hand they must be transformed to the standard Iris application form; and on the other hand the resulting token name must be distinguishable from other uses of the same token in source programs. For example, *x'first* is distinguished from *first(x)* by replacing *first* in *x'first* by *~first_of*. *%of* performs this name translation.

%reference This action is used when an identifier matched by *id* should be a reference rather than a token literal. It is used in the unusual case where the source language (semantically) permits expressions (i.e., references to) values of a given type but (syntactically) restricts the expressions to being single identifiers. That is, it enables overload resolution of identifiers specified by *id* instead of *exp*.

%select Distinguishes between select, timed entry and conditional entry statements and produces the correct tree.

%set_enum_list This action appears only in a single *exp* production:

```

enum_item ::= id #T (~preinit_decl $1 ()).
exp       ::= :
              | type id is '( {enum_item ,} ' )
                  #T (~decl $1 ~metatype (~derived (~enum $2))) %set_enum_list
              :

```

%set_enum_list fills in the second argument of the *~preinit_decl* with the overloadable identifier of the enumeration type.

%with_spec The parser locates the Iris trees associated with the library units named in a *with* clause. The reference nodes in the representation of the *with* clause are resolved to refer to the withed units.

A.2.10 Precedence

Operator precedence could be specified by a cascade of nonterminals. However, this is cumbersome and is contrary to our goal of having only a single primary nonterminal. However, there remains a need to express facts like *** has greater binding strength on the left than does *+* on the right, so

A.3 The Metagrammar

```
--
-- Grammar for Grammars (Metagrammar)
--
-- Copyright © 1990 by Incremental Systems Corporation. All rights reserved.

-- Permission to copy all or part of these materials is granted, provided that
-- the copies are not made or distributed for commercial advantage and that the
-- Incremental Systems Corporation copyright notice set forth here appears on
-- each copy.

-- This work was supported in part by the Defense Advanced Research Projects
-- Agency (Arpa Order 5057) monitored by the Department of the Navy, Space and
-- Naval Warfare Systems Command under contract N00039-85-C-0126 and by the
-- Defense Advanced Research Projects Agency (Arpa Order 6487-1) under
-- contract MDA972-88-C-0076.
--
```

```
::= | . $ % # * ^ ' [ ( { } ) ] @ N T E
```

grammar

```
root          ::= [{ any }] 'grammar list_of_subgrammars
list_of_subgrammars ::= { id '::= list_of_productions }
xl            ::= [{ exp }]
list_of_productions ::= { production @ '|' } '.
production    ::= concrete_syntax '# abstract_syntax
concrete_syntax ::= xl
abstract_syntax ::= 'T xl
                | 'E exp [exp]
                | 'N [exp]
exp           ::= '( xl ' )
                | '$ int_lit
                | '% id
                | '( ' )
                | '* id
                | '^ id
                | '' any
                | '[ xl ' ]
                | '{ xl ' }
                | '{ xl '@ xl ' }
                | '{ xl '$ xl ' }
```

```

cu::= language id exp      #T ("language_spec $1 $2")
    -- used only for language definitions
    | [{ prefix ; }] exp

    -- to the language package
    -- | [{ prefix ; }] exp
    -- the choice between "lib_body and "lib_spec
    -- is made by %context

    -- to the corresponding specification
    | [_, vprefix ; ] separate '( name ' ) exp
    name::= { _ , . , }

    prefix::= pragma id %reference ['( apl ')]
    | with { with_item 0 , }
    | use apl
    with_item::= id %with_spec
    -- builds trees with "with" and "indirect_with" operations

    apl::= {exp 0 ,}

    enum_item::= id

    null_or_xl::= null ;
    | { *exp ; }

    when_part::= { when exp => null_or_xl }

    sl::= { *st ; }

    if_body::= exp then sl if_tail end if
    if_tail::= elseif exp then sl if_tail
    | else sl
    |

#T ("lib_spec" parent_context $1 $2) %context
-- "parent_context is replaced with a reference

#T ("lib_body" parent_context $1 $2) %context

-- "parent_context is replaced by a reference

#T ("lib_subunit $2 $1 $3) %context.
#N.

#E "call_pragma
#E "with
#E "use.
#N.

#N.

#T ("preinit_decl $1 enum_name).
-- enum_name is set by %set_enum_list

#T ("list)
#N.
#N.

#N.
#E "list.
#N
#T "true $1
#N.

```

```

| exit exp when 15
| for id in exp loop s1 end loop %label_match
| for id in reverse exp loop s1 end loop %label_match
| for id ' ' %reference exp use 15
| for id use %reference 15
| for id use %reference at 15
| for id use %reference record at mod exp {exp at exp range exp ;} end record
| for id use %reference record {exp at exp range exp . end record
| function id [fpl] return 15
| function id [fpl] return exp is subprog_body [%id_match]
| function id [fpl] return exp renames 15
| function id is new 15
| generic [{ *fp ;}] function id [fpl] return 15
| generic [{ *fp ;}] package id is [dl] [private [dl]] end [%id_match]
| generic [{ *fp ;}] procedure id [fpl]
| goto 15
| if if_body
| loop s1 end loop %label_match
| new 15
| new 15 ' ' 15
| package id is [dl] [private [dl]] end [%id_match]
| package id is new 15
| package id renames 15
| package body id is subprog_body [%id_match]
| pragma id %reference ['( apl ')]
| procedure id fpl
| procedure id fpl is subprog_body [%id_match]
| procedure id fpl renames 15
| procedure id
| procedure id is subprog_body [%id_match]
| procedure id renames 15

#E ~exit_with_label
#T (~for_loop (~preinit_decl $1 $2) $3)
#T (~for_reverse_loop (~preinit_decl $1 $2) $3)
#E ~length_rep_spec
#E ~enum_rep_spec
#E ~address_rep_spec
#E ~record_rep_spec
#T (~record_rep_spec $1 '1 $2)
#T (~spec $1 (~function $2 $3) )
#T (~vis_decl $1 (~function $2 $3) $4)
#T (~rename_decl $1 (~function $2 $3) $4)
#T (~instantiation_decl $1 ~function $2)
#T (~spec $2 (~generic $1 (~function $3 $4)) )
#T (~%id_match]
#T (~vis_decl $2 (~generic $1 ~package) (~new_generic_package $3 $4 ))
#T (~spec $2 (~generic $1 (~function $3 ~statement)) )
#E ~goto
#E ~if
#E ~loop
#E ~access_allo
#E ~access_allo_default
#T (~vis_decl $1 ~package (~new_package $2 $3 ))
#T (~instantiation_decl $1 ~package $2)
#T (~rename_decl $1 ~package $2)
#T (~body_decl $1 ~package $2)
#E ~call_pragma
#T (~spec $1 (~function $2 ~statement) )
#T (~vis_decl $1 (~function $2 ~statement) $3)
#T (~rename_decl $1 (~function $2 ~statement) $3)
#T (~spec $1 (~function (~list) ~statement) )
#T (~vis_decl $1 (~function (~list) ~statement) $2)
#T (~rename_decl $1 (~function (~list) ~statement) $2)

```

```

| type id is digits 35
|
| type id is digits exp range 35
|
| type id is limited private
| type id is limited private ~fp
| type id is new 15
| type id is private
| type id is private ~fp
| type id is range <>
| type id is range 35
|
| type id is record {*id;} end record
| type id is record null; end record
| use apl
| while exp loop sl end loop %label_match
| with function id [fpl] return 15
| with function id [fpl] return exp is 15
| with procedure id [fpl]
| with procedure id [fpl] is 15
|
| id_list : constant 15
| id_list : constant exp := 15
| id_list : constant := 15
| id_list : exception
| id_list : in 15
| id_list : in exp := 15
| id_list : in out 15
| id_list : out 15
| id_list : out 15
| id_list : 15
| id_list : 15 ~fp
| id_list : 15 ~fd
| id_list : 15 ~st
| id_list : exp := 15

#T (~decl $1 ~metatype
  (~derived (~digits_constrain ~univ_float $2)))
#T (~decl $1 ~metatype (~derived (~range_constrain
  (~digits_constrain ~univ_float $2) $3)))
#T (~decl $1 ~metatype (~derived (~lim_private (~list) )))
#T (~fp_spec $1 (~metatype ~lim_private))
#T (~decl $1 ~metatype (~derived (~new_type $2)))
#T (~decl $1 ~metatype (~derived (~private (~list))))
#T (~fp_spec $1 (~metatype ~private))
#T (~fp_spec $1 (~metatype ~univ_integer))
#T (~decl $1 ~metatype
  (~derived (~range_constrain ~univ_integer $2)))
#T (~decl $1 ~metatype (~derived (~record (~list) $2)))
#T (~decl $1 ~metatype (~derived (~record (~list) (~list))))
#E ~use
#E ~while_loop
#T (~fp_spec $1 (~function $2 $3))
#T (~fp_spec $1 (~default (~function $2 $3) $4))
#T (~fp_spec $1 (~function $2 ~statement))
#T (~fp_spec $1 (~default (~function $2 ~statement) $3))
#T (~spec $1 $2 )
#E ~decl
#T (~rename_decl $1 (~static ~scalar) $2)
#T (~preinit_decl $1 ~exception)
#T (~fp_spec $1 (~in_mode $2))
#T (~fp_spec $1 (~in_mode (~default $2 $3)))
#T (~fp_spec $1 (~in_out_mode $2))
#T (~fp_spec $1 (~out_mode $2))
#T (~preinit_decl $1 (~in_out_mode $2))
#E ~fp_spec
#T (~fp_spec $1 (~value_mode $2))
#T (~label_decl $1 %identifier ~block_label $2)
#T (~preinit_decl $1 (~in_out_mode (~default $2 $3)))

```

```

| 70 - 75
| 70 & 75
| 80 ' * 85
| 80 / 85
| 80 mod 85
| 80 rem 85
| abs 95
| not 95
| 95 ** 95
| 100 ' . 110
| 100 ' , all
| 100 ' ( ap1 ' )

| 100 ' ' ( agg ' )
| 100 ' ' 110
| 100 ' ' 110 ' ( exp ' )

-- the following productions are necessary because delta, digits and range are reserved words
| 100 ' ' delta
| 100 ' ' digits
| 100 ' ' range
| 100 ' ' range ' ( exp ' )

-- nonAda productions
| ~type
| id_list : "quote 15

#E -
#E &
#E ' *
#E /
#E ~mod
#E ~rem
#E ~abs
#E ~not
#E **
#E ~dot_qual
#E ~dot_all
#T $1 $2 %apply
-- %apply result is $2 with list op replaced by $1
#E ~qual_exp
#T ($2 %of $1)
#T ($2 %of $1 $3)

#E ~delta_of
#E ~digits_of
#E ~range_of
#E ~range_of

#T ~metatype
#T (~fp_spec $1 (~quote_mode $2)).

```

Appendix B.

Updated Iris-Ada Grammar


```

--
--      Formal Ada Syntax
--
-- $Revision: 3.3 $ of $Date: 90/10/09 11:19:12 $
--
-- Copyright (c) 1989-1990 by Incremental Systems Corporation. All rights
-- reserved.
--
-- Permission to copy all or part of these materials is granted, provided that
-- the copies are not made or distributed for commercial advantage and that the
-- Incremental Systems Corporation copyright notice set forth above appears on
-- each copy.
--
-- This work was supported in part by the Defense Advanced Research Projects
-- Agency (Arpa Order 5057) monitored by the Department of the Navy, Space and
-- Naval Warfare Systems Command under contract N00039-85-C-0126 and by the
-- Defense Advanced Research Projects Agency under (Arpa Order 6487-1)
-- contract MDA972-88-C-0076.
--
-- Lexical analyzers and parsers will not do case shifting of any token string
-- syntax.
--
grammar ada is
reserved
  ~quote
  ~type
  abort
  abs
  accept
  access
  all
  and
  array
  at
  begin
  body
  case
  constant
  declare
  delay
  delta
  digits
  do
  else
  elsif
  end
  entry
  exception
  exit
  for
  function
  goto
  generic
  if
  in
  is
  limited
  loop
  mod
  new
  not
  null
  of
  or
  others
  out
  package
  pragma
  private
  procedure
  raise
  range
  record
  rem
  renames
  return
  reverse
  select
  separate
  subtype
  task
  terminate
  then
  type
  use
  when
  while
  with
  xor
  '
  =
  /=
  <
  <=
  *
  **
  /
  >
  >=
  ;
  ..
  :
  :
  &
  (
  )
  |
  <<
  >>
  ,
  +
  -
  ^
  _
  `
  ~
  &
  '
  =>
  .
  productions
  root ::= cu ;
  cu ::= language id exp
  | (( prefix ; )) exp
--
-- #N. (~language_spec $1 $2)
-- #T (~used only for language definitions)
-- #T (~library_spec ~parent_context $1 $2) %context

```

```

-- ~parent_context is replaced with a reference
-- to the language package
#T (~library_body ~parent_context $1 $2) %context
-- the choice between ~library_body and ~library_spec
-- is made by %context
-- ~parent_context is replaced by a reference
-- to the corresponding specification
#T (~library_subunit $2 $1 $3) %context.

#E ~call_pragma
#E ~with
#E ~use.
#N.

prefix::= pragma id %reference ['( apl ')]
| with { with_item @ , }
| use apl
with_item::= id %with_spec
-- %with_spec resolves the reference node to refer to the specification
-- of the withed unit

apl::= (exp @ , )

enum_item::= id

null_or_xl::= null ;
| { *exp ; }

when_part::= { when exp => null_or_xl }

sl::= { *st ; }
if_body::= exp then sl if_tail end if
if_tail::= elsif exp then sl if_tail
| else sl
|

select_alt::= when exp => *att; [sl]
| *att; [sl]
select_else::= else sl
|

dl::= {exp ; }
subprog_body::=
{dl} begin sl end
| {dl} begin sl exception when_part end
| {dl} end
| separate

fpl::= ' ( (*fp @ ; ) ' )
agg::= {exp @ , }
-- includes normal parenthesized expressions and all aggregate forms.
-- tree is either (~parentheses $1) or (~aggregate (~list $1 ... $n))

exp::=
abort apl
| accept id [do sl end [%id_match]]
| accept id %if_fpl fpl [do sl end [%id_match]]

#E ~abort
#T (~accept $1 %reference (~entry (~list) ~statement) $2)
#T (~accept $1 %reference (~entry $2 ~statement) $3)

```

```

| accept id '( exp ' ) [fpl] [do sl end [%id_match]]
| array '( apl ' ) of exp
| begin sl end %label_match
| begin sl exception when part end %label_match
| case exp is when part end case
| declare [dl] begin sl end %label_match
| declare [dl] begin sl exception when part end %label_match
| delay 15
| delay 15 ^att
| entry id
| entry id %if_fpl fpl
| entry id '( apl ' ) [fpl]
| exit
| exit when 15
| exit 15
| exit exp when 15
| for id in exp loop sl end loop %label_match
| for id in reverse exp loop sl end loop %label_match
| for id %reference ' ' size use 15
| for id %reference ' ' storage_size use 15
| for id %reference ' ' small use 15
| for id %reference use 15
| for id %reference use at 15
| for id %reference use record
|   at mod exp; [(id %reference at exp range exp ;)]
| end record
| for id %reference use record
|   [(id %reference at exp range exp ;)] end record
| function id [fpl] return 15
| function id [fpl] return exp is subprog_body [%id_match]
| function id [fpl] return exp renames 15
| function id is new 15
| generic [( *fp ;)] function id [fpl] return 15
| generic [( *fp ;)] package id is [dl] [private [dl]] end [%id_match]
| generic [( *fp ;)] procedure id [fpl]
|   goto 15
| if %body
|   loop sl end loop %label_match
|   new 15
|   new 15 ' ' 15
|   package id is [dl] [private [dl]] end [%id_match]
|   package id is new 15
|   package id renames 15
|   package body id is subprog_body [%id_match]
|   pragma id %reference [( ' ( apl ' ) )]
|   procedure id fpl
|   procedure id fpl is subprog_body [%id_match]
|   procedure id fpl renames 15
|   procedure id
|   procedure id is subprog_body [%id_match]
|   procedure id renames 15

#T (~accept (~subscript $1 $2) (~entry $3 ~statement) $4)
#T (~derived (~array $1 (~value_mode $2)))
#E ~compound
#E ~handler
#E ~case
#E ~scope
#T (~scope $1 (~handler $2 $3))
#E ~delay
#E ~timeout
#T (~preinit_decl $1 (~entry (~list) ~statement))
#T (~preinit_decl $1 (~entry $2 ~statement))
#T (~preinit_decl $1 (~array $2 (~entry $3 ~statement)))
#E ~exit
#E ~exit
#E ~exit_with_label
#E ~exit_with_label
#T (~for_loop (~preinit_decl $1 $2) $3)
#T (~for_reverse_loop (~preinit_decl $1 $2) $3)
#E ~length_size_rep_spec
#E ~length_storage_size_rep_spec
#E ~length_small_rep_spec
#E ~enum_rep_spec
#E ~address_rep_spec

#E ~record_rep_spec

#T (~record_rep_spec $1 ~unaligned $2)
#T (~spec $1 (~function $2 $3) )
#T (~vis_decl $1 (~function $2 $3) $4)
#T (~rename_decl $1 (~function $2 $3) $4)
#T (~instantiation_decl $1 ~function $2)
#T (~spec $2 (~generic $1 (~function $3 $4)) )
#T (~id_match)
#T (~vis_decl $2 (~generic $1 ~package) (~new_generic_package $3
#T (~spec $2 (~generic $1 (~function $3 ~statement))) )
#E ~goto
#E ~if
#E ~loop
#E ~access_allo
#E ~access_allo default
#T (~vis_decl $1 ~package (~new_package $2 $3 ))
#T (~instantiation_decl $1 ~package $2)
#T (~rename_decl $1 ~package $2)
#T (~body_decl $1 %reference ~package $2)
#E ~call_pragma
#T (~spec $1 (~function $2 ~statement) )
#T (~vis_decl $1 (~function $2 ~statement) $3)
#T (~rename_decl $1 (~function $2 ~statement) $3)
#T (~spec $1 (~function (~list) ~statement) )
#T (~vis_decl $1 (~function (~list) ~statement) $2)
#T (~rename_decl $1 (~function (~list) ~statement) $2)

```

```

| procedure id is new 15
| raise
| raise 15
| return
| return 15
| select (select_alt 0 or) select_else end select
| -- $select_produces the trees for ~select, ~timed_entry or ~cond_entry
| subtype id is 15
| task id [is [dl] end {fid_match}]
| task body id is subprog_body {fid_match}
| task type id [is [dl] end {fid_match}]
|
| terminate
| type id
| type id fpl
| type id fpl is limited private ^fp
| type id fpl is limited private ^fp
| type id fpl is private
| type id fpl is private ^fp
| type id fpl is record (*fd;) and record
| type id fpl is record null; end record
| type id is '(enum_item 0,)'
| type id is '<>'
| type id is access 15 ^fp
| type id is access 15
| type id is array '(apl)' of 15 ^fp
| type id is array '(apl)' of 15 ^fp
| type id is delta <>
| type id is delta 35
|
| type id is delta exp range *rtd
|
| type id is digits <>
| type id is digits 35
|
| type id is digits exp range *rtd
|
| type id is limited private
| type id is limited private ^fp
| type id is new 15
| type id is private
| type id is private ^fp
| type id is range <>
| type id is range *itd
|
| type id is record (*fd;) and record
| type id is record null; end record
| use apl
| while exp loop s1 end loop $label_match
| with function id [fpl] return 15
| with function id [fpl] return exp is 15
| with procedure id [fpl]
|
| ~instantiation_decl $1 (~function <> ~statement) $2)
|E ~raise
|T ~return
|E ~return
|T (~select $1 $2) $select
| or ~cond_entry
|T (~decl $1 ~metatype $2)
|T (~vis_decl $1 ~task (~new_package $2 (~list) ))
|T (~body_decl $1 $reference ~task $2)
|T (~vis_decl $1 (~metatype ~task)
| (~derived (~new_task_type $2)))
|
|E ~terminate
|T (~incomplete_spec $1 (~metatype (~limited_private (~list) )
| (~incomplete_spec $1 (~metatype (~limited_private $2 )))
| :
|T (~decl $1 ~metatype (~derived (~limited_private $2 )))
|T (~fp_spec $1 (~metatype (~limited_private $2 )))
|T (~decl $1 ~metatype (~derived (~private $2)))
|T (~fp_spec $1 (~metatype (~private $2)))
|T (~decl $1 ~metatype (~derived (~record $2 $3)))
|T (~decl $1 ~metatype (~derived (~record $2 (~list))))
|T (~vis_decl $1 ~metatype (~derived (~new_enum $2))) $set_enu
|T (~fp_spec $1 (~metatype ~discrete))
|T (~fp_spec $1 (~metatype (~access (~value_mode $2))))
|T (~decl $1 ~metatype (~derived (~access (~value_mode $2))))
|T (~decl $1 ~metatype (~derived (~array $2 (~value_mode $3))))
|T (~fp_spec $1 (~metatype (~array $2 (~value_mode $3))))
|T (~fp_spec $1 (~metatype ~universal_fixed))
|T (~decl $1 ~metatype
| (~derived (~delta_constrain ~universal_fixed $2)))
|T (~decl $1 ~metatype (~derived (~range_constrain
| (~delta_constrain ~universal_float $2
| (~metatype ~universal_float))
|T (~decl $1 ~metatype (~derived (~digits_constrain ~universal_float $2)
| (~derived (~digits_constrain ~universal_float $
| (~digits_constrain ~limited_private)))
|T (~decl $1 ~metatype (~derived (~limited_private)))
|T (~decl $1 ~metatype (~derived $2))
|T (~decl $1 ~metatype (~derived (~private (~list))))
|T (~fp_spec $1 (~metatype ~private))
|T (~fp_spec $1 (~metatype ~universal_integer))
|T (~decl $1 ~metatype
| (~derived (~range_constrain ~universal_integer $2
| (~derived (~record (~list) $2)))
|T (~decl $1 ~metatype (~derived (~record (~list) (~list))))
|E ~use
|E ~while_loop
|T (~fp_spec $1 (~function $2 $3))
|T (~fp_spec $1 (~default (~function $2 $3) $4))
|T (~fp_spec $1 (~function $2 ~statement))

```

```

| with procedure id [fpl] is 15
|
| id_list : constant 15
| id_list : constant exp := 15
| id_list : constant := 15
| id_list : exception
| id_list : in 15
| id_list : in exp := 15
| id_list : in out 15
| id_list : out 15
| id_list : 15
| id_list : 15 ^fp
| id_list : 15 ^fd
| id_list : 15 ^st
| id_list : exp := 15
| id_list : exp := 15 ^fp
| id_list : exp := 15 ^fd
|
| id_list : exp renames 15
| id_list : exception renames 15
|
| null
| others
| '(agg ' )
|
| << id >> 15
| 10 => 15
| 100 := 15
| 20 ' 25
| 30 delta 35
| 30 digits 35
| 30 range 35
| 40 or 41
| 40 or else 41
| 40 xor 41
| 40 and 41
| 40 and then 41
| 60 in 60
| 60 not in 60
| 60 = 60
| 60 /= 60
| 60 < 60
| 60 <= 60
| 60 > 60
| 60 >= 60
| 65 .. 65
| 65 .. 65 ^ltd
| 65 .. 65 ^rtd
|
| + 75
| - 75
| 70 + 75
| 70 - 75
|
|T (~fp_spec $1 (~default (~function $2 ~statement) $3))
|E (~spec $1 $2 )
|T (~rename_decl $1 (~static ~scalar) $2)
|T (~preinit_decl $1 ~exception)
|T (~fp_spec $1 (~in_mode $2))
|T (~fp_spec $1 (~in_mode (~default $2 $3)))
|T (~fp_spec $1 (~in_out_mode $2))
|T (~fp_spec $1 (~out_mode $2))
|T (~preinit_decl $1 (~in_out_mode $2))
|E ~fp_spec
|T (~fp_spec $1 (~value_mode $2))
|T (~label_decl $1 %identifier ~block_label $2)
|T (~preinit_decl $1 (~in_out_mode (~default $2 $3)))
|T (~fp_spec $1 (~default $2 $3))
|T (~fp_spec $1 (~value_mode (~default $2 $3)))
|
|T (~rename_decl $1 %identifier $2 $3)
|T (~rename_decl $1 %identifier ~exception $2)
|
|T ~null
|T ~others
|N
|T (<>
|T (~label_decl $1 ~goto_label $2)
|E ~named_expr
|E :=
|E '
|E ~delta_constrain
|E ~digits_constrain
|E ~range_constrain
|E or
|E ~or_else
|E xor
|E and
|E ~and_then
|E ~in
|E ~not_in
|E =
|T ((~ne_of =) $1 $2)
|E <
|E <=
|E >
|E >=
|E ..
|E ~integer_dot_dot
|E ~real_dot_dot
|E +
|E -
|E +
|E -

```

```

| 70 & 75
| 80 '*' 85
| 80 / 85
| 80 mod 85
| 80 rem 85
| abs 95
| not 95
| 95 ** 95
| 100 ' ' 110
| 100 ' ' all
| 100 ' ' "/"
| 100 ' ( apl ' )
| 100 ' ' ( agg ' )
| 100 ' ' . base
| 100 ' ' 110
| 100 ' ' 110 ' ( exp ' )
-- the following productions are necessary because delta, digits and range are reserved words
| 100 ' ' delta
| 100 ' ' digits
| 100 ' ' range
| 100 ' ' range ' ( exp ' )
| "/"
-- nonAda productions
| ~typo
| id_list : ~quote 15
| 100 ' { (exp 0 ; ) } '
| ' { (exp 0 ; ) } '
ond ada

#E &
#E '*'
#E /
#E mod
#E rem
#E abs
#E not
#E **
#E ~dot_qual
#E ~dot_all
#T (~ne_of (~dot_qual $1 "="))
#T $1 $2 %apply
-- %apply result is $2 with list op replaced by $1
#E ~qual_expr
#E ~base_of
#T ($2 %attribute $1)
#T ($2 %attribute $1) $3)
#E ~delta_attr
#E ~digits_attr
#E ~range_attr
#T (~range_attr $1) $2)
#T (~ne_of "=")

#T ~metatypo
#T (~fp_spec $1 (~quote_mode $2))
#T ($1 $2)
#N.

```

Appendix C.

Updated Ada Declarations

```

language ada
package standard is

-- $Revision: 3.8 $ of $Date: 90/11/12 20:00:46 $
-- Copyright (C) 1990 by Incremental Systems Corporation. All rights reserved.

-- Permission to copy all or part of these materials is granted, provided that
-- the copies are not made or distributed for commercial advantage and that the
-- Incremental Systems Corporation copyright notice set forth here appears on
-- each copy.

-- This work was supported in part by the Defense Advanced Research Projects
-- Agency (Arpa Order 5057) monitored by the Department of the Navy, Space and
-- Naval Warfare Systems Command under contract N00039--85--C--0126 and by the
-- Defense Advanced Research Projects Agency (Arpa Order 6487--1) under
-- contract MDA972--88--C--0076.

-- Functions
type ~function(
  fpl: ~quote ~list_of[~fp_item!];
  rt: ~type) is private;

function ~body(
  dl: ~quote ~list_of[~decl_item];
  sl: ~quote ~list_of[~statement]) return <>;
  -- return type depends on the context in which called
  -- return type can be any Ada return type, ~statement, ~package or ~generic
function ~return(x: <>) return ~statement; --+ return_value
  -- x is restricted to the return type of the function in which it is called.
function ~return return ~statement;
  -- may be called only within procedures and accept statements

-- Declarations
type ~decl_item is private;
type ~token is new ~scalar; -- a nonoverload resolved designator
function ~spec(
  name: ~token;
  t: ~type) return ~decl_item;
function ~preinit_decl( -- initialized constant declaration including enumeration items
  name: ~token;
  t: ~type) return ~decl_item;
function ~decl( -- value evaluated at elaboration of declaration
  -- ~decl is a composition of ~define and ~qual_expr where ~define is:
  -- function ~define(name: ~token; value: <>) return ~decl_item
  name: ~token;
  t: ~type;
  value: t) return ~decl_item;
function ~vis_decl( -- value evaluated at reference to declaration
  name: ~token;
  t: ~type;
  value: t) return ~decl_item;
function ~rename_decl(

```



```

name: ~token;
t: ~type;
value: ~quote t) return ~decl_item;
function ~label_decl{ -- for declarations in statement contexts
name: ~token;
t: ~type; -- block_label or goto_label
value: ~quote ~statement) return ~statement;
function ~incomplete_spec{ -- used only for incomplete types, similar to spec
name: ~token;
t: ~metatype(~type)) return ~decl_item;

-- Types
-- <> and ~limited_private are similar in that neither imposes any type
-- restrictions. However, ~limited_private is an Ada concept
-- and therefore follows the Ada rule that multiple uses of ~limited_private
-- in the same declaration implies dependence. <> is used where Ada
-- does not impose its normal dependence. In the case where
-- there is just one occurrence in a declaration, <> is used in preference
-- to ~limited_private.
type ~limited_private(fpl: ~quote ~list_of[~fp_item]) is limited_private;
<> : constant <> := <>; --+ box
-- unspecified value: substitutes for whatever value is needed
-- unspecified type: substitutes for whatever type is needed
-- The presence of <> often indicates additional Ada restrictions
-- that are not expressed formally here.

function ~parentheses(x: <>) return ~type of(x);
function ~in(x: <>) t: ~type of(~type of(x)) return ~nonderivable(boolean);
function ~not_in(x: <>) t: ~type of(~type of(x)) return ~nonderivable(boolean);
function ~dot_qual(x, y: <>) return ~type of(y);
-- The x argument can be a record, discriminated type, package, task or context.
-- y must be declared in the scope defined by x.
function ~constrained_attr(x: <>) return boolean;

function ~derived(t: ~type) return ~type;

type ~private is new ~limited_private;
function ~="(left, right: ~private) return ~nonderivable(boolean); --+ eq
function ~ne of(
x: ~function([left, right: in ~limited_private], boolean))
return ~type of(x);
-- ~ne_of is a functional that takes an equality function ("=") and returns
-- the corresponding inequality ("/=") function.
-- Every reference to "/"=" is changed by the parser to (~ne of "=").
function ~="(x: out ~private; y: ~type of(x)) return ~statement; --+assign

-- Metatypes
type ~metatype(t: ~type) is private;
-- the metatype of t
function ~qual_expr(t: ~type; x: t) return t;
function ~type_convert(t: ~type; x: <>) return t;
-- type_convert is implicit in Ada function form syntax

```

```

-- type_of(x) and t must have the same rep type
function ~type_of(x: ~quote <>) return ~type;
function ~base_of(t: ~type) return ~type;

function ~independent(t: ~type) return ~type_of(t);
-- allows parameters of the same type name to be independently derived
function ~nonderivable(t: ~type) return ~type_of(t);
-- prevents inheritance of subprogram relative to this parameter
-- not available to Ada users

-- Formal Parameters
type ~fp_item is private;
function ~in_mode(t: ~type) return ~type_of(t);
function ~in_out_mode(t: ~type) return ~type_of(t);
function ~out_mode(t: ~type) return ~type_of(t);
function ~value_mode(t: ~type) return ~type_of(~in_out_mode(t));
-- corresponding value of type t is copied to a local variable of that same type
function ~quote_mode(t: ~type) return ~type_of(t);
-- inhibits evaluation at point of call
function ~fp_spec(name: ~token; t: ~type) return ~fp_item;
function ~default(t: ~type; value: ~quote t) return ~type_of(t);
-- used for default formal parameters, variable initial values, and
-- default field values
function ~named_expr(name: ~token; x: <>) return ~type_of(x);
-- used for named actual parameters and aggregates

-- Scalar
type ~scalar is private;
function "<"(left, right: ~scalar) return ~nonderivable(boolean); --+ lt
function "<="(left, right: ~scalar) return ~nonderivable(boolean); --+ le
function ">"(left, right: ~scalar) return ~nonderivable(boolean); --+ ge
function ">"(left, right: ~scalar) return ~nonderivable(boolean); --+ gt
function ~range_constrain
  t: ~metatype(~scalar);
  s: ~set_of(t) return ~type_of(t);
function ~first_attr(t: ~metatype(~scalar)) return t;
function ~last_attr(t: ~metatype(~scalar)) return t;
function ~static(t: ~quote ~type) return ~type_of(t);
-- restricts values to Ada static expressions, not available to Ada users

type ~set_of(t: ~type) is private;
function "|"(left, right: ~static(~set_of)) return ~set_of;
function ".*(left, right: ~scalar) return ~type_of(~type_of(right)); --+ union
function ~integer_dot_dot(left, right: ~independent(~static(~universal_integer)))
  return ~type_of(~universal_integer);
function ~real_dot_dot(left, right: ~independent(~static(~universal_real)))
  return ~type_of(~universal_real);

function ~others return ~set_of;

-- Discrete Types

```



```

dc2: constant character := ~dc2;
dc4: constant character := ~dc4;
syn: constant character := ~syn;
can: constant character := ~can;
sub: constant character := ~sub;
fs: constant character := ~fs;
rs: constant character := ~rs;
del: constant character := ~del;
exclam: constant character := '!';
sharp: constant character := '#';
percent: constant character := '%';
colon: constant character := ':';
query: constant character := '?';
l_bracket: constant character := '[';
r_bracket: constant character := ']';
underline: constant character := '_';
l_brace: constant character := '{';
r_brace: constant character := '}';
lc_a: constant character := 'a';
lc_c: constant character := 'c';
lc_e: constant character := 'e';
lc_g: constant character := 'g';
lc_i: constant character := 'i';
lc_k: constant character := 'k';
lc_m: constant character := 'm';
lc_o: constant character := 'o';
lc_q: constant character := 'q';
lc_s: constant character := 's';
lc_u: constant character := 'u';
lc_w: constant character := 'w';
lc_y: constant character := 'y';
end ascii;

dc3: constant character := ~dc3;
nak: constant character := ~nak;
etb: constant character := ~etb;
em: constant character := ~em;
esc: constant character := ~esc;
gs: constant character := ~gs;
us: constant character := ~us;

quotation: constant character := '"';
dollar: constant character := '$';
ampersand: constant character := '&';
semicolon: constant character := ';';
at_sign: constant character := '@';
back_slash: constant character := '\';
circumflex: constant character := '^';
grave: constant character := '`';
bar: constant character := '|';
tilde: constant character := '~';
lc_b: constant character := 'b';
lc_d: constant character := 'd';
lc_f: constant character := 'f';
lc_h: constant character := 'h';
lc_j: constant character := 'j';
lc_l: constant character := 'l';
lc_n: constant character := 'n';
lc_p: constant character := 'p';
lc_r: constant character := 'r';
lc_t: constant character := 't';
lc_v: constant character := 'v';
lc_x: constant character := 'x';
lc_z: constant character := 'z';

--- Integer Types
type ~universal_integer is new ~discrete;
function "+"(right: ~universal_integer) return ~universal_integer;
function "-"(right: ~universal_integer) return ~universal_integer;
function "abs"(right: ~universal_integer) return ~universal_integer;
function "+"(left, right: ~universal_integer) return ~universal_integer;
function "-"(left, right: ~universal_integer) return ~universal_integer;
function "*" (left, right: ~universal_integer) return ~universal_integer;
function "mod"(left, right: ~universal_integer) return ~universal_integer;
function "rem"(left, right: ~universal_integer) return ~universal_integer;
function "**" (
    left: ~universal_integer;
    right: ~nonderivable(natural)) return ~universal_integer;
function "/"(left, right: ~universal_integer) return ~universal_integer;

type short_integer is new ~universal_integer range -128.. 127;
-- ranges are implementation dependent
type integer is new ~universal_integer range -32768 .. 32767;
--* negate_integer
type long_integer is new ~universal_integer range (-2147_483647 --* negate_integer

--+ identity_integer
--+ negate_integer
--+ abs_integer
--+ add_integer
--+ subtract_integer
--+ multiply_integer
--+ mod_integer
--+ rem_integer
--+ exponentiate_integer

--+ integer_divide_integer

```

```

) -1 .. 2147_483647;      --* subtract_integer

subtype positive is integer range 1 .. integer'last;  --* last_attr
subtype natural is integer range 0 .. integer'last;  --* last_attr

-- Universal Real
type ~universal_real is new ~scalar;
function "+"(right: ~universal_real) return ~universal_real;
function "-"(right: ~universal_real) return ~universal_real;
function "abs"(right: ~universal_real) return ~universal_real;
function "+"(left, right: ~universal_real) return ~universal_real;
function "-"(left, right: ~universal_real) return ~universal_real;
function "*" (
    left: ~nonderivable(~universal_real);
    right: ~nonderivable(~universal_integer)) return ~nonderivable(~universal_real);
function "*" (
    left: ~nonderivable(~universal_integer);
    right: ~nonderivable(~universal_real)) return ~nonderivable(~universal_real);
function "/" (
    left: ~nonderivable(~universal_real);
    right: ~nonderivable(~universal_integer)) return ~nonderivable(~universal_real);

function ~mantissa attr(t: ~metatype(~universal_real)) return ~universal_integer;
function ~large_attr(t: ~metatype(~universal_real)) return ~universal_real;
function ~small_attr(t: ~metatype(~universal_real)) return ~universal_real;
function ~safe_large_attr(t: ~metatype(~universal_real)) return ~universal_real;
function ~safe_small_attr(t: ~metatype(~universal_real)) return ~universal_real;
function ~machine_overflows attr(t: ~metatype(~universal_real)) return boolean;
function ~machine_rounds attr(t: ~metatype(~universal_real)) return boolean;

-- Floating Point Types
type ~universal_float is new ~universal_real;
function "*" (left, right: ~universal_float) return ~universal_float;
function "/" (left, right: ~universal_float) return ~universal_float;
function "*" (
    left: ~universal_float;
    right: ~nonderivable(integer)) return ~universal_float;
function ~digits constrain(
    t: ~metatype(~universal_float);
    digit: ~static(positive)) return ~type_of(t);

function ~digits attr(t: ~metatype(~universal_float)) return positive;
function ~emax attr(t: ~metatype(~universal_float)) return ~universal_integer;
function ~machine_emax attr(t: ~metatype(~universal_float)) return ~universal_integer;
function ~machine_emin attr(t: ~metatype(~universal_float)) return ~universal_integer;
function ~machine_mantissa attr(t: ~metatype(~universal_float)) return ~universal_integer;
function ~machine_mantissa_attr(t: ~metatype(~universal_float)) return ~universal_integer;
function ~safe_emax attr(t: ~metatype(~universal_float)) return ~universal_integer;
function ~epsilon attr(t: ~metatype(~universal_float)) return ~universal_real;

type float is digits 6;  -- digits values are implementation dependent

```

```

type long_float is digits 15;

-- Fixed Point Types
type ~universal_fixed is new ~universal_real;
function "*" (
  left: ~independent(~universal_fixed);
  right: ~independent(~universal_fixed)) return ~nonderivable(~universal_fixed);
function "*" (
  left: ~universal_fixed;
  right: ~nonderivable(integer)) return ~universal_fixed;
function "*" (
  left: ~nonderivable(integer)) return ~universal_fixed;
function "/" (
  left: ~independent(~universal_fixed);
  right: ~independent(~universal_fixed)) return ~nonderivable(~universal_fixed);
function "/" (
  left: ~universal_fixed;
  right: ~nonderivable(integer)) return ~universal_fixed;
function "/" (
  left: ~nonderivable(integer)) return ~universal_fixed;
function "delta_constrain" (
  t: ~metatype(~universal_fixed);
  del: ~static(~universal_real)) return ~type_of(t);
function ~delta_attr(t: ~metatype(~universal_fixed)) return ~universal_real;
function ~aft_attr(t: ~metatype(~universal_fixed)) return ~universal_integer;
function ~fore_attr(t: ~metatype(~universal_fixed)) return ~universal_integer;

-- Lists
type ~list_of(element_type: ~list_of(~type)) is private;
function ~list(x: ~list_of[<>]) return ~type_of(x);
-- ~list is the only operation which can have a variable number of parameters.
-- It cannot be overloaded.
-- Unlike other operations, ~list must be capable of handling large numbers
-- of operands.
-- ~list is implicit in the [...] syntax of extended Ada

-- Arrays
type ~array(
  index_list: ~list_of[~independent(~metatype(~discrete))];
  component_type: ~type) is private;
function ~aggregate(x: ~list_of[<>]) return ~array; --+ aggregate_array
-- Ada aggregates have special visibility rules and thus require special
-- case processing. The only special aggregate operations are
-- ~aggregate and ~named_expr. The arguments to an array aggregate
-- must be static if there are more than one.
function ~index_constrain(
  t: ~metatype(~array);
  constraint: ~list_of[~metatype(~discrete)]) return ~type_of(t);
function ~subscript(
  x: ~array;
  i: ~list_of(x.index_list))
  return x.component_type;
-- ~subscript is implicit in Ada function form syntax

```

```
function ~slice(
    x: ~array([<>], <>);
    i: ~type_of(x'range))
    return ~type_of(x);

    ---* range_array_attr

function ~first_attr(x: ~array) return ~discrete;
function ~first_attr(
    x: ~metatype(~array)) return ~discrete;
function ~first_attr(
    x: ~array)
    return ~function([~i: in ~static(~universal_integer)], ~discrete);
function ~first_attr(
    x: ~metatype(~array))
    return ~function([~i: in ~static(~universal_integer)], ~discrete);

function ~last_attr(x: ~array) return ~discrete;
function ~last_attr(
    x: ~metatype(~array)) return ~discrete;
function ~last_attr(
    x: ~array)
    return ~function([~i: in ~static(~universal_integer)], ~discrete);
function ~last_attr(
    x: ~metatype(~array))
    return ~function([~i: in ~static(~universal_integer)], ~discrete);

function ~length_attr(
    x: ~array)
    return ~universal_integer;
function ~length_attr(
    x: ~metatype(~array)) return ~universal_integer;
function ~length_attr(
    x: ~array)
    return ~function([~i: in ~static(~universal_integer)], ~universal_integer);
function ~length_attr(
    x: ~metatype(~array))
    return ~function([~i: in ~static(~universal_integer)], ~universal_integer);

function ~range_attr(x: ~array) return ~set_of(<>);
function ~range_attr(
    x: ~metatype(~array))
    return ~set_of(<>);
function ~range_attr(
    x: ~array)
    return ~function([~i: in ~static(~universal_integer)], ~set_of(<>));
function ~range_attr(
    x: ~metatype(~array))
    return ~function([~i: in ~static(~universal_integer)], ~set_of(<>));

function ~first_attr(
    x: ~access(~array)) return ~discrete;
function ~first_attr
```

```

x: ~metatype(~access(~array)))
return ~discrete;
function ~first_attr(
  x: ~access(~array))
  return ~function([~i: in ~static(~universal_integer)], ~discrete);
function ~first_attr(
  x: ~metatype(~access(~array)))
  return ~function([~i: in ~static(~universal_integer)], ~discrete);

function ~last_attr(
  x: ~access(~array))
  return ~discrete;
function ~last_attr(
  x: ~metatype(~access(~array)))
  return ~discrete;
function ~last_attr(
  x: ~access(~array))
  return ~function([~i: in ~static(~universal_integer)], ~discrete);
function ~last_attr(
  x: ~metatype(~access(~array)))
  return ~function([~i: in ~static(~universal_integer)], ~discrete);

function ~length_attr(
  x: ~access(~array))
  return ~universal_integer;
function ~length_attr(
  x: ~metatype(~access(~array)))
  return ~universal_integer;
function ~length_attr(
  x: ~access(~array))
  return ~function([~i: in ~static(~universal_integer)], ~universal_integer);
function ~length_attr(
  x: ~metatype(~access(~array)))
  return ~function([~i: in ~static(~universal_integer)], ~universal_integer);

function ~range_attr(
  x: ~access(~array))
  return ~set_of(<>);
function ~range_attr(
  x: ~metatype(~access(~array)))
  return ~set_of(<>);
function ~range_attr(
  x: ~access(~array))
  return ~function([~i: in ~static(~universal_integer)], ~set_of(<>));
function ~range_attr(
  x: ~metatype(~access(~array)))
  return ~function([~i: in ~static(~universal_integer)], ~set_of(<>));

function "&"(left, right: ~array([<>], <>)) return ~type_of(right);
function "&"(left: ~array([<>], <>); right: left.component_type) return ~type_of(left);
function "&"(left: <>; right: ~array([<>], ~type_of(left))) return ~type_of(right);
function "&"(left, right: ~limited_private) return ~array([<>], ~type_of(right));

```



```

function "not"(left: ~array{[<>], boolean}) return ~type_of(right);
function "and"(left, right: ~array{[<>], boolean}) return ~type_of(right);
function "or"(left, right: ~array{[<>], boolean}) return ~type_of(right);
function "xor"(left, right: ~array{[<>], boolean}) return ~type_of(right);
function "<"(left, right: ~array{[<>], ~discrete}) return ~nonderivable(boolean);
function ">"(left, right: ~array{[<>], ~discrete}) return ~nonderivable(boolean);
function "<="(left, right: ~array{[<>], ~discrete}) return ~nonderivable(boolean);
function ">="(left, right: ~array{[<>], ~discrete}) return ~nonderivable(boolean);

type string is array(positive range <>) of character;

-- Access Types
type ~access(val_type: ~type) is private;
function ~index_constrain(
  t: ~metatype(~access(~array));
  constraint: ~list_of(~metatype(~discrete))) return ~type_of(t);
function ~access_allo(t: ~type) return ~access(t);
function ~access_allo_default(t: ~type, x: t) return ~access(t);
function ~null_return ~access;
function ~dot_all(x: ~access) return x.val_type;
function ~access_subscript(
  x: ~access(~array);
  i: ~list_of(x.index_list)) return x.component_type;
  -- subscript is implicit in Ada function form syntax

function ~access_slice(
  x: ~access(~array{[<>], <>});
  i: ~type_of(x.range)) --* range_access_array_attr
  return x.val_type;
function ~access_dot_qual(x: ~access(~record); y: <>) return ~type_of(y);

-- Records
type ~record(fp1, cl: ~quote ~list_of[~fp_item]) is private;
function ~aggregate(x: ~list_of[<>]) return ~record; --* aggregate record
  -- Ada aggregates have special visibility rules and thus require special
  -- case processing. The only special aggregate operations are
  -- ~aggregate and ~named_expr.
function ~case(
  discrim: ~discrete;
  x: ~quote ~list_of[
    ~set_of(~static(~discrete));
    ~list_of[~fp_item]] return ~fp_item;

-- Sequential Control
type ~statement is private;
function ~compound(al: ~quote ~list_of[~statement]) return ~statement;
function ~scope( -- Ada declare

```

```

dl: ~quote ~list_of[~decl_item];
sl: ~quote ~list_of[~statement]) return ~statement;
function ~if(
  x: ~quote ~list_of[
    ~independent(boolean);
    ~list_of[~statement]]) return ~statement;

function ~case(
  discrim: ~discrete;
  x: ~quote ~list_of[
    ~set_of(~static(~discrete));
    ~list_of[~statement]]) return ~statement;
function ~loop(val: ~quote ~list_of[~statement]) return ~statement;
function ~while_loop(
  c: ~quote boolean;
  val: ~quote ~list_of[~statement]) return ~statement;
function ~for_loop(
  i: ~quote ~decl_item;
  val: ~quote ~list_of[~statement]) return ~statement;
function ~for_reverse_loop(
  i: ~quote ~decl_item;
  val: ~quote ~list_of[~statement]) return ~statement;

type ~block_label is new ~statement;
function ~exit(b: boolean := true) return ~statement;
function ~exit_with_label(bl: ~block_label; b: boolean := true) return ~statement;

type ~goto_label is new ~statement;
function ~goto(x: ~goto_label) return ~statement;

function ~null return ~statement;

-- Exception Handling
type ~exception is now ~scalar;
-- ~scalar type is required for exception handlers
-- However, not all ~scalar operations are available on exceptions
constraint_error, numeric_error, program_error,
storage_error, tasking_error: exception;
function ~raise(e: ~exception) return ~statement;
function ~reraise return ~statement;
function ~handler(
  sl: ~quote ~list_of[~statement];
  h: ~quote ~list_of[~set_of(~exception);
    ~list_of[~statement]]) return ~statement;

-- Packages
type ~package is private;
function ~new_package(
  vis_part: ~quote ~list_of[~decl_item];
  priv_part: ~quote ~list_of[~decl_item]) return ~package;
function ~new_generic_package(
  vis_part: ~quote ~list_of[~decl_item];
  priv_part: ~quote ~list_of[~decl_item]) return ~generic(<>, ~package);

```

```

function ~body_decl( -- must have a ~preinit:decl
name: <> -- restricted to package, task, task type, and generic package
t: ~metatype(~package);
value: ~package) return ~decl_item;
function ~use(x: ~list_of(~package)) return ~decl_item;

-- Duration, Task, and Entry
type duration is delta 0.000050 range -107374.1024..107374.1023; --* negate_real
-- actual values are implementation dependent
-- duration/5000 corresponds to 50 microseconds,
-- duration units are seconds,
-- range is -1.24 .. 1.24 days.
function ~delay(d: duration) return ~statement;

type ~task is new ~package;
function ~new_task_type(
via_part: ~quote ~list_of(~decl_item)) return ~metatype(~task);

function ~abort(x: ~list_of(~task)) return ~statement;
function ~callable_attr(x: ~task) return boolean;
function ~terminated_attr(x: ~task) return boolean;

function ~callable_attr(x: ~access(~task)) return boolean; --+ callable_access_task_attr:
function ~terminated_attr(x: ~access(~task)) return boolean; --+ terminated_access_task_attr

type ~entry is new ~function;
type ~accept_timeout_terminate is new ~statement;
function ~count_attr(e: ~entry) return ~universal_integer;
function ~accept(
e: ~entry;
t: ~metatype(~type_of(e));
sl: ~quote ~list_of(~statement)) return ~statement; --+ accept_in_select
e: ~entry;
t: ~metatype(~type_of(e));
sl: ~quote ~list_of(~statement)) return ~accept_timeout_terminate;
function ~timeout(d: duration) return ~accept_timeout_terminate;
-- ~timeout is introduced by the parser for a delay statement occurring
-- as a delay alternative in a selective wait statement

function ~terminate return ~accept_timeout_terminate;
function ~select(
alternatives: ~quote ~list_of(
~independent(boolean),
~accept_timeout_terminate,
~list_of(~statement));
else_part: ~quote ~list_of(~statement)) return ~statement;
function ~timed_entry(
entry_call: ~quote ~statement;
sl: ~quote ~list_of(~statement);
to: ~accept_timeout_terminate; -- only allowable statement is a timeout.
dsl: ~quote ~list_of(~statement)) return ~statement;

```

```

function ~conditional_entry(
  entry_call: ~quote ~statement;
  sl: ~quote ~list_of(~statement);
  else_part: ~quote ~list_of(~statement)) return ~statement;

-- Separate Compilation

type ~compilation_unit is private;

function ~with(wl: ~list_of(~compilation_unit)) return ~decl_item;

function ~language_spec(
  language_name: ~token;
  spec: ~quote ~decl_item) return ~compilation_unit;
function ~library_spec(
  language_cu: ~quote ~compilation_unit;
  prefix: ~quote ~list_of(~decl_item);
  spec: ~quote ~decl_item) return ~compilation_unit;
function ~library_body(
  spec_cu: ~quote ~compilation_unit;
  prefix: ~quote ~list_of(~decl_item);
  cu_body: ~quote ~decl_item) return ~compilation_unit;
function ~library_subunit(
  context: ~quote ~compilation_unit;
  prefix: ~quote ~list_of(~decl_item);
  su_body: ~quote ~decl_item) return ~compilation_unit;

function ~separate return <>; -- "is separate" in Ada

-- Generics
type ~generic( -- similar to function
  gpl: ~quote ~list_of(~fp_item);
  rt: ~type) is private;
function ~instantiation_decl( -- similar to ~vis_decl
  name: ~token;
  t: ~type;
  value: t) return ~decl_item;
function ~implicit_dot_qual(x, y: <>) return ~type of(y) renames ~dot_qual;
-- used only for references to fields of "used" instantiated generic packages.

-- Representation Specifications
type ~address is private;
-- the address type required in Ada package system is
-- subtype address is ~address;
function ~address_rep_spec(x: ~quote <>; a: ~address) return ~decl_item;
function ~length_size_rep_spec(
  t: ~type;
  val: ~static(~universal_integer)) return ~decl_item;
function ~length_storage_size_rep_spec( --+ length_storage_size_access_rep_spec
  t: ~notatypo(~access);
  val: ~universal_integer) return ~decl_item;
function ~length_storage_size_rep_spec( --+ length_storage_size_task_rep_spec

```

```

t: ~metatype(~task);
val: ~universal_integer) return ~decl_item;
function ~length_small_rep_spec(
t: ~metatype(~universal_fixed);
val: ~static(~universal_real)) return ~decl_item;
function ~enum_rep_spec(t: ~type; val: <>) return ~decl_item;
~unaligned: constant := 1;
function ~record_rep_spec(
t: ~metatype(~record);
alignment: ~static(positive);
component: ~list_of(
<>);
~static(~universal_integer);
~set_of (~independent (~static(~universal_integer))) return ~decl_item;
function ~address_attr(x: ~quote <>) return ~address;
function ~first_bit_attr(x: ~quote <>) return ~universal_integer;
function ~last_bit_attr(x: ~quote <>) return ~universal_integer;
function ~position_attr(x: ~quote <>) return ~universal_integer;
function ~size_attr(x: ~quote <>) return ~universal_integer;
function ~storage_size_attr(x: ~quote <>) return ~universal_integer;

-- ACE-specific declarations:

type ~ace_command is private;

function ~make_ace_command(s: ~statement) return ~ace_command; --+ ace_command_statement
function ~make_ace_command(d: ~decl_item) return ~ace_command; --+ ace_command_decl

-- A declaration or statement given to ACE is represented as an
-- application of ~make_ace_command. An unambiguous resolution of
-- the entire construct indicates correctness for ACE.

function ~ace_history_decl(
name: ~token;
history: ~list_of[~ace_command]) return ~decl_item;

-- A declarative operator that "records" the sequence of ACE commands.
-- It is intended as a short-term mechanism to allow the command
-- history to be stored in a library, primarily for debugging purposes.
-- An ACE history declaration is used as the third argument to an
-- application of ~library_spec. The name argument is used as the simple
-- name of the library unit.
--
-- Although ~ace_history_decl is a declarative operator, it differs
-- from normal declarative operators in many respects. It does not
-- make its name visible, and it has no type parameter (the second
-- argument of other declarative operators). Like other declarative
-- operators, it introduces a single new scope; all elements in the
-- history list act as if they were declared in that scope.

end standard;

```

```

t: ~metatype(~task);
val: ~universal_integer) return ~decl_item;
function ~length_small_rep_spec(
t: ~metatype(~universal_fixed);
val: ~static(~universal_real)) return ~decl_item;
function ~enum_rep_spec(t: ~type; val: <>) return ~decl_item;
~unaligned: constant := 1;
function ~record_rep_spec(
t: ~metatype(~record);
alignment: ~static(positive);
component: ~list_of(
<>);
~static(~universal_integer);
~set_of (~independent (~static(~universal_integer))) return ~decl_item;
function ~address_attr(x: ~quote <>) return ~address;
function ~first_bit_attr(x: ~quote <>) return ~universal_integer;
function ~last_bit_attr(x: ~quote <>) return ~universal_integer;
function ~position_attr(x: ~quote <>) return ~universal_integer;
function ~size_attr(x: ~quote <>) return ~universal_integer;
function ~storage_size_attr(x: ~quote <>) return ~universal_integer;

-- ACE-specific declarations:

type ~ace_command is private;

function ~make_ace_command(s: ~statement) return ~ace_command; --+ ace_command statement
function ~make_ace_command(d: ~decl_item) return ~ace_command; --+ ace_command decl

-- A declaration or statement given to ACE is represented as an
-- application of ~make_ace_command. An unambiguous resolution of
-- the entire construct indicates correctness for ACE.

function ~ace_history_decl(
name: ~token;
history: ~list_of[~ace_command]) return ~decl_item;

-- A declarative operator that "records" the sequence of ACE commands.
-- It is intended as a short-term mechanism to allow the command
-- history to be stored in a library, primarily for debugging purposes.
-- An ACE history declaration is used as the third argument to an
-- application of ~library_spec. The name argument is used as the simple
-- name of the library unit.
--
-- Although ~ace_history_decl is a declarative operator, it differs
-- from normal declarative operators in many respects. It does not
-- make its name visible, and it has no type parameter (the second
-- argument of other declarative operators). Like other declarative
-- operators, it introduces a single new scope; all elements in the
-- history list act as if they were declared in that scope.

end standard;

```

Experience Using Iris

Extended Abstract

David A. Mundie, David A. Fisher,
Deborah A. Baker, and Jonathan Shultis

*Incremental Systems Corporation
319 S. Craig Street
Pittsburgh PA 15213 U.S.A.¹*

Iris as an Internal Representation

An Overview of Iris. Iris is an internal form for representing utterances (i.e. programs) written in any formal language. Developed at Incremental Systems as part of an Ada compiler development project, Iris is intended as a common medium of information exchange in a fine-grained component-based programming environment based on multiple cooperating tools.

From a theoretical perspective, Iris is best seen as a generalization of the internal forms commonly used to represent first-order abstract syntax. It differs from those forms in that it is a *higher-order* system which treats operators as calls on operator-returning declarations. This frees Iris from any particular choice of operators, thus contributing to language independence. As we shall see, it also is critically important in ensuring that different tools can work independently on the same data base.

An Iris tree is composed of two kinds of nodes: *reference nodes* and *application nodes*. For example, the expression $f(x,g(y,z))$ consists of references to entities named f , x , g , y , and z , and applications of f and g . The corresponding Iris tree is shown in Figure 1, where circles

depict application nodes and squares depict reference nodes. It is important to note that Iris is a *semantically-based* system, in the sense that reference and application are not just *syntactic* constructs (as are the more common "nonterminal" and "terminal"), but rather are to be *interpreted* as references to declarations and as function application, respectively.

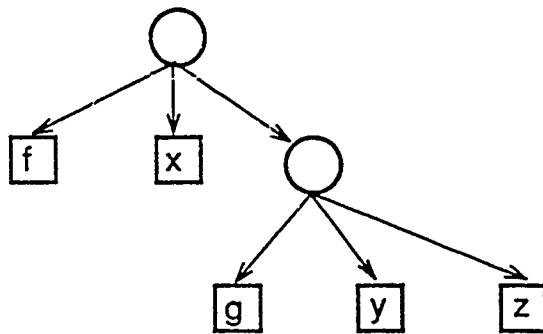


Figure 1: An Iris tree for the utterance "f(x,g(y,z))."

The first child of an application node is its *operator*, which identifies an *operation* applied to the remaining children, which are called *arguments* or *actual parameters*. Frequently, the operator is a reference to the declaration of a named operation, but it can be any operation-valued expression represented as an Iris tree.

In Iris, an *entity* is anything that can be defined, computed, or named. Depending on the language being supported, entities could include such things as constants, variables, types, functions, packages, exceptions, statements, declarations, axiomatic specifications, operational descriptions, and proofs. Some entities can be used as operations. Every operation has an associated *signature* which specifies the composition rules for the operation. In general all that Iris requires for the specification of composition rules is a concept of *type* for the purposes of type matching, and the concept of composition itself; everything else is language-specific. For example, in Ada the signature for an operation is defined by the

number, names, and types of its arguments and the type that an application of it yields.

Some operations, called *declarative operations*, associate identifiers with entities. An expression whose primary operator identifies a declarative operation is called a *declarative expression* or *declaration*. A declaration is an actual association of an identifier with an entity. The scope and visibility of a declaration is determined by other operations, consistent with the definition of the language in which the utterance is written.

Expectations for Iris. The principle benefits we expected from Iris were: (1) compact trees (2) ease of modification of the static semantics of the language being supported through modification of its signatures (3) flexibility in defining the attributes the tree possesses (4) independence of cooperating tools (5) compactness of code. In the remainder of this paper we examine our practical experience using Iris with an eye towards evaluating the degree to which those expectations were fulfilled.

Using Iris During Compiler Construction

Iris as an Intertool Insulator. The attribute system underlying Iris was designed to allow each tool in the environment to have its own view of the attributes of a program representation (Figure 2). Apart from the minimal skeleton required to define the tree structure itself, the attributes of the program tree are tool-specific. Tree nodes are not permanently defined and allocated records with a fixed number of fields, but are instead flexible, noncollocated bundles of attribute values. Each tool can select those fields that it needs, and need not concern itself with the fields that it does not use.

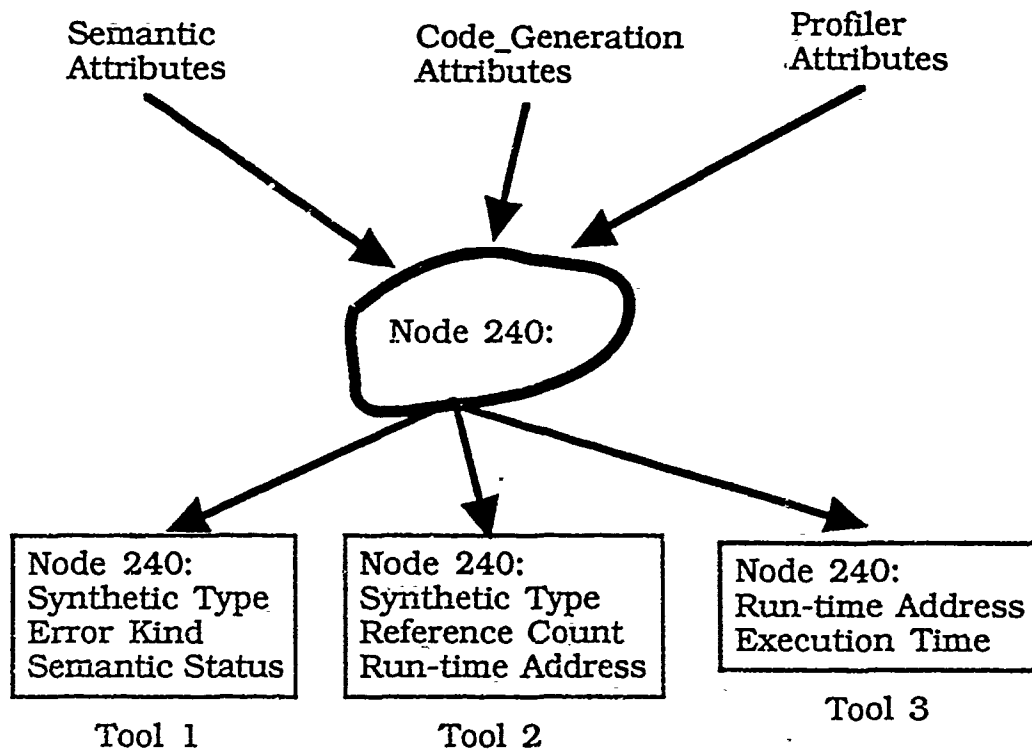


Figure 2: Each Tool can have its own view of what the program tree looks like, since nodes are partitioned horizontally.

This scheme greatly reduces the memory requirements for each individual tool, but its primary advantage is that it allows tools to operate *independently*. As we saw constantly during the compiler development, different tools can define their own local attributes without disrupting the other phases of the compiler. For example, at one point in development the Reference Counter decided that it needed an attribute to record whether or not a subprogram references global variables. Using Brand X, this would have been a major undertaking; a field would have had to be added to the global node definition, and every tool using the node definition would have had to be recompiled. Worse yet, that field would have become a permanent, globally visible attribute of the program tree. Using Iris, the Reference Counter

simply declared a local attribute to store the information and went on. To the Reference Counter, nodes had sprouted a new field, but to the other tools the node definition had remained unchanged.

Iris as an Intertool Communications Medium. The converse of the use of Iris to insulate tools from one another is its use as a medium of exchange among tools. Just as any tool could decide to add a local attribute, so any tool could decide to *reuse* an attribute without any impact on other tools. For example, the Semantic Analyzer, to optimize its processing, computed the representation type of program entities; this can dramatically speed up type checking, since it is known that two entities with different representation types cannot possibly be type-compatible—this cheap test can thus avoid the relatively expensive process of full-blown type checking. At one point the Code Generator got to the point where it needed to know the representation types of program entities. All that the Code Generator needed to do was to request that the *Rep_type* attribute be made visible to it; no global redefinition was required.

We have long felt that it is only this sort of *unanticipated* reuse that can lead to a free economy of tools where each component of the environment can borrow the results it needs from the other tools, and can in turn contribute its own results to the growing pool. What surprised us was how important this effect was even in a project as small as a compiler, since to us its primary utility will be in large-scale programming environments.

The flexibility of the Iris Attribute System comes in large measure from the fact that it places no restriction on the type of the attributes it manipulates. There are actually two questions here: (1) can tools declare the types of attributes, and declare them to be of arbitrary types? (2) does the system support type integrity? Although the Iris Attribute System in its current implementation does allow arbitrary user-defined types, there is currently no support

for type integrity within the attribute system—there is nothing to prevent a tool from looking at a string attribute as an array of integers, for example. What is missing is a method for tying together the type-checking mechanisms provided by Iris and the user-defined types provided by the attribute system. In a future implementation of the system, we intend to do just that through the mechanism of universal identifiers; each attribute type will be linked to a type declaration by means of that declaration's universal identifier.

Primitivelessness and self-definition. A key feature of Iris that turned out to be important in the development of the compiler was the fact that the static semantics of the language being implemented were described within the language itself. That is to say, the composition rules for Ada were specified in a "table" consisting of about one hundred and fifty Ada² signatures defining the operators of Ada. A sample of these specifications is given in Figure 3.

```
function "!="(x: out <>; y: type_of(x)) return statement;
type "function"(fpl: quote list_of [formal_parameter_item]; ret_type: type) is private;
function ".."(left, right: scalar) return set_of(scalar);
type "array"(index _type: list_of[independent(metatype(discrete))]; field_type: type) is
    private;
function "if"(x: quote list_of[boolean; statement]) return statement;
```

Figure 3: Some representative declarations from Iris-Ada.

The importance of this self-definition in constructing the compiler is difficult to overemphasize. All the traditional benefits of being table-driven accrue, but they extend to semantic analysis, and are not limited to syntactic processing. Instead of being embedded in the code for the compiler, the static semantics for the language is encoded in easy-to-read

²Actually, Ada extended with a few compile-time type-valued operators, and a couple of other minor additions.

Ada code that can be modified by simply changing signatures.

What this means is that operators can be added to the language *without* requiring modification of the tools that process the language. In a traditional system of first-order terms, all the operators of the language must be known to all tools, since there is no other way for the tool to "check all cases." This need for massive changes—scattered through tools whose very *existence* is difficult to detect in a large environment—is a major obstacle on the path to component-based environments. Because Iris is a higher-order system, tools can tell from the *declarator* which cases it needs not concern itself with, and that obstacle vanishes.

A further advantage to this approach is that the distinction between built-in operators and user-defined operators evaporates. The same representations and the same processing mechanisms are used for both. The importance of this primitivelessness lies in the fact that any given tool has a subset of operators which it "knows" about, that is, which it processes as special cases. The catch is that that subset varies from tool to tool, so that to build in any particular subset as primitive would condemn all tools for all time to treat that subset as special cases, even when they did not need to.

For example, consider the Semantic Analyzer and the Code Generator. The scope and visibility operators are of necessity special cases for Semantic Analysis, since they cause entries and deletions in the symbol table. To the Code Generator, however, they are *not* special cases; when doing reference counting, the Code Generator does not need to distinguish them from any other operators, and because Iris-Ada contains the signatures of those scope and visibility operators, that is exactly what the Code Generator does. On the other hand, operators such as the representation specification operator which are inherently special for code generation are plain vanilla for Semantic Analysis; because the specification for the representation specification operator is included in Iris-Ada, the semantic analyzer does not need to know about it, and can simply process it in the normal manner.

Regularity and general cases. A major concern of our compiler project was to minimize the complexity of the compiler by using algorithms that were as general as possible and therefore small and easily comprehended. We employed two principle means in achieving this end. The first was to generalize as much as possible the interpretations given to Ada operators. For example, entries were treated as subtypes of type "function," so that most processing could treat them uniformly.

The second means to generality was a set of conventions that ensured a uniform, regular representation of information in the Iris tree. For example, all declarators take the name of the thing being declared as their first parameter and its type as the second parameter. All of Ada's syntactically variegated type declarations were transformed to have a common tree shape.

This regularity of structure means that it is easy to extract information about types, declarations, signatures, bodies, and the like from the tree. It is difficult to convey the flavor of these algorithms without a lot of preparatory explanation, but a simple example is given in Figure 4, which shows the essence of the subprogram which computes the type of any node in the tree.

```
function type_of(exp: tree) return "type" is
begin
  if exp.is_reference then
    -- the second child of a declaration contains the type
    return exp.op.x[2];
  else -- in the case of a function application, exp.op.x[2] is "function";
    -- the second parameter to function is its result type
    return exp.op.x[2].x[2];
  end if;
end type_of;
```

Figure 4. A representative subprogram from the PDL code for the compiler.

Of course, the concrete syntax tree of an Ada program does not come with this regularity built in. To mould the tree into the requisite shape, numerous transformations are required while parsing, and to a lesser extent during semantic analysis. These transformations are of course not required by Iris, but are rather language-dependent optimizations to facilitate later processing.

Lessons Learned

Ever-greater regularity. The importance of transformations for ensuring regularity of structure during the early phases of the compiler was always clear to us. Their importance during the later phases of the compiler was clear to us also, but in spite of this we adopted a rule that we would refrain from making such transformations because of the nuisance of reusing tree space and doing garbage collection. We concluded that we had made the wrong choice, and that in the long run it would have been cheaper to build in garbage collection, only after we had expended significant effort making special cases out of constructs that could have been treated as general cases if we *had* allowed transformations in the back end. For example, the declaration of the loop control variable in for loops should have been treated like any other variable declaration, but could not be because the tree was not of the right shape. One thing that convinced us we had made the wrong choice was that many of the attributes the Code Generator added to the tree were in effect attributes that recorded the transformations that the Code Generator *would* have made if it could have; it then used those attributes to simulate the transformations on the fly.

Language support. The use of Iris could be greatly facilitated by language support for noncollocated records. Although simple in concept, such records must be implemented in

infelicitous notation in most existing programming languages. For example, the Ada rendering of the statement "return exp.op.x[2].x[2]" from Figure 4 would be something like:

```
exp_op := expression(exp.op);  
exp_op_x2 := expression(exp_op.x[2]);  
return exp_op_x2.x[2];
```

Rigorous adherence to coding conventions reduced the problem to manageable proportions, but everyone who actually coded the compiler yearns for the day when record types are freed from the collocation assumption.

This is especially so since any programming language that supports records has the basic machinery to support noncollocated attributes. Only two language additions are needed. The first is an "extensible" record specification mechanism, and the global coordination of field names that that requires. The second is a mechanism for specifying which fields are to be collocated.

Conclusions

Our expectations about the beneficial effects of Iris on both tree size and code size were born out by our experience. Considering just the "core" attributes needed to describe the skeleton of the tree, reference nodes in Iris trees occupy 4 bytes, and application nodes 7 bytes. We thus can handle a compilation unit of 2,000 lines in less than 512K bytes of main memory.

The compiler itself, including an optimizing code generator, consists of fewer than 30,000 lines of code, compared to the 200,000-500,000 lines in most commercial Ada compilers. The semantic analyzer, generally the largest portion of an Ada compiler, is fewer than 2,000 lines.

What is harder to assess is the qualitative impact of using Iris on the compiler effort. In terms of human resources, the project was not significantly smaller than other Ada compiler

efforts. We suspect that this was to a large extent the effect of the learning curve, and feel that using Iris for a new language or a rewritten compiler *would* have substantial benefits in terms of programming costs. In the last analysis, though, the principle benefit was that the compiler which resulted uses algorithms whose correctness it is feasible to check by inspection and a language definition whose static semantics are scrutinizable by mortals.

Management of Small- and Large-Grained Objects in the Iris Framework

Deborah A. Baker, David A. Fisher, Frank P. Tadman

Incremental Systems Corporation*
319 South Craig Street
Pittsburgh, Pennsylvania 15213, USA

March 30, 1990

1 The Iris Framework for Information Management

There have been many advances in the last twenty years in technology, tools and environments for the design, implementation and maintenance of software applications. While many of these component technologies are demonstrably effective for limited aspects of the software process, there has been no way for them to work cooperatively and there has not been a marked improvement in software productivity.

Iris¹ is a flexible, open-ended and easily extended framework for information representation. Iris information (i.e. object) management enables sharing and communication of information among tools and tool components, thus promoting their integration.

Objects (i.e. information) in a software environment are diverse. They obviously include requirements, designs, specifications, implementations and results of programs. Also included are representations of these objects in the form of source text, internal representations, and object and target code. Artifacts from analysis, documentation, testing, project management and maintenance processes are objects. It is crucial that the various tools and tool generators of the environment are themselves first class objects. The objects in a software environment are diverse in type, vary in their relationships, and may exist simultaneously in a variety of versions and configurations. Objects exist from a wide range of both logical and physical granularity from the very small (a bit or a binary digit) to the very large (many megabytes or an entire database). One of the challenges of object management systems is to remain effective across the entire spectrum of object granularity.

Iris provides a powerful infrastructure for the representation and management of the diverse objects of a software environment. A full Iris system includes an information structure, small-grained object management and large-grained object management.

*This work supported in part by the Defense Advanced Research Projects Agency (Arpa Order 6487-1) under contract MDA972-88-C-0076.

¹Iris is the Greek goddess of the rainbow, and messenger of the gods.

In Section 2, the Iris information structure is introduced. Small- and large-grained object management, and the user perspective thereon, are discussed in Section 3. Finally, the work at Incremental Systems that supports the Iris framework is discussed in Section 4. In the full paper, each of these sections will be expanded, and include more detail.

2 The Iris Information Structure

An Iris information structure is a representation of an utterance in some formal language (such as a specification, implementation or design language). At the highest level of abstraction, the Iris information structure is a tree composed of *applications* and *references*. Corresponding to this, an Iris tree is composed of two kinds of nodes: *reference nodes* and *application nodes*. For example, the expression $f(x, g(y, z))$ consists of references to entities named f , x , g , y , and z and applications of f and g . The corresponding Iris tree is shown in Figure 1. Circles depict application nodes and squares depict reference nodes.

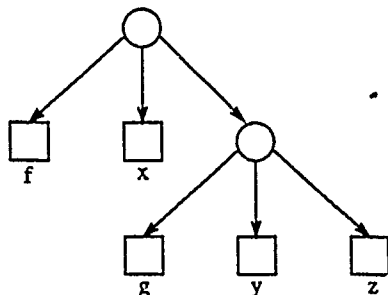


Figure 1: An Iris Tree

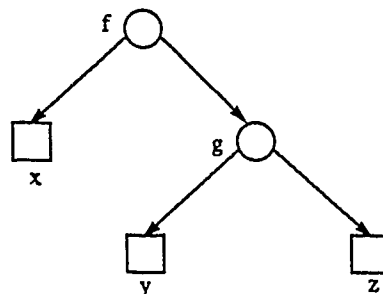


Figure 2: A Simplified Iris Tree

The first child of an application node is its *operator*. The operator identifies an *operation* which is applied to the remaining children, which are called *arguments* (actual parameters). Frequently, the operator is a reference to the declaration of a named operation, but it can be any operation-valued expression represented as an Iris tree.

To avoid clutter, Iris trees are often drawn in the style shown in Figure 2 where the name of the operation is shown next to the application node. In this case it is understood that the operator is a single reference node referring to the named operation and is not shown.

In Iris, an *entity* is anything that can be defined, computed, or named. Entities include such things as constants, variables, types, functions, packages, exceptions, statements, declarations, axiomatic specifications, operational descriptions and proofs. Some entities can be used as operations. Every operation has an associated *signature* which specifies the number and type of the arguments that an application of it requires and the type that the application yields.

The Iris information structure is similar to commonly used internal forms for (first-order) abstract syntax, but differs from them in that it demotes the operator from a nonterminal class to a distinguished subtree. This frees Iris from any particular choice of operators, thus contributing to its language independence.

3 Iris Small- and Large-Grained Object Management

Current state-of-the-art object management systems distinguish between large- and small-grained objects. The distinction is based on the use of an object, not merely its size (as it is possible to have physically small large-grained objects as well as physically large small-grained objects).

Large-grained objects tend to be independent entities. Each large-grained object can be placed (moved) independently and has a unique, universal, location independent identity. Small-grained objects, on the other hand, are grouped into collections, with each collection typically represented as a large-grained object. Each small-grained object is then placed and identified only as a member of a collection.

There are certain tradeoffs that are made in choosing between large- and small-grained objects. Factors that would influence the choice include frequency of access, the nature of relationships with other objects and performance requirements.

In the Iris framework, a *segment* is a container for an indexed collection of small-grained objects called *items*; each segment is a large-grained object. An *item manager* implements a particular form of small-grained object management. An *object manager* provides the facilities needed for large-grained object management.

4 Basis for Iris Research and Implementation

Incremental Systems Corporation has several ongoing projects involving both research and practical development which provide the experience and basis for the Iris work described in this abstract.

- Ada-to-Iris: For this project, a tool to translate Ada source text into a common Iris representation is being developed. It includes the Iris tree form, a variety of small-grained object management mechanisms and rudimentary large-grained object management. Iris-Ada is the name of this specialization of Iris for Ada. Under Iris-Ada, each Iris tree represents an Ada library unit. An Iris-Ada tree is represented as collections of attributes with each node consisting of its identity and its attributes. Attribute management is a particular kind of small-grained object management while management of Ada library units constitutes large-grained object management in this project.
- Full spectrum language: The goal of this effort is to produce a persistent framework in which information can be expressed, captured, reused, improved and built upon. A fully generalized Iris system will provide the internal form for expression of this information and will include both small- and large-grained object management.
- Mechanisms: For this project, a set of *mechanisms* for (large-grained) persistent object management in a distributed software development environment were designed.

In addition to our own projects, Iris has been adopted by research and research and development groups at several universities and corporations.

Draft
Nov 12, 1988 13:29

Draft Report
on
Requirements for a Common Prototyping System

Chairman: Robert Balzer
Editor: Richard P. Gabriel

Common Prototyping Working Group: Frank Belz, Robert Dewar, David Fisher,
John Guttag, Paul Hudak, Mitchell Wand.

Draft Dated: Nov 12, 1988 13:29
Unlimited Distribution and Not for Publication
All Rights Reserved

Prototyping 1

Optimized Overload Resolution and Type Matching for Ada

David A. Mundie
David A. Fisher

Proceedings of the Symposium on
Environments and Tools for Ada (SETA-1)

Rodondo Beach, California
April 1990

Technical Report Number 900401

Incremental Systems Corporation
319 South Craig Street
Pittsburgh, Pennsylvania

Optimized Overload Resolution and Type Matching for Ada*

David A. Mundie and David A. Fisher

*Incremental Systems Corporation
319 S. Craig Street
Pittsburgh PA 15213*

1 Introduction

The challenge of implementing Ada's overload resolution has elicited a number of different algorithms and implementation strategies. Most algorithms use varying numbers of alternating top-down and bottom-up passes: the top-down passes use inherited information to eliminate candidates that do not produce the desired result type, while the bottom-up passes use synthesized information to eliminate candidates that do not have the correct types for their parameters. The number of passes required has decreased steadily from unbounded [Ichbiah 79] to four [Ganziger 80] to two [Pennello 80] to one [Baker 82, Stockton 85]. However, the smaller number of passes have been purchased at the price of increased storage, increased complexity, and often, surprisingly, increased processing time.

The algorithms in this class have two major drawbacks, especially when implementing an incremental programming environment. First, they use auxiliary storage to maintain the candidate sets which are manipulated during the walking of the tree. This storage is potentially quite large, and the complexity of maintaining it is unattractive. Secondly, they are essentially batch-oriented algorithms. Their application requires that the entire subtree under consideration be processed; nothing in them permits pruning branches of the tree that can be shown to be unaffected by an editing change.

The algorithm presented in this paper takes a different approach. Our method does away with

candidate sets entirely by adopting the recursive strategy first proposed by Cormack [Cormack 81]. This greatly simplifies the method, and makes it more suitable for integration into an incremental environment. We also have incorporated into the algorithm a number of pruning heuristics which improve its practical performance. The most important of these heuristics prunes the tree at nodes which have no more than one candidate—a high percentage of the nodes in most programs.

2 The Baseline Algorithm

For clarity of presentation we will develop the fully optimized algorithm in three stages. In this section we shall examine a baseline algorithm and in the next we present our refinements of it.

All versions of our algorithm assume that the input to overload resolution will be generalized parse trees represented in IRIS, the internal representation developed at Incremental Systems for sharing information in tool-based environments [Fisher 89, Mundie 89]. Tree nodes in IRIS are classified as either *application* nodes representing function application or *reference* nodes representing references to entities. Every application node in the parse tree consists of an operator node followed by a varying number of arguments. After parsing, the operator points to the string which is the name of the operator. The task of overload resolution is to determine what operation is intended by that name, and to fill into the operator a pointer to the declaration of that operation, so that subsequent passes of the compiler will have immediate access to the information (types of parameters, result type, etc.) which is contained within that declaration. Figure 1 illustrates this process using a simple example. The operator "f" has as its declaration "function f(i: integer) return boolean." After parsing, the call "f(4)" is represented as an operator pointing to the string "f", followed by a single operand pointing to the integer literal "4." The overload resolver, when processing this node, must decide which of all the f's in the program is the one referred to by this call and assign the correct declaration of "f" to

* This work was supported in part by the Defense Advanced Research Projects Agency (Arpa Order 5057) monitored by the Department of the Navy, Space and Naval Warfare Systems Command under contract N00039-85-C-0126.

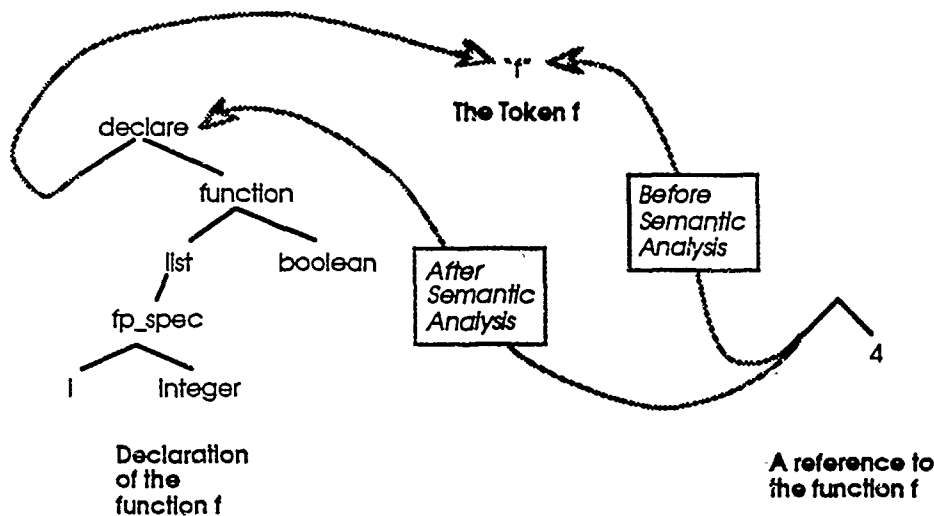


Figure 1. Overload resolution determines and retains the correct operator declaration for each node.

the operator of the node.

The baseline algorithm, shown in Figure 2, generates every legal configuration of the tree. It traverses the parse tree top-down; at each node, the list of currently visible declarations is searched for entries which have the same name as the current operator, and whose formal parameter list is compatible in terms of number of parameters, the names of named actual parameters, and default parameters. For each such entry, we check whether the actual parameters to the current node match the corresponding formal parameters. If the types of all the parameters match, then we have found a legal configuration for the current node.

The algorithm does a **coreturn** once for each legal configuration. When the list of candidates is exhausted, it falls off the end and does a normal return. For clarity, the algorithm uses an "embedded for loop" construction to represent an arbitrary number of nested loops. For example, the embedded for loop in Figure 2,

```

embedded for i in 1 .. length of apl do
  for each resolve(apl(i)) loop
    If type of fpl(i) matches type of apl(i) then od...

```

would get expanded, assuming the length of the apl were 3, into:

```

for each resolve(apl(1)) loop
  If type of fpl(1) matches type of apl(1) then
    for each resolve(apl(2)) loop
      If type of fpl(2) matches type of apl(2) then
        for each resolve(apl(3)) loop
          If type of fpl(3) matches type of apl(3) then...

```

That is, the loop body is nested into as many for

generator resolve(tree) is
begin

If tree is a reference then

```

  op:= tree.op;
  for each visible declaration d loop
    If name of d = op then
      tree.op:= d;
      coreturn;

```

else -- tree is an application

apl:= actual parameter list of tree;

for each resolve(tree.op) loop

If tree.op is a function then

fpl:= formal parameter list of tree.op;

apl:= actual parameter list of tree;

If number of parameters, parameter names, and defaults match between fpl and apl then
embedded for i in 1.. length of apl do

for each resolve(apl(i)) loop

If type of fpl(i) matches type of apl(i) then od
coreturn;

end resolve;

Figure 2. The baseline algorithm.

loops as there are actual parameters.

This algorithm is similar to the one presented by Cormack. We chose it as the starting point of our research because of its clarity and simplicity. Because it is a top-down, recursive algorithm, it lends itself to analyzing trade-offs, identifying possibilities for pruning, and experimental modifications.

3 Synthetic Refinements to the Algorithm

Despite its merits, the baseline algorithm given

type config_status = (reprocess, unique, no_match); ①
 — the status field is initialized to reprocess in all nodes

generator resolve(tree) is
 begin

```

  If tree.status = reprocess then ;
    n := 0; — begin count of synthetic configurations ③
    If tree is a reference then
      op := tree.op;
      for each visible declaration d loop
        If name of d = op then
          tree.op := d;
          n := n + 1; — count synthetic configurations ③
          coreturn;
        else — tree is an application
          apl := actual parameter list of tree;
          for each resolve( tree.op ) loop
            If tree.op is a function then
              fpl := formal parameter list of tree.op;
              If number of parameters, parameter names, and
                defaults match between fpl and apl then
                embedded for i in 1.. length of apl do
                  for each resolve( apl(i) ) loop
                    If type of fpl(i)
                      matches type of apl(i) then od
                    n := n + 1;
                    — count synthetic configurations ③
                    coreturn;
            If n = 1 then ④
              tree.status := unique; — mark subtree as unique
            elsif n = 0 then ⑤
              stop_error := false;
              If not tree is a leaf and tree.op is a function then
                for i := 1 to length of apl loop
                  If apl(i).status = reprocess then ⑥
                    resolve( apl(i) ); — process all nodes
                  elsif apl(i).status = no_match then
                    stop_error := true;
              If stop_error then ⑦
                tree.status := unique; — inhibit error propagation
                coreturn;
            else
              tree.status := no_match; — mark subtree as no match
              report_error( tree, 'no resolution of subtree' );
            elsif tree.status = unique then ⑧
              coreturn; — return one configuration for unique
            — if tree.status = no_match, just bounce off ⑨
  end resolve;

```

Figure 3. Synthetic optimizations and error recovery.

above suffers from two serious drawbacks which make it unusable in a production-quality or incremental system. (1) Its performance is exponential in the size of the tree in both the worst case and in the expected case. (2) It has two serious deficiencies in the area of error handling. First, the children of a node are never processed if the node is il-

legal; thus an overload error at the top of a parse subtree results in the user's program going unprocessed, and any errors there being masked and unreported. Secondly, errors are propagated up the tree: if a given node is found to be illegal, its parent will be also, and its parent, and so on, so that a single overload error near the bottom of the tree results in large numbers of nodes being marked illegal. Taken together, these aspects of the algorithm mean that correct operator selection will be performed *only when the entire tree is free of errors*. This is, of course, understandable, since a primary purpose of overload resolution is to determine whether the tree as a whole is correct. For meaningful error reporting, however, the error status of individual nodes must often be determined by ignoring certain errors in their ancestors and descendants.

The improvements we have made to the baseline algorithm address both of these areas. The basic idea behind the improvements is that the reprocessing of nodes may be greatly improved by recording in each node the history of its previous processing, particularly those properties which will remain invariant during subsequent reprocessing. This history is captured during processing as the number of legal configurations found for the subtree, and then recorded in the tree as a status field indicating how future processing of the node should be done.

We consider first the improvements that can be made by exploiting the synthetic information provided by the analysis of the actual parameters to a node. The changes introduced are shown in italics in Figure 3. The first change (labeled ①) is to mark each node of the tree with a configuration status field with three values: *reprocess*, *unique*, and *no_match*. This field is initialized to *no_match* by the parser. It is this field that encodes the history information required for the optimizations and error handling described above.

The second change is to count the number of legal configurations for the given node. A counter is initialized on entry to the generator and incremented whenever a candidate operator is found whose formal parameter list type-matches with the types of its actual parameters (③).

The next change is to use the number of legal configurations to compute the configuration status of the node. The most important case is where only one legal configuration was found (④). What this means is that overload resolution can be performed for this node *using synthetic information alone*. That is to say, no inherited information from the parent, viz. the type the parent expected this node to turn out to be, can possibly affect the overload resolution for this node—there simply were no other legal configurations. When this is the case, we mark the tree's status as *unique*. The benefit from this is that in subsequent reprocessing, we can

simply skip the body of the routine, and do a single coreturn reflecting the fact that there is a unique legal configuration (©).

The reason we attach such importance to this optimization is that we expect it to cover the vast majority of the cases in typical Ada programs. Even in programs that make extensive use of overloaded operators, it is rare that synthetic information is not sufficient by itself to do the resolution. In fact, almost no language other than Ada even permits overloading based on return type. This optimization alone reduces the time for overload resolution in the expected case from exponential to linear.

The second special case in computing the status is where there were no legal configurations at all (©). This means that based solely on synthetic information we were able to determine that this is an illegal node with no valid choice of operators. As with unique nodes, subsequent processing can simply "bounce off" this node, since no further information could change the outcome; in the *no_match* case, however, we do not even need to do the coreturn, since there is no legal configuration to generate (©).

Two complications arise, however, at the point that we detect the *no_match* case. The logic of the generator is such that it quits processing actual parameters as soon as the first failure is encountered. From an optimization point of view this is correct, since once one parameter has failed, we know the node is illegal, and as Cormack points out, we are really not interested in exactly *how* illegal it is. From an error correction point of view, however, it has the undesirable consequence that actual parameters beyond the first failure may be left unprocessed. These parameters might, however, contain errors which the user should know about, so leaving them unprocessed is unsatisfactory. Our solution is simply to loop over the actual parameter list looking for unprocessed nodes and calling *resolve* on them to guarantee that they are all processed (©).

The second complication is that we must prevent errors which result from type matching failures from being propagated up the tree. We accomplish this by checking for children marked *no_match* at the same time we are looking for unprocessed children (©). If any are found, we simply mark the current node as *unique*¹, so that future calls will simply bounce off (©).

As an aside, we should point out that the

¹ The algorithm as shown marks nodes unique during the second call on *resolve*. In actuality, the node should be marked unique during the first call on *resolve*, not the second. This optimization does not affect the order of the algorithm, but is a significant savings in practice especially because the inherited optimization discussed in the following section depends on the parent node being marked unique. The details of achieving this have been left out so as not to cloud the algorithm.

put(f(3, g(1) * h(a+9)), invert(g(2) + h(1)))

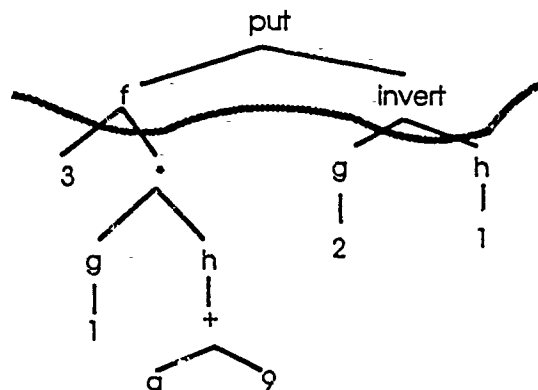


Figure 2. Unique Nodes Act as Dikes.

unique nodes used above to optimize reprocessing can in an incremental environment be used as "dikes" which prevent editing changes from being propagated throughout the tree. Consider for example the program fragment "put(f(3, g(1) * h(a+9)), invert(g(2) + h(1)))" as shown in Figure 3. Suppose that because of an editing change this subtree must be re-analyzed. Without our heuristic, the entire subtree would have to be re-traversed, as would be done by the baseline algorithm. If, however, "f" and "invert" have only one candidate based on name and number of parameters, then we do *not* have to re-examine their parameters, since we know that nothing below them could change as a result of changes above them. The restriction of overload resolution to sub-statements and sub-declarations falls out of this strategy naturally without treating statements and declarations as special cases.

4 Inherited Optimizations to the Algorithm

The performance of the algorithm can be enhanced still further by a judicious exploitation of inherited information. The key observation here is that although passing *all possible* desired types down to the children by means of candidate lists is unappealing, a practical and low-cost approximation to that strategy is to pass down the *single* desired type in the very common case where there is only a single legitimate choice for the parent.

Figure 4 shows the required changes in the algorithm, as well as the changes needed to add ambiguity detection. A new parameter to *resolve* (©) passes down the inherited type required; if the parent is not unique, then *null* is passed instead. Before doing operator selection (©) and before processing the actual parameter list (©), the required type is checked against the type which would be returned by the configuration; if the types do not match, then the subsequent processing can be skipped. At those places where *resolve* is called re-

cursively but it is not known that the configuration is unique, we must be careful to pass null as the required type (④), and we must split the unique case out during the processing of the actual parameter list (⑥).

The final algorithm shown in Figure 4 also adds a new status type to handle ambiguous trees (①). Whenever more than one legal configuration has been found for a node, it is marked ambiguous; this allows us to minimize reprocessing by returning only a single configuration on subsequent calls (⑦)

5 Analysis of the Algorithm.

As mentioned above, the baseline algorithm itself is exponential in the size of the tree. We define m = the average number of candidates for a given node, k = the average number of parameters to a given non-terminal, n = the number of non-terminal nodes in the parse tree, and h = the height of the parse tree. The performance of the baseline algorithm is $O(m^h n)$ for nonterminal nodes, since it visits each node once recursively for every candidate of its parents. In addition, bottom nonterminal nodes must process each of their k terminal descendants for a total time of $O(km^h n)$. In the case of the baseline algorithm, there is no time difference between best case, worse case, and expected case. The space requirements are constant per node, i.e. $O(n)$.² We note that on the average $h = \log_k n$ and that for large Ada programs $k \approx 2.1$.

As mentioned above, the optimizations we have given greatly improve the expected-case performance of the algorithm. It is in fact *linear* whenever nodes can be resolved using only synthetic information or when their parent is unique, since in these cases the node will be marked **unique** and subsequent calls on resolve will simply bounce off. Statistically, this covers the vast majority of cases. The optimizations do *not* change the worst-case performance of the algorithm, however, although due to the optimizations we have added it is more difficult to construct such a situation than it is with the baseline algorithm. Despite the overloading of arithmetic operators, for example, expressions like " $x := a+b+c+d$ " are not exponential due to the pruning effected by inherited information: our algorithm does not examine all possible "+" operators, only those whose result type matches the inherited type of x which is unique by virtue of being a variable. A worst case *can* be constructed, though. Consider the following:

```
function f(a: t1; b: t3; c: t5) return integer;
```

```
function f(a: t2; b: t4; c: t6) return integer;
function f(a: t1; b: t3; c: t7) return integer;
function f(a: t2; b: t4; c: t8) return integer;
function g return t1;
function g return t2;
function h return t3;
function h return t4;
k: t8;
...
f(g,h,k)...
```

Because the formal parameter types of a and b toggle back and forth between two different types during the processing of successive candidates, their subtrees must be re-analysed each time. This is why Cormack proposed storing the results of previous passes. It is our contention that because the worst case is rare, the additional time savings on worst case time performance will be less than the cost of managing the retained values when the algorithm is applied to real programs. This is especially so because even in the worst case, the algorithm is not exponential in the height of the whole tree, but only in the height of the non-unique node-chain between unique nodes. That is, unique nodes above or below serve to cut off the exponential processing. This is particularly significant in view of the fact that statement operators and declaration operators are all unique in Ada, so that exponential behavior will never extend beyond statement or declaration boundaries:

6 Using the Algorithm in IRIS-Ada

The algorithm discussed above does not address a number of special challenges which Ada presents for overload resolution. In this section we discuss the strategy we have adopted in inserting the algorithm into our compiler.

1. Assignment. Throughout semantic analysis we attempt to exploit the the language's type system to do as much of the work for us as possible. Instead of limiting overload resolution and type matching to user-defined constructs, we treat *all* constructs of the language as subject to the same type-based composition rules. This requires providing a complete language definition expressed in Ada—or to be precise in a somewhat generalized version of the language, since some of the constructs of Ada cannot be expressed in Ada itself³. The advantage is that semantic analysis can be driven by that language definition, so that most constructs which would otherwise require special-case processing can be treated using the general algorithm.

Our treatment of assignment provides an example. One method of treating assignment is to evalu-

² This is quite different from the analysis in [Baker 82]. Baker's algorithm is entirely bottom up with $O(km^2n)$ time and $O(mn)$ space.

³ For example, the abstract syntax for a renaming declaration says that a renaming takes a name, a type t , and an object of type

cursively but it is not known that the configuration is unique, we must be careful to pass null as the required type (ⓐ), and we must split the unique case out during the processing of the actual parameter list (ⓑ).

The final algorithm shown in Figure 4 also adds a new status type to handle ambiguous trees (ⓒ). Whenever more than one legal configuration has been found for a node, it is marked ambiguous; this allows us to minimize reprocessing by returning only a single configuration on subsequent calls (ⓓ).

5 Analysis of the Algorithm.

As mentioned above, the baseline algorithm itself is exponential in the size of the tree. We define m = the average number of candidates for a given node, k = the average number of parameters to a given non-terminal, n = the number of non-terminal nodes in the parse tree, and h = the height of the parse tree. The performance of the baseline algorithm is $O(m^h n)$ for nonterminal nodes, since it visits each node once recursively for every candidate of its parents. In addition, bottom nonterminal nodes must process each of their k terminal descendants for a total time of $O(km^h n)$. In the case of the baseline algorithm, there is no time difference between best case, worse case, and expected case. The space requirements are constant per node, i.e. $O(n)$.² We note that on the average $h = \log_k n$ and that for large Ada programs $k \approx 2.1$.

As mentioned above, the optimizations we have given greatly improve the expected-case performance of the algorithm. It is in fact *linear* whenever nodes can be resolved using only synthetic information or when their parent is unique, since in these cases the node will be marked **unique** and subsequent calls on resolve will simply bounce off. Statistically, this covers the vast majority of cases. The optimizations do *not* change the worst-case performance of the algorithm, however, although due to the optimizations we have added it is more difficult to construct such a situation than it is with the baseline algorithm. Despite the overloading of arithmetic operators, for example, expressions like " $x := a+b+c+d$ " are not exponential due to the pruning effected by inherited information: our algorithm does not examine all possible "+" operators, only those whose result type matches the inherited type of x which is unique by virtue of being a variable. A worst case *can* be constructed, though. Consider the following:

```
function f(a: t1; b: t3; c: t5) return integer;
```

```
function f(a: t2; b: t4; c: t6) return integer;
function f(a: t1; b: t3; c: t7) return integer;
function f(a: t2; b: t4; c: t8) return integer;
function g return t1;
function g return t2;
function h return t3;
function h return t4;
k: t8;
...
f(g,h,k)...
```

Because the formal parameter types of a and b toggle back and forth between two different types during the processing of successive candidates, their subtrees must be re-analysed each time. This is why Cormack proposed storing the results of previous passes. It is our contention that because the worst case is rare, the additional time savings on worst case time performance will be less than the cost of managing the retained values when the algorithm is applied to real programs. This is especially so because even in the worst case, the algorithm is not exponential in the height of the whole tree, but only in the height of the non-unique node-chain between unique nodes. That is, unique nodes above or below serve to cut off the exponential processing. This is particularly significant in view of the fact that statement operators and declaration operators are all unique in Ada, so that exponential behavior will never extend beyond statement or declaration boundaries:

6 Using the Algorithm in IRIS-Ada

The algorithm discussed above does not address a number of special challenges which Ada presents for overload resolution. In this section we discuss the strategy we have adopted in inserting the algorithm into our compiler.

1. Assignment. Throughout semantic analysis we attempt to exploit the the language's type system to do as much of the work for us as possible. Instead of limiting overload resolution and type matching to user-defined constructs, we treat *all* constructs of the language as subject to the same type-based composition rules. This requires providing a complete language definition expressed in Ada—or to be precise in a somewhat generalized version of the language, since some of the constructs of Ada cannot be expressed in Ada itself³. The advantage is that semantic analysis can be driven by that language definition, so that most constructs which would otherwise require special-case processing can be treated using the general algorithm.

Our treatment of assignment provides an example. One method of treating assignment is to evalu-

² This is quite different from the analysis in [Baker 82]. Baker's algorithm is entirely bottom up with $O(km^2n)$ time and $O(mn)$ space.

³ For example, the abstract syntax for a renaming declaration says that a renaming takes a name, a type t , and an object of type

Implementation.

[Pennello 80] T. Pennello, F. DeRemer, and R. Meyers, "A Simplified Operator Identification Scheme for Ada". ACM SIGPLAN Notices, v. 15, n. 11, Nov. 1980, p. 47.

[Stockton 85] R. G. Stockton, "Overload Resolution in Ada+." Carnegie-Mellon University Technical Report CMU-CS-85-186.

[Wallis 80] R. J. Wallis and R. W. Silverman, "Efficient Implementation of the Ada Overloading Rules". Inf. Process. Lett., v. 10, n. 3, Apr. 1980, p. 120.