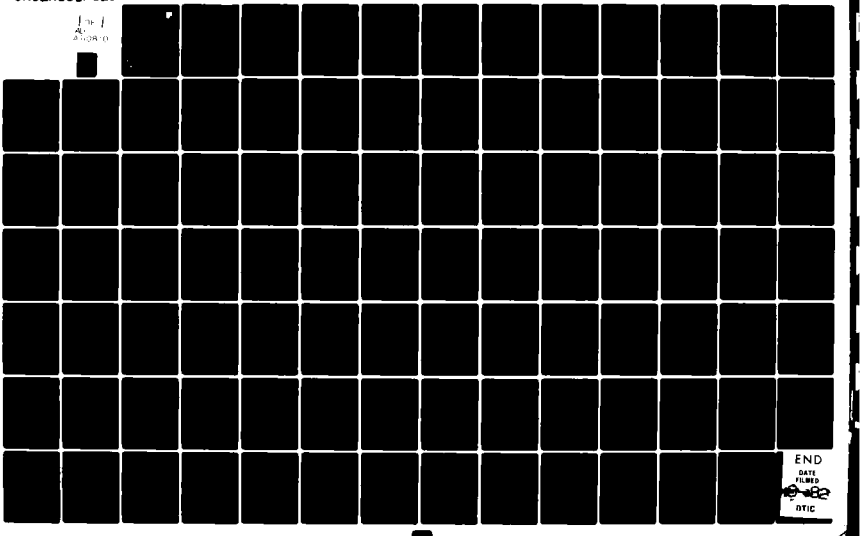
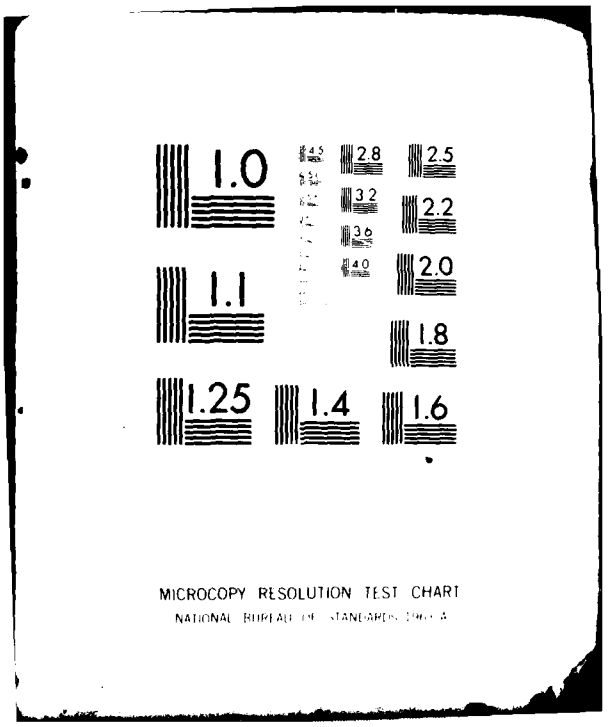
AD-A110 810 SYRACUSE UNIV NY SCHOOL OF COMPUTER AND INFORMATION --ETC F/G 9/2
PROVING PROGRAM CORRECTNESS. VOLUME V.(U)
NOV 81 J C REYNOLDS F30602-77-C-0235
UNCLASSIFIED RADC-TR-80-379-VOL-5 NL



END
DATE FILMED
10-82
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

RADC-TR-80-379, Vol V (of five)
Final Technical Report
November 1981

PROVING PROGRAM CORRECTNESS

Syracuse University

John C. Reynolds

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-80-379, Vol V (of five) has been reviewed and is approved for publication.

APPROVED: Clement D. Falzarano

CLEMENT D. FALZARANO
Project Engineer

APPROVED:

JOHN J. MARCINIAK, Colonel, USAF
Chief, Command and Control Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIS) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-80-379, Vol V (of five)	2. GOVT ACCESSION NO. AD-A110816	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) PROVING PROGRAM CORRECTNESS		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report 1 Oct 77 - 30 Sep 80
		6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s) John C. Reynolds		8. CONTRACT OR GRANT NUMBER(s) F30602-77-C-0235
9. PERFORMING ORGANIZATION NAME AND ADDRESS Syracuse University School of Computer & Information Science Syracuse NY 13210		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62702F 55811903
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISIS) Griffiss AFB NY 13441		12. REPORT DATE November 1981 13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Clement D. Falzarano (CO)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Programming Systems
Programming Languages
Programming Grammars
Proving Programs Correct
Computer Modeling
Simulation
Scheduling Algorithm
Logic Programming

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The "Language Studies" contract is divided into four project areas, all of which are directed to the problems of effectively, reliably and efficiently using modern computers in a wide range of applications.

Three of the projects deal with methods of communicating with computers. — Task 1. Very High Level Programming Systems (P.I.: J.A. Robinson). This group is working towards combining the features developed to support work in the area of artificial intelligence and those used in general program

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

development into a new conceptual framework that can be understood and used by a large community of users. Task 2. Proving Program Correctness (P.I.: J.C. Reynolds). This group is working towards programming language designs which increase the probability that specification errors will be detected by the compiler or interpreter and to provide the language facilities so that users will more nearly be able to prove that programs perform as they are specified than is currently possible. Task 3. Grammars of Programming (P.I.: E.F. Storm). This group is working towards the development of methods which will allow users to communicate with computer programs in terms more normal to their every day communication forms. Task 4. Systems Studies (P.I.: R.G. Sargent). This group is working towards developing more sophisticated and efficient models of computer systems which can predict system performance when given particular parameter values. The current efforts concern models of transaction processing systems (TPS).

Preface

This report describes efforts completed in the Language Studies project at Syracuse University under RADC contract F30602-77-C-0235. The work covers the period October 1, 1977 through September 30, 1980.

The report is produced in five volumes to facilitate single volume distribution.

Volume 1. Report from the Very High Level Programming Systems task. Report title is "Logic Programming in Lisp".

Volume 2. Report from the Systems Studies task. Report title is "Multiple Finite Queueing Model with Fixed Priority Scheduling".

Volume 3. Report from the Systems Studies task. Report title is "An Algorithmic Solution for a Queueing Model of a Computer System with Interactive and Batch Jobs.

Volume 4. Report from the Grammars of Programming task. Report title is "Integrated Parallel Processes: The Elements of Meaning in Language".

Volume 5. Report from the Proving Program Correctness task. Report title is "Proving Program Correctness"

The main goal of our research over the last three years has been the development of a programming language with the basic character of Algol 60, but without the major deficiencies of this language.

Of course, Algol 60 had a pivotal influence on language theory and design when it was first introduced nearly twenty years ago. However, the long-term result of this influence has been languages that are quite different than Algol 60, and which overcome its deficiencies at the expense of introducing new, quite different limitations.

On the one hand, Algol 60 inspired the development of semantic models, particularly by Strachey and Landin, which in turn led to the development of languages such as ISWIM, PAL, GEDANKEN, and, in a somewhat different line of development, Algol 68. All of these languages are "higher level" than Algol 60; in particular, they require a heap (garbage-collectable store) for their implementation, and make it difficult to determine whether a particular data item is stored in a stack or a heap.

On the other hand, the machine implementation of Algol 60 led to the design of languages that avoided various inherently inefficient features of that language. At the same time, Hoare's development of axiomatic language definitions has encouraged the abandonment of certain features, such as procedural parameters and call by name, that are difficult to treat axiomatically. This line of development has lead to languages such as PASCAL, EUCLID, MESA, and ADA, which are all "lower level" than Algol 60.

Our own goal has been to improve and extend Algol 60 without changing its basic character. In particular, we want to retain both the use of stack storage allocation and the power of the Algol procedure mechanism.

A first step in this direction has been the development of an idealization of Algol that is described in the first part of Appendix A. In this language, the type structure has been refined to permit the complete syntactic detection of procedure parameter mismatches, lambda expressions and fixed-point operators of all types have been introduced, and a wide variety of language features have been described as abbreviations for more basic structures. (In Landin's phrase, they have been reduced to "syntactic sugar".)

The main shortcoming of this language, as of Algol 60 itself, is the phenomenon of interference, which includes both variable aliasing and various kinds of procedural side effects. To deal with this phenomenon, we have explored two quite different approaches. The first, called the syntactic control of interference, is to restrict the language so as to make potential interference syntactically detectable. The second, which is embodied in specification logic, is to regard noninterference as a relation between pairs of language phrases that must be proved.

In the syntactic control of interference, described in Appendix A, the language is restricted so that distinct identifiers always denote noninterfering entities, while interfering entities must be named by qualifications of the same identifier. This approach leads to certain syntactic difficulties: the natural abstract syntax is ambiguous, and syntactic correctness is violated by certain beta reductions.

These difficulties were an initial motivation for the development of a generalization of many-sorted algebras, called category-sorted algebras, which is described in Appendix B. In their most obvious application, these algebras are a language design tool for controlling the interaction between type conversions and generic operators. The underlying idea is to permit an abstract syntax to be ambiguous while insuring that this ambiguity does not produce an ambiguity of meaning.

Specification logic is a new approach to proving the correctness of programs written in an Algol-like language. Its central novelty is to regard specifications such as Hoare's {P} S {Q} as predicates about environments (in the sense of Strachey and Landin). By introducing new forms of specifications it is possible to formulate universal specifications that are true in all environments, and to give rules for the inference of such universal specifications. This logical system goes beyond such approaches as Hoare's axiomatic semantics, Dijkstra's weakest preconditions, and Pratt's dynamic logic in its ability to treat interference phenomena, call by name, and statement parameters. Moreover, by introducing lambda expressions and beta reduction, it is possible to use simpler and more abstract inference rules than in other logics that treat procedures.

The semantics of specifications, and rules for their inference are described in Appendix C.

In addition to the above developments, which are related to the design of an Algol-like language, we have also investigated a variety of concepts, laws, and notations for making precise yet intelligible assertions about arrays. This work is based upon Hoare's idea that an array is a variable-like

entity whose value is a function on an interval of integers. Interval and partition diagrams are introduced to make assertions about intervals without recourse to inequalities. A variety of functional concepts, such as restriction, images, pointwise-extended relations, ordering, and rearrangement, are used to minimize quantifiers in assertions about array values.

Our early work in this area is described in Appendix D. More recently, we have made further progress by generalizing the concept of shift equivalence to that of realignment, introducing a kind of abstract concatenation based upon the disjoint union, and using preimages and related concepts. This work is described in Appendix E.

APPENDIX A

SYNTACTIC CONTROL OF INTERFERENCE

John C. Reynolds
School of Computer and Information Science
Syracuse University

ABSTRACT In programming languages which permit both assignment and procedures, distinct identifiers can represent data structures which share storage or procedures with interfering side effects. In addition to being a direct source of programming errors, this phenomenon, which we call interference can impact type structure and parallelism. We show how to eliminate these difficulties by imposing syntactic restrictions, without prohibiting the kind of constructive interference which occurs with higher-order procedures or SIMULA classes. The basic idea is to prohibit interference between identifiers, but to permit interference among components of collections named by single identifiers.

The Problem

It has long been known that a variety of anomalies can arise when a programming language combines assignment with a sufficiently powerful procedure mechanism. The simplest and best-understood case is aliasing or sharing between variables, but there are also subtler phenomena of the kind known vaguely as "interfering side effects".

In this paper we will show that these anomalies are instances of a general phenomenon which we call interference. We will argue that it is vital to constrain a language so that interference is syntactically detectable, and we will suggest principles for this constraint.

Between simple variables, the only form of interference is aliasing or sharing. Consider, for example, the factorial-computing program:

```
procedure fact(n, f); integer n, f;
   begin integer k;
   k := 0; f := 1;
   while k ≠ n do
      begin k := k + 1; f := k × f end
   end .
```

Suppose n and f are called by name as in Algol, or by reference as in FORTRAN, and consider the effect of a call such as fact(z, z), in which both actual parameters are the same. Then the formal parameters n and f will be aliases, i.e., they will interfere in the sense that assigning to either one will affect the value of the other. As a consequence, the assignment f := 1 will obliterate the value of n so that fact(z, z) will not behave correctly.

In this case the problem can be solved by changing n to a local variable which is initialized to the value of the input parameter; this is tantamount to calling n by value. But while this solution is adequate for simple variables, it can become impractical for arrays. For example, the procedure

```
procedure transpose(X, Y); real array X, Y;
   for i := 1 until 50 do
      for j := 1 until 50 do
         Y(i, j) := X(j, i)
```

will malfunction for a call such as transpose(Z, Z) which causes X and Y to be aliases. But changing X to a local variable only solves this problem at the expense of gross inefficiency in both time and space. Certainly, this inefficiency should not be imposed upon calls which do not produce interference. On the other hand, in-place transposition is best done by a completely different algorithm. This suggests that it is reasonable to permit procedures such as transpose, but to prohibit calls of such procedures with interfering parameters.

Although these difficulties date back to Algol and FORTRAN, more recent languages have introduced new features which exacerbate the problem of interference. One such feature is the union of data types. Suppose x is a variable whose value can range over the union of the disjoint data types integer and character. Then the language must provide some construct for branching on whether the current value of x is an integer or a character, and thereafter treating x as one type or the other. For example, one might write

```
unioncase x of (integer S; character: S') ,
```

where x may be used as an identifier of type integer in S and as an identifier of type character in S'. However, consider

```
unioncase x of
   (integer: (y := "A"; n := x + 1);
   character: noaction) .
```

It is evident that aliasing between x and y can cause a type error in the expression x + 1. Thus, in the presence of a union mechanism, interference can destroy type security. This problem occurs with variant records in PASCAL [1], and is only avoided in Algol 68 [2] at the expense of copying union values.

The introduction of parallelism also causes serious difficulties. Hoare [3,4] and Brinch-Hansen [5] have argued convincingly that intelligible programming requires all interactions between parallel processes to be mediated by some mechanism such as a critical region or monitor. As a consequence, in the absence of any critical regions or monitor calls, the parallel execution of two statements, written $S_1 \,||\, S_2$, can only be permitted when S_1 and S_2 do not interfere with one another. For example,

$$x := x + 1 \,||\, y := y * 2$$

would not be permissible when x and y were aliases.

In this paper, we will not consider interacting parallel processes, but we will permit the parallel construct $S_1 \,||\, S_2$ when it is syntactically evident that S_1 and S_2 do not interfere. Although this kind of determinate parallelism is inadequate for practical concurrent programming, it is sufficient to make the consequences of interference especially vivid. For example, when x and y are aliases, the above statement becomes equivalent to

$$z := z + 1 \,||\, z := z * 2$$

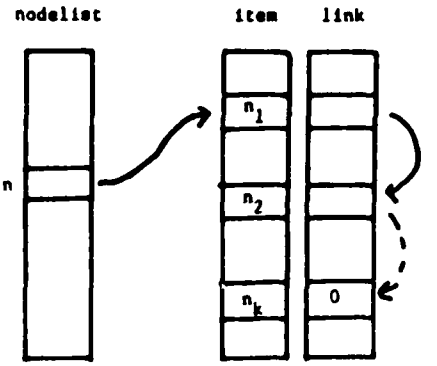
whose meaning, if any, is indeterminate, machine-dependent, and useless.

These examples demonstrate the desirability of constraining a language so that variable aliasing is syntactically detectable. Indeed, several authors have suggested constraints which would eliminate aliasing completely [6,7].

However, aliasing is only the simplest case of the more general phenomenon of interference, which can occur between a variety of program phrases. We have already spoken of two statements interfering when one can perform any action which affects the other. Similarly, two procedures interfere when one can perform a global action which has a global effect upon the other.

Interference raises the same problems as variable aliasing. For example, P(3) || Q(4) is only meaningful if the procedures P and Q do not interfere. Thus the case for syntactic detection extends from aliasing to interference in general. However, the complete prohibition of interference would be untenably restrictive since, unlike variables, interfering expressions, statements, and procedures can have usefully different meanings.

Both the usefulness and the dangers of interference between procedures arise when procedures are used to encapsulate data representations. As an example, consider a finite directed graph whose nodes are labelled by small integers. Such a graph might be represented by giving, for each node n, a linked list of its immediate successors $n_1, \ldots, n_k$:



This representation is used by the procedure

```
procedure itersucc(n,p); integer n; procedure p;
   begin integer k;
   k := nodelist(n);
   while k ≠ 0 do
      begin p(item(k)); k := link(k) end
   end
```

which causes the procedure p to be applied to each immediate successor of the node n.

If the graph is ever to change, then something - probably a procedure such as "addedge" or "deleteedge" - must interfere with itersucc by assigning to the global arrays nodelist, item, and link. On the other hand, the correct operation of itersucc requires that the procedure parameter p must not assign to these arrays, i.e., that p must not interfere with itersucc. Indeed, if itersucc involved parallelism, e.g. if the body of the while statement were

```
      begin integer m;
         m := item(k);
         begin p(m) || k := link(k) end
      end ,
```

then noninterference between p and itersucc would be required for meaningfulness rather than just correctness.

Of course, the need for interfering procedures would vanish if the graph representation were a parameter to the procedures which use it. But this would preclude an important style of programming - epitomized by SIMULA 67 [8] - in which data abstraction is realized by using collections of procedures which interfere via hidden global variables.

In summary, these examples motivate the basic goal of this paper: to design a programming language in which interference is possible yet syntactically detectable. To the author's knowledge, the only current language which tries to meet this goal is Euclid [7]. The approach used in Euclid is quite different than that given here, and apparently precludes procedural parameters and call-by-name.

The Basic Approach

Before proceeding further, we must delineate the idea of interference more precisely. By a phrase we mean a variable, expression, statement, or procedure denotation. In the first three cases, we speak of exercising the phrase P, meaning: either assigning or evaluating P if it is a variable, evaluating P if it is an expression, or executing P if it is a statement.

For phrases P and Q, we write P # Q to indicate that it is syntactically detectable that P and Q do not interfere. More precisely, # is a syntactically decidable symmetric relation between phrases such that:

(1) If neither P nor Q denotes a procedure, then P # Q implies that, for all ways of exercising P and Q, the exercise of P will have no effect on the exercise of Q (and vice-versa). Thus the meaning of exercising P and Q in parallel is well-defined and determinate.

(2) If P denotes a procedure, $A_1, \ldots, A_n$ are syntactically appropriate actual parameters, P # Q, and A_1 # Q, ... , A_n # Q, then $P(A_1, \ldots, A_n)$ # Q. (Thus P # Q captures the idea that P cannot interfere with Q via global variables.)

It should be emphasized that these rules have a fail-safe character: P # Q implies that P and Q cannot interfere, but not the converse. Indeed, the rules are vacuously satisfied by defining # to be universally false, and there is a probably endless sequence of satisfactory definitions which come ever closer to the semantic relation of non-interference at the expense of increasing complexity. Where to stop is ultimately a question of taste: P # Q should mean that P and Q obviously do not interfere.

Our own approach is based upon three principles:

(I) If I # J for all identifiers I occurring free in P and J occurring free in Q, then P # Q.

In effect, all "channels" of interference must be named by identifiers. For the language discussed in this paper, this principle is trivial, since the only such channels are variables. In a richer language, the principle would imply, for example, that all I/O devices must be named by identifiers.

(II) If I and J are distinct identifiers, then I # J.

This is the most controversial of our principles, since it enforces a particular convention for distinguishing between interfering and noninterfering phrases. Interfering procedures (and other entities) are still permissible, but they must occur within a collection which is named by a single identifier. (An example of such a collection is a typical element in a SIMULA [8] class. Indeed, the idea of using such collections was suggested by the SIMULA class mechanism, although we will permit collections which do not belong to any class.)

(III) Certain types of phrases, such as expressions, and procedures which do not assign to global variables, are said to be passive. When P and Q are both passive, P # Q.

Passive phrases perform no assignments or other actions which could cause interference. Thus they cannot interfere with one another or even with themselves, although an active phrase and a passive phrase can interfere.

An Illustrative Language

To illustrate the above principles we will first introduce an Algol-based language which, although it satisfies Principle (I), permits uncontrolled interference. We will then impose Principle (II) to make interference syntactically detectable. Finally, we will explore the consequences of Principle (III).

Unlike Algol, the illustrative language is completely typed, so that reduction (i.e. application of the copy rule) cannot introduce syntax errors. It provides lambda expressions and fixed-point operators for all program types, and a named Cartesian product, which is needed for the collections discussed under Principle II. Procedure declarations, multiple-parameter procedures, and classes are treated as syntactic sugar, i.e., as abbreviations which are defined in terms of more basic linguistic constructs.

Arrays, call-by-value, jumps and labels, unions of types, references, input-output, and critical regions are not considered.

We distinguish between data types, which are the types of values of simple variables, and program types, which are the types which can be declared for identifiers and specified for parameters. The only data types are integer, real, and Boolean, as in Algol, but there are an infinite number of program types. Specifically, the set of program types is the smallest set such that:

(T1) If δ is a data type, then δ var (meaning variable) and δ exp (meaning expression) are program types.

(T2) sta (meaning statement) is a program type.

(T3) If ω and ω' are program types, then $\omega \to \omega'$ is a program type.

(T4) If $\bar{\omega}$ is a function from a finite set of identifiers into program types, then $\Pi(\bar{\omega})$ is a program type.

A formal parameter specified to have type δ var can be used on either side of assignment statements, while a formal parameter specified to have type δ exp can only be used as an expression. The program type $\omega \to \omega'$ describes procedures whose single parameter has type ω and whose call has type ω'. For example, the Algol procedures

```
procedure p1(n); integer n; n := 1;
real procedure p2(x); real x; p2 := x * x;
```

would have types integer var → sta and real exp → real exp respectively.

The program type $\Pi(\bar{\omega})$ is a Cartesian product in which components are indexed by identifiers rather than by consecutive integers. Specifically, $\Pi(\bar{\omega})$ describes collections in which each i in the domain of $\bar{\omega}$ indexes a component of type $\bar{\omega}(i)$. The function $\bar{\omega}$ will always be written as a list of pairs of the form argument:value. Thus, for example, Π(inc: sta, val: integer exp) describes collections in which inc indexes a statement and val indexes an integer expression. A typical phrase of this type might be (inc: n := n + 1; val: n × n).

To simplify the description of syntax we will ignore aspects of concrete representation such as parenthesation, and we will adopt the fiction that each identifier has a fixed program type (except when used as a component index), when in fact the program type of an identifier will be specified in the format I:ω when the identifier is bound.

We write <ω id> and <ω> to denote the sets of identifiers and phrases with program type ω. Then the syntax of the illustrative language is given by the following production schemata, in which δ ranges over all data types, ω, ω', ω_1, ... ω_n range over program types, and i_1, ... , i_n, range over identifiers:

<δ exp> ::= <δ var>

<integer exp> ::= 0

| <integer exp> + <integer exp>

<Boolean exp> ::= true

| <integer exp> = <integer exp>

| <Boolean exp> & <Boolean exp>

(and similarly for other constants and operations on data types)

<sta> ::= <δ var> := <δ exp>

<sta> ::= noaction

| <sta> ; <sta>

| while <Boolean exp> do <sta>

<sta> ::= new <δ var id> in <sta>

<ω> ::= <ω id>

<$\omega \to \omega'$> ::= λ <ω id>. <ω'>

<ω'> ::= <$\omega \to \omega'$> (<ω>)

<$\Pi(i_1{:}\omega_1, \ldots, i_n{:}\omega_n)$> ::= $(i_1{:}\langle\omega_1\rangle, \ldots, i_n{:}\langle\omega_n\rangle)$

<ω_k> ::= <$\Pi(i_1{:}\omega_1, \ldots, i_n{:}\omega_n)$> . i_k

<ω> ::= if <Boolean exp> then <ω> else <ω>

<ω> ::= Y(<$\omega \to \omega$>)

Although a formal semantic specification is beyond the scope of this paper, the meaning of our language can be explicated by various reduction rules. For lambda expressions, we have the usual rule of beta-reduction:

$$(\lambda I.\ P)\ (Q) \ \Rightarrow\ P\big|_{I \to Q}$$

where the right side denotes the result of substituting Q for the free occurrences of I in P, after changing bound identifiers in P to avoid conflicts with free identifiers in Q. Note that this rule implies call by name: If P does not contain a free occurrence of I then (λI. P)(Q) reduces to P even if Q is nonterminating or causes side effects. For collection expressions, we have

$$(i_1{:}\ P_1, \ldots, i_n{:}\ P_n)\,.\,i_k \ \Rightarrow\ P_k\ .$$

For example,

$$(\text{inc: } n := n+1,\ \text{val: } n{\times}n)\,.\,\text{inc} \ \Rightarrow\ n := n+1\ .$$

Again, there is a flavor of call-by-name, since the above reduction would still hold if n×n were replaced by a nonterminating expression. The fixed-point operator Y can also be elucidated by a reduction rule:

$$Y(f) \ \Rightarrow\ f(Y(f))\ .$$

In addition to lambda expressions, the only other binding mechanism in our language is the declaration of new variables. The statement new I: $\left[\begin{smallmatrix}\text{integer}\\ \text{real}\\ \text{Boolean}\end{smallmatrix}\right]$ in S has the same meaning as the Algol statement begin $\left[\begin{smallmatrix}\text{integer}\\ \text{real}\\ \text{Boolean}\end{smallmatrix}\right]$ I; S end.

By themselves, lambda expressions and new variable declarations are an austere vocabulary for variable binding. But they are sufficient to permit other binding mechanisms to be defined as abbreviations. This approach is vital for the language constraints which will be given below, since it insures that all binding mechanisms will be affected uniformly.

Multiple-parameter procedures are treated following Curry [9]:

$$P(A_1, \ldots, A_n) \ \equiv\ P(A_1) \ldots (A_n)$$

$$\lambda(I_1, \ldots, I_n).\ B \ \equiv\ \lambda I_1. \ldots \lambda I_n.\ B$$

and definitional forms, including procedure declarations are treated following Landin [10]:

$$\text{let } I = Q \text{ in } P \ \equiv\ (\lambda I.\ P)(Q)$$

$$\text{let rec } I = Q \text{ in } P \ \equiv\ (\lambda I.\ P)(Y(\lambda I.\ Q))\ .$$

(However, unlike Landin, we are using call-by-name.) We will omit type specifications from let and let rec expressions when the type of I is apparent from Q.

As shown in the Appendix, classes (in a slightly more limited sense than in SIMULA) can also be defined as abbreviations.

As an example, the declaration of the procedure fact shown at the beginning of this paper, along with a statement S in the scope of this declaration, would be written as:

```
let fact = λ(n :integer exp, f: integer var).
    new k: integer in
        (k := 0; f := 1;
        while k ≠ n do (k := k+1; f := k×f))
in S .
```

After eliminating abbreviations, this becomes

```
(λfact: integer exp → (integer var → sta). S)
    (λn: integer exp. λf: integer var.
        new k: integer in
            (k := 0; f := 1;
            while k ≠ n do (k := k+1; f := k×f))) .
```

Controlling Interference

The illustrative language already satisfies Principle I. If we can constrain it to satisfy Principle II as well, then P # Q will hold when P and Q have no free identifiers in common. By assuming the most pessimistic definition of # compatible with this result (and postponing the consequences of Principle III until the next section), we get

$$P \# Q \text{ iff } F(P) \cap F(Q) = \{\},$$

where $F(P)$ denotes the set of identifiers which occur free in P.

To establish Principle II, we must consider each way of binding an identifier. A new variable declaration causes no problems, since new variables are guaranteed to be independent of all previously declared entities. But a lambda expression can cause trouble, since its formal parameter will interfere with its global identifiers if it is ever applied to an actual parameter which interferes with the global identifiers, or equivalently, with the procedure itself. To avoid this interference, we will restrict the call P(A) of a procedure by imposing the requirement P # A.

The following informal argument shows why this restriction works. Consider a beta-reduction $(\lambda I.\ P)(Q) \Rightarrow P|_{I \to Q}$. Within P there may be a pair of identifiers which are syntactically required to satisfy the #-relationship, and therefore must be distinct. If so, it is essential that the substitution I → Q preserve the #-relationship. No problem occurs if neither identifier is the formal parameter I. On the other hand, if one identifier is I, then the other distinct identifier must be global. Thus the #-relation will be preserved if K # Q holds for all global identifiers K, i.e., for all identifiers occurring free in $\lambda I.\ P$. This is equivalent to $(\lambda I.\ P) \# Q$.

More formally, one can show that, with the restriction on procedure calls:

$$\langle\omega'\rangle ::= \langle\omega \to \omega'\rangle(\langle\omega\rangle) \quad \text{when} \quad \langle\omega \to \omega'\rangle \# \langle\omega\rangle ,$$

syntactic correctness is preserved by beta reduction (and also by reduction of collection expressions), and continues to be preserved when other productions restricted by # are added, e.g.,

$$\langle sta\rangle ::= \langle sta_1\rangle \ ||\ \langle sta_2\rangle \text{ when } \langle sta_1\rangle \# \langle sta_2\rangle .$$

The restriction P # A on P(A) also affects the language constructs which are defined as abbreviations. For let I = Q in P ≡ $(\lambda I.\ P)(Q)$, and for let rec I = Q in P ≡ $(\lambda I.\ P)(Y(\lambda I.\ Q))$, we see that, except for I, no free identifier of Q can occur free in P. Thus, although one can declare a procedure or a collection of procedures which use global identifiers (the free identifiers of Q), these globals are masked from occurring in the scope P of the declaration, where they would interfere with the identifier I.

For multi-parameter procedures, $P(A_1, \ldots, A_n) \equiv P(A_1) \ldots (A_n)$ implies the restrictions $P \# A_1$, $P(A_1) \# A_2$, ..., $P(A_1) \ldots (A_{n-1}) \# A_n$, which are equivalent to requiring $P \# A_i$ for each parameter and $A_i \# A_j$ for each pair of distinct parameters.

For example, consider the following procedure for a "repeat" statement:

let repeat = λ(s: sta, b: Boolean exp).
(s; while ¬ b do s) .

In any useful call repeat(A_1, A_2), the statement A_1 will interfere with the Boolean expression A_2. Although this is permitted in the unconstrained illustrative language, as in Algol, it is prohibited by the restriction $A_1 \# A_2$. Instead, one must group the interfering parameters into a collection:

let repeat = λx: Π(s: sta, b: Boolean exp).
(x.s; while ¬ x.b do x.s) ,

and use calls of the form repeat((s:A_1, b:A_2)).

This example is characteristic of Principle II. Although interfering parameters are permitted, they require a somewhat cumbersome notation. In compensation, it is immediately clear to the reader of a procedure body when interference between parameters is possible.

Passive Phrases

In making interference syntactically detectable, we have been unnecessarily restrictive. For example, we have forbidden parallel constructs such as

x := n || y := n

or

let twice = λs: sta. (s; s) in
(twice (x := x+1) || twice(y := y*2)) .

Moreover, the right side of the reduction rule Y(f) ⇒ f(Y(f)) violates the requirement f # Y(f), giving a clear sign that there is a problem with recursion.

In the first two cases, we have failed to take into account that the expression n and the procedure twice are passive: They do no assignment (to global variables in the case of procedures), and therefore do not interfere with themselves. Similarly, when f is passive, f # Y(f) holds, and the reduction rule for Y(f) becomes valid. This legitimizes the recursive definition of procedures which do not assign to global variables.

(Recursive procedures which assign to global variables are a more difficult problem. Within the body of such a procedure, the global variables and the procedure itself are interfering entities, and must therefore be represented by components of a collection named by a single identifier. This situation probably doesn't pose any fundamental difficulties, but we have not pursued it.)

The following treatment of passivity is more tentative than the previous development. Expressions in our language are always passive, since they never cause assignment to free variables. Procedures may be active or passive, independently of their argument and result types. Thus we must distinguish the program type $\omega \to_p \omega'$ describing passive procedures from the program type $\omega \to \omega'$ describing (possibly) active procedures.

More formally, we augment the definition of program types with

(T5) If ω and ω' are program types, then $\omega \to_p \omega'$ is a program type.

and we define passive program types to be the smallest set of program types such that

(P1) ϕ exp is passive.

(P2) $\omega \to_p \omega'$ is passive.

(P3) If $\bar{\omega}(i)$ is passive for all i in the domain of $\bar{\omega}$, then $\Pi(\bar{\omega})$ is passive.

Next, for any phrase r, we define $A(r)$ to be the set of identifiers which have at least one free occurrence in r which is outside of any subphrase of passive type. Note that, since identifier occurrences are themselves subphrases, $A(r)$ never contains identifiers of passive type, and since r is a subphrase of itself, $A(r)$ is empty when r has passive type.

Then we relax the definition of P # Q to permit P and Q to contain free occurrences of the same identifier, providing every such occurrence is within a passive subphrase. We define:

$$P \# Q \equiv A(P) \cap F(Q) = \{\} \,\&\, F(P) \cap A(Q) = \{\} .$$

Finally, we modify the abstract syntax. We define a passive procedure to be one in which no global identifier has an active occurrence:

$$\langle \omega \to_p \omega' \rangle ::= \lambda \langle \omega \text{ id} \rangle . \langle \omega' \rangle \quad \text{when } A(\langle \omega' \rangle) - \{\langle \omega \text{ id} \rangle\} = \{\} .$$

Passive procedures can occur in any context which permits active procedures:

$$\langle \omega \to \omega' \rangle ::= \langle \omega \to_p \omega' \rangle ,$$

but only passive procedures can be operands of the fixed-point operator:

$$\langle \omega \rangle ::= \underline{Y}(\langle \omega \to_p \omega \rangle) .$$

Some Unresolved Questions

Our abstract syntax is ambiguous, in the sense that specifying the type of a phrase does not always specify a unique type for each subphrase. For example, in the original illustrative language, the subphrase *if* p *then* x *else* y might be either a variable or an expression in contexts such as

```
z := if p then x else y
(a: if p then x else y, b: 3).b
```

Similarly, the introduction of passive procedures permits the subphrase λs: sta. (s; s) to have either type sta → sta or sta →p sta in the context

```
(λs: sta. (s; s))(x := x+1)
```

Although these ambiguities could probably be eliminated, our intuition is to retain them, while insisting that they must not lead to ambiguous meanings. Indeed, it may be fruitful to extend this attitude to a wider variety of implicit conversions.

In normal usage, a procedure call will be active if and only if either the procedure itself or its parameter are active. Although other cases are syntactically permissible they seem to have only trivial instances. Thus it might be desirable to limit the program types of procedures to the cases:

$$\theta \to_p \theta' \quad \alpha \to_p \alpha' \quad \theta \to \alpha \quad \alpha \to \alpha'$$

where θ and θ' are passive types and α and α' are nonpassive types.

The most serious problem with our treatment of passivity is our inability to retain the basic property that beta-reduction preserves syntactic correctness. Consider, for example, the reduction

```
(λp: mixed. (x := p.a || y := p.a))
    ((a: n+1, b: n := 0))
→ x := (a: n+1, b: n := 0).a
    || y := (a: n+1, b: n := 0).a
→ x := n+1 || y := n+1
```

where "mixed" stands for the program type Π(a: integer exp, b: sta). Although the first and last lines are perfectly reasonable, the intermediate line is rather dubious, since it contains assignments to the same variable n within two statements to be executed in parallel. Nevertheless, our definition of # still permits the intermediate line, on the grounds that assignments within passive phrases cannot be executed.

However, if we accept

```
x := (a: n+1, b: n := 0).a
  # y := (a: n+1, b: n := 0).a ,
```

then it is hard to deny

```
λs: sta. x := (a: n+1, b: (n := 0 || s)).a
  # y := (a: n+1, b: n := 0).a .
```

But this permits the reduction

```
(λs: sta. x := (a: n+1, b: (n := 0 || s)).a)
    (y := (a: n+1, b: n := 0). a)
→ x := (a: n+1, b:
    (n := 0 || y := (a: n+1, b: n := 0).a)
    ).a)
→ x := n+1
```

Here the intermediate step, in which the underlined statement is clearly illegal, is prohibited by our syntax.

This kind of problem is compounded by the possibility of collection-returning procedures. For instance, in the above examples, one might have silly(n+1, n := 0), where silly has type integer exp → (sta → mixed), in place of the collection (a: n+1, b: n := 0).

A possible though unesthetic solution to these problems might be to permit illegal phrases in contexts where passivity guarantees nonexecution. A more hopeful possibility would be to alter the definition of substitution to avoid the creation of illegal phrases in such contexts.

Directions for Further Work

Beyond dealing with the above questions, it is obviously essential to extend these ideas to other language mechanisms, particularly arrays.

In addition, the interaction between these ideas and the axiomatization of program correctness needs to be explored. We suspect that many rules of inference might be simplified by using a logic which imposes ϕ-preservation upon substitutions.

A somewhat tangential aspect of this work is the distinction between data and program types, which obviously has implications for user-defined types. (Note the absence of this distinction in Algol 68 [2].) In less Algol-like languages, data types might have as much structure as program types, and user definitions might be needed for both "types" of type. Indeed, there may be grounds for introducing more than two "types" of type.

Finally, these ideas may have implications for the optimization of call-by-name, perhaps to an extent which will overcome the aura of hopeless inefficiency which surrounds this concept. For example, when an expression is a single parameter to a procedure, as opposed to a component of a collection which is a parameter, then its repeated evaluation within the procedure must yield the same value (although nontermination is still possible). This suggests a possible application of the idea of "lazy evaluation" [11, 12].

APPENDIX

Classes as Syntactic Sugar

In a previous paper, we have argued that classes are a less powerful data abstraction mechanism than either higher-order procedures or user-defined types [14]. The greater generality of higher-order procedures permits the definition of classes (in the reference-free sense of Hoare [13] rather than SIMULA itself) as abbreviations in our illustrative language. In fact, the basic idea works in Algol 60, although the absence there of lambda expressions and named collections of procedures makes its application cumbersome.

We consider a class declaration with scope S of the form:

$$\underline{\text{class}}\ C(\text{DECL};\ \text{INIT};\ I_1{:}P_1,\ \ldots,\ I_n{:}P_n)\ \underline{\text{in}}\ S \qquad (1)$$

which defines C to be a class with component names $I_1, \ldots, I_n$. Here DECL is a list of declarations of variables and procedures which will be private to a class element, INIT is an initialization statement to be executed when each class element is created, and each P_k is the procedure named by I_k, in which the private variables may occur as globals.

Within the scope S, one may declare X to be a new element of class C by writing the statement

$$\underline{\text{newelement}}\ X{:}\ C\ \underline{\text{in}}\ S' \quad . \qquad (2)$$

Then within the statement S' one may write $X.I_k$ to denote the component P_k of the class element X.

To express these notations in terms of procedures, suppose $P_1, \ldots, P_n$ have types $\omega_1, \ldots, \omega_n$ respectively. Then we define (1) to be an abbreviation for:

$$\underline{\text{let}}\ C = \lambda b{:}\ \Pi(I_1{:}\omega_1,\ \ldots,\ I_n{:}\omega_n) \to \text{sta}.\ (\text{DECL};\ \text{INIT};\ b(\langle I_1{:}P_1,\ \ldots,\ I_n{:}P_n\rangle))\ \underline{\text{in}}\ S\ ,$$

where b is an identifier not occurring in the original class declaration, and where DECL must be expressed in terms of new and let declarations. Then we define (2) to be an abbreviation for:

$$C(\lambda X{:}\ \Pi(I_1{:}\omega_1,\ \ldots,\ I_n{:}\omega_n).\ S')\ .$$

As an example, where for simplicity P_1 and P_2 are parameterless procedures:

```
class counter(integer n; n := 0;
                  inc: n := n+1, val: n) in
... newelement k: counter in
... (k.inc; x := k.val)
```

is an abbreviation for

```
let counter =
   λb: Π(inc: sta, val: integer exp) → sta.
      new n: integer in
         (n := 0; b(⟨inc: n := n+1, val: n⟩))
in
... counter(λk: Π(inc: sta, val: integer exp).
   ... (k.inc; x := k.val)) ,
```

which eventually reduces to

```
new n: integer in (n := 0;
     ... (n := n+1; x := n)) .
```

In the process of reduction, identifiers will be renamed to protect the privacy of n.

The only effect of our interference-controlling constraints is that C must be a passive procedure, i.e., INIT and $P_1, \ldots, P_n$ cannot assign to any variables which are more global than those declared by DECL. This insures that distinct class elements will not interfere with one another. Otherwise, if C is not passive, then S' in the definition of (2) cannot contain calls of C, so that multiple class elements cannot coexist.

ACKNOWLEDGEMENTS

Most of this research was done during a delightful and stimulating sabbatical at the University of Edinburgh. Special thanks are due to Rod Burstall and Robin Milner for their encouragement and helpful suggestions, and to the members of IFIP working group 2.3, especially Tony Hoare, for establishing the viewpoint about programming which underlies this work.

REFERENCES

[1] Wirth, N. The Programming Language PASCAL. Acta Informatica 1, (1971), pp. 35-63.

[2] van Wijngaarden, A. (ed.), Mailloux, B. J., Peck, J. E. L., and Koster, C. H. A. Report on the Algorithmic Language ALGOL 68. MR 101, Mathematisch Centrum, Amsterdam, February 1969.

[3] Hoare, C. A. R. Towards a Theory of Parallel Programming. In Operating Systems Techniques, Academic Press, New York, 1972.

[4] Hoare, C. A. R. Monitors: An Operating System Structuring Concept. Comm. ACM 17 (October 1974), pp. 549-557.

[5] Brinch-Hansen, P. Structured Multiprogramming. Comm. ACM 15 (July 1972), pp. 574-577.

[6] Hoare, C. A. R. Procedures and Parameters: An Axiomatic Approach. In Symposium on the Semantics of Algorithmic Languages (ed. E. Engeler). Springer, Berlin-Heidelberg-New York, 1971.

[7] Popek, G. J., Horning, J. J., Lampson, B. W., Mitchell, J. G., and London, R. L. Notes on the Design of Euclid. In Proceedings of an ACM Conference on Language Design for Reliable Software, SIGPLAN Notices 12, no. 3 (March 1977), pp. 11-18.

[8] Dahl, O. -J. Hierarchical Program Structures. In Structured Programming, Academic Press, New York 1972.

[9] Curry, H. B., and Feys, R. Combinatory Logic, Volume I. North-Holland, Amsterdam 1958.

[10] Landin, P. J. A Correspondence Between ALGOL 60 and Church's Lambda Notation. Comm ACM 8 (February and March 1965), pp. 89-101 and 158-165.

[11] Henderson, P., and Morris, J. H., Jr. A Lazy Evaluator. Third ACM Symposium on Principles of Programming Languages (1976), pp. 95-103.

[12] Friedman, D. P., and Wise, D. S. CONS Should Not Evaluate its Arguments. Third Int'l Colloquium on Automata, Languages, and Programming, Edinburgh University Press 1976, pp. 257-284.

[13] Hoare, C. A. R. Proof of Correctness of Data Representations. Acta Informatica 1, pp. 271-281 (1972).

[14] Reynolds, J. C. User-Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction. In New Directions in Algorithmic Languages 1975, ed. S. A. Schuman, I.R.I.A. 1975, pp. 157-168.

APPENDIX B

USING CATEGORY THEORY TO DESIGN IMPLICIT CONVERSIONS AND GENERIC OPERATORS

John C. Reynolds
Syracuse University
Syracuse, New York

ABSTRACT A generalization of many-sorted algebras, called category-sorted algebras, is defined and applied to the language-design problem of avoiding anomalies in the interaction of implicit conversions and generic operators. The definition of a simple imperative language (without any binding mechanisms) is used as an example.

Introduction

A significant problem in the design of programming languages is the treatment of implicit conversions, sometimes called coercions, between types. A failure to provide implicit conversions can degrade the conciseness and readability of a language. On the other hand, unless great care is taken in the design of such conversions, and their interaction with operators which can be applied to operands of several types, the resulting language will exhibit anomalies that will be a rich source of programming errors. (In the author's opinion, PL/I and Algol 68 exemplify this danger.)

As a simple illustration, consider assigning the sum of two integer variables to a real variable. In the absence of an implicit conversion from integer to real, one would have to write either

```
x := integer-to-real(m) + integer-to-real(n)
```

or

```
x := integer-to-real(m + n) .
```

Clearly, one would prefer to write x := m + n. If the language permits this, however, one can ask whether the implicit conversion precedes or follows the addition, i.e., which of the above statements is equivalent to x := m + n.

It is generally believed that a precise language definition must answer this question unambiguously. However, if one were to ask the question of a mathematician (at least one who didn't know too much about programming), he would probably reply that it doesn't matter, since both of the above statements have the same meaning, and that indeed the whole point of permitting the same operator + to be applied to arguments of different type which are connected by an implicit conversion is that the resulting ambiguity should not affect the meaning.

In a sense, of course, the mathematician is wrong: some computers provide a floating-point representation with such limited precision that the ambiguity in question does affect meaning. But in a deeper sense the mathematician is right. One intuitively expects that the above statements should have nearly the same meaning, and in analogous cases where numerical approximation or overflow is not involved, one expects exactly the same meaning.

To see this, replace real by character string in the above example, and suppose that integers are implicitly converted into character strings giving their decimal representation, and that + denotes both addition of integers and concatenation of strings. Then the two possible meanings of x := m + n are radically different. This case is clearly a mistake in language design which would be likely to cause programming errors.

In this paper we will describe a method for avoiding such errors. The underlying mathematical tool will be a generalization of many-sorted algebras called category-sorted algebras, which are closely related to the order-sorted algebras invented by Goguen.[1]

Beyond the specific goal of treating implicit conversions, our presentation is intended to illustrate the potential of category theory in the area of language definition and to suggest that the "standard" denotational semantics developed by Scott and Strachey may not be the final solution to the language-definition problem. There is nothing incorrect about the Scott-Strachey methodology, and it has provided fundamental insights into many aspects of programming languages such as recursion. But it has not been so helpful in other areas of language design such as type structure. We suspect that clearer insights into these areas will require quite different applications of mathematics.

Conventional Many-Sorted Algebras

Our use of algebras is based on the ideas of Goguen, Thatcher, Wagner, and Wright,[2] which have roots as far back as Burstall and Landin.[3] In (2) a language is viewed as an initial algebra and its semantic function as the unique homomorphism from this initial algebra into some target algebra, so that defining the target algebra is tantamount to defining semantics. Here we will adopt the slightly more elaborate view that (roughly speaking) a language is the free algebra generated by some set of identifiers, that an environment is a mapping of these identifiers into the carrier of the target algebra, and that the semantic function is the function which maps each environment into its unique extension as a homomorphism from the free algebra to the target algebra.

We propose to treat implicit conversions in this framework by generalizing the concept of an algebra appropriately. To motivate this proposal we will proceed through a sequence of increasingly general definitions of "algebra".

The standard concept of a many-sorted algebra used in algebraic semantics is due to Birkhoff and Lipson,[4] who called it an "heterogeneous" algebra. According to Birkhoff and Lipson, but with changes of notation and terminology to reveal the similarity to later definitions:

(1) A signature consists of:

(1a) A set Ω of sorts. (Informally, the sorts correspond to types in a programming language.)

(1b) A family, indexed by nonnegative integers, of disjoint sets Δ_n of operators of rank n.

(1c) For each $n \geq 0$ and $\delta \in \Delta_n$, a specification $\Gamma_\delta \in \Omega^n \times \Omega$. (Informally, if $\Gamma_\delta = \langle\langle \omega_1, \ldots, \omega_n \rangle, \omega \rangle$ then the operator δ accepts operands of sorts $\omega_1, \ldots, \omega_n$ and yields a result of sort ω.)

(2) An ΩΔΓ-algebra consists of:

(2a) A carrier B, which is an Ω-indexed family of sets. (Informally $B(\omega)$ is the set of meanings appropriate for phrases of type ω.)

(2b) For each $n \geq 0$ and $\delta \in \Delta_n$, an interpretation $\gamma_\delta \in B(\omega_1) \times \ldots \times B(\omega_n) \rightarrow B(\omega)$, where $\langle\langle\omega_1, \ldots, \omega_n\rangle, \omega\rangle = \Gamma_\delta$.

(3) If B, γ and B', γ' are ΩΔΓ-algebras, then a homomorphism from B, γ to B', γ' is an Ω-indexed family of functions $\theta(\omega) \in B(\omega) \rightarrow B'(\omega)$ such that, for all $n \geq 0$ and $\delta \in \Delta_n$, the diagram

$$\begin{array}{ccc} B(\omega_1) \times \ldots \times B(\omega_n) & \xrightarrow{\gamma_\delta} & B(\omega) \\ \downarrow \theta(\omega_1) \times \ldots \times \theta(\omega_n) & & \downarrow \theta(\omega) \\ B'(\omega_1) \times \ldots \times B'(\omega_n) & \xrightarrow{\gamma'_\delta} & B'(\omega) \end{array}$$

commutes. Here $\langle\langle\omega_1, \ldots, \omega_n\rangle, \omega\rangle = \Gamma_\delta$ and $f_1 \times \ldots \times f_n$ denotes the function such that $(f_1 \times \ldots \times f_n)(x_1, \ldots, x_n) = \langle f_1(x_1), \ldots f_n(x_n)\rangle$.

Unfortunately, it is difficult to pose the implicit-conversion problem within this concept of algebra since there is no mechanism for grouping operators which are represented by the same symbol. For example, integer addition and real addition would be distinct members of Δ_2 (with specifications $\langle\langle$integer, integer$\rangle$, integer$\rangle$ and $\langle\langle$real, real$\rangle$, real$\rangle$), and there is no mechanism for relating their interpretations more closely than, say, integer addition and multiplication.

Many-Sorted Algebras with Generic Operators

To solve this problem, we will employ an alternative concept of many-sorted algebras due to Higgins.[5] In this approach, the operators are (in programming jargon) generic. The specification of an operator of rank n is a partial function from Ω^n to Ω, which is defined for the combinations of sorts of operands to which the operator is applicable, and which maps each such combination into the sort of the result yielded by the operand. (Notice that this captures the idea of bottom-up type determination.) Then the interpretation of the operator is a family of n-ary functions indexed by the domain of its specification.

In our own development we will insist that the specification be a total function from Ω^n to Ω. At first sight, this simplification might appear to be untenable since it implies that every operator can be applied to operands of arbitrary sorts. Formally, however, the situation can be saved by introducing a "nonsense" sort ns, which is the sort of "type-incorrect" phrases. (If a phrase is type-incorrect whenever any of its subphrases are type-incorrect, then every specification will yield ns whenever any of the sorts to which it is applied is ns. However, one can conceive of contexts, such as the application of a constant function, where this assumption might be relaxed.)

With this simplification, and a few changes of notation and terminology, Higgins' concept of a many-sorted algebra is:

(1) A signature consists of:

(1a) (as before) A set Ω of sorts.

(1b) (as before) A family, indexed by nonnegative integers, of disjoint sets Δ_n of operators of rank n.

(1c) For each $n \geq 0$ and $\delta \in \Delta_n$, a specification $\Gamma_\delta \in \Omega^n \rightarrow \Omega$. (Informally, $\Gamma_\delta(\omega_1, \ldots, \omega_n)$ is the sort of result yielded by the generic operator δ when applied to operands of sorts $\omega_1, \ldots, \omega_n$.)

(2) An $\Omega\Delta\Gamma$-algebra consists of:

(2a) (as before) A carrier B, which is an Ω-indexed family of sets.

(2b) For each $n \geq 0$ and $\delta \in \Delta_n$, an interpretation γ_δ, which is an Ω^n-indexed family of functions $\gamma_\delta(\omega_1, \ldots, \omega_n) \in B(\omega_1) \times \ldots \times B(\omega_n) \to B(\Gamma_\delta(\omega_1, \ldots, \omega_n))$. (Informally, $\gamma_\delta(\omega_1, \ldots, \omega_n)$ is the interpretation of the version of the generic operator δ which is applicable to sorts $\omega_1, \ldots, \omega_n$.)

(3) If B,γ and B',γ' are $\Omega\Delta\Gamma$-algebras, then an homomorphism from B,γ to B',γ' is an Ω-indexed family of functions $\theta(\omega) \in B(\omega) \to B'(\omega)$ such that, for all $n \geq 0$, $\delta \in \Delta_n$, and $\omega_1, \ldots, \omega_n \in \Omega$, the diagram

$$\begin{array}{ccc} B(\omega_1) \times \ldots \times B(\omega_n) & \xrightarrow{\gamma_\delta(\omega_1, \ldots, \omega_n)} & B(\Gamma_\delta(\omega_1, \ldots, \omega_n)) \\ \downarrow \theta(\omega_1) \times \ldots \times \theta(\omega_n) & & \downarrow \theta(\Gamma_\delta(\omega_1, \ldots, \omega_n)) \\ B'(\omega_1) \times \ldots \times B'(\omega_n) & \xrightarrow{\gamma'_\delta(\omega_1, \ldots, \omega_n)} & B'(\Gamma_\delta(\omega_1, \ldots, \omega_n)) \end{array} \quad (I)$$

commutes.

Algebras with Ordered Sorts

We can now introduce the notion of implicit conversion. When there is an implicit conversion from sort ω to sort ω', we write $\omega \leq \omega'$ and say that ω is a subsort (or subtype) of ω'. Syntactically, this means that a phrase of sort ω can occur in any context which permits a phrase of sort ω'.

It is reasonable to expect that $\omega \leq \omega$ and that $\omega \leq \omega'$ and $\omega' \leq \omega''$ implies $\omega \leq \omega''$. Thus the relation $\leq$ is a preordering (sometimes called a quasiordering) of the set Ω. Actually, in all of the examples in this paper $\leq$ will be a partial ordering, i.e., $\omega \leq \omega'$ and $\omega' \leq \omega$ will only hold when $\omega = \omega'$. However, our general theory will not impose this additional requirement upon $\leq$.

Now suppose δ is an operator of rank n, and $\omega_1, \ldots, \omega_n$ and $\omega_1', \ldots, \omega_n'$ are sorts such that $\omega_i \leq \omega_i'$ for each i from one to n. Then a context which permits a phrase of sort $\Gamma_\delta(\omega_1', \ldots \omega_n')$ will permit an application of δ to operands of sorts $\omega_1', \ldots, \omega_n'$. But the context of the ith operand will also permit an operand of sort ω_i, so that the overall context must also permit an application of δ to operands of sort $\omega_1, \ldots, \omega_n$, which has sort $\Gamma_\delta(\omega_1, \ldots, \omega_n)$. Thus we expect that $\Gamma_\delta(\omega_1, \ldots, \omega_n) \leq \Gamma_\delta(\omega_1', \ldots, \omega_n')$ or, more abstractly, that the specification Γ_δ will be a monotone function.

If $\omega \leq \omega'$ then an algebra must specify a conversion function from the set $B(\omega)$ of meanings appropriate to ω to the set $B(\omega')$of meanings appropriate to ω'. At first sight, one might expect that this can only occur when $B(\omega)$ is a subset of $B(\omega')$, and that the conversion function must be the corresponding identity injection. For example, integer can be taken as a subsort of real because the integers are a subset of the reals.

However there are other situations in which this is too limited a view of implicit conversion. For example, we would like to say that integer variable is a subsort of integer expression, so that integer variables can occur in any context which permits an integer expression. But it is difficult to regard the meanings of integer variables as a subset of the meanings of integer expressions. In fact, we will regard the meaning of an integer variable as a pair of functions: an acceptor function, which maps integers into state transformations, and an evaluator function, which maps states into integers. Then the meaning of an expression will just be an evaluator function, and the implicit conversion function from variables to expressions will be a function on pairs which forgets their first components.

In general, we will permit implicit conversion functions which forget information and are therefore not injective. To paraphrase Jim Morris,[6] subtypes are not subsets. This is the main difference between our approach and that of Goguen.[1] (There are some more technical differences, particularly in the definition of signatures, whose implications are not completely clear to this author.)

However, there are still some restrictions that should be imposed upon implicit conversion functions. The conversion function from any type to itself should be an identity function. Moreover, if $\omega \leq \omega'$ and $\omega' \leq \omega''$ then the conversion function from $B(\omega)$ to $B(\omega'')$ should be the composition of the functions from $B(\omega)$ to $B(\omega')$ and from $B(\omega')$ to $B(\omega'')$. This will insure that a conversion from one sort to another will not depend upon the choice of a particular path in the preordering of sorts.

These restrictions can be stated more succinctly by invoking category theory. A preordered set such as Ω can be viewed as a category with the members of Ω as objects, in which there is a single morphism from ω to ω' if $\omega \leq \omega'$ and no such morphism otherwise. Suppose we write $\omega \leq \omega'$ to stand for the unique morphism from ω to ω' (as well as for the condition that this morphism exists), and require the carrier B to map each $\omega \leq \omega'$ into the conversion function from $B(\omega)$ to $B(\omega')$. Then we have

(i) $B(\omega \leq \omega') \in B(\omega) \rightarrow B(\omega')$.

(ii) $B(\omega \leq \omega) = I_{B(\omega)}$.

(iii) If $\omega \leq \omega'$ and $\omega' \leq \omega''$ then
$B(\omega \leq \omega'') = B(\omega \leq \omega');B(\omega' \leq \omega'')$.

(Throughout this paper we will use semicolons to indicate composition in diagrammatic order, i.e., $(f;g)(x) = g(f(x))$.) These requirements are equivalent to saying that B must be a functor from Ω to the category SET, in which the objects are sets and the morphisms from S to S' are the functions from S to S'.

This leads to the following definition:

(1) A signature consists of:

(1a) A preordered set Ω of sorts.

(1b) (as before) A family, indexed by nonnegative integers, of disjoint sets Δ_n of operators of rank n.

(1c) For each $n \geq 0$ and $\delta \in \Delta_n$, a specification Γ_δ, which is a monotone function from Ω^n to Ω.

(2) An $\Omega\Delta\Gamma$-algebra consists of:

(2a) A carrier B, which is a functor from Ω to SET.

(2b) For each $n \geq 0$ and $\delta \in \Delta_n$, an interpretation γ_δ, which is an Ω^n-indexed family of functions $\gamma_\delta(\omega_1, \ldots, \omega_n) \in B(\omega_1) \times \ldots \times B(\omega_n) \rightarrow B(\Gamma_\delta(\omega_1, \ldots, \omega_n))$ such that, whenever $\omega_1 \leq \omega_1', \ldots, \omega_n \leq \omega_n'$, the diagram

$$\begin{array}{ccc} B(\omega_1) \times \ldots \times B(\omega_n) & \xrightarrow{\gamma_\delta(\omega_1, \ldots, \omega_n)} & B(\Gamma_\delta(\omega_1, \ldots, \omega_n)) \\ \downarrow B(\omega_1 \leq \omega_1') \times \ldots \times B(\omega_n \leq \omega_n') & & \downarrow B(\Gamma_\delta(\omega_1, \ldots, \omega_n) \leq \Gamma_\delta(\omega_1', \ldots \omega_n')) \\ B(\omega_1') \times \ldots \times B(\omega_n') & \xrightarrow{\gamma_\delta(\omega_1', \ldots, \omega_n')} & B(\Gamma_\delta(\omega_1', \ldots, \omega_n')) \end{array} \qquad \text{(II)}$$

commutes.

The above diagram asserts the relationship between generic operators and implicit conversions which originally motivated our development. To recapture our original example, suppose integer, real $\in \Omega$, integer $\leq$ real, $+ \in \Delta_2$, Γ_+(integer, integer) = integer, and Γ_+(real, real) = real. Then a particular instance of the above diagram is

$$\begin{array}{ccc} B(\text{integer}) \times B(\text{integer}) & \xrightarrow{\gamma_+(\text{integer},\text{integer})} & B(\text{integer}) \\ \downarrow B(\text{integer} \leq \text{real}) \times B(\text{integer} \leq \text{real}) & & \downarrow B(\text{integer} \leq \text{real}) \\ B(\text{real}) \times B(\text{real}) & \xrightarrow{\gamma_+(\text{real}, \text{real})} & B(\text{real}) \end{array}$$

In other words, the result of adding two integers and converting their sum to a real number must be the same as the result of converting the integers and adding the converted operands.

In essence, the key to insuring that implicit conversions and generic operators mesh nicely is to require a commutative relationship between these entities. An analogous relationship must also be required between implicit conversions and homomorphisms:

(3) If B,γ and B',γ' are $\Omega\Delta\Gamma$-algebras, then an homomorphism from B,γ to B',γ' is an Ω-indexed family of functions $\theta(\omega) \in B(\omega) \rightarrow B'(\omega)$ such that, whenever $\omega \leq \omega'$, the diagram

$$\begin{array}{ccc} B(\omega) & \xrightarrow{\theta(\omega)} & B'(\omega) \\ \downarrow{\scriptstyle B(\omega \leq \omega')} & & \downarrow{\scriptstyle B'(\omega \leq \omega')} \\ B(\omega') & \xrightarrow{\theta(\omega')} & B'(\omega') \end{array} \qquad \text{(III)}$$

commutes, and (as before) for all $n \geq 0$, $\delta \in \Delta_n$, and $\omega_1, \ldots, \omega_n \in \Omega$, the diagram (I) commutes.

Category-Sorted Algebras

By viewing the preordered set of sorts as a category, we have been able to use the category-theoretic concept of a functor to express appropriate restrictions on implicit conversion functions. In a similar vein, we can use the concept of a natural transformation to express the relationship between implicit conversion functions and interpretations given by diagram (II) and the relationship between implicit conversion functions and homomorphisms given by diagram (III).

In fact, diagram (III) is simply an assertion that the homomorphism θ is a natural transformation from the functor B to the functor B'. Diagram (II), however, is more complex. To express this diagram as a natural transformation, we must first define some notation for the exponentiation of categories and functors, and for the Cartesian product functor on SET:

(1) For any category K, we write:

(a) $|K|$ for the set (or collection) of objects of K.

(b) $X \vec{K} X'$ for the set of morphisms from X to X' in K.

(c) I^K_X for the identity morphism of X in K.

(d) $;_K$ for composition in K.

(2) For any category K, we write K^n to denote the category such that:

(a) $|K^n| = |K|^n$, i.e. the n-fold Cartesian product of $|K|$.

(b) $\langle X_1, \ldots, X_n \rangle \overset{\rightarrow}{K^n} \langle X'_1, \ldots, X'_n \rangle$

$= (X_1 \vec{K} X'_1) \times \ldots \times (X_n \vec{K} X'_n)$.

(c) $I^{K^n}_{\langle X_1, \ldots, X_n \rangle} = \langle I^K_{X_1}, \ldots, I^K_{X_n} \rangle$.

(d) $\langle \rho_1, \ldots, \rho_n \rangle ;_{K^n} \langle \rho'_1, \ldots, \rho'_n \rangle = \langle \rho_1 ;_K \rho'_1, \ldots, \rho_n ;_K \rho'_n \rangle$.

(Notice that when K is a preorder (e.g. Ω) this definition is consistent with the usual notion (e.g. Ω^n) of exponentiation of a preorder.)

(3) For any functor F from K to K', we write F^n to denote the functor from K^n to K'^n such that:

(a) $F^n(X_1, \ldots, X_n) = \langle F(X_1), \ldots, F(X_n) \rangle$.

(b) $F^n(\rho_1, \ldots, \rho_n) = \langle F(\rho_1), \ldots, F(\rho_n) \rangle$.

(4) We write $\times^{(n)}$ to denote the functor from SET^n to SET such that:

(a) $\times^{(n)}(S_1, \ldots, S_n) = S_1 \times \ldots \times S_n$.

(b) $\times^{(n)}(f_1, \ldots, f_n) = f_1 \times \ldots \times f_n$.

Next, we note that when Ω^n and Ω are viewed as categories, the monotone function Γ_δ can be viewed as a functor from Ω^n to Ω by defining its action on morphisms to be $\Gamma_\delta(\omega_1 \leq \omega'_1, \ldots, \omega_n \leq \omega'_n) = \Gamma_\delta(\omega_1, \ldots, \omega_n) \leq \Gamma_\delta(\omega'_1, \ldots, \omega'_n)$.

Then

$$\Omega^n \xrightarrow{B^n} SET^n \xrightarrow{\times^{(n)}} SET$$

and

$$\Omega^n \xrightarrow{\Gamma_\delta} \Omega \xrightarrow{B} SET$$

are compositions of functors which can be used to rewrite diagram (II) as:

$$\begin{array}{ccc}
(B^n;\times^{(n)})(\omega_1, \ldots, \omega_n) & \xrightarrow{\gamma_\delta(\omega_1, \ldots, \omega_n)} & (\Gamma_\delta;B)(\omega_1, \ldots, \omega_n) \\
\downarrow (B^n;\times^{(n)})(\omega_1 \leq \omega_1', \ldots, \omega_n \leq \omega_n') & & \downarrow (\Gamma_\delta;B)(\omega_1 \leq \omega_1', \ldots, \omega_n \leq \omega_n') \\
(B^n;\times^{(n)})(\omega_1', \ldots, \omega_n') & \xrightarrow{\gamma_\delta(\omega_1', \ldots, \omega_n')} & (\Gamma_\delta;B)(\omega_1', \ldots, \omega_n')
\end{array}$$

In this form, the diagram is clearly an assertion that γ_δ is a natural transformation from the functor $B^n;\times^{(n)}$ to the functor $\Gamma_\delta;B$.

At this stage we have come to regard Ω entirely as a category. Indeed, we can justify the term "category-sorted algebra" by extending our definition to the case where Ω is an arbitrary category:

(1) A signature consists of:

(1a) A category Ω of sorts.

(1b) A family, indexed by nonnegative integers, of disjoint sets Δ_n of operators of rank n.

(1c) For each $n \geq 0$ and $\delta \in \Delta_n$, a specification Γ_δ, which is a functor from Ω^n to Ω.

(2) An $\Omega\Delta\Gamma$-algebra consists of:

(2a) A carrier B, which is a functor from Ω to SET.

(2b) For each $n \geq 0$ and $\delta \in \Delta_n$, an interpretation γ_δ, which is a natural transformation from $B^n;\times^{(n)}$ to $\Gamma_\delta;B$.

(3) If B,γ and B',γ' are $\Omega\Delta\Gamma$-algebras, then an homomorphism from B,γ to B',γ' is a natural transformation from B to B' such that, for all $n \geq 0$, $\delta \in \Delta_n$, and $\omega_1, \ldots, \omega_n \in \Omega$, the diagram (I) commutes.

This is a clear illustration of what we mean by applying category theory to language definition. Our intention is not to use any deep theorems of category theory, but merely to employ the basic concepts of this field as organizing principles. This might appear as a desire to be concise at the expense of being esoteric. But in designing a programming language, the central problem is to organize a variety of concepts in a way which exhibits uniformity and generality. Substantial leverage can be gained in attacking this problem if these concepts can be defined concisely within a framework which has already proven its ability to impose uniformity and generality upon a wide variety of mathematics.

It is easy to verify that $\Omega\Delta\Gamma$-algebras and their homomorphisms form a category, which we will call $ALG_{\Omega\Delta\Gamma}$. It is also evident that these category-sorted algebras reduce to the Higgins algebras (with total specifications) discussed earlier when Ω is a discrete category (i.e., a partially ordered set in which $\omega \leq \omega'$ only holds when $\omega = \omega'$.)

Algebraic Semantics

We can now explicate our claim that defining semantics is tantamount to defining a target algebra. Suppose the target algebra is a category-sorted $\Omega\Delta\Gamma$-algebra B,γ. Then $B(\omega)$ is the set of meanings of type ω. Thus we can define the set M of all meanings to be the disjoint union of $B(\omega)$ over $\omega \in |\Omega|$, i.e.,

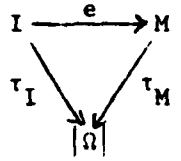
$$M = \{\omega,x \mid \omega \in |\Omega| \text{ and } x \in B(\omega)\} .$$

We can also define the function $\tau_M \in M \to |\Omega|$ such that

$$\tau_M(\omega, x) = \omega ,$$

which gives the type of each meaning in M.

Now let I be a set of identifiers and $\tau_I \in I \to |\Omega|$ be an assignment of types to each identifier in I. Then an environment e for I,τ_I is a function from I to M which maps each identifier into a meaning of the appropriate type, i.e., which makes the diagram

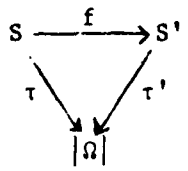


of functions commute.

To describe this situation in category-theoretic terms,we define the category $SET\downarrow|\Omega|$ of sets with type assignments. This is the category such that

(a) The objects of $SET\downarrow|\Omega|$ are pairs S,τ, where S is a set and $\tau \in S \to |\Omega|$,

(b) $S,\tau \underset{SET\downarrow|\Omega|}{\rightarrow} S',\tau'$ is the set of functions f from S to S' such that the diagram



commutes,

(c) Composition and identities in $SET\downarrow|\Omega|$ are the same as in SET.

Then an environment for I,τ_I is a morphism in $I,\tau_I \underset{SET\downarrow|\Omega|}{\rightarrow} M,\tau_M$. We call this set $Env(I,\tau_I)$.

Next we define U to be the functor from $ALG_{\Omega\Delta\Gamma}$ to $SET\downarrow|\Omega|$ whose action on an $\Omega\Delta\Gamma$-algebra B,γ is given by

$U(B,\gamma) = S,\tau$ where

$S = \{\omega,x \mid \omega \in |\Omega| \text{ and } x \subset B(\omega)\}$,

$\tau \in S \to |\Omega|$ is the function such that $\tau(\omega,x) = \omega$,

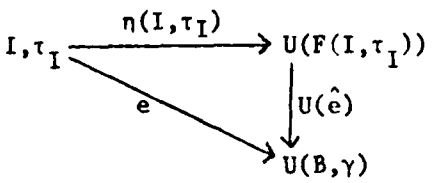
and whose action on an homomorphism θ from B,γ to B',γ' is given by

$U(\theta) \in U(B,\gamma) \underset{SET\downarrow|\Omega|}{\rightarrow} U(B',\gamma')$ is the function such that

$U(\theta)(\omega,x) = \omega,\theta(\omega)(x)$.

Then M,τ_M is the result of applying U to the target algebra B,γ, so that $Env(I,\tau_I) = I,\tau_I \overset{\rightarrow}{SET\downarrow|\Omega|} U(B,\gamma)$. More generally, U is the "forgetful" functor which forgets both interpretations and implicit conversions, and maps a category-sorted algebra into the disjoint union of its carrier, along with an appropriate assignment of types to this disjoint union.

In the appendix, we will show that for any object I,τ_I of $SET\downarrow|\Omega|$ there is an algebra $F(I,\tau_I)$, called the <u>free $\Omega\Delta\Gamma$-algebra generated by</u> I,τ_I, and a morphism $\eta(I,\tau_I) \in I,\tau_I \overset{\rightarrow}{SET\downarrow|\Omega|} U(F(I,\tau_I))$, called the <u>embedding of I,τ_I into its free algebra</u>, such that:

> For any $B,\gamma \in |ALG_{\Omega\Delta\Gamma}|$ and $e \in I,\tau_I \overset{\rightarrow}{SET\downarrow|\Omega|} U(B,\gamma)$, there is exactly one homomorphism $\hat{e} \in F(I,\tau_I) \overset{\rightarrow}{ALG}_{\Omega\Delta\Gamma} B,\gamma$ such that the diagram
>
> 
>
>
> in $SET\downarrow|\Omega|$ commutes.

Suppose $F(I,\tau_I) = B_0,\gamma_0$. Then each $B_0(\omega)$ is the set of phrases of type ω which can be constructed from identifiers in I whose types are given by τ_I. Each $\hat{e}(\omega)$ maps the phrases of type ω into their meanings in $B(\omega)$. Moreover, suppose $R,\tau_R = U(B_0,\gamma_0) = U(F(I,\tau_I))$. Then R is the set of phrases of all types, τ_R maps these phrases into their types, and $U(\hat{e})$ maps these phrases into their meanings in a way which preserves types.

The embedding $\eta(I,\tau_I)$ maps each identifier into the phrase which consists of that identifier. Thus the above diagram shows that the meaning $U(\hat{e})(\eta(I,\tau_I)(i))$ of the phrase consisting of i is the meaning e(i) given to i by the environment e.

For a given I,τ_I, one can define the $|\Omega|$-indexed family of semantic functions

$$\mu(\omega) \in B_0(\omega) \rightarrow (Env(I,\tau_I) \rightarrow B(\omega))$$

such that

$$\mu(\omega)(r)(e) = \hat{e}(\omega)(r) .$$

Then each $\mu(\omega)$ maps phrases of type ω into functions from environments to meanings of type ω. Alternatively, one can define the single semantic function

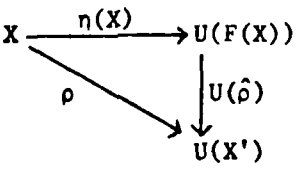
$$\mu \in R \to (Env(I,\tau_I) \to M)$$

such that

$$\mu(r)(e) = U(\hat{e})(r) \ .$$

This function maps phrases of all types into functions from environments to meanings.

It is evident that the linguistic application of category-sorted algebras depends crucially upon the existence of free algebras or, more abstractly, upon the existence of a left adjoint to the forgetful functor U. In general, if U is any functor from a category K' to a category K, F is a functor from K to K', and η is a natural transformation from I_K to F;U such that:

> For all $X \in |K|$, $X' \in |K'|$, and $\rho \in X \underset{K}{\to} U(X')$, there is exactly one morphism $\hat{\rho} \in F(X) \underset{K'}{\to} X'$ such that
>
> 
>
>
> commutes in K,

then F is said to be a <u>left adjoint of</u> U, with <u>associated natural transformation</u> η. The triple F, U, η is called an <u>adjunction</u> from K to K'.

In the appendix, we show the existence of free category-sorted algebras by constructing a left adjoint and associated natural transformation for the forgetful functor U from $ALG_{\Omega\Delta\Gamma}$ to $SET^{\downarrow|\Omega|}$.

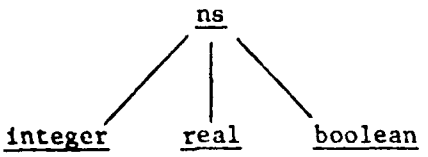
Data Algebras

To illustrate the application of category-sorted algebras, we will consider several variations of Algol 60. However, since we do not yet know how to treat binding mechanisms elegantly in an algebraic framework, we will limit ourselves to the subset of Algol which excludes the binding of identifiers, i.e., to the simple imperative language which underlies Algol. Although this is a substantial limitation, we will still be able to show the potential of our methodology for disciplining the design of implicit conversions and generic operators.

As discussed in (7) and (8), we believe that a fundamental characteristic of Algol-like languages is the presence of two kinds of type: data types, which describe variables (or expressions) and their ranges of values, and phrase types (called program types in (7)) which describe identifiers (or phrases which can be bound to identifiers) and their sets of meanings.

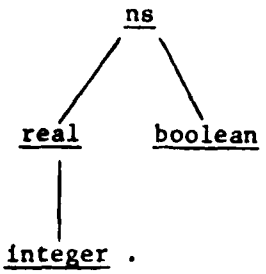
Algebraically, Ω should be a set of data types in order to define the values of expressions. In this case, the carrier of the free algebra is a data-type-indexed family of sets of expressions, and the carrier of the target algebra, which we will call a data algebra, is a data-type-indexed family of sets of values.

In Algol 60 itself there are three data types: integer, real, and boolean, to which we must add the nonsense type ns. To avoid implicit conversions, we would take Ω to be

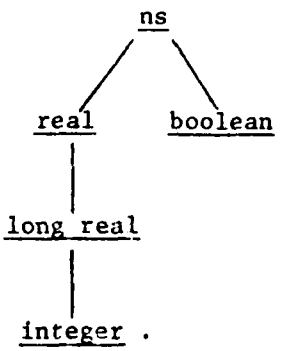


Notice that ns is the greatest element in this partial ordering, reflecting the notion that any sensible expression can occur in a context which permits nonsense.

On the other hand, to introduce an implicit conversion from integer to real, we would take integer to be a subtype of real:



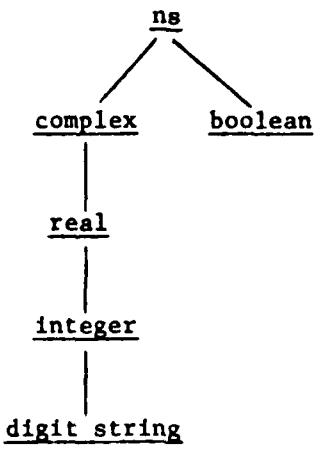
A more interesting situation arises when long real is introduced. One might expect real to be a subtype of long real, but an implicit conversion from real to long real would be dangerous from the viewpoint of numerical analysis, since a real value does not provide enough information to completely determine a long real value. In fact, it is the opposite implicit conversion which is numerically safe, so that long real should be a subtype of real:



In a language definition which was sufficiently concrete to make sense of the distinction between real and long real, one might take B(real) and B(long real) to be sets of real numbers with single and double precision representations, respectively, and B(long real $\leq$ real) to be the truncation or roundoff function from B(long real) to B(real). Notice that this function is not an injection, reflecting the fact that a conversion from long real to real loses information.

However, although this is suggestive, our methodology is not really adequate for dealing with the problems of roundoff or overflow. For this reason, we will omit the type long real and define our language at the level of abstraction where roundoff and overflow are ignored.

In the rest of this paper we will take Ω to be:



It should be emphasized that this choice of Ω - particularly the use of digit string - is purely for illustrative purposes, and is not put forth as desirable for a real programming language.

In the carrier of our target algebra we will have:

B(digit string) = the set of strings of digits,
B(integer) = the set of integers,
B(real) = the set of real numbers,
B(complex) = the set of complex numbers,
B(boolean) = {true, false},

with the conversion functions

B(digit string ≤ integer) = the function which maps each digit string into the integer of which it is a decimal representation.
B(integer ≤ real) = the identity injection from integers to real numbers.
B(real ≤ complex) = the identity injection from real numbers to complex numbers.

Notice that, because of the possible presence of leading zeros, the function B(digit string ≤ integer) is not an injection.

We must also specify B(ns) and the conversion functions into this set. For these conversion functions to exist, B(ns) must be nonempty, i.e., we must give some kind of meaning to nonsense expressions. The closest we can come to saying that they do not make sense is to give them all the same meaning by taking B(ns) to be a singleton set. This insures (since a singleton set is a terminal element in the category SET), that there will be exactly one possible conversion function from any data type to ns:

$B(\underline{ns}) = \{<>\}$,

$B(\omega \leq \underline{ns})$ = the unique function from $B(\omega)$ to $\{<>\}$.

As an example of an operator, let + be a member of Δ_2, with the specification

$$\Gamma_+(\omega_1,\omega_2) = \underline{\text{if}}\ \omega_1 \leq \underline{\text{integer}}\ \underline{\text{and}}\ \omega_2 \leq \underline{\text{integer}}\ \underline{\text{then}}\ \underline{\text{integer}}$$
$$\underline{\text{else if}}\ \omega_1 \leq \underline{\text{real}}\ \underline{\text{and}}\ \omega_2 \leq \underline{\text{real}}\ \underline{\text{then}}\ \underline{\text{real}}$$
$$\underline{\text{else if}}\ \omega_1 \leq \underline{\text{complex}}\ \underline{\text{and}}\ \omega_2 \leq \underline{\text{complex}}\ \underline{\text{then}}\ \underline{\text{complex}}$$
$$\underline{\text{else}}\ \underline{\text{ns}}$$

and the interpretation

$$\gamma_+(\omega_1,\omega_2) = \underline{\text{if}}\ \omega_1 \leq \underline{\text{integer}}\ \underline{\text{and}}\ \omega_2 \leq \underline{\text{integer}}\ \underline{\text{then}}$$
$$\lambda(x,y).\ \underline{\text{let}}\ x' = B(\omega_1 \leq \underline{\text{integer}})(x)\ \underline{\text{and}}\ y' = B(\omega_2 \leq \underline{\text{integer}})(y)$$
$$\underline{\text{in}}\ \text{integer-addition}(x',y')$$
$$\underline{\text{else if}}\ \omega_1 \leq \underline{\text{real}}\ \underline{\text{and}}\ \omega_2 \leq \underline{\text{real}}\ \underline{\text{then}}$$
$$\lambda(x,y).\ \underline{\text{let}}\ x' = B(\omega_1 \leq \underline{\text{real}})(x)\ \underline{\text{and}}\ y' = B(\omega_2 \leq \underline{\text{real}})(y)$$
$$\underline{\text{in}}\ \text{real-addition}(x',y')$$
$$\underline{\text{else if}}\ \omega_1 \leq \underline{\text{complex}}\ \underline{\text{and}}\ \omega_2 \leq \underline{\text{complex}}\ \underline{\text{then}}$$
$$\lambda(x,y).\ \underline{\text{let}}\ x' = B(\omega_1 \leq \underline{\text{complex}})(x)\ \underline{\text{and}}\ y' = B(\omega_2 \leq \underline{\text{complex}})(y)$$
$$\underline{\text{in}}\ \text{complex-addition}(x',y')$$
$$\underline{\text{else}}\ \lambda(x,y).\ <> .$$

Although the above definition makes + a purely numerical operator, it can be extended to encompass nonnumerical"addition":

$$\Gamma_+(\omega_1,\omega_2) = \underline{\text{if}}\ \omega_1 \leq \underline{\text{boolean}}\ \underline{\text{and}}\ \omega_2 \leq \underline{\text{boolean}}\ \underline{\text{then}}\ \underline{\text{boolean}}$$
$$\underline{\text{else if}}\ \omega_1 \leq \underline{\text{digit string}}\ \underline{\text{and}}\ \omega_2 \leq \underline{\text{digit string}}\ \underline{\text{then}}\ \underline{\text{digit string}}$$
$$\underline{\text{else}}\ \ \ldots\ \ \text{(as before)}$$

$$\gamma_+(\omega_1,\omega_2) = \underline{\text{if}}\ \omega_1 \leq \underline{\text{boolean}}\ \underline{\text{and}}\ \omega_2 \leq \underline{\text{boolean}}\ \underline{\text{then}}$$
$$\lambda(x,y).\ \underline{\text{let}}\ x' = B(\omega_1 \leq \underline{\text{boolean}})(x)\ \underline{\text{and}}\ y' = B(\omega_2 \leq \underline{\text{boolean}})(y)$$
$$\underline{\text{in}}\ \text{boolean-addition}(x',y')$$
$$\underline{\text{else if}}\ \omega_1 \leq \underline{\text{digit string}}\ \underline{\text{and}}\ \omega_2 \leq \underline{\text{digit string}}\ \underline{\text{then}}$$
$$\lambda(x,y).\ \underline{\text{let}}\ x' = B(\omega_1 \leq \underline{\text{digit string}})(x)$$
$$\underline{\text{and}}\ y' = B(\omega_2 \leq \underline{\text{digit string}})(y)$$
$$\underline{\text{in}}\ \text{digit-string-addition}(x',y')$$
$$\underline{\text{else}}\ \ \ldots\ \ \text{(as before)}\ .$$

Since there are no implicit conversions between boolean and any other type than ns, we are free to choose "boolean addition" to be any function from pairs of truth values to truth values. On the other hand, "digit-string addition" is tightly constrained by the implicit conversion from digit string to integer, which gives rise to the requirement that

$$\begin{array}{ccc} B(\underline{\text{digit string}}) \times B(\underline{\text{digit string}}) & \xrightarrow{\text{digit-string addition}} & B(\underline{\text{digit string}}) \\ \Big\downarrow B(\underline{\text{digit string}} \leq \underline{\text{integer}}) \times B(\underline{\text{digit string}} \leq \underline{\text{integer}}) & & \Big\downarrow B(\underline{\text{digit string}} \leq \underline{\text{integer}}) \\ B(\underline{\text{integer}}) \times B(\underline{\text{integer}}) & \xrightarrow{\text{integer addition}} & B(\underline{\text{integer}}) \end{array}$$

commute. In other words, the sum of two digit strings must be a decimal representation of the sum of the integers which are represented by those two strings. The only freedom we have in defining digit-string addition is in the treatment of leading zeros in the result.

The definition of + suggests that a typical operator will have a significant specification and interpretation for certain "key" sorts of operands, and that its specification and interpretation for other sorts of operands can be obtained by implicitly converting the operands to key sorts. To formalize this idea, let

(1) Λ_δ be a category of <u>keys</u>.

(2) Φ_δ be a functor from Λ to Ω^n.

(3) $\overline{\Gamma}_\delta$ be a functor from Λ to Ω.

(4) $\overline{\gamma}_\delta$ be a natural transformation from $\Phi_\delta;B^n;\times^{(n)}$ to $\overline{\Gamma}_\delta;B$.

Intuitively, for each key $\lambda \in |\Lambda_\delta|$, $\Phi_\delta(\lambda)$ is the n-tuple of sorts to which the "λ-version" of δ is applicable, $\overline{\Gamma}_\delta(\lambda)$ is the sort of the result of the λ-version of δ, and $\overline{\gamma}_\delta(\lambda) \in \times^{(n)}(B^n(\Phi_\delta(\lambda))) \rightarrow B(\overline{\Gamma}_\delta(\lambda))$ is the interpretation of the λ-version of δ.

These entities can be extended to all sorts of operands if the functor Φ_δ possesses a left adjoint Ψ_δ, which will be a functor from Ω^n to Λ, and an associated natural transformation η_δ, which will be a natural transformation from I_{Ω^n} to $\Psi_\delta;\Phi_\delta$. Then we can define the specification

$$\Gamma_\delta = \Psi_\delta;\overline{\Gamma}_\delta \in \Omega^n \rightarrow \Omega ,$$

and the interpretation

$$\gamma_\delta(\omega_1, \ldots, \omega_n) = \times^{(n)}(B^n(\eta_\delta(\omega_1, \ldots, \omega_n));\overline{\gamma}_\delta(\Psi_\delta(\omega_1, \ldots, \omega_n)) ,$$

which can easily be shown to be a natural transformation from $B^n;\times^{(n)}$ to $\Gamma_\delta;B$. Intuitively, $\Psi_\delta(\omega_1, \ldots, \omega_n)$ can be thought of as the key determining the version of δ to be used for operands of sorts $\omega_1, \ldots, \omega_n$, and $\eta_\delta(\omega_1, \ldots, \omega_n)$ as the implicit conversion to be applied to these operands.

In the special case where Λ_δ and Ω are partially ordered sets, it can be shown (9, p. 93) that Ψ_δ will be a left adjoint of Φ_δ if and only if $\overline{\omega} \leq \Phi_\delta(\Psi_\delta(\overline{\omega}))$ for all $\overline{\omega} \in \Omega^n$ and $\Psi_\delta(\Phi_\delta(\lambda)) \leq \lambda$ for all $\lambda \in \Lambda_\delta$. In this case $\eta_\delta(\overline{\omega})$ will be the unique morphism $\overline{\omega} \leq \Phi_\delta(\Psi_\delta(\overline{\omega}))$, and γ_δ will be

$$\gamma_\delta(\overline{\omega}) = \times^{(n)}(B^n(\overline{\omega} \leq \Phi_\delta(\Psi_\delta(\overline{\omega}))));\overline{\gamma}_\delta(\Psi_\delta(\overline{\omega})) .$$

Moreover, as shown by the following proposition, Ψ_δ will be uniquely determined by Φ_δ:

Proposition Suppose Φ is a monotone function from Λ to Ω^n, where Λ and Ω^n are partially ordered sets, such that

(1) For all $\bar{\omega} \in \Omega^n$, the set $\{\lambda \mid \lambda \in \Lambda \text{ and } \bar{\omega} \leq \Phi(\lambda)\}$ has a greatest lower bound in Λ.

(2) For all $\bar{\omega} \in \Omega^n$, $\Phi(\sqcap_\Lambda \{\lambda \mid \lambda \in \Lambda \text{ and } \bar{\omega} \leq \Phi(\lambda)\})$ is the greatest lower bound in Ω^n of $\{\Phi(\lambda) \mid \lambda \in \Lambda \text{ and } \bar{\omega} \leq \Phi(\lambda)\}$.

Then $\Psi(\bar{\omega}) = \sqcap_\Lambda \{\lambda \mid \lambda \in \Lambda \text{ and } \bar{\omega} \leq \Phi(\lambda)\}$ is the unique monotone function from Ω^n to Λ such that Ψ is a left adjoint of Φ.

Proof: Ψ is obviously monotone. For any $\lambda \in \Lambda$, $\Psi(\Phi(\lambda))$ is the greatest lower bound of $\{\lambda' \mid \lambda' \in \Lambda \text{ and } \Phi(\lambda) \leq \Phi(\lambda')\}$ and, since λ belongs to this set, $\Psi(\Phi(\lambda)) \leq \lambda$. For any $\bar{\omega} \in \Omega^n$, $\Phi(\Psi(\bar{\omega})) = \Phi(\sqcap_\Lambda \{\lambda \mid \lambda \in \Lambda \text{ and } \bar{\omega} \leq \Phi(\lambda)\})$ is the greatest lower bound of $\{\Phi(\lambda) \mid \lambda \in \Lambda \text{ and } \bar{\omega} \leq \Phi(\lambda)\}$ and, since $\bar{\omega}$ is a lower bound of this set, $\bar{\omega} \leq \Phi(\Psi(\bar{\omega}))$.

Suppose Ψ is a left adjoint of Φ. If $\bar{\omega} \leq \Phi(\lambda)$ then $\Psi(\bar{\omega}) \leq \Psi(\Phi(\lambda)) \leq \lambda$. Thus $\Psi(\bar{\omega})$ is a lower bound of $\{\lambda \mid \lambda \in \Lambda \text{ and } \bar{\omega} \leq \Phi(\lambda)\}$. Moreover, this set contains $\Psi(\bar{\omega})$ since $\bar{\omega} \leq \Phi(\Psi(\bar{\omega}))$. Thus any lower bound of this set must be less than $\Psi(\bar{\omega})$, so that $\Psi(\bar{\omega})$ is the greatest lower bound.

The conditions in this proposition will hold if Λ contains greatest lower bounds of all of its subsets, i.e., if Λ is a complete lattice, and Φ preserves all greatest lower bounds. However, we will sometimes use Λ's which are not complete lattices.

As an example, the purely numeric definition of + given earlier can be recast more concisely by using the set of keys

$$\Lambda_+ = \{\underline{\text{integer}}, \underline{\text{real}}, \underline{\text{complex}}, \underline{\text{ns}}\}$$

with the same partial ordering as Ω. Then the specification Γ_+ is determined by the functions Φ_+ and $\bar{\Gamma}_+$ such that

λ	$\Phi_+(\lambda)$	$\overline{\Gamma}_+(\lambda)$
integer	integer,integer	integer
real	real,real	real
complex	complex,complex	complex
ns	ns,ns	ns

and the interpretation γ_+ is determined by

$\overline{\gamma}_+(\underline{\text{integer}})$ = integer addition

$\overline{\gamma}_+(\underline{\text{real}})$ = real addition

$\overline{\gamma}_+(\underline{\text{complex}})$ = complex addition

$\overline{\gamma}_+(\underline{\text{ns}}) = \lambda(x,y).\ \langle\rangle$.

To extend this definition to nonnumeric types, one adds <u>boolean</u> and <u>digit string</u> to Λ_+, with

λ	$\Phi_+(\lambda)$	$\overline{\Gamma}_+(\lambda)$
boolean	boolean,boolean	boolean
digit string	digit string,digit string	digit string

and

$\overline{\gamma}_+(\underline{\text{boolean}})$ = boolean addition

$\overline{\gamma}_+(\underline{\text{digit string}})$ = digit-string addition .

(Notice that in this case Λ_+ is not a complete lattice, but the necessary conditions for the existence of a left adjoint to Φ_+ are still met.)

In the remainder of this section we will illustrate our approach by defining a few other binary operators. In each case Λ_+ is the listed subset of Ω, with the same partial ordering as Ω.

For the division operators / and ÷ we can define

λ	$\Phi_{/}(\lambda)$	$\overline{\Gamma}_{/}(\lambda)$
real	real,real	real
complex	complex,complex	complex
ns	ns,ns	ns

$\overline{\gamma}_{/}$(real) = real division

$\overline{\gamma}_{/}$(complex) = complex division

$\overline{\gamma}_{/}$(ns) = λ(x,y). <>

and

λ	$\Phi_{\div}(\lambda)$	$\overline{\Gamma}_{\div}(\lambda)$
integer	integer,integer	integer
ns	ns,ns	ns

$\overline{\gamma}_{\div}$(integer) = λ(x,y). the unique integer q such that
x = q × y + r where
if $x \geq 0$ then $0 \leq r < |y|$ else $-|y| < r \leq 0$,

$\overline{\gamma}_{\div}$(ns) = λ(x,y). <> .

These operations cannot be combined into a single operator since, for example, 3/2 = 1.5 but 3 ÷ 2 = 1. On the other hand, since the definition of $\gamma_{\div}$(integer) extends sensibly to the case where x and y are real, one could generalize ÷ by taking $\Phi_{\div}$(integer) = real,real.

Since nonnegative integers have not been introduced as a data type and, for example, 3^{-2} is not an integer, exponentiation cannot be defined to yield an integer result for any sort of operands. If exponents are limited to integers, one can define

$\lambda_{\uparrow}$	$\Phi_{\uparrow}(\lambda)$	$\overline{\Gamma}_{\uparrow}(\lambda)$
real	real,integer	real
complex	complex,integer	complex
ns	ns,ns	ns

$\overline{\gamma}_{\uparrow}(\underline{\text{real}}) = \lambda(x,n).\ x^n$

$\overline{\gamma}_{\uparrow}(\underline{\text{complex}}) = \lambda(x,n).\ x^n$

$\overline{\gamma}_{\uparrow}(\underline{\text{ns}}) = \lambda(x,y).\ \langle\rangle$.

This can be extended to noninteger exponents by taking $\Phi_{\uparrow}(\underline{\text{complex}}) =$ $\underline{\text{complex}},\underline{\text{complex}}$, but the multi-valued nature of complex exponentiation (as well as the time required to compute the necessary logarithms and exponentials) would probably make this unwise.

Finally, we define an equality operation:

λ	$\Phi_{=}(\lambda)$	$\overline{\Gamma}_{=}(\lambda)$
boolean	boolean,boolean	boolean
integer	integer,integer	boolean
real	real,real	boolean
complex	complex,complex	boolean
ns	ns,ns	ns

$\overline{\gamma}_{=}(\lambda)$ = $\underline{\text{if}}$ $\lambda \neq \underline{\text{ns}}$ $\underline{\text{then}}$ the equality relation for $B(\lambda)$
$\underline{\text{else}}$ $\lambda(x,y).\ \langle\rangle$.

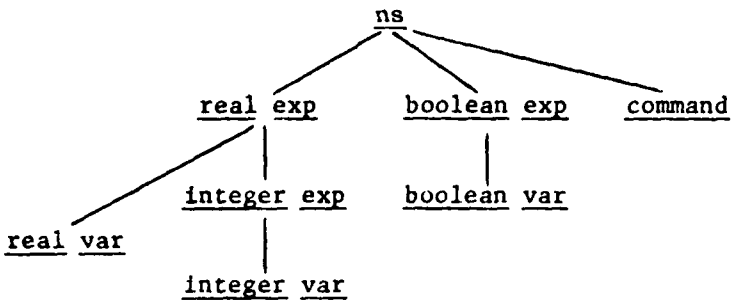
One might be tempted to add $\underline{\text{digit string}}$ to $\Lambda_{=}$, with $\Phi_{=}(\underline{\text{digit string}}) =$ $\underline{\text{digit string}},\underline{\text{digit string}}$, $\overline{\Gamma}_{=}(\underline{\text{digit string}}) = \underline{\text{boolean}}$, and $\overline{\gamma}_{=}(\underline{\text{digit string}})$ = the equality relation for $B(\underline{\text{digit string}})$. However, the diagram

$$\begin{array}{ccc} B(\underline{\text{digit string}}) \times B(\underline{\text{digit string}}) & \xrightarrow{=_{\underline{\text{digit string}}}} & B(\underline{\text{boolean}}) \\ \Big\downarrow B(\underline{\text{digit string}} \leq \underline{\text{integer}}) \times B(\underline{\text{digit string}} \leq \underline{\text{integer}}) & & \Big\downarrow I_{B(\underline{\text{boolean}})} \\ B(\underline{\text{integer}}) \times B(\underline{\text{integer}}) & \xrightarrow{=_{\underline{\text{integer}}}} & B(\underline{\text{boolean}}) \end{array}$$

does not commute, since $\mathcal{B}$(digit string $\leq$ integer) is not an injection. (For example, 6 and 06 are unequal digit strings which convert to equal integers.) Indeed, one can never use the same operator for the equality relation on different data types when the data types are connected by an implicit conversion function which is not an injection. (At the more concrete level where roundoff error is taken into account, this suggests, quite correctly, that there are special perils surrounding an equality operation for real numbers.)

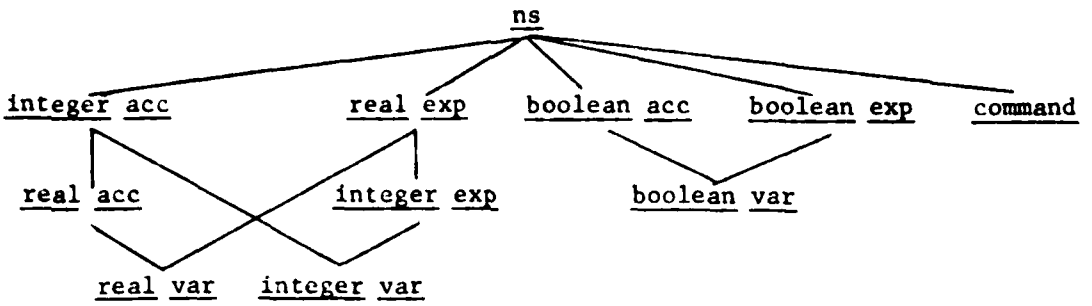
Algebras for Simple Imperative Languages

Now we move from data algebras, which describe languages of expressions, to algebras which describe simple imperative programming languages, i.e., languages with variables, expressions, and commands, but without binding operations. The sorts of our algebras will change from data types to phrase types, which can be thought of as phrase class names of the abstract syntax for the language being defined. For example, in place of the set of data types {integer, real, boolean}, Ω might be the following partially ordered set of phrase types:



It is evident that for each data type τ there will be two phrase types τ exp(ression) and τ var(iable), and that τ exp will be a subtype of τ' exp whenever the data type τ is a subtype of τ'. Moreover, τ var will be a subtype of τ exp since a variable can be used in any context which permits an expression of the same data type. On the other hand, the subtype relation will never hold between variables of distinct data types. For example, an integer variable cannot be used as a real variable since it cannot accept a noninteger value, and a real variable cannot be used as an integer variable since it might produce a noninteger value.

This kind of phrase-type structure, which describes many programming languages, is unpleasantly asymmetric. For each data type, there are variables, which can accept or produce values, and expressions, which can only produce values. Thus one might expect another kind of phrase, called an acceptor, which can only accept values. If acceptors for each data type are added to Ω, we have:



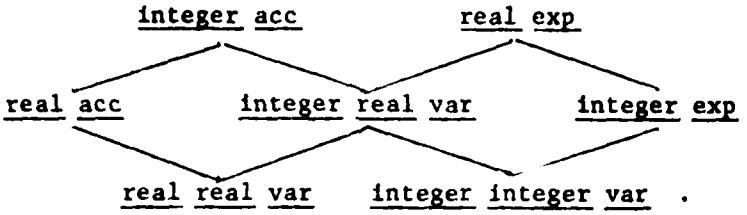
Notice that the subtype relation among acceptors is the dual of that for data types or expressions. For example, a real acceptor can be used as an integer acceptor since an integer value can be converted into a real value.

The above partial ordering has the peculiarity that there is a pair of phrase types, real var and integer var, which have no least upper bound. In general this might not be a problem, but we will find that there is one language construct, the general conditional phrase, which requires the existence of binary least upper bounds. To see the problem, suppose n is an integer variable and x is a real variable, and consider the conditional variable

if p then n else x .

In a context which calls for an expression, this phrase must be considered a real expression, since when p is false it can produce a noninteger value. But *in a context which calls for an acceptor*, the phrase must be considered an integer acceptor, since when p is true it cannot accept a noninteger value. The phrase type which describes this situation must be a subtype of both integer acc and real exp which in turn has real var and integer var as its subtypes. In other words, it must be the least upper bound of real var and integer var.

The way out of this difficulty is to characterize variables by both the data type which they accept and the data type which they produce. For example, a real var is actually a "real-accepting, real-producing" variable, an integer var is actually an "integer-accepting, integer-producing" variable, and the above conditional variable is an "integer-accepting, real-producing" variable. If we write $\tau_1\ \tau_2$ var to abbreviate "τ_1-accepting, τ_2-producing" variable, then we have the ordering



Implicit in this discussion is the idea that phrase types are constructed from data types. More generally, since the meaning of expressions can be described by a data algebra, and expressions are a major constituent of an imperative programming language, it should be possible to define the algebra describing the programming language in terms of the data algebra describing its expressions. To emphasize this possibility we will construct a programming-language algebra for an arbitrary data algebra, with signature Ω^D, Δ^D, Γ^D, carrier B^D, and interpretation γ^D. The main restrictions we will place upon this data algebra are that ns must be the greatest sort in Ω^D, and that $\Gamma^D(\omega_1, \ldots, \omega_n) = \text{ns}$ must hold when any ω_i is ns.

The set of phrase types is

$$\Omega = \{\tau\ \text{exp} \mid \tau \in \Omega^D - \{\text{ns}\}\} \cup \{\tau\ \text{acc} \mid \tau \in \Omega^D - \{\text{ns}\}\}$$
$$\cup \{\tau_1\tau_2\ \text{var} \mid \tau_1, \tau_2 \in \Omega^D - \{\text{ns}\}\} \cup \{\text{comm}, \text{ns}\} ,$$

with the least partial ordering such that

if $\tau \leq^D \tau'$ then τ exp $\leq \tau'$ exp
if $\tau' \leq^D \tau$ then τ acc $\leq \tau'$ acc
if $\tau_1' \leq^D \tau_1$ and $\tau_2 \leq^D \tau_2'$ then $\tau_1\tau_2$ var $\leq \tau_1'\tau_2'$ var
$\tau_1\tau_2$ var $\leq \tau_1$ acc
$\tau_1\tau_2$ var $\leq \tau_2$ exp
$\omega \leq$ ns .

Our target algebra describes direct semantics. (Continuation semantics can be treated in much the same way, but it leads to more complex definitions without providing any additional insights into the concerns of this paper.) The carrier of this target algebra will map each sort into a domain (a partially ordered set containing a least element $\perp$ and least upper bounds of its directed subsets), with implicit conversion functions which are strict and continuous (i.e., which preserve $\perp$ and least upper bounds of directed sets). Specifically, the following carrier is appropriate for direct semantics:

$$B(\tau\ \underline{\text{exp}}) = S \to [B^D(\tau)]_\perp$$

$$B(\underline{\text{comm}}) = S \to [S]_\perp$$

$$B(\tau\ \underline{\text{acc}}) = B^D(\tau) \to B(\underline{\text{comm}})$$

$$B(\tau_1\tau_2\ \underline{\text{var}}) = B(\tau_1\ \underline{\text{acc}}) \times B(\tau_2\ \underline{\text{exp}})$$

$$B(\underline{\text{ns}}) = \{\perp\}$$

$$B(\tau\ \underline{\text{exp}} \leq \tau'\ \underline{\text{exp}}) = \lambda v.\ v; [B^D(\tau \leq \tau')]_\perp$$

$$B(\tau\ \underline{\text{acc}} \leq \tau'\ \underline{\text{acc}}) = \lambda a.\ B^D(\tau' \leq \tau);\ a$$

$$B(\tau_1\tau_2\ \underline{\text{var}} \leq \tau_1\ \underline{\text{acc}}) = \lambda(a, v).\ a$$

$$B(\tau_1\tau_2\ \underline{\text{var}} \leq \tau_2\ \underline{\text{exp}}) = \lambda(a, v).\ v$$

$$B(\tau_1\tau_2\ \text{var} \leq \tau_1'\tau_2'\ \underline{\text{var}}) = B(\tau_1\ \underline{\text{acc}} \leq \tau_1'\ \underline{\text{acc}}) \times B(\tau_2\ \underline{\text{exp}} \leq \tau_2'\ \underline{\text{exp}})$$

$$B(\omega \leq \underline{\text{ns}}) = \lambda x.\ \perp_{B(\underline{\text{ns}})}.$$

Here S is an unspecified set of store states. For any set X, $[X]_\perp$ denotes the flat domain obtained by adding $\perp$ to X. For any function $f \in X \to X'$, $[f]_\perp$ denotes the strict extension of f to $[X]_\perp \to [X']_\perp$.

Basically, the meaning of a command is a state transition function (with result $\perp$ for nontermination), the meaning of an acceptor is a function from data values to state transition functions, and the meaning of a variable is a pair giving both the meaning of an acceptor and of an expression. Notice that this way of defining variables avoids the mention of any entities such as Strachey's L-values. (As a consequence, our definition permits strangely behaved variables akin to the implicit references in GEDANKEN.[10])

Next we consider operators. Each operator of the data algebra becomes an expression-producing operator of the imperative-language algebra. If $\delta \in \Delta_n^D$, then $\delta \in \Delta_n$, with the specification given by:

$$\Lambda_\delta = (\Omega^D)^n$$

$$\Phi_\delta = \phi^n \text{ , where } \phi \in \Omega^D \to \Omega \text{ is the function such that}$$

$$\phi(\tau) = \underline{\text{if}}\ \tau = \underline{\text{ns}}\ \underline{\text{then}}\ \underline{\text{ns}}\ \underline{\text{else}}\ \tau\ \underline{\text{exp}} \text{ ,}$$

$$\overline{\Gamma}_\delta = \Gamma_\delta^D;\phi \text{ .}$$

To define the interpretation of δ we must give a natural transformation $\overline{\gamma}_\delta$ from $\Phi_\delta;B^n;\times^{(n)} = \phi^n;B^n;\times^{(n)}$ to $\overline{\Gamma}_\delta;B = \Gamma_\delta^D;\phi;B$. Thus $\overline{\gamma}_\delta(\tau_1, \ldots, \tau_n)$ must be a function from $B(\phi(\tau_1)) \times \ldots \times B(\phi(\tau_n))$ to $B(\phi(\Gamma_\delta^D(\tau_1, \ldots, \tau_n)))$. If $\Gamma_\delta^D(\tau_1, \ldots, \tau_n)$ is $\underline{\text{ns}}$, then $\overline{\gamma}_\delta(\tau_1, \ldots, \tau_n)$ will be the unique function from $B(\phi(\tau_1)) \times \ldots \times B(\phi(\tau_n))$ to $B(\underline{\text{ns}})$. Otherwise, none of the τ_i will be $\underline{\text{ns}}$, and $\overline{\gamma}_\delta(\tau_1, \ldots, \tau_n)$ will be the function from $B(\tau_1\ \underline{\text{exp}}) \times \ldots \times B(\tau_n\ \underline{\text{exp}}) = (S \to [B^D(\tau_1)]_\perp) \times \ldots \times (S \to [B^D(\tau_n)]_\perp)$ to $B(\Gamma_\delta^D(\tau_1, \ldots, \tau_n)\ \underline{\text{exp}}) = S \to [B^D(\Gamma_\delta^D(\tau_1, \ldots, \tau_n))]_\perp$ such that

$$\overline{\gamma}_\delta(\tau_1, \ldots, \tau_n)(v_1, \ldots, v_n) = \lambda\sigma \in S.\ [\gamma_\delta^D(\tau_1, \ldots, \tau_n)]_{\perp\perp}(v_1(\sigma), \ldots, v_n(\sigma)) \text{ ,}$$

where $[\gamma_\delta^D(\tau_1, \ldots, \tau_n)]_{\perp\perp}$ denotes the extension of $\gamma_\delta^D(\tau_1, \ldots, \tau_n)$ such that $[\gamma_\delta^D(\tau_1, \ldots, \tau_n)]_{\perp\perp}(x_1, \ldots, x_n) = \perp$ if any $x_i = \perp$.

Assignment is an operator $:= \in \Delta_2$. This is the one case which we cannot define by using an adjunction from a set of keys. The specification is

$$\Gamma_{:=}(\omega_1,\omega_2) = \underline{\text{if}}\ (\exists\tau \in \Omega^D - \{\underline{\text{ns}}\})\ \omega_1 \leq \tau\ \underline{\text{acc}}\ \underline{\text{and}}\ \omega_2 \leq \tau\ \underline{\text{exp}}$$

$$\underline{\text{then}}\ \underline{\text{comm}}\ \underline{\text{else}}\ \underline{\text{ns}}$$

If a data type τ meeting the above condition exists, then the interpretation is

$$\gamma_{:=}(\omega_1,\omega_2) = \lambda(a,v).\ \underline{\text{let}}\ a' = B(\omega_1 \leq \tau\ \underline{\text{acc}})(a)\ \underline{\text{and}}\ v' = B(\omega_2 \leq \tau\ \underline{\text{exp}})(v)$$

$$\underline{\text{in}}\ D_{\text{comm}}(v';[a']_{\mathcal{Q}}) \text{ ;}$$

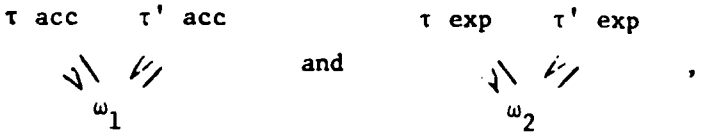
otherwise

$$\gamma_{:=}(\omega_1,\omega_2) = \lambda(a,v).\ \perp_{B(\underline{ns})} .$$

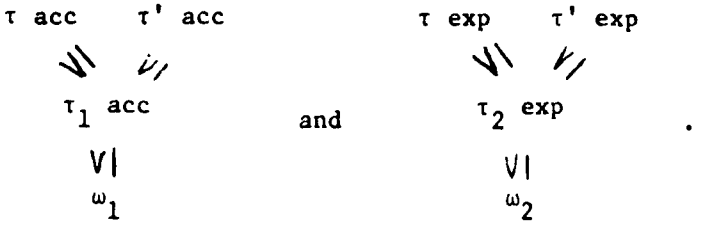
Here $[a']_{\textcircled{\perp}}$ is the $\perp$-preserving extension of a' from $B^D(\tau) \to B(\underline{comm})$ to $[B^D(\tau)]_\perp \to B(\underline{comm})$, and $D_{comm} \in (S \to B(\underline{comm})) \to B(\underline{comm})$ is the diagonalizing function such that

$$D_{comm}(h)(\sigma) = h(\sigma)(\sigma).$$

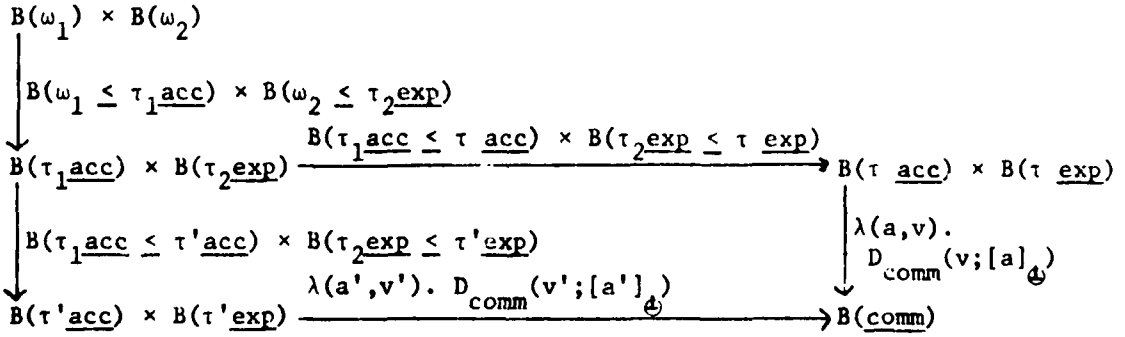
A subtlety in this definition is that the data type τ may not be unique. For example, if ω_1 is $\underline{real}\ \underline{acc}$ and ω_2 is $\underline{integer}\ \underline{exp}$ then τ can be either $\underline{integer}$ or $\underline{real}$. However, the definition still gives a unique meaning to $\gamma_{:=}$. Basically, this is because the structure of Ω insures that, if



then there are data types τ_1 and τ_2 such that



Then the definition of B for the implicit conversion of acceptors and expressions implies that the diagram



of functions commutes. A slight extension of this argument shows that $\gamma_{:=}$ is a natural transformation.

Next we consider conditional phrases. It is trivial to define a particular type of conditional phrase such as a conditional command, but the definition of a generic conditional, applicable to arbitrary phrase types, is more challenging. Obviously, boolean must be a data type, with $B^D(\underline{\text{boolean}}) = \{\underline{\text{true}},\underline{\text{false}}\}$. Less obviously, Ω must possess all binary least upper bounds. (Note that this imposes a restriction upon Ω^D.)

Under these conditions, we can define $\underline{\text{if}} \in \Delta_3$, with the specification

$\Lambda_{\underline{\text{if}}} = \Omega$

$\Phi_{\underline{\text{if}}} \in \Omega \to \Omega^3$ is the function such that

$$\Phi_{\underline{\text{if}}}(\omega) = \underline{\text{if}}\ \omega = \underline{\text{ns}}\ \underline{\text{then}}\ \langle \underline{\text{ns}},\underline{\text{ns}},\underline{\text{ns}}\rangle\ \underline{\text{else}}\ \langle \underline{\text{boolean exp}},\omega,\omega\rangle$$

$\bar{\Gamma}_{\underline{\text{if}}} = I_\Omega$.

Then the left adjoint of $\Phi_{\underline{\text{if}}}$ is the function $\Psi_{\underline{\text{if}}} \in \Omega^3 \to \Omega$ such that

$$\Psi_{\underline{\text{if}}}(\omega_1,\omega_2,\omega_3) = \underline{\text{if}}\ \omega_1 \leq \underline{\text{boolean exp}}\ \underline{\text{then}}\ \omega_2 \sqcup \omega_3\ \underline{\text{else}}\ \underline{\text{ns}} .$$

(From the proposition in the previous section, it can be shown that if there are ω_2, ω_3 in Ω which do not possess a least upper bound then Φ has no left adjoint.)

To determine the interpretation of if, we must give a natural transformation $\bar{\gamma}_{\underline{\text{if}}}$ from $\Phi;B^3;\times^{(3)}$ to $\bar{\Gamma};B = B$. When $\omega = \underline{\text{ns}}$, $\bar{\gamma}_{\underline{\text{if}}}(\omega)$ is the unique function from $B(\underline{\text{ns}}) \times B(\underline{\text{ns}}) \times B(\underline{\text{ns}})$ to $B(\underline{\text{ns}})$. Otherwise it is the function from $B(\underline{\text{boolean exp}}) \times B(\omega) \times B(\omega)$ to $B(\omega)$ such that

$$\bar{\gamma}_{\underline{\text{if}}}(\omega)(v,f,g) = D_\omega(v;[\lambda b \in \{\underline{\text{true}},\underline{\text{false}}\}.\ \underline{\text{if}}\ b\ \underline{\text{then}}\ f\ \underline{\text{else}}\ g]_\omega) ,$$

where D is the Ω-indexed family of diagonalizing functions, $D_\omega \in (S \to B(\omega)) \to B(\omega)$ such that

$$D_{\tau\ \text{exp}} = \lambda h \in S \to (S \to [B^D(\tau)]_\perp).\ \lambda\sigma \in S.\ h(\sigma)(\sigma)$$

$$D_{\text{comm}} = \lambda h \in S \to (S \to [S]_\perp).\ \lambda\sigma \in S.\ h(\sigma)(\sigma)$$

$$D_{\tau\ \text{acc}} = \lambda h \in S \to (B^D(\tau) \to (S \to [S]_\perp)).\ \lambda x \in B^D(\tau).\ \lambda\sigma \in S.\ h(\sigma)(x)(\sigma)$$

$$D_{\tau_1\tau_2\ \text{var}} = \lambda h \in S \to B(\tau_1\ \underline{\text{acc}}) \times B(\tau_2\ \underline{\text{exp}}).$$
$$\langle D_{\tau_1\ \text{acc}}(h;(\lambda(a,v).a)),\ D_{\tau_2\ \text{exp}}(h;(\lambda(a,v).v))\rangle$$

$$D_{\underline{\text{ns}}} = \lambda h \in S \to B(\underline{\text{ns}}).\ \perp_{B(\underline{\text{ns}})} .$$

(Notice that D_{comm} also occurred in the definition of assignment.) This family has the property that, for all $\omega,\omega' \in \Omega$ such that $\omega \leq \omega'$ and all $h \in S \to B(\omega)$,

$$B(\omega \leq \omega')(D_\omega(h)) = D_{\omega'}(h;B(\omega \leq \omega')) .$$

It is this property that insures that $\overline{\gamma}_{\underline{if}}$ is a natural transformation.

Finally, for completeness, we define operators for statement sequencing and a _while_ statement. Since these operators are not generic, their definition is straightforward:

$; \in \Delta_2$, $\underline{while} \in \Delta_2$

$\Lambda_; = \Lambda_{\underline{while}} = \{\underline{comm}, \underline{ns}\}$ with the same partial ordering as Ω.

$\Phi_;(\underline{comm}) = \langle \underline{comm},\underline{comm} \rangle$, $\Phi_{\underline{while}}(\underline{comm}) = \langle \underline{boolean\ exp},\underline{comm} \rangle$

$\Phi_;(\underline{ns}) = \Phi_{\underline{while}}(\underline{ns}) = \langle \underline{ns},\underline{ns} \rangle$

$\overline{\Gamma}_;(\underline{comm}) = \overline{\Gamma}_{\underline{while}}(\underline{comm}) = \underline{comm}$

$\overline{\Gamma}_;(\underline{ns}) = \overline{\Gamma}_{\underline{while}}(\underline{ns}) = \underline{ns}$

$\overline{\gamma}_;(\underline{ns}) = \overline{\gamma}_{\underline{while}}(\underline{ns})$ is the unique function from $B(\underline{ns}) \times B(\underline{ns})$ to $B(\underline{ns})$.

$\overline{\gamma}_;(\underline{comm}) = \lambda(c_1 \in S \to [S]_\perp,\ c_2 \in S \to [S]_\perp).\ c_1;[c_2]_\omega$

$\overline{\gamma}_{\underline{while}}(\underline{comm}) = \lambda(v \in S \to [\{\underline{true},\underline{false}\}]_\perp,\ c_1 \in S \to [S]_\perp).$

$$Y(\lambda c_2 \in S \to [S]_\perp.\ D_{comm}(v;[\lambda b.\ \underline{if}\ b\ \underline{then}\ (c_1;[c_2]_\Phi)\ \underline{else}\ J]_\Phi)) .$$

Here J is the identity injection from S to $[S]_\perp$ and Y is the least-fixed-point operator for the domain $S \to [S]_\perp$.

Future Directions

The approach described in this paper is still far from being able to encompass a full-blown programming language. In particular, the following areas need investigation:

(1) Binding mechanisms, i.e. declarations and procedures.

(2) Products of types, i.e. records or class elements.

(3) Sums of types, i.e. disjoint unions.

(4) Type definitions, including recursive type definitions.

(5) Syntactic control of interference.[7]

In the first three of these areas, our ideas have progressed far enough to suggest the form of the partially ordered set of phrase types. One wants a set Ω satisfying

$$\Omega = \Omega_{primitive} + \Omega_{procedure} + \Omega_{product} + \Omega_{sum} .$$

Here + denotes some kind of sum of partially ordered sets. (At present, it is not clear how this sum should treat the greatest type ns or a possible least type.) The partially ordered set $\Omega_{primitive}$ is similar to the Ω described in the previous section, and

$$\Omega_{procedure} = \{\omega_1 \rightarrow \omega_2 \mid \omega_1, \omega_2 \in \Omega\}$$

$$\Omega_{product} = \{\underline{product}(\omega_1, \ldots, \omega_n) \mid n \geq 0 \text{ and } \omega_1, \ldots, \omega_n \in \Omega\}$$

$$\Omega_{sum} = \{\underline{sum}(\omega_1, \ldots, \omega_n) \mid n \geq 0 \text{ and } \omega_1, \ldots, \omega_n\}$$

The main novelty is the partial ordering of $\Omega_{procedure}$. One wants procedure types to satisfy

$$(\omega_1 \rightarrow \omega_2) \leq (\omega_1' \rightarrow \omega_2') \text{ if and only if } \omega_1' \leq \omega_1 \text{ and } \omega_2 \leq \omega_2' ,$$

so that the type operator $\rightarrow$ is antimonotone in its first argument. For example, suppose integer exp $\leq$ real exp. Then a procedure of type real exp $\rightarrow$ boolean exp, which can accept any real expression as argument, can also accept any integer expression as argument, and should therefore be permissible in any context which permits a procedure of type integer exp $\rightarrow$ boolean exp. Thus (real exp $\rightarrow$ boolean exp) $\leq$ (integer exp $\rightarrow$ boolean exp).

It follows that $\Omega_{procedure}$ will be isomorphic to $\Omega^{op} \times \Omega$, where Ω^{op} denotes the dual of Ω. This raises the question of how one solves the recursive equation describing Ω. The simplest answer is to impose an appropriate ordering on the least set satisfying this equation. The resulting Ω, however, will not contain certain limits which will be needed to deal with recursive type definitions. One would like to use Scott's methods to treat recursive definitions, but these methods do not encompass the operation of dualizing a partial ordering.

This difficulty does not arise for products or sums, where conventional pointwise ordering seems natural. However, a richer ordering becomes attractive when named, rather than numbered, products and sums are considered. Suppose we redefine

$$\Omega_{\text{product}} = \{\underline{\text{product}}(\overline{\omega}) \mid \overline{\omega} \in N \to \Omega \text{ for some finite set } N \text{ of names}\},$$

and similarly for Ω_{sum}. Then the following ordering can be used:

$\underline{\text{product}}(\overline{\omega}) \leq \underline{\text{product}}(\overline{\omega}')$ whenever

$$\text{domain}(\overline{\omega}) \supseteq \text{domain}(\overline{\omega}') \text{ and } (\forall n \in \text{domain}(\overline{\omega}'))\ \overline{\omega}(n) \leq \overline{\omega}'(n),$$

$\underline{\text{sum}}(\overline{\omega}) \leq \underline{\text{sum}}(\overline{\omega}')$ whenever

$$\text{domain}(\overline{\omega}) \subseteq \text{domain}(\overline{\omega}') \text{ and } (\forall n \in \text{domain}(\overline{\omega}))\ \overline{\omega}(n) \leq \overline{\omega}'(n).$$

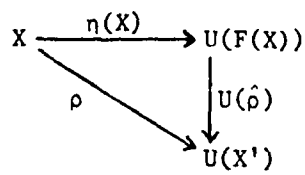
The first ordering permits implicit record conversions which forget fields. The second ordering permits implicit conversions of disjoint unions which broaden the number of alternatives in a union.

In particular, the second ordering solves a long-standing problem in the type-checking of disjoint union expressions. Suppose p is a phrase of type ω, and make-n denotes the injection into a disjoint union corresponding to the alternative named n. Using bottom-up type analysis, how does one determine the type of make-n(p)? The answer is that the type is $\underline{\text{sum}}(n{:}\omega)$, which is a subtype of any sum of the form $\underline{\text{sum}}(\ \ldots\ ,\ n{:}\omega,\ \ldots)$.

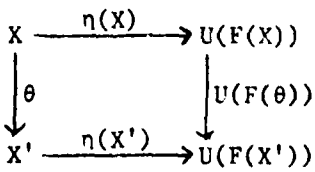
APPENDIX

In this appendix we will demonstrate the existence of free category-sorted algebras by constructing an appropriate adjunction. Our basic approach will be to connect category-sorted algebras with ordinary one-sorted algebras in order to use the known existence of free ordinary algebras. We begin by stating several general properties of adjunctions which will be used in our development.

Proposition Suppose U is a functor from K' to K, F is a function from $|K|$ to $|K'|$, and η is a $|K|$-indexed family of morphisms $\eta(X) \in X \underset{K}{\rightarrow} U(F(X))$ such that:

> For all $X \in |K|$, $X' \in |K'|$, and $\rho \in X \underset{K}{\rightarrow} U(X')$ there is exactly one morphism $\hat{\rho} \in F(X) \underset{K'}{\rightarrow} X'$ such that
>
> 
>
>
> commutes in K.

Then there is exactly one way of extending F to be a functor from K to K' such that F is the left adjoint of U with η as the associated natural transformation. Namely, for each $\theta \in X \underset{K}{\rightarrow} X'$, $F(\theta)$ must be the unique morphism such that



commutes in K.

We omit the proof (11,p. 116), the main point of which is to show that the extension of F preserves composition and identities. The utility of this proposition is that, in specifying adjunctions it is only necessary to specify the object part of the left adjoint.

Next, we consider the composition of adjunctions:

Proposition Suppose U is a functor from K' to K with left adjoint F and associated natural transformation η, and U' is a functor from K'' to K' with left adjoint F' and associated natural transformation η'. Let

$$U'' = U';U$$

$$F'' = F;F'$$

$$\eta''(X) = \eta(X);_K U(\eta'(F(X))) .$$

Then U'' is a functor from K'' to K with left adjoint F'' and associated natural transformation η''.

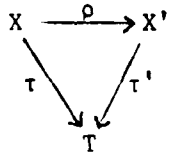
Again we omit the proof (9, p. 101).

Finally, we introduce the construction of categories over distinguished objects, and show that an adjunction between such categories can be built out of an adjunction between the categories from which they have been constructed.

Let K be a category and $T \in |K|$. Then $K{\downarrow}T$, called the category of objects over T, is the category such that

(a) $|K{\downarrow}T| = \{X, \tau \mid X \in |K| \text{ and } \tau \in X \vec{K} T\}$,

(b) $X,\tau \; {}_{K{\downarrow}T}\overset{\rightarrow}{} \; X',\tau'$ is the set of morphisms $\rho \in X \vec{K} X'$ such that



commutes in K.

(c) Composition and identities are the same as in K.

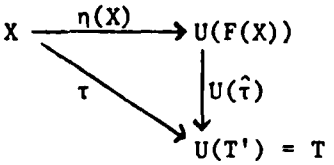
Then:

Proposition Suppose U is a functor from K' to K with left adjoint F and associated natural transformation η. Suppose $T' \in |K'|$ and $T = U(T')$. Let $\overline{U}$ be the functor from $K'\downarrow T'$ to $K\downarrow T$ such that $\overline{U}(X',\tau') = U(X'),U(\tau')$ and $\overline{U}(\rho) = U(\rho)$. Then $\overline{U}$ has a left adjoint $\overline{F}$ and an associated natural transformation $\overline{\eta}$ such that

$$\overline{F}(X,\tau) = F(X),\hat{\tau}$$

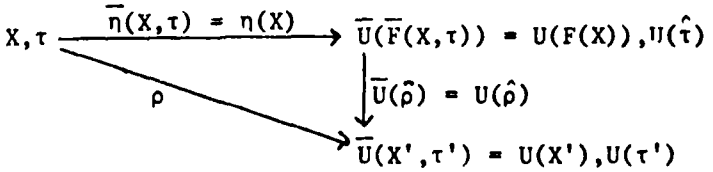
$$\overline{\eta}(X,\tau) = \eta(X) ,$$

where $\hat{\tau} \in F(X) \underset{K'}{\rightarrow} T'$ is the unique morphism such that



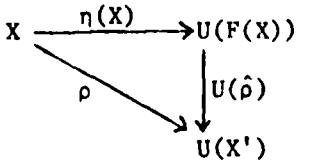
commutes in K.

Proof: We leave it to the reader to verify that $\overline{U}$ is a functor from $K'\downarrow T'$ to $K\downarrow T$, that $\overline{F}$ is (the object part of) a functor from $K\downarrow T$ to $K'\downarrow T'$, and that $\overline{\eta}(X,\tau) \in X,\tau \underset{K\downarrow T}{\rightarrow} \overline{U}(\overline{F}(X,\tau))$. To show the adjunction property, suppose $X,\tau \in |K\downarrow T|$, $X',\tau' \in |K'\downarrow T'|$, and $\rho \in X,\tau \underset{K\downarrow T}{\rightarrow} \overline{U}(X',\tau')$. Then we must show that there is exactly one $\hat{\rho} \in \overline{F}(X,\tau) \underset{K'\downarrow T'}{\rightarrow} X',\tau'$ such that



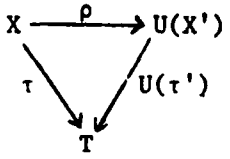
commutes in $K\downarrow T$.

Since composition is the same in $K\downarrow T$ as in K, $\hat{\rho}$ can only be the unique morphism in $F(X) \underset{K'}{\rightarrow} X'$ such that

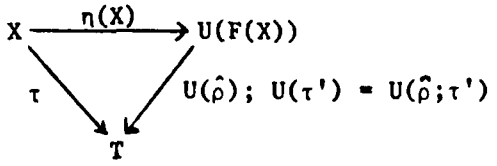


commutes in K.

However, we must show that $\hat{\rho}$ actually belongs to the more restricted set of morphisms $\overline{F}(X,\tau)\ {}_{K'\downarrow\overrightarrow{T}}\ X',\tau'$. To establish this, we note that $\rho \in X,\tau\ {}_{K\downarrow\overrightarrow{T}}\ \overline{U}(X',\tau') = X,\tau\ {}_{K\downarrow\overrightarrow{T}}\ U(X'),U(\tau')$ implies that



commutes in K, which in conjunction with the previous diagram implies that



commutes in K. Then the uniqueness of $\hat{\tau}$ gives $\hat{\rho};\tau' = \hat{\tau}$, so that $\hat{\rho} \in F(X),\hat{\tau}\ {}_{K'\downarrow\overrightarrow{T}}\ X',\tau' = \overline{F}(X,\tau)\ {}_{K'\downarrow\overrightarrow{T}}\ X',\tau'$.

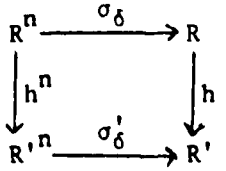
Now we can apply these general results to the specific case of interest. Let $\Omega\Delta\Gamma$ be a fixed but arbitrary category-sorted signature, let CALG (called $ALG_{\Omega\Delta\Gamma}$ in the main text) be the category of $\Omega\Delta\Gamma$-algebras and their homomorphisms, and let ALG be the category of Δ-algebras and their homomorphisms:

(1) A Δ-algebra consists of:

(1a) A <u>carrier</u> R, which is a set.

(1b) For each $n \geq 0$ and $\delta \in \Delta_n$, an <u>interpretation</u> $\sigma_\delta \in R^n \to R$.

(2) If R,σ and R',σ' are Δ-algebras, then a <u>homomorphism</u> from R,σ to R',σ' is a function $h \in R \to R'$ such that, for all $n \geq 0$ and $\delta \in \Delta_n$, the diagram



of functions commutes.

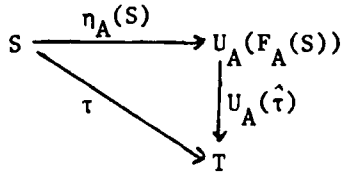
The known existence of ordinary free algebras can be stated in the language of adjunctions by:

> Let U_A be the functor from ALG to SET which maps algebras into their carriers and homomorphisms into themselves. Then U_A possesses a left adjoint F_A with an associated natural transformation η_A.

Here $F_A(S)$ is the free Δ-algebra generated by S, and $\eta_A(S)$ is the embedding of S into the carrier of $F_A(S)$.

Of particular importance is the Δ-algebra, which we will call T, in which the carrier members are sorts and the interpretation of each operator is its category-sorted specification. More precisely, T is the Δ-algebra $|\Omega|, \Gamma_{ob}$, where each $\Gamma_{ob,\delta}$ is the object part of the functor Γ_δ.

We now introduce the categories ALG$\downarrow$T and SET$\downarrow|\Omega|$. An object of ALG$\downarrow$T can be thought of as a Δ-algebra equipped with an assignment of sorts to the members of its carrier. Similarly, an object of SET$\downarrow|\Omega|$ can be thought of as a set equipped with an assignment of sorts to its members. Since $|\Omega| = U_A(T)$, our last general proposition gives:

> Let U_T be the functor from ALG$\downarrow$T to SET$\downarrow|\Omega|$ such that $U_T(\langle R,\sigma\rangle,\tau) = U_A(R,\sigma), U_A(\tau) = R,\tau$, and $U_T(h) = U_A(h) = h$. Then U_T has a left adjoint F_T and an associated natural transformation η_T such that
>
> $$F_T(S,\tau) = F_A(S), \hat{\tau}$$
> $$\eta_T(S,\tau) = \eta_A(S) ,$$
>
> where $\hat{\tau} \in F_A(S) \xrightarrow[ALG]{} T$ is the unique morphism such that
>
> 
>
>
> commutes in SET.

Informally, a type assignment to a set can be extended to the free Δ-algebra generated by that set by using the specification Γ to interpret the operators in Δ.

Our final (and most complicated) task is to construct an adjunction from ALG↓T to CALG. Let U_C be a functor from CALG to ALG↓T whose action on objects is given by:

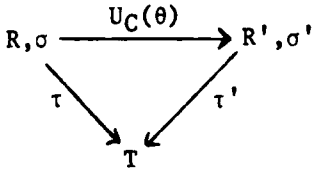
$$
\begin{aligned}
&U_C(B',\gamma') = \langle R',\sigma'\rangle,\tau' \quad \text{where} \\
&\quad R' = \{\omega,x' \mid \omega \in |\Omega| \text{ and } x' \in B'(\omega)\} , \\
&\quad \sigma'_\delta \in R'^{n} \to R' \text{ is the function such that} \\
&\qquad \sigma'_\delta(\langle\omega_1,x'_1\rangle, \ldots , \langle\omega_n,x'_n\rangle) = \\
&\qquad\quad \Gamma_\delta(\omega_1, \ldots , \omega_n), \gamma'_\delta(\omega_1, \ldots , \omega_n)(x'_1, \ldots , x'_n) , \\
&\quad \tau' \in R' \to |\Omega| \text{ is the function such that } \tau'(\omega,x') = \omega .
\end{aligned}
\tag{1}
$$

(The variables in this definition have been primed to facilitate its application to later developments.) The reader may verify that τ' is an homomorphism from R',σ' to T, so that $\langle R',\sigma'\rangle,\tau'$ is an object of ALG↓T. Intuitively, the action of U_C on objects is to forget the morphism part of B' (i.e., the implicit conversion functions) and to collapse the object part of B' into a disjoint union R' of its components, with a type assignment τ' which remembers which component of B' was the source of each member of R'.

To specify the action of U_C on morphisms, suppose $\theta \in B,\gamma \xrightarrow[CALG]{} B',\gamma'$, and let $\langle R,\sigma\rangle,\tau = U_C(B,\gamma)$ and $\langle R',\sigma'\rangle,\tau' = U_C(B',\gamma')$. Then

$$
\begin{aligned}
&U_C(\theta) \in R \to R' \text{ is the function such that} \\
&\quad U_C(\theta)(\omega,x) = \omega,\theta(\omega)(x) .
\end{aligned}
$$

The reader may verify that $U_C(\theta)$ is an homomorphism from R,σ to R',σ' (which depends upon the fact that θ is an homomorphism from B,γ to B',γ'), that



commutes in ALG, so that $U_C(\theta) \in \langle R,\sigma\rangle,\tau \xrightarrow[ALG↓T]{} \langle R',\sigma'\rangle,\tau'$, and that U_C preserves composition and identities.

Next, let F_C be the functor from ALG↓T to CALG such that

$F_C(\langle R,\sigma\rangle,\tau) = B,\gamma$ where

$B(\omega) = \{r,\iota \mid r \in R \text{ and } \iota \in \tau(r) \overset{\rightarrow}{\Omega} \omega\}$,

$B(\rho \in \omega \overset{\rightarrow}{\Omega} \omega') \in B(\omega) \to B(\omega')$ is the function such that

$B(\rho)(r,\iota) = r,(\iota;_{\Omega}\rho)$,

$\gamma_\delta(\omega_1, \ldots, \omega_n) \in B(\omega_1) \times \ldots \times B(\omega_n) \to B(\Gamma_\delta(\omega_1, \ldots, \omega_n))$ (2)

is the function such that

$$\gamma_\delta(\omega_1, \ldots, \omega_n)(\langle r_1,\iota_1\rangle, \ldots, \langle r_n,\iota_n\rangle) = \sigma_\delta(r_1, \ldots, r_n),\Gamma_\delta(\iota_1, \ldots, \iota_n) .$$

To see that $\gamma_\delta(\omega_1, \ldots, \omega_n)$ is a function of the correct type, suppose that, for $1 \leq i \leq n$, $\langle r_i,\iota_i\rangle \in B(\omega_i)$. Then each $\iota_i \in \tau(r_i) \overset{\rightarrow}{\Omega} \omega$. Thus $\Gamma_\delta(\iota_1, \ldots, \iota_n) \in \Gamma_\delta(\tau(r_1), \ldots, \tau(r_n)) \overset{\rightarrow}{\Omega} \Gamma_\delta(\omega_1, \ldots, \omega_n)$. But since τ is an homomorphism from R,σ to $T = |\Omega|,\Gamma_{ob}$, this set is also $\tau(\sigma_\delta(r_1, \ldots, r_n)) \overset{\rightarrow}{\Omega} \Gamma_\delta(\omega_1, \ldots, \omega_n))$. Thus $\langle\sigma_\delta(r_1, \ldots, r_n), \Gamma_\delta(\iota_1, \ldots, \iota_n)\rangle \in B(\Gamma_\delta(\omega_1, \ldots, \omega_n))$. The reader may also verify that B is a functor from Ω to SET and γ_δ is a natural transformation from $B^n;\times^{(n)}$ to $\Gamma;B$.

Intuitively, one can think of τ as assigning a "minimal" type to each member of R, and of a member of $B(\omega)$ as a member of R paired with an implicit conversion from its minimal type to ω.

For any object $\langle R,\sigma\rangle,\tau$ of ALG↓T,

$U_C(F_C(\langle R,\sigma\rangle,\tau)) = \langle\overline{R},\overline{\sigma}\rangle,\overline{\tau}$ where

$\overline{R} = \{\omega,\langle r,\iota\rangle \mid \omega \in |\Omega| \text{ and } r \in R \text{ and } \iota \in \tau(r) \overset{\rightarrow}{\Omega} \omega\}$,

$\overline{\sigma}_\delta \in \overline{R}^n \to \overline{R}$ is the function such that

$$\overline{\sigma}_\delta(\langle\omega_1,\langle r_1,\iota_1\rangle\rangle, \ldots, \langle\omega_n,\langle r_n,\iota_n\rangle\rangle) = \Gamma_\delta(\omega_1, \ldots, \omega_n), \langle\sigma_\delta(r_1, \ldots, r_n),\Gamma_\delta(\iota_1, \ldots, \iota_n)\rangle ,$$

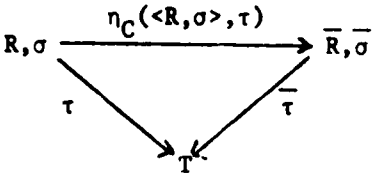
$\overline{\tau} \in \overline{R} \to |\Omega|$ is the function such that $\overline{\tau}(\omega,\langle r,\iota\rangle) = \omega$.

Let

$\eta_C(\langle R,\sigma\rangle,\tau) \in R \to \overline{R}$ be the function such that

$$\eta_C(\langle R,\sigma\rangle,\tau)(r) = \tau(r),\langle r, I^{\Omega}_{\tau(r)}\rangle .$$

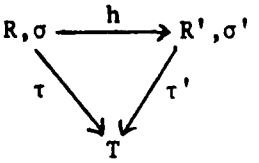
The reader may verify that $\eta_C(\langle R,\sigma\rangle,\tau)$ is an homomorphism from R,σ to $\overline{R},\overline{\sigma}$ (which depends upon the fact that τ is an homomorphism from R,σ to $T = |\Omega|,\Gamma_{ob}$), and that



commutes in ALG. Thus $\eta_C(\langle R,\sigma\rangle,\tau) \in \langle R,\sigma\rangle,\tau \underset{ALG\downarrow T}{\rightarrow} \langle \overline{R},\overline{\sigma}\rangle,\overline{\tau} = \langle R,\sigma\rangle,\tau \underset{ALG\downarrow T}{\rightarrow} U_C(F_C(\langle R,\sigma\rangle,\tau))$.

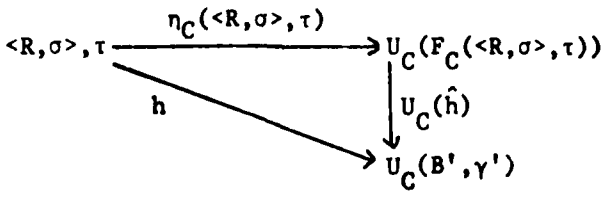
Now we will show that F_C is a left adjoint of U_C, with associated natural transformation η_C. Let $\langle R,\sigma\rangle,\tau$ be an object of ALG↓T, let B',γ' be an object of CALG, and let h be a morphism in ALG↓T from $\langle R,\sigma\rangle,\tau$ to $U_C(B',\gamma')$, where $U_C(B',\gamma') = \langle R',\sigma'\rangle,\tau'$ is described by (1).

Since h is a function from R to R', the definition of R' implies that h(r) will be a pair ω,x', where $x' \in B'(\omega)$. Moreover, since h is a morphism in ALG↓T,



must commute in ALG, so that $\tau(r) = \tau'(h(r)) = \tau'(\omega,x') = \omega$. Thus $[h(r)]_1 = \tau(r)$ and $[h(r)]_2 \in B'(\tau(r))$.

Now suppose $\hat{h}$ is any morphism in $F_C(\langle R,\sigma\rangle,\tau) \underset{CALG}{\rightarrow} B',\gamma'$, where $F_C(\langle R,\sigma\rangle,\tau) = B,\gamma$ is described by (2), and consider the diagram



(D)

in ALG↓T.

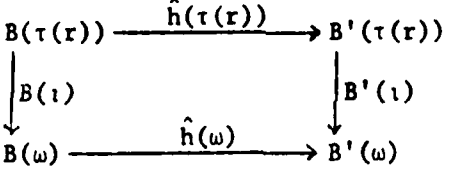
From the definitions of η_C and of the action of U_C on morphisms, we have

$$U_C(\hat{h})(\eta_C(\langle R,\sigma\rangle,\tau)(r)) = U_C(\hat{h})(\tau(r),\langle r,I^{\Omega}_{\tau(r)}\rangle) = \tau(r),\hat{h}(\tau(r))(r,I^{\Omega}_{\tau(r)}) .$$

Thus the diagram (D) will commute if and only if, for all $r \in R$,

$$\hat{h}(\tau(r))(r,I^{\Omega}_{\tau(r)}) = [h(r)]_2 .$$

Moreover, since $\hat{h}$ is a category-sorted homomorphism from B,γ to B',γ', it is a natural transformation from B to B'. Thus for all $r \in R$, $\omega \in |\Omega|$, and $\iota \in \tau(r) \overset{\rightarrow}{\Omega} \omega$,



commutes in SET. In conjunction with the action of B on morphisms, this gives

$$\hat{h}(\omega)(\langle r,\iota\rangle) = \hat{h}(\omega)(B(\iota)(r,I^{\Omega}_{\tau(r)})) = B'(\iota)(\hat{h}(\tau(r))(r,I^{\Omega}_{\tau(r)})) .$$

Thus diagram (D) will commute if and only if

$$\hat{h}(\omega)(\langle r,\iota\rangle) = B'(\iota)([h(r)]_2)$$

holds for all $r \in R$, $\omega \in |\Omega|$, and $\iota \in \tau(r) \overset{\rightarrow}{\Omega} \omega$.

Since this equation completely determines $\hat{h}$, the adjunction property will hold if the resulting $\hat{h}$ is actually a category-sorted homomorphism from B,γ to B',γ'. We leave it to the reader to verify that $\hat{h}(\omega) \in B(\omega) \to B'(\omega)$, and that, because of the action of B on morphisms, $\hat{h}$ is a natural transformation from B to B'. The one nontrivial property to be shown is that $\hat{h}$ satisfies the homomorphic relationship with the interpretations γ and γ', i.e., that for all $n \geq 0$, $\delta \in \Delta_n$, and $\omega_1, \ldots, \omega_n \in |\Omega|$,

$$\begin{array}{ccc} B(\omega_1) \times \ldots \times B(\omega_n) & \xrightarrow{\gamma_\delta(\omega_1, \ldots, \omega_n)} & B(\Gamma_\delta(\omega_1, \ldots, \omega_n)) \\ \downarrow \hat{h}(\omega_1) \times \ldots \times \hat{h}(\omega_n) & & \downarrow \hat{h}(\Gamma_\delta(\omega_1, \ldots, \omega_n)) \\ B'(\omega_1) \times \ldots \times B'(\omega_n) & \xrightarrow{\gamma'_\delta(\omega_1, \ldots, \omega_n)} & B'(\Gamma_\delta(\omega_1, \ldots, \omega_n)) \end{array}$$

commutes in SET.

To see this, suppose $\langle r_1,\iota_1\rangle, \ldots, \langle r_n,\iota_n\rangle \in B(\omega_1) \times \ldots \times B(\omega_n)$. Then

$$\hat{h}(\Gamma_\delta(\omega_1, \ldots, \omega_n))(\gamma_\delta(\omega_1, \ldots, \omega_n)(\langle r_1,\iota_1\rangle, \ldots, \langle r_n,\iota_n\rangle))$$

$$= \hat{h}(\Gamma_\delta(\omega_1, \ldots, \omega_n))(\sigma_\delta(r_1, \ldots, r_n), \Gamma_\delta(\iota_1, \ldots, \iota_n))$$

$$= B'(\Gamma_\delta(\iota_1, \ldots, \iota_n))([h(\sigma_\delta(r_1, \ldots, r_n))]_2)$$

$$= B'(\Gamma_\delta(\iota_1, \ldots, \iota_n))([\sigma'_\delta(h(r_1), \ldots, h(r_n))]_2)$$

since h is an homomorphism from R,σ to R',σ'

$$= B'(\Gamma_\delta(\iota_1, \ldots, \iota_n))(\gamma'_\delta(\tau(r_1), \ldots, \tau(r_n))([h(r_1)]_2, \ldots, [h(r_n)]_2))$$

by the definition of σ'_δ given in (1)

$$= \gamma'_\delta(\omega_1, \ldots, \omega_n)(B'(\iota_1)([h(r_1)]_2), \ldots, B'(\iota_n)([h(r_n)]_2))$$

since γ'_δ is a natural transformation from $B'^n;\times^{(n)}$ to $\Gamma_\delta;B'$

$$= \gamma'_\delta(\omega_1, \ldots, \omega_n)(\hat{h}(\omega_1)(r_1,\iota_1), \ldots, \hat{h}(\omega_n)(r_n,\iota_n))$$

In summary, we have constructed the adjunctions

$$\text{SET}\downarrow|\Omega| \underset{U_T}{\overset{F_T}{\rightleftarrows}} \text{ALG}\downarrow T \underset{U_C}{\overset{F_C}{\rightleftarrows}} \text{CALG}$$

with associated natural transformations η_T and η_C. The adjunction used in the main text is the composition of these adjunctions:

$$U = U_C;U_T$$

$$F = F_T;F_C$$

$$\eta(S,\tau_S) = \eta_T(S,\tau_S);_{\text{SET}\downarrow|\Omega|}U_T(\eta_C(F_T(S,\tau_S))) .$$

The free $\Omega\Delta\Gamma$-algebra $F(S,\tau_S)$ generated by S,τ_S is given explicitly by (2), where R,σ is the free Δ-algebra generated by S and $\tau \in R \to |\Omega|$ is the unique homomorphism such that $\eta_A(S);\tau = \tau_S$.

In the special case where Ω is a preordered set, there is at most one $\iota \in \tau(r) \underset{\Omega}{\downarrow} \omega$, so that (2) is isomorphic to the much simpler definition:

$$B(\omega) = \{r \mid r \in R \text{ and } \tau(r) \leq \omega\}$$

$B(\omega \leq \omega')$ is the identity inclusion from $B(\omega)$ to $B(\omega')$,

$$\gamma_\delta(\omega_1, \ldots, \omega_n)(r_1, \ldots, r_n) = \sigma_\delta(r_1, \ldots, r_n).$$

In this case, $B(\omega)$ is simply the subset of the terms of the ordinary free Δ-algebra whose minimal sort is a subsort of ω, the implicit conversion functions are all identity inclusions, and the operators are interpreted the same way as in the ordinary free algebra.

REFERENCES

1. Goguen, J. A., "Order Sorted Algebras: Exceptions and Error Sorts, Coercions and Overloaded Operators", Semantics and Theory of Computation Report #14, Computer Science Department, U.C.L.A., (December 1978). To appear in Journal of Computer and Systems Science.

2. Goguen, J. A., Thatcher, J. W., Wagner, E. G., and Wright, J. B., "Initial Algebra Semantics and Continuous Algebras", Journal ACM 24 (1) pp. 68-95 (January 1977).

3. Burstall, R. M., and Landin, P. J., "Programs and Their Proofs: An Algebraic Approach", in Machine Intelligence 4, B. Meltzer and D. Michie, Eds., Edinburgh University Press, pp. 17-43 (1969).

4. Birkhoff, G., and Lipson, J. D., "Heterogeneous Algebras", Journal of Combinatorial Theory 8, pp. 115-133 (1970).

5. Higgins, P. J., "Algebras with a Schema of Operators", Math. Nachr. 27, pp. 115-132 (1963).

6. Morris, J. ".", "Types are not Sets", Proc. ACM Symposium on Principles of Programming Languages, pp. 120-124, Boston (1973).

7. Reynolds, J. C., "Syntactic Control of Interference", Proc. Fifth ACM Symposium on Principles of Programming Languages, pp. 39-46, Tucson (1978).

8. Reynolds, J. C., The Craft of Programming, in preparation.

9. MacLane, S., Categories for the Working Mathematician, Springer-Verlag, New York (1971).

10. Reynolds, J. C., "GEDANKEN - A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept", Comm. ACM 13 (5), pp. 308-319 (May 1970).

11. Arbib, M. A., and Manes, E. G., Arrows, Structures, and Functors - The Categorical Imperative, Academic Press, New York (1975).

APPENDIX C: SPECIFICATION LOGIC

This presentation of specification logic is based upon a subset of Algol W that has been augmented by refining its type structure and introducing lambda expressions, as in idealized Algol.

The phrases of this language are categorized by phrase types, which are described by the following grammar:

<data type> ::= integer | real | logical

<phrase type> ::= <data type> variable | <data type> expression
| <data type> array variable (<dimension list>)
| <data type> array expression (<dimension list>)
| statement | assertion
| procedure (<phrase type list>)
| <data type> procedure (<phrase type list>)

<phrase type list> ::= <phrase type>
| <phrase type list> , <phrase type>

<dimension list> ::= * | <dimension list> , *

The symbols exp and var are often used to abbreviate expression and variable.

Let M_θ be the set of meanings appropriate to the phrase type θ. In particular, let

$$M_{\tau\ \text{expression}} = S \to V_\tau$$
$$M_{\text{assertion}} = S \to \{\text{true}, \text{false}\}$$
$$M_{\text{statement}} = S \to (S^* \cup S^\omega) ,$$

where S is the set of states (mappings of variables into values) and V_τ is the set of values appropriate to the data type τ.

Here the form of $M_{statement}$ reflects the partially operational view that the meaning of a statement maps a state σ into the finite or infinite sequence of states that occur during execution of the statement starting with σ. The inclusion of intermediate states in this definition is necessary for the definition of noninterference specifications.

An environment is a mapping on the set of identifiers that maps each identifier of phrase type θ into a member of M_θ. We write $[\![P]\!]_\eta$ for the meaning of a phrase P in an environment η.

Then the meanings of the various forms of specifications used in specification logic can be defined as follows:

(1) If P and Q are assertions and S is a statement then $[\![\{P\}\ S\ \{Q\}]\!]_\eta$ is true if and only if, for any state σ such that $[\![P]\!]_\eta(\sigma)$ is true, the sequence $[\![S]\!]_\eta(\sigma)$ is either infinite or concludes with a final state σ_f such that $[\![Q]\!]_\eta(\sigma_f)$ is true.

(2) If P is an assertion, then $[\![\{P\}]\!]_\eta$ is true if and only if $[\![P]\!]_\eta(\sigma)$ is true for all states σ.

(3) For $n \geq 1$, if S_1, ... , S_n and S are specifications then $[\![S_1 \ \& \ \ldots \ \& \ S_n \Rightarrow S]\!]_\eta$ is true if and only if either $[\![S]\!]_\eta$ is true or some $[\![S_i]\!]_\eta$ is false.

(4) If $\underline{I}$ is an identifier and S is a specification such that the free occurrences of $\underline{I}$ in S have phrase type θ, then $[\![(\forall\, \theta\ \underline{I})\ S]\!]_\eta$ is true if and only if, for all meanings m appropriate to θ, $[\![S]\!]_{[\eta \mid I:\, m]}$ is true.

(5a) If S is a statement and E is a τ expression or assertion then $[\![S \# E]\!]_\eta$ is true if and only if, for all states σ and σ' such that σ' occurs in the sequence $[\![S]\!]_\eta(\sigma)$, $[\![E]\!]_\eta(\sigma') = [\![E]\!]_\eta(\sigma)$.

(5b) If V is a τ variable, E is a τ' expression or assertion, and I is an identifier not occurring free in V or E then, for all environments, V # E has the same meaning as
$(\forall \tau \text{ exp } I)\ (V := I) \# E$.

(5c) If X is an n-dimensional τ array variable, E is a τ' expression or assertion, and $I_1, \ldots, I_n$ are distinct identifiers not occurring free in X or E then, for all environments, X # E has the same meaning as

$$(\forall \text{integer exp } I_1) \ldots (\forall \text{integer exp } I_n)\ X(I_1, \ldots, I_n) \# E .$$

(5d) If H is a procedure$(\theta_1, \ldots, \theta_n)$, E is a τ expression or assertion, $I_1, \ldots, I_n$ are distinct identifiers that do not occur free in H or E, and $\theta_{i_1}, \ldots, \theta_{i_k}$ are the statement-like members of $\{\theta_1, \ldots, \theta_n\}$ then, for all environments, H # E has the same meaning as

$$(\forall \theta_1\ I_1) \ldots (\forall \theta_n\ I_n)$$
$$(I_{i_1} \# E\ \&\ \ldots\ \&\ I_{i_k} \# E \Rightarrow H(I_1, \ldots, I_n) \# E) .$$

(5e) If S is a statement-like phrase, Y is an n-dimensional τ array expression, and $I_1, \ldots, I_n$ are distinct identifiers not occurring free in S or Y then, for all environments, S # Y has the same meaning as

$$(\forall \underline{\text{integer exp}}\ I_1) \ldots (\forall \underline{\text{integer exp}}\ I_n)$$
$$(S \# I_1 \ \& \ \ldots \ \& \ S \# I_n \Rightarrow S \# Y(I_1, \ldots, I_n)) .$$

(5f) If S is a statement-like phrase, F is a τ procedure$(\theta_1, \ldots, \theta_n)$ or an assertion procedure$(\theta_1, \ldots, \theta_n)$, $I_1, \ldots, I_n$ are distinct identifiers not occurring free in S or F, and $\theta_{i_1}, \ldots, \theta_{i_k}$ are the expression-like members of $\{\theta_1, \ldots, \theta_n\}$ then, for all environments, S # F has the same meaning as

$$(\forall \theta_1\ I_1) \ldots (\forall \theta_n\ I_n)$$
$$(S \# I_{i_1} \ \& \ \ldots \ \& \ S \# I_{i_k} \Rightarrow S \# F(I_1, \ldots, I_n)) .$$

(6) If V is a τ variable, and E and Π are distinct identifiers that do not occur free in V then, for all environments, gv(V) has the same meaning as

$$(\forall \tau\ \underline{\text{exp}}\ E)\ (\forall \underline{\text{assertion procedure}}(\tau\ \underline{\text{exp}})\ \Pi)$$
$$(V \# \Pi \Rightarrow \{\Pi(E)\}\ V := E\ \{\Pi(V)\}) .$$

Specification logic is a system for inferring universal specifications, which are specifications that are true in all environments. It includes both axioms, which are particular universal specifications, and rules of inference. Inferences may also be made by alpha conversion and (forward or backward) beta reduction, as in the lambda calculus.

An inference rule consists of zero or more premises and a conclusion. An instance of the rule is obtained by replacing metavariables, denoted by upper case letters, by appropriate phrases, subject to restrictions that may preface the rule. If all of the premises of an instance are universal, then the conclusion of the instance is universal.

In the form S_1 & ... & $S_n \Rightarrow S$, the specifications on the left, called assumptions, are regard as a finite set. The metavariable Σ is used for such a set, while S is used for a single specification. Σ & Σ' abbreviates $\Sigma \cup \Sigma'$, while Σ & S abbreviates $\Sigma \cup \{S\}$. When Σ is empty, $\Sigma \Rightarrow S$ stands for S.

Phrase types are classified as statement-like and/or expression-like as follows:

Phrase Type	Statement-like	Expression-like
τ variable	X	X
τ expression		X
τ array variable(*, ... , *)	X	X
τ array expression(*, ... , *)		X
statement	X	
assertion		X
procedure(θ_1, ... , θ_n)	X	
τ procedure(θ_1, ... , θ_n)		X
assertion procedure(θ_1, ... , θ_n)		X

An occurrence in P of an identifier is statement-like (expression-like) if the type of every subphrase of P enclosing the occurrence is statement-like (expression-like). We write $F_{\underset{\sim}{\text{sta-like}}}(P)$ ($F_{\underset{\sim}{\text{exp-like}}}(P)$) for the set of identifiers having statement-like (expression-like) free occurrences in P.

The following rules of inference and axioms have been developed:

(1) **Self-Implication**

$$\frac{}{S \Rightarrow S}$$

(2) Adding Assumptions

$$\frac{\Sigma \Rightarrow S}{\Sigma \ \& \ \Sigma' \Rightarrow S}$$

(3) Separating Assumptions

$$\frac{\Sigma \ \& \ \Sigma' \Rightarrow S}{\Sigma \Rightarrow (\Sigma' \Rightarrow S)}$$

(4) Combining Assumptions

$$\frac{\Sigma \Rightarrow (\Sigma' \Rightarrow S)}{\Sigma \ \& \ \Sigma' \Rightarrow S}$$

(5) Modus Ponens

$$\frac{\begin{array}{l}\Sigma_1 \Rightarrow S_1 \\ \quad \vdots \\ \Sigma_n \Rightarrow S_n \\ \Sigma \ \& \ S_1 \ \& \ \ldots \ \& \ S_n \Rightarrow S\end{array}}{\Sigma \ \& \ \Sigma_1 \ \& \ \ldots \ \& \ \Sigma_n \Rightarrow S}$$

(6) Quantifier Introduction

If $\underline{I}$ is an identifier of phrase type θ that does not occur free in Σ then

$$\frac{\Sigma \Rightarrow S}{\Sigma \Rightarrow (\forall \theta\ I)\ S}\ .$$

(7) Quantifier Removal

If $I_1, \ldots, I_n$ are distinct identifiers of phrase types $\theta_1, \ldots \theta_n$, and $A_1, \ldots, A_n$ are phrases of types $\theta_1, \ldots, \theta_n$ then

$$\frac{}{(\forall \theta_1\ I_1) \ldots (\forall \theta_n\ I_n)\ S \Rightarrow S|_{I_1, \ldots, I_n \rightarrow A_1, \ldots, A_n}}\ .$$

(8) Free Substitution

If $S|_{I_1, \ldots, I_n \rightarrow A_1, \ldots, A_n}$ is a type-correct substitution, then

$$\frac{S}{S|_{I_1, \ldots, I_n \rightarrow A_1, \ldots, A_n}}\ .$$

(9) Mathematical Fact Introduction

If P is an assertion that is a mathematical fact then

$$\frac{}{\{P\}}\ .$$

(10) Reductio ad Absurdum

$$\frac{}{\{\underline{\text{false}}\} \Rightarrow S}\ .$$

(11) Static Implication

If P and Q are assertions then

$$\frac{}{\{P\}\ \&\ \{P\ \underline{\text{implies}}\ Q\} \Rightarrow \{Q\}}\ .$$

(12) Statement Compounding (Axiom)

$\{p\}\ s_1\ \{q\}\ \&\ \{q\}\ s_2\ \{r\} \Rightarrow \{p\}\ s_1;\ s_2\ \{r\}$

(13) Strengthening Precedent (Axiom)

$\{p \text{ implies } q\}\ \&\ \{q\}\ s\ \{r\} \Rightarrow \{p\}\ s\ \{r\}$

(14) Weakening Consequent (Axiom)

$\{p\}\ s\ \{q\}\ \&\ \{q \text{ implies } r\} \Rightarrow \{p\}\ s\ \{r\}$

(15) while statement (Axiom)

$\{i \text{ and } \ell\}\ s\ \{i\} \Rightarrow \{i\} \text{ while } \ell \text{ do } s\ \{i \text{ and } \neg\ell\}$

(16) Two-way Conditional Statement (Axiom)

$\{p \text{ and } \ell\}\ s_1\ \{q\}\ \&\ \{p \text{ and } \neg\ell\}\ s_2\ \{q\}$

$\Rightarrow \{p\} \text{ if } \ell \text{ then } s_1 \text{ else } s_2\ \{q\}$

(17) One-way Conditional Statement (Axiom)

$\{p \text{ and } \ell\}\ s\ \{q\}\ \&\ \{(p \text{ and } \neg\ell) \text{ implies } q\}$

$\Rightarrow \{p\} \text{ if } \ell \text{ then } s\ \{q\}$

(18) Empty Statement (Axiom)

$\{p\}\ \{p\}$

(19) Specification Conjunction (Axiom)

$\{p_1\}\ s\ \{q_1\}\ \&\ \{p_2\}\ s\ \{q_2\} \Rightarrow \{p_1 \text{ and } p_2\}\ s\ \{q_1 \text{ and } q_2\}$

(20) Specification Disjunction (Axiom)

$\{p_1\}\ s\ \{q_1\}\ \&\ \{p_2\}\ s\ \{q_2\} \Rightarrow \{p_1 \text{ or } p_2\}\ s\ \{q_1 \text{ or } q_2\}$

(21) Left-Side Noninterference Decomposition

If S is a statement-like phrase, E is an expression-like phrase, and $F_{\text{sta-like}}(S) = \{I_1, \ldots, I_n\}$, then

$$\frac{}{I_1 \# E\ \&\ \ldots\ \&\ I_n \# E \Rightarrow S \# E}\ .$$

(22) Right-Side Noninterference Decomposition

If S is a statement-like phrase, E is an expression-like phrase, and $F_{\text{exp-like}}(E) = \{I_1, \ldots, I_n\}$, then

$$\frac{}{S \# I_1 \;\&\; \ldots \;\&\; S \# I_n \Rightarrow S \# E} \quad .$$

(23) Constancy (Axiom)

$$s \# p \;\&\; \{q\}\ s\ \{r\} \Rightarrow \{q \text{ and } p\}\ s\ \{r \text{ and } p\} \quad .$$

(24) Simple Assignment

Let X be a τ variable identifier, E be a τ expression, and P be an assertion such that all free occurrences of X in P have type τ expression. Let $\{I_1, \ldots, I_n\} = F_{\text{exp-like}}(P) - \{X\}$. Then

$$\frac{}{gv(X) \;\&\; X \# I_1 \;\&\; \ldots\; X \# I_n \Rightarrow \{P|_{X \to E}\}\ X := E\ \{P\}} \quad .$$

(25) Simple Variable Declarations

If X is a τ variable identifier, P and Q are assertions, $E_1, \ldots, E_m$ are expression-like phrases, $S_1, \ldots, S_n$ are statement-like phrases, and X does not occur free in Σ, P, Q, $E_1, \ldots, E_m, S_1, \ldots, S_n$, then

$$\frac{\Sigma \;\&\; gv(X) \;\&\; X \# E_1 \;\&\; \ldots \;\&\; X \# E_m \;\&\; S_1 \# X \;\&\; \ldots \;\&\; S_n \# X \Rightarrow \{P\}\ B\ \{Q\}}{\Sigma \Rightarrow \{P\}\ \text{begin } \tau\ X;\ B\ \text{end}\ \{Q\}} \quad .$$

(26) Proper Procedure Declarations

Suppose

$F_1, \ldots, F_n, G_1, \ldots, G_k, H$ are distinct identifiers of phrase types $\theta_1, \ldots, \theta_n, \theta_1', \ldots, \theta_k'$, procedure$(\theta_1, \ldots, \theta_n)$,

B_{proc}, B are statements,

P_{proc}, Q_{proc}, P, Q are assertions,

Σ, Σ', Σ_{pa} are finite sets of specifications,

such that

$\Sigma' \subseteq \Sigma$,

$F_1, \ldots, F_n$ do not occur free in Σ',

$G_1, \ldots, G_k$ do not occur free in B_{proc} or Σ',

H does not occur free in P_{proc}, Q_{proc}, P, Q, Σ, Σ', or Σ_{pa}.

Let Σ_{proc} be

$$(\forall \theta_1 F_1) \ldots (\forall \theta_n F_n)(\forall \theta_1' G_1) \ldots (\forall \theta_k' G_k)$$
$$(\Sigma_{pa} \Rightarrow \{P_{proc}\}\ H(F_1, \ldots, F_n)\ \{Q_{proc}\})$$
$$\& (\forall \text{exp-like } E)\ (I_1 \# E \ \& \ \ldots \ \& \ I_m \# E \Rightarrow H \# E),$$

where $\{I_1, \ldots, I_m\} = F_{\text{sta-like}}(B_{proc}) - \{F_1, \ldots, F_n, H\}$ and E is some identifier that is distinct from $I_1, \ldots, I_m$ and H. Then

$$\frac{\Sigma' \ \& \ \Sigma_{pa} \ \& \ \Sigma_{proc} \Rightarrow \{P_{proc}\}\ B_{proc}\ \{Q_{proc}\} \qquad \Sigma \ \& \ \Sigma_{proc} \Rightarrow \{P\}\ B\ \{Q\}}{\Sigma \Rightarrow \{P\}\ \text{begin procedure } H(\theta_1 F_1; \ \ldots \ ; \theta_n F_n);\ B_{proc};\ B\ \text{end}\ \{Q\}}.$$

(27) Simple Assignment (Axiom)

$$\underline{gv}(x)\ \&\ x \# \pi \Rightarrow \{\pi(e)\}\ x := e\ \{\pi(x)\}\ .$$

(28) Good Variables (Axiom)

$$(\forall \tau\ \underline{exp}\ e)(\forall \underline{assertion}\ \underline{procedure}(\tau\ \underline{exp})\ \pi)$$
$$(x \# \pi \Rightarrow \{\pi(e)\}\ x := e\ \{\pi(x)\})$$
$$\Rightarrow \underline{gv}(x)\ .$$

(29) Nonrecursive Proper Procedure Declarations (Axiom)

$$\{p\}\ \sigma(m)\ \{q\} \Rightarrow$$
$$\{p\}$$
$$\underline{begin}\ \underline{procedure}\ h(\theta_1 f_1;\ \ldots\ ;\ \theta_n f_n);\ m(f_1, \ldots, f_n);$$
$$\sigma(h)$$
$$\underline{end}$$
$$\{q\}\ .$$

(30) Array Assignment

Let X be an identifier of type τ array variable(*), S be an integer expression, E be a τ expression, and P be an assertion such that all free occurrences of X in P have type τ array expression(*). Let $\{I_1, \ldots, I_n\} = F_{\underline{exp\text{-}like}}(P) - \{X\}$. Then

$$\frac{}{X(S) \# I_1\ \&\ \ldots\ \&\ X(S) \# I_n \Rightarrow \{P|_{X,\ [X|S|E]}\}\ X(S) := E\ \{P\}\ .}$$

(31) Good Array Designators (Axiom)

$$x(s) \# s \Rightarrow \underline{gv}(x(s))\ .$$

(32) Array Element Noninterference (Axiom)

$\{s \neq t\}$ & $x(s)$ # $t \Rightarrow x(s)$ # $x(t)$.

(33) Array Segment Noninterference (Axiom)

$\{s \notin \nu\}$ & $x(s)$ # $\nu \Rightarrow x(s)$ # $x\lceil\nu$.

(34) Array Declarations

If X is a τ array variable(*) identifier, P and Q are assertions, L and U are integer expressions, $E_1, \ldots, E_m$ are expression-like phrases, $S_1, \ldots, S_n$ are statement-like phrases, and X does not occur free in Σ, P, Q, L, U, $E_1, \ldots, E_m, S_1, \ldots, S_n$, then:

$$\Sigma \;\&\; X \# E_1 \;\&\; \ldots \;\&\; X \# E_m \;\&\; S_1 \# X \;\&\; \ldots\; S_n \# X \Rightarrow \{P \text{ and } \underline{\text{dom}}\ X = \boxed{L \;\; U}\}\ B\ \{Q\}$$

$$\Sigma \Rightarrow \{P\}\ \underline{\text{begin}}\ \tau\ \underline{\text{array}}\ X\ (L::U);\ B\ \underline{\text{end}}\ \{Q\} \quad .$$

(35) Domain Constancy (Axiom)

s # dom x .

APPENDIX D

Programming Languages
J.J. Horning Editor

Reasoning About Arrays

John C. Reynolds
Syracuse University

A variety of concepts, laws, and notations are presented which facilitate reasoning about arrays. The basic concepts include intervals and their partitions, functional restriction, images, pointwise extension of relations, ordering, single-point variation of functions, various equivalence relations for array values, and concatenation. The effectiveness of these ideas is illustrated by informal descriptions of algorithms for binary search and merging, and by a short formal proof.

Key Words and Phrases: arrays, assertions, program proving, intervals, partitions, pointwise extension, ordering, concatenation, binary search, merging

CR Categories: 4.0, 4.22, 5.21, 5.24

1. Introduction

The use of assertions to describe programs and prove their correctness [4–6] has developed to the point where the necessary assertions are often at least as lengthy and difficult to comprehend as the program which they describe. A major cause is the use of languages and proof methods—typically the first-order predicate calculus—which are taken from classical logic and are not oriented towards programming.

Perhaps the most glaring example of these difficulties is the use of arrays. One need only compare the assertions needed to describe a program such as log n exponentiation, which does not involve arrays or other compound data structures, with the assertions for a program such as binary search, which is intuitively no more complex, but uses arrays. In the first case, the assertions are clear and concise, and reasoning about them involves only the familiar laws of elementary algebra. But when arrays are introduced, the assertions become lengthy and filled with quantifiers, and their manipulation seems only tenuously connected with the programmer's intuition.

Superficially, we need a better notation for assertions about arrays. But more fundamentally, we need concepts and laws which are not only correct but also reflect our intuitive understanding of arrays, just as the concepts of addition and multiplication, and the associative, commutative, and distributive laws reflect our intuitive understanding of numbers. Once the right concepts and laws have been found, it is comparatively trivial to design a notation which facilitates their application.

This paper presents a variety of concepts, laws, and notations for reasoning about arrays—some borrowed from mathematics and others original—which we believe meet the above criteria. Their utility will be demonstrated both by informal descriptions of program behavior and by a short formal proof of program correctness.

The consideration of both informal and formal proofs reflects our belief that the relationship between the two is a critical issue in program proving. Ideally, an informal description of "why a program works" should provide enough information that an intelligent reader could produce a formal correctness proof by filling in details, without any significant invention or change of concepts.

As an illustrative programming language, we will use Algol 60 with the following changes:

(1) **while** statements.

(2) Round rather than square brackets for array subscripts (to emphasize the view that array values are functions).

(3) Integer expressions of the form **lower** X and **upper** X, denoting the minimum and maximum subscripts of a one-dimensional array X.

(4) Empty arrays, obtained by permitting array declarations in which a lower subscript bound is larger than the corresponding upper bound.

We have purposely stayed close to Algol to avoid inadvertently choosing a programming language which hid the defects of our assertion language. In particular, we have refrained from introducing our notation for assertions into the programming language itself (except for **lower** and **upper**, which were irresistibly attractive). Moving in this direction seems to lead to a very high-level language, closer to APL than to Algol, which is outside the scope of this paper.

On the other hand, even the choice of Algol has had subtle effects on the ensuing development. For example, switching to a programming language with the novel approach to arrays described in [3, Ch. 11] would necessitate minor changes to many concepts, such as abandoning the uniqueness of the array value with an empty domain.

To an even greater extent than is indicated by the explicit references, this work is built upon the ideas of C.A.R. Hoare [7–9]. Mention should also be made of distinct but related work on arrays by D.C. Cooper [2]

Work supported by National Science Foundation Grant MCS 75-22002 and Rome Air Force Development Center Contract F30602-77-C-0235.

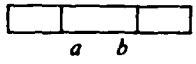
Author's address: School of Computer and Information Science, Syracuse University, 313 Link Hall, Syracuse NY 13210

and of work by R. Burstall [1] which, roughly speaking, does for lists what we are trying to do for arrays.

2. Interval and Partition Diagrams

Before considering arrays themselves, we introduce some diagrammatic expressions for making assertions about subscripts. Basically, these expressions are a formalization of the diagrams which are traditionally drawn by programmers when describing arrays.

For example, in describing the program for binary search to be developed in Section 5, one might draw



to assert a relationship between the integer variables a and b and the domain of permissible subscripts of the array X. We will regard this diagram as an assertion that the subscript domain is partitioned into three subsets: $\{i|\textbf{lower } X \le i < a\}$, $\{i|a \le i \le b\}$, and $\{i|b < i \le \textbf{upper } X\}$.

Of course, an equivalent assertion can be given in the predicate calculus, but this sacrifices the intuitive content of the diagram. (For example, the above assertion is equivalent to **lower** $X - 1 \le a - 1 \le b \le$ **upper** X or **lower** $X - 1 \ge a - 1 \ge b \ge$ **upper** X.) A better approach is to formalize and give rigorous meaning to the diagram itself. The only change we will make is to place expressions such as a and b within, rather than below, the relevant boxes. In addition to making the notation more nearly linear, this curtails the tendency of such expressions to migrate across boundaries when written hastily.

Before defining such *partition diagrams*, however, we must introduce the simpler concept of an *interval diagram*. An *interval* is a finite consecutive set of integers. If a and b are expressions denoting integers, then $a[\ \ b]$, called an *interval diagram*, is an expression denoting the interval

$$a[\ \ b] = \{i|a < i \le b\}.$$

When formulating general properties of interval diagrams (or partition diagrams) we will always use the standard form $a[\ \ b]$. But when using the diagrams to make assertions, we will permit more flexibility. Specifically, at either end of an interval diagram, $|a$ may be written instead of $a - 1|$. Also, $[a]$ may be written as an abbreviation for $[a\ \ a]$. Thus $[a\ \ b] = \{i|a \le i \le b\}$, $[a\ \ )b = \{i|a \le i < b\}$, $a(\ \ )b = \{i|a < i < b\}$, and $[a] = \{a\}$.

For any finite set S, we write $\#S$ to denote the size, or number of elements in S. Thus

$$\#a[\ \ b] = \textbf{if } b - a \ge 0 \textbf{ then } b - a \textbf{ else } 0. \qquad (2.1)$$

This use of a conditional expression to describe a fundamental property of a data structure is a clear symptom of a potential source of error, i.e. the possibility that a program may be correct for one case of the conditional but not the other. To emphasize this situation, we say that the interval $a[\ \ b]$ is *regular* when $b - a \ge 0$, or *irregular* when $b - a < 0$. It is evident that a nonempty interval is always regular, but the empty interval can be either regular or irregular. (This is a slight abuse of language; it is really the interval diagram, rather than the interval itself, which is regular or irregular.)

Partition diagrams are concatenations of interval diagrams which assert that the corresponding intervals form a partition. More precisely, if $a_0, a_1, \ldots, a_n$ are expressions denoting integers, then:

(a) $a_0[\ \ a_1|\ \ \ldots\ \ a_{n-1}|\ \ a_n]$ is called a *partition diagram*.

(b) $a_0[\ \ a_1), \ldots, a_{n-1}[\ \ a_n]$, i.e. the intervals denoted by diagrams obtained by eliminating all but an adjacent pair of lines, are called the *component intervals* of the partition diagram.

(c) $a_0[\ \ a_n]$, i.e. the interval denoted by the diagram obtained by eliminating interior lines, is called the *total interval* of the partition diagram.

(d) The partition diagram is a logical expression which is true iff the component intervals are a partition of the total interval, i.e. iff the component intervals are disjoint and their union is the total interval.

As with interval diagrams, $\ldots\ \ |a\ \ \ldots$ may be written in place of $\ldots\ \ a-1|\ \ \ldots$, and $\ldots\ \ [a]\ \ \ldots$ in place of $\ldots\ \ [a\ \ a]\ \ \ldots$. Thus for example, $[a\ \ b|\ \ c)$ is a partition diagram which is true iff the component intervals $[a\ \ )b = \{i|a \le i < b\}$, $[b] = \{b\}$, and $b[\ \ c) = \{i|b < i < c\}$ are disjoint and their union is the total interval $[a\ \ c] = \{i|a \le i \le c\}$.

The nature of partitions implies that the size of the total interval is the sum of the sizes of the component intervals.

$$a_0[\ \ a_1|\ \ \ldots\ \ a_{n-1}|\ \ a_n] \text{ implies} \qquad (2.2)$$
$$\#a_0[\ \ a_n] = \sum_{i=1}^{n} \#a_{i-1}[\ \ a_i].$$

As shown in the Appendix, (2.2) implies the following fundamental property of partition diagrams.

$$a_0[\ \ a_1|\ \ \ldots\ \ a_{n-1}|\ \ a_n] \text{ iff either} \qquad (2.3)$$
$$a_0 \le a_1 \le \ldots \le a_{n-1} \le a_n \text{ or}$$
$$a_0 \ge a_1 \ge \ldots \ge a_{n-1} \ge a_n.$$

Note that the first inequality asserts that every component interval is regular, while the second inequality asserts that every component interval is empty.

From (2.3), the following simple cases are obvious:

$$a[\ \ b] \text{ is always true.} \qquad (2.4)$$

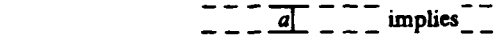
$$[a\ \ )b] \text{ iff } [a\ \ )[b] \text{ iff } a < b \text{ iff} \qquad (2.5)$$
$$[a\ \ b] \text{ is nonempty}$$

$\boxed{a \quad b \quad c}$ iff $a \le b \le c$ iff $b \in \boxed{a \quad c}$. (2.6)

By (2.4), partition diagrams without interior lines are tautologies, so that in practice such diagrams will not occur in assertions. This circumvents the problem that such diagrams can only be distinguished from interval diagrams by their context.

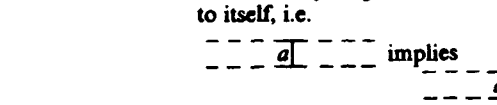
More interestingly, one can easily derive several "diagrammatically natural" rules of inference. (Here "line" refers to any vertical line in a diagram, including its associated expression.)

Erasure. From a partition diagram one can infer any diagram obtained by deleting a line, i.e.



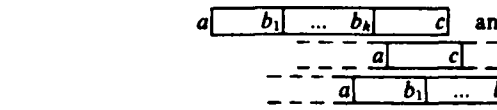
(2.7)

Adjacent Duplication. From a partition diagram one can infer any diagram obtained by replicating a line next to itself, i.e.



(2.8)

Substitution. From two partition diagrams such that the end lines of the first match some pair of adjacent lines in the second, one can infer the diagram obtained by substituting the first diagram for the adjacent lines in the second:



(2.9)

It should be emphasized that (2.4) to (2.9) are useful, but not complete rules, i.e. they cannot completely replace (2.2), (2.3), or the definition of partition diagrams. The use of these rules is illustrated by the following inferences, which will be pertinent to the binary search example to be given in Section 5:

(a) For any integers l and u, (2.4) and (2.8) show that $\boxed{l \quad l \quad u \quad u}$ holds.

(b) Suppose $\boxed{l \quad a \quad b \quad u}$ and $a \le j \le b$. Then by (2.6) and (2.9), $\boxed{l \quad a \quad j \quad b \quad u}$ holds. In turn, by (2.7), this implies $\boxed{l \quad j \quad u}$, $\boxed{l \quad j+1 \quad b \quad u}$, and $\boxed{l \quad a \quad j-1 \quad u}$.

In conclusion, it should be noted that the definitions of interval and partition diagrams have been motivated by a definite attitude towards empty and irregular intervals, and towards arrays with such intervals as their domain of subscripts. Although there are exceptions, such as finding the subscript of a maximum element, most array-manipulating algorithms can be extended without complication to handle the empty array. In the author's opinion, it is invariably good practice to do so, and the linguistic prohibition of empty arrays (as in Algol 60) is a design mistake—akin to prohibiting **for** statements which execute their bodies zero times.

However, one could permit intervals to be empty without permitting their irregular representation by regarding $a\boxed{\quad b}$ as well-defined when $a = b$, but undefined when $a > b$. Our decision to permit irregular representations has several motivations:

(1) Undefined expressions are a potential source of confusion.

(2) If **for** $i := a$ **until** b **do** s is regarded as iterating over the interval $\boxed{a \quad b}$ (as in [8]), then most Algol-based languages permit $\boxed{a \quad b}$ to be irregular in this context.

(3) The author has never encountered an array-manipulating algorithm which handles the empty array yet cannot be extended without complication to handle irregular subscript domains.

A potential counterargument is that even though an algorithm may extend smoothly to the irregular case, its proof of correctness may require extra case analysis. But in the author's experience, this case analysis can be avoided by using partition diagrams instead of inequalities - basically this avoids the **or** lurking in Proposition (2.3).

Nevertheless, a consistent case can be made for avoiding irregular intervals. F.L. Morris has explored the use of interval and partition diagrams in this context. His basic approach is to regard any occurrence of an interval diagram $a\boxed{\quad b}$ within an assertion as having the "side effect" of asserting $a \le b$. Then the partition diagram $a_0\boxed{a_1 \quad \ldots \quad a_n}$ is defined to mean $a_0 \le a_1 \le \ldots \le a_n$, which implies both that the component intervals are well-defined and that they form a partition of the total interval. In this approach, Propositions (2.2) and (2.4) to (2.9) remain true.

3. Functions as Array Values

There are two quite different concepts of an array. The more traditional view is that an array of, say, real numbers is a function from subscripts into variables, which in turn possess real values. The more recent view, expounded by Hoare [7, 9] and Dijkstra [3], is that an array of real numbers is a variable whose value is a function from subscripts into real numbers. In this paper, we take the latter view. The effect is to banish the possibility of "sharing" or "aliasing" among array elements, which would greatly complicate the problems of proving program correctness.

Specifically, we assume that an array declared by τ **array** $X(a{:}b)$ is a variable whose values range over the set of functions from the interval $\boxed{a \quad b}$ into the set τ.

We write $\langle\ \rangle$ to denote the unique function whose domain is the empty set $\{\ \}$. For any function X, we write **dom** X for the domain of X, and when this domain is an interval, **lower** X and **upper** X for the integers such that **dom** $X = \boxed{\textbf{lower } X \quad \textbf{upper } X}$. This definition of **lower** X and **upper** X is intentionally incomplete for the case where $X = \langle\ \rangle$. We assume that there are integers

l_0 and u_0 such that **lower** $\langle\ \rangle = l_0$, **upper** $\langle\ \rangle = u_0$, and $l_0 > u_0$, but we leave these integers unspecified to avoid making arguments which might depend upon their arbitrary values.

When $S \subseteq$ **dom** X, we write $X \upharpoonright S$, called the *restriction* of X to S, to denote the function such that

$$\textbf{dom}(X \upharpoonright S) = S \tag{3.1}$$

$$(\forall i \in S)\ (X \upharpoonright S)(i) = X(i). \tag{3.2}$$

(Usually, but not necessarily, S will be an interval.) This concept, which mirrors the informal idea of (the value of) a subarray or segment of an array, satisfies

$$\text{If } S' \subseteq S \subseteq \textbf{dom}\ X \quad \text{then} \quad (X \upharpoonright S) \upharpoonright S' = X \upharpoonright S' \tag{3.3}$$

$$X \upharpoonright \{\ \} = \langle\ \rangle. \tag{3.4}$$

As an example, consider the program

begin integer *i*; **integer array** *Squares*(−5:5);
integer array *Possquares*(0:5);
integer array *Nosquares*(14:5);
for *i* := −5 **until** 5 **do** *Squares*(*i*) := *i* × *i*;
for *i* := 0 **until** 5 **do** *Passquares*(*i*) := *i* × *i*;
...
end

At the program point indicated by the ellipsis, the following assertions will hold:

dom *Squares* = [−5 5]
lower *Squares* = −5
upper *Squares* = 5
(∀*i* ∈ [−5 5]) *Squares*(*i*) = *i* × *i*
Possquares = *Squares* ↑ [0 5]
Nosquares = *Squares* ↑ { } = ⟨ ⟩
lower *Nosquares* > **upper** *Nosquares*.

The expressions **lower** X and **upper** X occur so frequently in interval and partition diagrams that it is useful to adopt conventions for eliding them. We will permit the name of a function X to be attached as a label to an interval or partition diagram. In the presence of such a label, lower X may be omitted from the right of the leftmost line of the diagram, and **upper** X may be omitted from the left of the rightmost line. For Example, X: [a b] stands for [**lower** X a b **upper** X], X: [k] stands for [k **upper** X], and X: [ ] stands for **dom** X. Moreover, when an interval diagram is used to restrict a function X, the label X: can also be elided. For example, $X \upharpoonright$ [a] stands for $X \upharpoonright$ [**lower** X a].

For a function X, we write $\{X\}$, called the *image* of X, to denote the set $\{X(i) | i \in \textbf{dom}\ X\}$ of values obtained by applying X to members of its domain. (On the other hand, when x is not a function, $\{x\}$ will denote the singleton set containing x.) Thus for example,

$\{Possquares\} = \{0, 1, 4, 9, 16, 25\}$
$\{Possquares \upharpoonright [1\ \ 3]\} = \{1, 4, 9\}$
$\{Squares \upharpoonright [-2\ \ 2]\} = \{0, 1, 4\}$.

It is easily seen that images possess the following properties:

$$S \subseteq \textbf{dom}\ X \quad \text{implies} \quad \{X \upharpoonright S\} \subseteq \{X\} \tag{3.5}$$

$$\{\langle\ \rangle\} = \{\ \} \tag{3.6}$$

$$S \cup S' = \textbf{dom}\ X \quad \text{implies} \quad \{X\} = \{X \upharpoonright S\} \cup \{X \upharpoonright S'\} \tag{3.7}$$

$$\{X \upharpoonright [i]\} = \{X(i)\} \tag{3.8}$$

$$\#\{X\} \leq \#\ \textbf{dom}\ X \quad \text{when} \quad \textbf{dom}\ X \text{ is finite} \tag{3.9}$$

4. Operations on Relations

There are several operations on relations which can often be used to reduce the number of quantifiers in assertions.

Suppose ρ is a binary relation between two sets U and U'. Then ρ^*, called the *pointwise extension of* ρ, is the binary relation between the set of subsets of U and the set of subsets of U', such that $S\ \rho^*\ S'$ holds if and only if $x\ \rho\ x'$ holds for all x in S and all x' in S'.

When U and U' are both the set of integers, ρ could be any of the relational operators of Algol. For example, $\{2, 3\} \leq^* \{3, 4\}$ and $\{2, 3\} \neq^* \{4, 5\}$ are both true, while $\{2, 3\} <^* \{3, 4\}$, $\{2, 3\} =^* \{2, 3\}$, and $\{2, 3\} \neq^* \{2, 3\}$ are all false. The last two examples demonstrate that $\neq^*$ is not the negation of $=^*$ (and thereby show the importance of making * explicit).

The pointwise extension of any relation satisfies the following laws:

$$(S\ \rho^*\ S' \ \&\ T \subseteq S) \quad \text{implies} \quad T\ \rho^*\ S' \tag{4.1a}$$

$$(S\ \rho^*\ S' \ \&\ T' \subseteq S') \quad \text{implies} \quad S\ \rho^*\ T' \tag{4.1b}$$

$$\{\ \}\ \rho^*\ S' \tag{4.2a}$$

$$S\ \rho^*\ \{\ \} \tag{4.2b}$$

$$(S \cup T)\ \rho^*\ S' \text{ iff } (S\ \rho^*\ S' \ \&\ T\ \rho^*\ S') \tag{4.3a}$$

$$S\ \rho^*\ (S' \cup T') \text{ iff } (S\ \rho^*\ S' \ \&\ S\ \rho^*\ T') \tag{4.3b}$$

$$\{x\}\ \rho^*\ \{x'\} \text{ iff } x\ \rho\ x'. \tag{4.4}$$

Occasionally, one needs the pointwise extension of a relation with regard to only a single argument. The simplest way of encompassing this case is to regard $x\ \rho^*\ S'$ as an abbreviation for $\{x\}\ \rho^*\ S'$ and $S\ \rho^*\ x'$ as an abbreviation for $S\ \rho^*\ \{x'\}$.

Another concept involving relations, somewhat more specialized than pointwise extension, is ordering. The usual idea of an ordered array can be generalized to an arbitrary relation in a way which unifies several important cases. Let X be a function whose domain is a set of integers, and let ρ be a binary relation appropriate to the type of result of X. Then X is *ordered with regard to* ρ, written **ord**$_\rho$ X, if and only if, for all i and j in the domain of X, $i < j$ implies $X(i)\ \rho\ X(j)$.

The following "orderings" appear as specific cases:

ord$_{\le}$ X: increasing order
ord$_{<}$ X: strict increasing order
ord$_{\ge}$ X: decreasing order
ord$_{>}$ X: strict decreasing order
ord$_{=}$ X: all elements equal
ord$_{\neq}$ X: all elements distinct

Moreover, the generalization satisfies the following essential laws of ordering:

$$\mathbf{ord}_\rho\, X \;\&\; S \subseteq \mathbf{dom}\, X \quad \text{implies} \quad \mathbf{ord}_\rho(X \restriction S) \tag{4.5}$$

$$\#\,\mathbf{dom}\, X \le 1 \quad \text{implies} \quad \mathbf{ord}_\rho\, X \tag{4.6}$$

$$\begin{aligned}&\text{If} \quad S \cup T = \mathbf{dom}\, X \;\&\; S <^{*} T \quad \text{then}\\ &\quad (\mathbf{ord}_\rho\, X \quad \text{iff} \quad (\mathbf{ord}_\rho(X \restriction S) \;\&\; \mathbf{ord}_\rho(X \restriction T)\\ &\qquad\qquad \&\; \{X \restriction S\}\ \rho^{*}\ \{X \restriction T\})).\end{aligned} \tag{4.7}$$

An important special case of (4.7) is obtained by taking S and T to be two components of a partition:

$$\begin{aligned}&\text{If} \quad X\colon \boxed{\ k\ } \quad \text{then}\\ &\quad (\mathbf{ord}_\rho\, X \quad \text{iff} \quad (\mathbf{ord}_\rho(X \restriction \boxed{\ k}) \\ &\qquad\qquad \&\; \mathbf{ord}_\rho(X \restriction k\boxed{}) \\ &\qquad \&\; \{X \restriction \boxed{\ k}\}\ \rho^{*}\ \{X \restriction k\boxed{}\})).\end{aligned} \tag{4.8}$$

For particular relations ρ, there will be additional significant laws about ρ^{*} and $\mathbf{ord}_\rho$. Although we cannot approach completeness in this area, the following laws are relevant to the examples we will give:

If $x\ \rho\ y$ implies $x\ \rho'\ y$ for all x and y, then $S\ \rho^{*}\ T$ implies $S\ \rho'^{*}\ T$ for all S and T, and $\mathbf{ord}_\rho\ X$ implies $\mathbf{ord}_{\rho'}\ X$ for all X. (4.9)

If $x\ \rho\ y$ and $y\ \rho'\ z$ implies $x\ \rho''\ z$ for all x, y, and z, then $S\ \rho^{*}\ y$ and $y\ \rho'^{*}\ T$ implies $S\ \rho''^{*}\ T$ for all S, y, and T. (4.10)

If $x\ \rho\ x$ for all x, and if $\mathbf{dom}\ X$ is a nonempty interval, then $\mathbf{ord}_\rho\ X$ implies $X(\mathbf{lower}\ X)\ \rho^{*}\ \{X\}$ and $\{X\}\ \rho^{*}\ X(\mathbf{upper}\ X)$. (4.11)

5. Binary Search

We have now introduced enough of our notation to demonstrate its use in describing—precisely yet intelligibly—why a program works. As an example, we describe an algorithm for binary search.

Given an ordered array X and a test value y, the program should set the boolean variable *found* to indicate whether any element of X is equal to y. If *found* is true, then the integer variable j should be set to a subscript of X such that $X(j) = y$. More precisely, if $\mathbf{ord}_{\le}\ X$, then executing the program should achieve the goal

$$\begin{aligned}&\textbf{if } \textit{found} \textbf{ then} \quad X\colon \boxed{\ j\ } \quad \&\ X(j) = y \textbf{ else}\\ &\quad \{X\} \not\approx^{*} y.\end{aligned}$$

Throughout program execution, *found* will only be set to true if $X\colon \boxed{\ j\ }\ \&\ X(j) = y$ is achieved. On the other hand, when *found* is false, it will not be known that y occurs nowhere in X, but only that it does not occur in either of two segments at the left and right ends of X. If we use the local variables a and b to delineate these segments, we have the invariant:

$$\begin{aligned}&\textbf{if } \textit{found} \textbf{ then} \quad X\colon \boxed{\ j\ } \quad \&\ X(j) = y \textbf{ else}\\ &\quad X\colon \boxed{\ a\quad b\ } \quad \&\ \{X \restriction \boxed{\ a}\} \not\approx^{*} y\\ &\quad \&\ \{X \restriction b\boxed{}\} \not\approx^{*} y.\end{aligned}$$

On the one hand, this invariant can be achieved initially by setting *found* to **false** and making the end segments of X empty. On the other hand, it is easy to see that the invariant implies the goal of the program if either *found* is **true** or $\boxed{a\quad b}$ is empty. This is obvious if *found* is **true**, while if *found* is **false** and $\boxed{a\quad b}$ is empty then the partition diagram $X\colon \boxed{\ a\quad b\ }$ implies $\mathbf{dom}\ X = \boxed{\ a} \cup \boxed{b\ }$, so that $\{X \restriction \boxed{\ a}\} \not\approx^{*} y\ \&\ \{X \restriction b\boxed{}\} \not\approx^{*} y$ implies $\{X\} \not\approx^{*} y$. Thus, since the emptiness of $\boxed{a\quad b}$ can be tested by $a > b$, our program has the form:

```
begin integer a, b,
a = lower X, b = upper X, found = false,
while ¬ (found or a > b) do ...
end
```

When execution of the body of the **while** statement begins, both the invariant and the **while** test will be true. Since $\boxed{a\quad b}$ will be nonempty, we can perform an operation "Pick j" (whose details will be considered later) which sets j to some integer in $\boxed{a\quad b}$. At this stage, we will have

$$\begin{aligned}&X\colon \boxed{\ a\ \ j\ \ b\ }\\ &\qquad \&\ \{X \restriction \boxed{\ a}\} \not\approx^{*} y\ \&\ \{X \restriction b\boxed{}\} \not\approx^{*} y,\end{aligned}$$

and we can compare $X(j)$ with y. There are three cases:

(1) If $X(j) = y$, the invariant will be preserved if *found* is set to **true**.

(2) If $X(j) < y$, then $\mathbf{ord}_{<}\ X$ insures that $\{X \restriction \boxed{\ j}\} \not\approx^{*} y$. Thus $\{X \restriction \boxed{\ a}\} \not\approx^{*} y$ will be preserved if a is set to $j + 1$.

(3) If $X(j) > y$, then a similar argument justifies setting b to $j - 1$.

The following is a more detailed justification of Case (2): From (4.5) and (4.11), $\mathbf{ord}_{<}\ X$ and the nonemptiness of $X\colon \boxed{\ j\ }$ imply $\{X \restriction \boxed{\ j}\} \le^{*} X(j)$. Along with $X(j) < y$, this implies $\{X \restriction \boxed{\ j}\} <^{*} y$ by (4.10), and $\{X \restriction \boxed{\ j}\} \not\approx^{*} y$ by (4.9). (In a more formal presentation, $\mathbf{ord}_{<}\ X$ would occur in all assertions, reflecting the obvious fact that the program does not change the array X.)

Thus our program is:

```
begin integer a, b,
a = lower X, b = upper X, found = false;
while ¬ (found or a > b) do
  begin
  "Pick j";
  if X(j) = y then found = true else
    if X(j) < y then a = j + 1 else b = j - 1
  end
end
```

Termination is guaranteed by the fact that each iteration either sets *found* to **true**, which immediately stops further iterations, or else decreases the size of $\boxed{a \quad b}$, whose emptiness will cause termination. The absence of subscript errors is guaranteed since X: $\boxed{\quad | j | \quad}$ holds at the program points where $X(j)$ is evaluated.

It should be noticed that this discussion of binary search does not exclude the possibility that $\boxed{\textbf{lower } X \quad \textbf{upper } X}$, and therefore $\boxed{a \quad b}$, might be irregular. This illustrates our contention, at the end of Section 2, that partition diagrams permit reasoning about intervals to include the irregular case without extra case analysis.

To complete our program, we must digress from the topic of arrays to specify "Pick j". In this case, the problem is not to find a correct realization—either $j := a$ or $j := b$ would be correct—but to find an efficient one. The need to shrink $\boxed{a \quad b}$ as much as possible suggests choosing j as close as possible to the midpoint of $\boxed{a \quad b}$, i.e. $j := (a + b) \div 2$.

However, we must be sure that if $a \le b$, then $j := (a + b) \div 2$ will achieve $a \le j \le b$, despite the fact that integer division involves rounding. Although it is standardized in Algol 60, the rounding behavior of hardware-implemented division can vary for different machines, especially when $a + b$ is negative. Fortunately, it is enough to know that division by two is a monotonic function which is exact for even numbers. For $a \le b$ implies $a + a \le a + b \le b + b$, so that monotonicity gives $(a + a) \div 2 \le (a + b) \div 2 \le (b + b) \div 2$, and exactness for even numbers gives $a \le (a + b) \div 2 \le b$.

(S. Winograd has pointed out that $j := (a + b) \div 2$ is unnecessarily prone to overflow, in comparison with, for example, $j := a + (b - a) \div 2$. We leave it to the reader to show that the correctness of this improvement can still be proved with a monotonicity argument.)

6. Array Assignment

We must now move beyond programs such as binary search which merely use arrays, to consider programs which change arrays. Our treatment of such programs follows the ideas of Hoare [7, 9], which are based upon earlier work by McCarthy and Painter [10].

In programming languages at the level of Algol, the fundamental agent of change is an assignment statement which alters a single array element, e.g. $X(i) := e$. To deal with this statement from the viewpoint that an array is a function-valued variable, we must regard it as an abbreviation for the assignment $X := [X|i|e]$, where $[X|i|e]$ denotes the function which is similar to X except that it maps i into e. More formally, $[X|i|e]$ is defined when $i \in \textbf{dom } X$, in which case it is the function satisfying

$$\textbf{dom } [X|i|e] = \textbf{dom } X \tag{6.1}$$

$$[X|i|e](i) = e \tag{6.2}$$

$$[X|i|e](j) = X(j) \quad \text{when} \quad j \neq i, \tag{6.3}$$

and, as an immediate consequence of (6.3),

$$[X|i|e] \restriction S = X \restriction S \quad \text{when} \quad S \subseteq \textbf{dom } X \text{ and } i \notin S. \tag{6.4}$$

Once $X(i) := e$ is seen as an abbreviation for $X := [X|i|e]$, the usual axiom of assignment [5]:

$$P|_{x \to e}\{x := e\}\, P \tag{6.5}$$

(where $P|_{x \to e}$ denotes the result of substituting e for x in P) extends to an axiom of array assignment [9]:

$$P|_{X \to [X|i|e]}\{X(i) := e\}\, P. \tag{6.6}$$

Because of (6.1), when this axiom is used, the substitution $X \to [X|i|e]$ need not be applied to occurrences of X in **dom** X, **lower** X, **upper** X, or in a label attached to an interval or partition diagram.

7. Equivalence Relations for Arrays

For many programs which alter arrays, such as sorting programs, a full specification will stipulate both that the final value of the array will possess some property, such as being ordered, and that the final value will be related to the initial value in some way, such as being a rearrangement. Often—even when the situation is intuitively obvious—a formidable technical apparatus is needed to formulate and prove the latter kind of specification.

To deal with these problems it is useful to introduce several equivalence relations for array values. Suppose X and Y are both functions whose domains are sets of integers. Then:

(a) We write $X \sim\sim Y$, and say that X is a *redistribution* of Y iff $\{X\} = \{Y\}$.

(b) We write $X \sim Y$, and say that X is a *rearrangement* of Y iff there is a bijection B (sometimes called a one to one correspondence or a permutation) from **dom** X to **dom** Y such that $(\forall i \in \textbf{dom } X)\ Y(B(i)) = X(i)$.

(c) We write $X \simeq Y$, and say that X is a *shift* of Y iff there is a bijection as in (b) with the special form $B(i) = i + s$ for some integer s.

This defines an increasingly stringent sequence of equivalence relations. Thus where ρ is $\sim\sim$, $\sim$, or $\simeq$:

Transitivity $X \rho Y \,\&\, Y \rho Z$ implies $X \rho Z$ (7.1)

Symmetry $X \rho Y$ implies $Y \rho X$ (7.2)

Reflexivity $X \rho X$ (7.3)

$X \simeq Y$ implies $X \sim Y$ (7.4)

$X \sim Y$ implies $X \sim\sim Y$. (7.5)

Finally, we have three more specific laws. Exchanging a pair of elements produces a rearrangement:

$$(\forall i, j \in \textbf{dom } X)\ [[X|i|X(j)]|j|X(i)] \sim X, \tag{7.6}$$

two one-element arrays with equal values are shifts of one another:

$$\boxed{i} = \mathbf{dom}\ X\ \&\ \boxed{j} = \mathbf{dom}\ Y\ \&\ X(i) = Y(j) \quad \text{implies } X \simeq Y, \tag{7.7}$$

and a shift of an ordered array is ordered:

$$X \simeq Y\ \&\ \mathbf{ord}_{\le} X \text{ implies } \mathbf{ord}_{\le} Y. \tag{7.8}$$

As Hoare has pointed out [6], for any program which only alters an array by performing exchanges, (7.1), (7.3), and (7.6) are sufficient to show that the final array value is a rearrangement of the initial value. However, to deal with programs which move information from one array to another, we must also consider the concatenation of array values.

8. Concatenation

Let X and Y be functions whose domains are intervals with sizes m and n respectively. Then $X^\frown Y$, called the *concatenation* of X and Y, is a function such that

$$\mathbf{dom}\ (X^\frown Y) = \boxed{l \quad m+n+l-1}$$
$$(X^\frown Y) \upharpoonright \boxed{l \quad m+l-1} \simeq X$$
$$(X^\frown Y) \upharpoonright \boxed{m+l \quad m+n+l-1} \simeq Y,$$

where $l = \mathbf{lower}\ (X^\frown Y)$. To make this definition unique, we would have to specify the integer function **lower** $(X^\frown Y)$; we refrain from doing so to preclude arguments which might depend upon this arbitrarily chosen function.

Let $\langle\ \rangle$ denote the unique function whose domain is empty. Then concatenation satisfies the following laws:

$$X^\frown \langle\ \rangle \simeq X \tag{8.1}$$
$$\langle\ \rangle^\frown X \simeq X \tag{8.2}$$
$$(X^\frown Y)^\frown Z \simeq X^\frown (Y^\frown Z) \tag{8.3}$$
$$X \simeq X'\ \&\ Y \simeq Y' \text{ implies } X^\frown Y \simeq X'^\frown Y' \tag{8.4}$$
$$X^\frown Y \sim Y^\frown X \tag{8.5}$$
$$X \sim X'\ \&\ Y \sim Y' \text{ implies } X^\frown Y \sim X'^\frown Y' \tag{8.6}$$
$$X\colon \boxed{\quad a \quad} \text{ implies } X \simeq (X \upharpoonright \boxed{\quad a}\,)^\frown (X \upharpoonright a\,\boxed{\quad}) \tag{8.7}$$
$$\{X^\frown Y\} = \{X\} \cup \{Y\} \tag{8.8}$$
$$\mathbf{ord}_\rho (X^\frown Y) \text{ iff } \mathbf{ord}_\rho X\ \&\ \mathbf{ord}_\rho Y\ \&\ \{X\}\ \rho^*\ \{Y\}. \tag{8.9}$$

The first four laws show that array values form a monoid under concatenation, provided that shift equivalence is used in place of true equality. The next two laws show that this monoid becomes commutative when the less stringent equivalence of rearrangement is used. (Technically, one can make these statements precise by working with the quotient of the set of array values under the equivalence relations $\simeq$ or $\sim$.)

The last three laws establish the basic connections between concatenation and partitions, images, and ordering. In particular, (8.9) is a consequence of (4.8) and (7.8).

9. Merging

As a second example of program description, we consider the problem of merging: Given two ordered arrays X and Y, set Z to an ordered rearrangement of the concatenation of X and Y. We assume that Z is just the right size to hold the result. Thus if

$$\mathbf{ord}_{\le} X\ \&\ \mathbf{ord}_{\le} Y\ \&\ \#\ \mathbf{dom}\ Z = (\#\ \mathbf{dom}\ X + \#\ \mathbf{dom}\ Y),$$

then executing the program should achieve the goal

$$\mathbf{ord}_{\le} Z\ \&\ Z \sim (X^\frown Y).$$

During execution, each array will be partitioned into a processed part on the left and an unprocessed part on the right, the processed part of Z will be an ordered rearrangement of the concatenation of the processed parts of X and Y, the unprocessed part of Z will be the right size to hold the unprocessed parts of X and Y, and all processed elements in Z will be smaller or equal to all unprocessed elements in X or Y. (The last condition is needed to insure that the unprocessed elements can be moved into Z without rearranging the already processed elements.) Thus we have the invariant:

$$I \equiv X\colon \boxed{\quad kx} \ \&\ Y\colon \boxed{\quad ky} \ \&\ Z\colon \boxed{\quad kz} \tag{a}$$
$$\&\ \mathbf{ord}_{\le} Z \upharpoonright \boxed{\quad kz} \tag{b}$$
$$\&\ Z \upharpoonright \boxed{\quad kz} \sim (X \upharpoonright \boxed{\quad kx}^\frown Y \upharpoonright \boxed{\quad ky}) \tag{c}$$
$$\&\ \#\ Z\colon \boxed{kz} = \#\ X\colon \boxed{kx} + \#\ Y\colon \boxed{ky} \tag{d}$$
$$\&\ (Z \upharpoonright \boxed{\quad kz}) \le^* (X \upharpoonright \boxed{kx}) \cup (Y \upharpoonright \boxed{ky}). \tag{e}$$

(The conciseness and clarity of this notation in comparison with predicate calculus can be seen by comparing this invariant with the nearly equivalent one given in Reynolds [11].)

The invariant can be achieved initially by making the processed parts all empty, and it will imply the goal of the program when the unprocessed parts are all empty, which by (d) will occur when the unprocessed part of Z is empty. Thus we can use a program of the form:

begin integer kx, ky, kz;
$kx :=$ **lower** X; $ky :=$ **lower** Y; $kz :=$ **lower** Z;
while $kz \le$ **upper** Z **do** "Copy One Element"
end.

In "Copy One Element," a single element will be moved from the unprocessed part of X or Y into the processed part of Z. To preserve condition (e) the element to be moved must be the smallest member of $(X \upharpoonright \boxed{kx}) \cup (Y \upharpoonright \boxed{ky})$. Since both X and Y are ordered, this will be the smaller of the leftmost unprocessed elements, $X(kx)$ or $Y(ky)$, providing both unprocessed parts are nonempty. However, if only one unprocessed part is nonempty, its leftmost element will be the element to be moved.

More precisely, when "Copy One Element" begins, Z: $\boxed{kz\ \ }$ and at least one of X: $\boxed{kx\ \ }$ and Y: $\boxed{ky\ \ }$ will be nonempty. Suppose X: $\boxed{kx\ \ }$ is nonempty and Y: $\boxed{ky\ \ }$ is empty. Since $\mathbf{ord}_{\leq} X$, (4.5) and (4.11) imply $X(kx) \leq^{\bullet} \{X \upharpoonright \boxed{kx\ \ }\}$, and since $\{Y \upharpoonright \boxed{ky\ \ }\}$ is empty,

$$IX \equiv X\colon \boxed{kx\ \ } \ \&\ Z\colon \boxed{kz\ \ } \tag{f}$$
$$\&\ X(kx) \leq^{\bullet} (\{X \upharpoonright \boxed{kx\ \ }\} \cup \{Y \upharpoonright \boxed{ky\ \ }\}) \tag{g}$$

will hold as well as the invariant I. (Note that X: $\boxed{kx\ \ }$ is an abbreviation for the partition diagram $\boxed{kx\ \ \ \mathbf{upper}\ X}$, which asserts that the unprocessed part of X is nonempty.) By a similar argument, if X: $\boxed{kx\ \ }$ is empty and Y: $\boxed{ky\ \ }$ is nonempty, then

$$IY \equiv Y\colon \boxed{ky\ \ } \ \&\ Z\colon \boxed{kz\ \ }$$
$$\&\ Y(ky) \leq^{\bullet} (\{X \upharpoonright \boxed{kx\ \ }\} \cup \{Y \upharpoonright \boxed{ky\ \ }\})$$

will hold. Finally, if both unprocessed segments are nonempty, then

$$X\colon \boxed{kx\ \ } \ \&\ Y\colon \boxed{ky\ \ } \ \&\ Z\colon \boxed{kz\ \ }$$
$$\&\ X(kx) \leq^{\bullet} \{X \upharpoonright \boxed{kx\ \ }\}$$
$$\&\ Y(ky) \leq^{\bullet} \{Y \upharpoonright \boxed{ky\ \ }\}$$

will hold. In this case, by (4.10) and the transitivity of $\leq$, $X(kx) \leq Y(ky)$ implies IX, while $Y(ky) \leq X(kx)$ implies IY.

Thus if we define

"Copy One Element" $\equiv$
  if $ky >$ **upper** Y **then** "Copy X" **else**
  if $kx >$ **upper** X **then** "Copy Y" **else**
  if $X(kx) \leq Y(ky)$ **then** "Copy X" **else** Copy Y",

then I & IX will hold before the execution of (either occurrence of) "Copy X", and I & IY will hold before the execution of "Copy Y".

If "Copy X" moves $X(kx)$ out of the unprocessed part of X and into the processed part of Z, then (g) insures that (e) will be preserved. Moreover, (e) insures that $X(kx)$ will be larger or equal to the elements which have previously been moved into Z. Thus the ordering (b) will be preserved if $X(kx)$ is placed at the right of the processed part of Z. This leads to:

"Copy X" $\equiv$
  begin $Z(kz) := X(kx)$; $kx := kx + 1$; $kz := kz + 1$ **end**.

and by a similar argument

"Copy Y" $\equiv$
  begin $Z(kz) := Y(ky)$; $ky := ky + 1$; $kz := kz + 1$ **end**.

Formally, in the notation of Hoare [5], "Copy X" must meet the specification

I & IX {"Copy X"} I.

To exemplify the application of the various laws we have stated, we give a formal proof of this specification. The assignment axioms (6.5) and (6.6) imply I' {"Copy X"} I, where

$$I' \equiv I_{|kz \to kz+1|kx \to kx+1|Z \to (Z|kz|X(kx))|}$$
$$= X\colon \boxed{kx\ \ } \ \&\ Y\colon \boxed{\ \ ky} \tag{a'}$$
$$\&\ Z\colon \boxed{kz\ \ }$$
$$\&\ \mathbf{ord}_{\leq} [Z|kz|X(kx)] \upharpoonright \boxed{\ \ kz} \tag{b'}$$
$$\&\ [Z|kz|X(kx)] \upharpoonright \boxed{\ \ kz}$$
$$\sim (X \upharpoonright \boxed{kx\ \ } \frown Y \upharpoonright \boxed{\ \ }ky) \tag{c'}$$
$$\&\ \#\, Z\colon kz\boxed{\ \ }$$
$$= \#\, X\colon kx\boxed{\ \ } + \#\, Y\colon \boxed{ky\ \ } \tag{d'}$$
$$\&\ \{[Z|kz|X(kx)] \upharpoonright \boxed{\ \ kz}\}$$
$$\leq^{\bullet} \{X \upharpoonright kx\boxed{\ \ }\} \cup \{Y \upharpoonright \boxed{ky\ \ }\} \tag{e'}$$

(Here we have made the simplification of replacing occurrences of $\boxed{\ \ kx+1\ \ }$ and $\boxed{kz+1}$ by the equivalent forms $\boxed{\ \ kx}$ and $\boxed{kz}$.) Thus we must show that I & IX implies I', i.e. that lines (a) through (g) imply (a') through (e').

By the rule (2.9) of substitution, (a) and (f) imply

$$X\colon \boxed{\ \ kx\ \ } \ \&\ Y\colon \boxed{\ \ ky\ \ } \ \&\ Z\colon \boxed{\ \ kz\ \ } \tag{h}$$

which, by the rule (2.7) of erasure, implies (a') as well as various partition diagrams used in the sequel. In particular, by (2.2) and (2.1), X: $\boxed{kx\ \ }$ implies $\#X$: $\boxed{kx\ \ }$ $= \#X$: $kx\boxed{\ \ } + 1$, and Z: $\boxed{kz\ \ }$ implies $\#Z$: $\boxed{kz\ \ }$ $= \#Z$: $kz\boxed{\ \ } + 1$, so that (d) implies (d').

Next, we have

$$[Z|kz|X(kx)] \upharpoonright \boxed{\ \ kz}$$
$$\simeq [Z|kz|X(kx)] \upharpoonright \boxed{\ \ }kz \frown [Z|kz|X(kx)] \upharpoonright \boxed{kz}$$
by Z: $\boxed{\ \ kz}$, (8.7), (3.3)
$$= Z \upharpoonright \boxed{\ \ }kz \frown [Z|kz|X(kx)] \upharpoonright \boxed{kz}$$
by (6.4)
$$\simeq Z \upharpoonright \boxed{\ \ }kz \frown X \upharpoonright \boxed{kx}$$
by (7.7), (8.4), (6.2)
$$\sim (X \upharpoonright \boxed{\ \ }kx \frown Y \upharpoonright \boxed{\ \ }ky) \frown X \upharpoonright \boxed{kx}$$
by (c), (8.6)
$$\sim (X \upharpoonright \boxed{\ \ }kx \frown X \upharpoonright \boxed{kx}) \frown Y \upharpoonright \boxed{\ \ }ky$$
by (8.3), (8.5), (8.6), (7.4)
$$\simeq X \upharpoonright \boxed{\ \ kx} \frown Y \upharpoonright \boxed{\ \ }ky$$
by X: $\boxed{\ \ kx}$ (8.7), (3.3)

which establishes (c'), and also

$$[Z|kz|X(kx)] \upharpoonright \boxed{\ \ kz} \simeq Z \upharpoonright \boxed{\ \ }kz \frown X \upharpoonright \boxed{kx}. \tag{i}$$

Then

$$\{[Z|kz|X(kx)] \upharpoonright \boxed{\ \ kz}\}$$
$$= \{Z \upharpoonright \boxed{\ \ }kz \frown X \upharpoonright \boxed{kx}\}$$
by (i), (7.4), (7.5)
$$= \{Z \upharpoonright \boxed{\ \ }kz\} \cup \{X \upharpoonright \boxed{kx}\}$$
by (8.8)

$= \{Z \upharpoonright [\,)kz\} \cup \{X(kx)\}$
by (3.8)

$\leq^* \{X \upharpoonright [kx\ \ )\} \cup \{Y \upharpoonright [ky\ \ )\}$
by (e), (g), (4.3a)

$= \{X \upharpoonright [kx] \frown X \upharpoonright kx[\,)\} \cup \{Y \upharpoonright [ky\ \ )\}$
by X: $[kx\ \ )$, (8.7), (3.3)

$= \{X \upharpoonright [kx)\} \cup \{X \upharpoonright kx[\,)\} \cup \{Y \upharpoonright [ky\ \ )\}$
by (8.8)

so that (4.1a) and (4.1b) give (e') and

$$\{Z \upharpoonright [\,)kz\} \leq^* \{X \upharpoonright [kx)\}. \qquad \text{(j)}$$

Finally, (3.1), (2.1), and (4.6) imply $\mathbf{ord}_{\leq} X \upharpoonright [kx)$, which with (b), (j), and (8.9) implies $\mathbf{ord}_{\leq}(Z \upharpoonright [\,)kz \frown X \upharpoonright [kx)$, which with (i) and (7.8) implies (b').

10. Multidimensional Arrays

Although the concepts we have presented were developed and tested in the context of one-dimensional arrays, most of them extend to the multidimensional case. The major additional concept which is needed is the *Cartesian product*:

$$S_1 \times \ldots \times S_n = \{\langle i_1, \ldots, i_n\rangle \mid i \in S_1 \,\&\, \ldots \,\&\, i_n \in S_n\}.$$

A Cartesian product of intervals is called a *block*. The values of the array declared by τ **array** $X(a_1: b_1, \ldots, a_n: b_n)$ are functions whose domain is the block $[a_1\ \ b_1) \times \ldots \times [a_n\ \ b_n)$.

It is evident that the values of subarrays of X such as rows and columns are restrictions of X to certain blocks. For example, with some fairly obvious conventions about eliding lower and upper bounds, the following assertion specifies that $\langle i, j\rangle$ is a saddle point of the two-dimensional array X:

$$\{X \upharpoonright ([i] \times [\,)\}) \leq^* X(i, j)$$
$$\&\ X(i, j) \leq^* \{X \upharpoonright ([\,) \times [j])\}.$$

11. Conclusion

The content of this paper is only a small beginning. It is largely limited to one-dimensional integer-subscripted arrays, and even within this domain it is based upon the careful study of perhaps a dozen simple programs. Moreover, program proving has been viewed as a purely human endeavor and the possibility of mechanization has been ignored.

Thus further study is certain to produce significant extensions and reformulations. Nevertheless, we believe that we have gone far enough to demonstrate the value of the underlying approach: We have formulated concepts, laws, and notations which are powerful tools for the precise yet intelligible description of a significant aspect of programming.

Hopefully, this work suggests guidelines for further progress: One should focus upon particular mechanisms such as arrays rather than generalities which pertain to all computation. Concepts and laws are more fundamental than notation per se, and should reflect intuitive understanding. Most important, the crucial test is the ability to describe real programs in a way which is not only precise but also intelligible to the human reader.

Acknowledgments. I am indebted to the members of IFIP Working Group 2.3, who have provided motivation, inspiration, and helpful criticism. I am also grateful for the hospitality of the University of Edinburgh and the support of the Science Research Council during the period when this paper was written.

Appendix. Proof of Proposition (2.3)

We leave it to the reader to verify that either $a_0 \leq a_1 \leq \ldots \leq a_n$ or $a_0 \geq a_1 \geq \ldots \geq a_n$ implies $a_0[\,)a_1[\,)\ \ldots\ [\,)a_n]$. The following proof of the converse was found by F.L. Morris.

Suppose $a_0[\,)a_1[\,)\ \ldots\ [\,)a_n]$. From (2.2) we have

$$\sharp\, a_0[\,)a_n] = \sum_{i=1}^{n} \sharp\, a_{i-1}[\,)a_i], \qquad \text{(a)}$$

where

$$\sharp\, a[\,)b] = \textbf{if } b - a \geq 0 \textbf{ then } b - a \textbf{ else } 0$$

is always nonnegative and is zero iff $a[\,)b]$ is empty. For arbitrary a_i's simple cancellation gives

$$a_n - a_0 = \sum_{i=1}^{n} a_i - a_{i-1}.$$

Then subtraction of (a) from both sides gives

$$f(a_0, a_n) = \sum_{i=1}^{n} f(a_{i-1}, a_i), \qquad \text{(b)}$$

where

$$f(a, b) = b - a - \sharp\, a[\,)b]$$
$$= \textbf{if } b - a \geq 0 \textbf{ then } 0 \textbf{ else } b - a$$

is always nonpositive and is zero iff $a[\,)b]$ is regular.

The interval $a_0[\,)a_n]$ must be either empty or regular (or both). Suppose it is empty. Then (a) asserts that a sum of nonnegative terms is zero, which implies that each term is zero. Thus for each i, $a_{i-1}[\,)a_i]$ is empty, and $a_{i-1} \geq a_i$.

On the other hand, suppose $a_0[\,)a_n]$ is regular. Then (b) asserts that a sum of nonpositive terms is zero, which implies that each term is zero. Thus for each i, $a_{i-1}[\,)a_i]$ is regular, and $a_{i-1} \leq a_i$.

Received July 1977, revised May 1978

References

1. Burstall, R.M. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence 7* (Nov. 1972), 23–49.
2. Cooper, D.C. Proofs about programs with one-dimensional arrays. Unpublished.
3. Dijkstra, E.W. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1976.
4. Floyd, R.W. Assigning meanings to programs. Proc. Symp. in Applied Mathematics, Vol. 19, Amer. Math. Soc., Providence, R.I., 1967, pp. 19–32.
5. Hoare, C.A.R. An axiomatic basis for computer programming. *Comm. ACM 12*, 10 (Oct. 1969), 576–581.
6. Hoare, C.A.R. Proof of a program: FIND. *Comm. ACM 14*, 1 (Jan. 1971), 39–45.
7. Hoare, C.A.R. Notes on data structuring. In *Structured Programming*, O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, Academic Press, N.Y., 1972, pp. 83–174.
8. Hoare, C.A.R. A note on the FOR statement. *BIT 12*, 3 (1972), 334–341.
9. Hoare, C.A.R., and Wirth, N. An axiomatic definition of the programming language PASCAL. *Acta Informatica 2* (1973), 335–355.
10. McCarthy, J., and Painter, J. Correctness of a compiler for arithmetic expressions. Proc. Symp. in Applied Mathematics, Vol. 19, Amer. Math. Soc., Providence, R.I., 1967, pp. 33–41.
11. Reynolds, J.C. Programming with transition diagrams. In *Programming Methodology, A Collection of Papers by Members of IFIP WG 2.3*, D. Gries, Ed., Springer-Verlag, 1978, pp. 153–165.

APPENDIX E: RECENT WORK ON ARRAY CONCEPTS

In "Reasoning about Arrays" (Comm. ACM 22 (1979) 290-299), we defined the concept of rearrangement:

> When X and Y are functions with the same codomain, $X \sim Y$ (X is a rearrangement of Y) holds if and only if there is a bijection B: $\underset{\sim}{dom}\ X \to \underset{\sim}{dom}\ Y$ such that $X = B \cdot Y$,

and of shift equivalence:

> When X and Y are functions with the same codomain and domains that are intervals of the same size, $X \underset{\sim}{\sim} Y$ (X is shift equivalent to Y) holds if and only if there is a constant s such that $X(i) = Y(i + s)$ holds for all i in $\underset{\sim}{dom}\ X$.

This year we discovered the usefulness of generalizing the latter concept as follows:

> When X and Y are functions with the same codomain and totally ordered domains, $X \underset{\sim}{\sim} Y$ (X is a realignment of Y) holds if and only if there is a monotone bijection B: $\underset{\sim}{dom}\ X \to \underset{\sim}{dom}\ Y$ such that $X = B \cdot Y$.

It is easily seen that realignment is an equivalence relation that implies rearrangement and reduces to shift equivalence in the special case where $\underset{\sim}{dom}\ X$ and $\underset{\sim}{dom}\ Y$ are intervals of the same size. Morever,

$$\text{If } X \underset{\sim}{\sim} Y \text{ and } \underset{\sim}{ord}_{\rho}\ Y \text{ then } \underset{\sim}{ord}_{\rho}\ X$$

and

$$\text{If } X \underset{\sim}{\sim} Y \text{ then } X \cdot Z \underset{\sim}{\sim} Y \cdot Z\ .$$

The advantage of realignment lies in its ability to deal with functions whose domains are sets of integers that are not intervals, or even sets of nonintegers. An example is the following annotation of a program for left-shifting an array:

$\{\boxed{a \quad b}$ and $\boxed{a \quad b} \subseteq \mathrm{dom}\ X$ and $X = X_0\}$

begin integer k;

k := a;

$\{$**whileinv:** $\boxed{a \quad k \quad b}$ and $X \upharpoonright (\boxed{a}\, k \cup k\, \boxed{b}) \simeq X_0 \upharpoonright a\, \boxed{b}\}$

while k < b do

begin k := k + 1; X(k-1) := X(k) end

end

$\{X \upharpoonright \boxed{a}\, b \simeq X_0 \upharpoonright a\, \boxed{b}\}$

Notice that the invariant expresses the idea of an array with a hole in the middle by using a function whose domain $\boxed{a}\, k \cup k\, \boxed{b}$ **is not an interval.**

Another advantage is that we can replace the usual notion of concatenation by a kind of concatenation based on "source tupling" of functions. For sets S and T, let

$$S + T = \{1\} \times S \cup \{2\} \times T$$

with the ordering

$$\langle x, y\rangle \leq \langle x', y'\rangle \text{ iff } x < x' \text{ or } (x = x' \text{ and } y \leq y') .$$

Then, for functions $X: S \to U$ and $Y: T \to U$, let $X \oplus Y: S + T \to U$ be the function such that

$$(\forall i \in S)\ (X \oplus Y)(\langle 1, i\rangle) = X(i)$$

$$(\forall j \in T)\ (X \oplus Y)(\langle 2, j\rangle) = Y(j) .$$

Then

a. $\underset{\sim}{\mathrm{dom}}(X \oplus Y)$ is the union of the disjoint sets $\{1\} \times \underset{\sim}{\mathrm{dom}}\ X$ and $\{2\} \times \underset{\sim}{\mathrm{dom}}\ Y$,

b. $(X \oplus Y) \upharpoonright (\{1\} \times \underset{\sim}{\mathrm{dom}}\ X) \underset{\sim}{\sim} X$,

c. $(X \oplus Y) \upharpoonright (\{2\} \times \underset{\sim}{\mathrm{dom}}\ Y) \underset{\sim}{\sim} Y$,

d. $\{1\} \times \underset{\sim}{\mathrm{dom}}\ X <^{*} \{2\} \times \underset{\sim}{\mathrm{dom}}\ Y$.

establish that $\oplus$ is a kind of concatenation. In particular, if X and Y are sequences, then $X \oplus Y$ is a realignment of the usual sequence-concatenation of X and Y. However, unlike the usual notion of concatenation, $X \oplus Y$ is defined for any pair of functions with the same codomain.

Further laws include:

If $S \subseteq \underset{\sim}{\mathrm{dom}}\ X$ and $T \subseteq \underset{\sim}{\mathrm{dom}}\ Y$ then

$(X \oplus Y) \upharpoonright (S + T) = (X \upharpoonright S) \oplus (Y \upharpoonright T)$.

$(X \oplus Y) \cdot Z = X \cdot Z \oplus Y \cdot Z$.

$\{X \oplus Y\} = \{X\} \cup \{Y\}$.

$\underset{\sim}{\mathrm{ord}}_{\rho}\ (X \oplus Y)$ if and only if

(a) $\underset{\sim}{\mathrm{ord}}_{\rho}\ X$

and (b) $\underset{\sim}{\mathrm{ord}}_{\rho}\ Y$

and (c) $\{X\}\ \rho^{*}\ \{Y\}$.

$(X \bullet Y) \bullet Z \underset{\sim}{\sim} X \bullet (Y \bullet Z)$.

$X \bullet \langle\rangle \underset{\sim}{\sim} X$.

$\langle\rangle \bullet X \underset{\sim}{\sim} X$.

$X \bullet Y \sim Y \bullet X$.

If $X \sim X'$ and $Y \sim Y'$ then $X \bullet Y \sim X' \bullet Y'$.

If $X \underset{\sim}{\sim} X'$ and $Y \underset{\sim}{\sim} Y'$ then $X \bullet Y \underset{\sim}{\sim} X' \bullet Y'$.

If $\underset{\sim}{\text{dom}}\ X = S \cup T$ and S and T are disjoint

then $X \sim (X \upharpoonright S) \bullet (X \upharpoonright T)$.

If $\underset{\sim}{\text{dom}}\ X = S \cup T$ and $S <^* T$

then $X \underset{\sim}{\sim} (X \upharpoonright S) \bullet (X \upharpoonright T)$.

If $\underset{\sim}{\text{dom}}\ X = a\boxed{\ \ c}$ and $a\boxed{\ b\ }\boxed{\ c\ }$

then $X \underset{\sim}{\sim} (X \upharpoonright a\boxed{\ \ b}) \bullet (X \upharpoonright b\boxed{\ \ c})$.

For example, in proving the above left-shifting program, one must show that

$\boxed{a\ \ }\boxed{k-1}\boxed{k}\boxed{\ \ b}$ and $X \upharpoonright (\boxed{a\ \ }k\text{-}1 \cup k\text{-}1\boxed{\ \ b}) \underset{\sim}{\sim} X_0 \upharpoonright a\boxed{\ \ b}$

implies

$[X \mid k\text{-}1\text{: } X(k)] \upharpoonright (\boxed{a\ \ }k \cup k\boxed{\ \ b}) \underset{\sim}{\sim} X_0 \upharpoonright a\boxed{\ \ b}$.

This can be proved by a sequence of realignments involving concatenations:

$$[X \mid k\text{-}1\colon X(k)] \uparrow (\boxed{a}\, k \cup k\, \boxed{b})$$

$$\simeq ([X \mid k\text{-}1\colon X(k)] \uparrow \boxed{a}\, k) \bullet ([X \mid k\text{-}1\colon X(k)] \uparrow k\, \boxed{b})$$

$$\simeq ([X \mid k\text{-}1\colon X(k)] \uparrow \boxed{a}\, k\text{-}1) \bullet ([X \mid k\text{-}1\colon X(k)] \uparrow \boxed{k\text{-}1})$$

$$\bullet ([X \mid k\text{-}1\colon X(k)] \uparrow k\, \boxed{b})$$

$$\simeq (X \uparrow \boxed{a}\, k\text{-}1) \bullet (X \uparrow \boxed{k}) \bullet (X \uparrow k\, \boxed{b})$$

$$\simeq (X \uparrow \boxed{a}\, k\text{-}1) \bullet (X \uparrow k\text{-}1\, \boxed{b})$$

$$\simeq X \uparrow (\boxed{a}\, k\text{-}1 \cup k\text{-}1\, \boxed{b})$$

$$\simeq X_0 \uparrow a\, \boxed{b} \ .$$

Further applications of realignment arise in conjunction with preimages and related concepts. For a function X and a set $U \subseteq \underset{\sim}{\text{cod}}\ X$, let

$$P(U, X) = \{i \mid i \in \underset{\sim}{\text{dom}}\ X \text{ and } X(i) \in U\}$$

be the preimage of U under X. Then

If $U' \subseteq U$ then $P(U', X) \subseteq P(U, X)$,

$P(U \cup U', X) = P(U, X) \cup P(U', X)$,

$P(U \cap U', X) = P(U, X) \cap P(U', X)$,

$P(U - U', X) = P(U, X) - P(U', X)$,

$P(U, X) = \underset{\sim}{\text{dom}}\ X$ if and only if $\{X\} \subseteq U$,

$P(U, X) = \{\}$ if and only if U and $\{X\}$ are disjoint .

$p(U, X \cdot Y) = p(p(U, Y), X)$,

$p(U', I_U) = U'$.

$P(U, X \uparrow S) = P(U, X) \cap S$.

$S \subseteq P(\{X \uparrow S\}, X)$.

$\{X \uparrow P(U, X)\} = U \cap \{X\}$.

$P(U, X \bullet Y) = P(U, X) + P(U, Y)$.

If $X \sim Y$ then $X \restriction P(U, X) \sim Y \restriction P(U, Y)$,

If $X \underset{\sim}{\sim} Y$ then $X \restriction P(U, X) \underset{\sim}{\sim} Y \restriction P(U, Y)$.

For a function X and set U, let

$$X \mathbin{\dot{\cap}} U = X \restriction P(\underset{\sim}{\mathrm{cod}}\ X \cap U, X) ,$$

$$X \mathbin{\dot{-}} U = X \restriction P(\underset{\sim}{\mathrm{cod}}\ X \mathbin{\dot{-}} U, X) .$$

Then

$X \mathbin{\dot{\cap}} U = X$ if and only if $\{X\} \subseteq U$.

$X \mathbin{\dot{\cap}} U = \langle\rangle$ if and only if U and $\{X\}$ are disjoint .

$(X \mathbin{\dot{\cap}} U) \mathbin{\dot{\cap}} U' = X \mathbin{\dot{\cap}} (U \cap U')$.

$(X \oplus Y) \mathbin{\dot{\cap}} U = (X \mathbin{\dot{\cap}} U) \oplus (Y \mathbin{\dot{\cap}} U)$.

$\{X \mathbin{\dot{\cap}} U\} = \{X\} \cap U$,

If $X \sim Y$ then $X \mathbin{\dot{\cap}} U \sim Y \mathbin{\dot{\cap}} U$,

If $X \underset{\sim}{\sim} Y$ then $X \mathbin{\dot{\cap}} U \underset{\sim}{\sim} Y \mathbin{\dot{\cap}} U$.

and

$X \mathbin{\dot{-}} U = X$ if and only if U and $\{X\}$ are disjoint ,

$X \mathbin{\dot{-}} U = \langle\rangle$ if and only if $\{X\} \subseteq U$,

$(X \mathbin{\dot{-}} U) \mathbin{\dot{-}} U' = X \mathbin{\dot{-}} (U \cup U')$,

$(X \oplus Y) \mathbin{\dot{-}} U = (X \mathbin{\dot{-}} U) \oplus (Y \mathbin{\dot{-}} U)$,

$\{X \mathbin{\dot{-}} U\} = \{X\} - U$,

If $X \sim Y$ then $X \mathbin{\dot{-}} U \sim Y \mathbin{\dot{-}} U$,

If $X \underset{\sim}{\sim} Y$ then $X \mathbin{\dot{-}} U \underset{\sim}{\sim} Y \mathbin{\dot{-}} U$.

In effect, $X \mathbin{\dot\cap} U$ and $X \mathbin{\dot-} U$ can be regarded as the intersection and difference of the function X and the set U.

In conjunction with realignment these concepts can be used to specify programs such as the following, which deletes array elements with values outside of the interval $\boxed{r\ \ s}$:

$\{\boxed{a\ \ b} \subseteq \mathrm{dom}\ X \text{ and } X = X_0\}$

begin integer d; c := a; d := a;

{**whileinv**: $\boxed{a}\boxed{c}\boxed{d\ \ b}$ and $X \upharpoonright \boxed{a}\, c \simeq (X_0 \upharpoonright \boxed{a}\, d) \mathbin{\dot\cap} \boxed{r\ \ s}$

and $X \upharpoonright \boxed{d\ \ b} = X_0 \upharpoonright \boxed{d\ \ b}$ }

while d < b **do**

if (X(d) < r) **or** ($\neg$ r ≤ X(d)) **then** d := d + 1

else begin X(c) := X(d); c := c + 1; d := d + 1 **end**

end

$\{\boxed{a}\boxed{c\ \ b}$ and $X \upharpoonright \boxed{a}\, c \simeq (X_0 \upharpoonright \boxed{a\ \ b}) \mathbin{\dot\cap} \boxed{r\ \ s}\}$

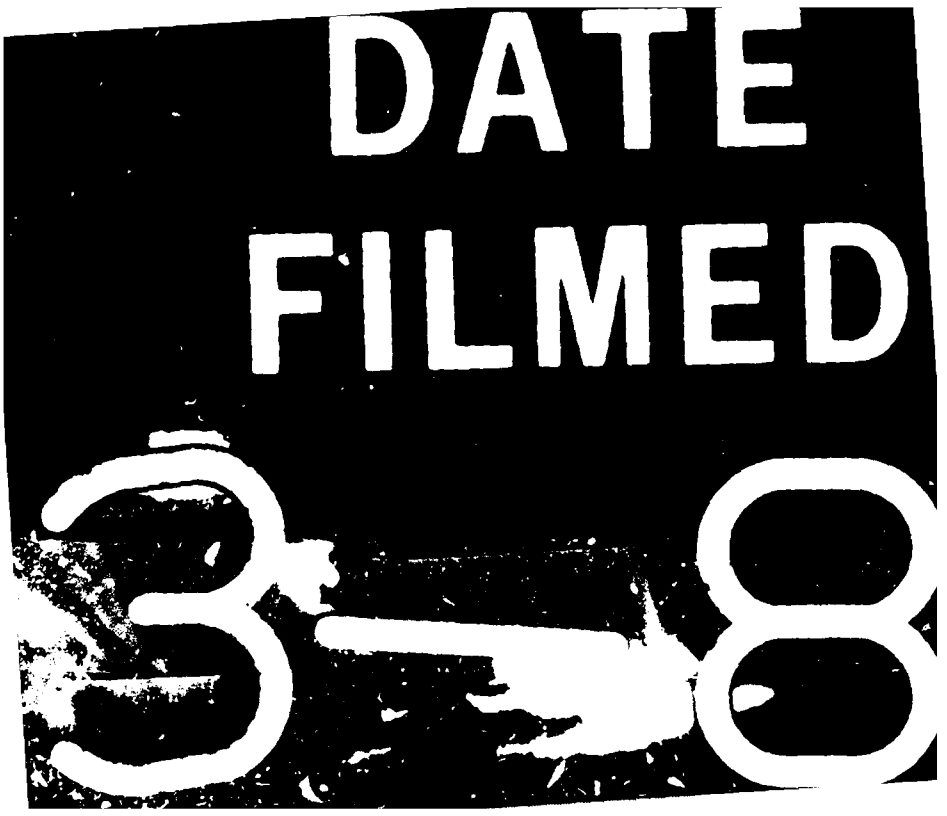
Another application is the following definition of stability (in the sense of stable sorting):

Suppose X, Y, and K are functions such that cod X = cod Y = dom K. Then X is a <u>stable rearrangement of Y with respect to K</u> when

$$(\forall k \in \mathrm{cod}\ K)\ X \mathbin{\dot\cap} P(\{k\}, K) \simeq Y \mathbin{\dot\cap} P(\{k\}, K) \quad .$$

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.


DATE
FILMED
3-88