Advanced Languages for Systems Software

The Fox Project in 1994

Robert Harper and Peter Lee

January 1994
CMU-CS-94-104

Carnegie Mellon

0

Advanced Languages for Systems Software

The Fox Project in 1994

Robert Harper and Peter Lee

January 1994
CMU–CS–94–104

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Also published as Fox Memorandum CMU-CS-FOX-94-01

This research was sponsored by the Defense Advanced Research Projects Agency, CSTO, under the title "The Fox Project: Advanced Development of Systems Software", ARPA Order No. 8313, issued by ESD/AVS under Contract No. F19628-91-C-0168.

Abstract

It has been amply demonstrated in recent years that careful attention to the structure of systems software can lead to greater flexibility, reliability, and ease of implementation, without incurring an undue penalty in performance. It is our contention that advanced programming languages—particularly languages with a mathematically rigorous semantics, and featuring higher-order functions, polymorphic types, first-class continuations, and a useful and powerful module system—are ideally suited to expressing such structure. Indeed, our previous research has shown that the use of an advanced programming language can have a fundamental effect on system design, leading naturally to system architectures that are highly modular, efficient, and allow re-use of code.
We are thus working to demonstrate the viability and benefits of advanced languages for programming real-world systems. To achieve this, we have organized our research into the three areas of *language design*, *compiler technology*, and *systems building*. This report describes the current plans for this effort, which we refer to as the Fox project.

Authors' electronic mail addresses:
Robert.Harper@cs.cmu.edu
Peter.Lee@cs.cmu.edu

Introduction

It has been amply demonstrated in recent years that careful attention to the structure of systems software can lead to greater flexibility, reliability, and ease of implementation, without incurring an undue penalty in performance. Specific examples of well-structured systems include the microkernel architecture of the Mach 3.0 operating system and the modular layered architecture of the x-kernel implementation of network communication protocols.

It is our contention that advanced programming languages — particularly languages with a mathematically rigorous semantics, and featuring higher-order functions, polymorphic types, first-class continuations, and a useful and powerful module system — are ideally suited to expressing such structure. Indeed, our previous research has shown that the use of an advanced programming language can have a fundamental effect on system design, leading naturally to system architectures that are highly modular, efficient, and allow re-use of code [12].

Unfortunately, in practice advanced languages typically have not been used to implement systems software. Until recently, advanced languages have lacked features that are crucial for systems programming, and their implementations have been too inefficient for use in practical systems. Furthermore, although the rigorous foundations underlying advanced languages should in principle provide many benefits for software development, such languages have rarely been used in the development of programs of realistic size and complexity. As a result, the advantages of advanced programming languages and program development techniques—greater reliability, ease of maintenance, better adaptability, improved code re-use, and so forth—have not been realized for systems software in the "real world". We believe that this has led to undesirable compromises in the design of present-day systems software.

We are thus proposing to demonstrate the viability and benefits of advanced languages for programming real-world systems. To achieve this, we have organized our research into three areas:

- **Language design.** The development of the foundations of programming languages, and their application to the design of new advanced programming languages.
- **Compiler technology.** The development of techniques for compiling and optimizing advanced programming languages.
- **Systems building.** The use of advanced programming languages in the design and implementation of real-world systems.

More specifically, we have two goals. The first is to advance the art of language design by using advanced languages to build systems software. Building real systems provides a discriminating test of language design principles by applying them in real-world settings. It also suggests new language features and modifications to existing features, while at the same time encouraging the study of semantic, type-theoretic, and implementation issues to uncover hidden complications and problems. And of course, successful implementations demonstrate the usefulness of advanced programming languages.

The second goal is to advance the art of software development by using advanced languages to build advanced systems software. By exploiting language structure, we expect to achieve more reliable, maintainable, and elegant software systems. System building also allows us to exhibit real code as proof of concept and demonstration of the viability of advanced languages, and provides a basis for studying the costs (especially with respect to performance) of advanced languages in systems software. Finally, we expect that this experience will suggest alternative organizations and techniques in the development of software systems.

Our approach thus far has involved the design and implementation of extensions to the Standard ML programming language, and use of these extensions in implementing a suite of efficient, industry-standard network communications protocols. For the next three years, we are proposing to continue this development of Standard ML and its use in the design and implementation of systems software. This includes, as in the past, network communication protocols, but now also networking applications, device drivers, and operating system kernels for embedded systems. As has been our previous experience, we expect the design of these systems to be affected in fundamental ways by the use of an advanced language. At the same time, the systems-building experience provides requirements for the language design, thus guiding the development of the theoretical foundations for the design of extensions to the language. Such foundations provide the basis for the development of techniques for compiling and optimizing programs, and the systems-building experience provides a real-world test of the effectiveness of optimizations. The compiler technology in turn enables aspects of system design and is an important consideration in the design of new language features.

These three areas, though separate in technical content and approach, are carried out in a cooperative manner that promotes an extraordinarily high level of interaction and feedback. (Indeed, all of the research personnel are involved in at least two, and usually all three areas.) Thus we have been extremely successful in realizing a crucial synergy that we believe can help to define fundamentally new approaches to system software and language design.

Technical Approach

The central assumption underlying our approach to the project's research is as follows: a systematic, principled approach to language design can and will lead to an advanced programming language that will provide demonstrable benefits for system software development, and in fact will improve the practice of software development in general. By *systematic* we refer specifically to the use (and development) of mathematical frameworks in defining and reasoning about the semantics of programming languages. When we say *principled*, we mean that the design of the language should be guided by a certain aesthetic, inspired primarily by the formal connections between mathematical logic and the fundamental nature of computation.

In fact, there is already overwhelming evidence that this assumption is true, *but only in those application domains for which advanced programming languages can be usefully applied*, such as list processing, program translation and transformation, and other (usually small) applications in symbolic computation. The key challenge, then, is to demonstrate that the assumption holds up in larger "real-world" application domains, and of particular interest for us is the domain of systems software, including operating systems, network communications, and real-time embedded systems.

Perhaps the most obvious technical hurdle is performance; advanced languages such as Standard ML have hard-to-implement features such as higher-order functions, polymorphic types, first-class continuations, concurrent threads of control, and a powerful module system. Hence, the development of new compiler technology will be necessary to overcome the apparent costs of such language features. Less obvious is the problem of expressiveness of the language for performing necessary tasks in systems programming. For example, our previous experience in implementing network communications protocols in Standard ML exposed the requirement for precise control over data layout; perhaps surprisingly, such a feature has been missing from advanced programming languages and thus had to be developed by us (and is still under development). Finally, a claim for the viability of advanced languages for systems software must go beyond simply implementing practical systems in such a language. The benefits in software engineering must be demonstrated, such as reduced development costs, greater reliability, improved code re-use, and so forth. But perhaps even more importantly, it must be shown that *the use of advanced languages can lead to important new ideas and structuring concepts in the design of future systems.*

All of these considerations lead us to organize our research around the three areas of *language design*, *compiler technology*, and *systems building*, and within each of the areas, into *near-term* and *long-term* research activities. These three areas, though separate in technical content and approach, are carried out in a cooperative manner that promotes an extraordinarily high level of interaction and feedback. (Indeed, all of the research personnel are involved in at least two, and usually all three areas.) Thus, we have been extremely successful in realizing a crucial synergy, leading to good research progress especially at the "boundaries" of the three areas. The distinction between near-term and long-term activities is also useful in the project organization. The near-term activities provide an immediate focus for project personnel, leading to a team atmosphere and the development of "demonstration systems". Also important is the fact that this kind of experience provides immediate guidance and feedback for ideas generated by the long-term research work. In turn, the long-term activities take place in an environment that is less constrained by immediate real-world requirements, thereby fostering the development of radically new ideas in system and language design.

In what follows, then, we describe in greater detail our specific research plans, both near-term and long-term, in each of the three research areas.

Systems Building

Near Term

As a first step we plan to complete our implementation of the TCP/IP protocol stack. The most important remaining feature to be implemented are IP options and fragmentation/reassembly. Once this is completed we will conduct a careful performance analysis of our implementation using a 10Mb/s isolated Ethernet connection between dedicated processors. This will provide a firm basis for comparing our implementation with the x-kernel and Mach implementations, and will provide essential information for improving the performance of our code. Since we expect that the 10Mb/s capacity of current Ethernet technology will be a limiting factor, particularly on the emerging fast RISC processors, we also plan to implement either an ATM or high-performance Ethernet device layer, and to again measure the performance of our system and competing implementations. Our goal is to demonstrate that SML is not significantly less efficient as an implementation language for high-speed network protocols than C, and to understand such limitations and performance bottlenecks as we do encounter. This information will provide an important reference point for future work both in the systems area and in language design and implementation.

Upon completion of the implementation and evaluation of the TCP/IP protocol stack, we will expand our work on protocol design by considering higher-level protocols such as RPC for remote procedures and ISIS-like protocols for reliable distributed computing. These protocols can then be used to experiment with higher-level distributed computing paradigms such as have already been explored by the Venari Project at CMU and the DML effort at Cornell. This will establish an important link with these closely-related efforts, and will provide further evidence for (or against) the viability of SML as a vehicle for distributed computing. We are also considering pushing the protocol work further to include the development of network file system implementations similar to Sun's NFS network file system or the CMU Coda mobile computing file system. These systems will again provide an important case study for systems programming in SML, and are an enabling technology for our longer-term proposed work in embedded systems (more on which below). Finally, we will explore other high-level applications of the FoxNet network technology by implementing ML-to-ML communication, the ability to send and receive ML data objects over the network. This provides a natural arena for exploring the use of demand-driven transfer protocols for decreased communication latency by providing first those portions of structured data items that are needed first to complete a particular computation.

The current FoxNet is built on top of the Mach 3.0 kernel which provides for us the basic primitives required to communicate with a network device driver. Crossing the kernel boundary is an expensive operation, both because of the context switching overhead and because of the need to pass in and out of the ML run-time system. An important direction for future research is to quantify the overhead associated with the kernel boundary and to explore the possibility of reducing reliance on the kernel by putting more functionality into the SML implementation. It may also be possible to exploit the type safety of the SML language to allow for the efficient execution of user code in kernel space to avoid the overhead associated with typical protection mechanisms. These investigations are the starting point for gaining a deeper understanding of the potential for using SML on a "bare machine" to implement device drivers and kernels, particularly for embedded systems. They also provide an opportunity to explore kernel abstractions from a linguistic standpoint, an important first step in understanding the implementation of composable kernel services in SML.

Long Term

An important long-term goal of our work is to explore the design of new software systems from a "language-centric" viewpoint. Here again our current work makes clear that the mechanisms provided by Standard ML suggest fresh new perspectives on software design that are difficult o impossible to achieve in low-level languages such as C. For example, we make essential use of higher-order functions in our implementation of TCP/IP, for example to provide "staged" computation whereby network services can be specialized on connection-specific data to improve performance on data transmission. More significantly, we make essential use of the SML module language to express the structure of the protocol stack and to ensure that microprotocols are fully composable. An important goal is to explore further the use of such high-level language constructs in the design and implementation of systems software, with the hope of improving not only their utility and elegance, but also their performance, relative to existing approaches.

We see microkernels as a first step towards building a completely modular operating system from a set of composable services. However, just as the idea of a microprotocol in the x-kernel is a design principle, rather than a formally realized construct, the modularity of microkernels is largely a matter of programming practice enforced using hardware protection mechanisms and system calls. By working in a language such as SML it should be possible to give full expression to the modular structure of operating systems in a manner roughly parallel to that used in the FoxNet protocol stack. This would open the way to building embedded operating systems (for example. for robotics or industrial control systems) from "off-the-shelf" software components. Furthermore. since SML is a type-safe language, many of the precautions necessary in a general operating system environment need not be taken in the context of an embedded operating system, making it possible to eliminate context-switching overhead.

We therefore view the exploration of the construction of embedded systems in SML as an important long-term direction for the project. The short-term goals mentioned above are important stepping stones towards this longer-term goal. For example, experience gained in exploring the kernel boundary in the FoxNet implementation will include an understanding of the design of the interface to kernel services. We will also be exploring execution of user code in kernel space (for example, for packet filters), relying on the type- and storage-safety properties of SML programs to avoid high-overhead protection mechanisms. This naturally leads to formalizing a number of operating system services as composable modules, and to the construction of embedded systems as the composition of a number of services. Possible application areas are robot controllers, a telemetry station, or other small, non-critical, self-contained systems. These explorations are likely to involve relatively weak real-time constraints, giving us the opportunity to explore meeting real-time requirements in a language with automatic storage management.

Language Design

Near Term

In the near term our work on language design will continue to emphasize the enrichment of Standard ML with extensions to support systems programming, and careful study of the associated semantic and type-theoretic properties of these extensions.

In the first phase of the project we extended SML with first-class continuations, an essential step towards supporting co-routines and threads. Our investigations uncovered a serious flaw in the naive (and widely-accepted) typing rules for `callcc`, exposing an insecurity in the type system. Detailed analysis of the failure of soundness led to a correct typing rule for `callcc` and suggested a fundamental analysis of the typing properties of operations that have control and store effects.

We also undertook a study of the typing properties of the CPS transform, a compilation technique of fundamental importance to functional languages. In addition to shedding light on the semantic problems arising from the combination of polymorphism with control effects, this work is an important step in developing a general theory of compilation for typed languages. Current compilers for SML take little or no advantage of type information during translation, relying only on the assumption that the program is well-typed, and not on specific information about the types of constituent phrases. Our study of the typing properties of the CPS transform, and in particular the relationship between explicit and implicit polymorphism, is a first step towards developing a type-based compilation method for SML and related languages. (Concurrently with this work Leroy has developed a type-based translation that attempts to minimize the use of "boxing" in the compilation of polymorphic programs, lending further support to our conviction that type-based methods are essential to the efficient compilation of SML programs.)

The implementation of TCP/IP in SML suggests that an important limitation of Standard ML in connection with systems programming is the inability to gain tight control over the layout of compound data structures and to limit the amount of copying associated with manipulating these structures. Experience in building network protocols shows that copying data is an important source of inefficiency, and must be carefully controlled to achieve reasonable performance. Furthermore, the definitions of network protocols and operating systems interfaces place very detailed constraints on the layout of data structures such as network packets and IPC messages. Since SML provides only a very abstract view of compound data structures, there is a fundamental mismatch between the expectations imposed by the external world and the representations that are kept private to the SML compiler. To bridge this gap it is necessary to extend the SML type system with ***flat types*** and ***locatives*** that provide the ability to gain precise control over data layout, including satisfying external constraints on the representation of compound values, and to support references into the "middle" of compound objects, providing many of the benefits of pointer arithmetic without sacrificing type safety. The fundamental obstacle to adding these features to SML is how to combine them with polymorphism, modularity, and separate compilation. Specifically, there is a fundamental conflict between the need to control data representation and the need to compile programs in the absence of complete information about the types of the values they manipulate.

We plan to investigate two approaches to this problem. One is based on extending the tagging scheme already provided by SML/NJ to support garbage collection and polymorphic equality. By enriching the set of type tags associated with data values in SML we can introduce "flat" types whose representations are guaranteed to be free of pointers (other than those explicitly introduced by the programmer), and will satisfy certain layout guarantees. Each "flat" data object comes with a top level descriptor for use by the garbage collector and primitive operations for manipulating such flat structures. This implementation will provide important experience with these new language features, and serve as a testbed for experimenting with design alternatives. But it is not an acceptable long-term solution, relying as it does on an elaborate and intrusive tagging scheme. We therefore plan to build on our previous work on type-based compilation methods for SML to develop an approach to compiling "flat" types that relies on a translation into an explicitly polymorphic language that passes type parameters at run-time. Layout guarantees are achieved through the use of *non-parametric* polymorphic operations (*i.e.* those that make essential use of their type parameters, in contrast to *parametric* operations that work uniformly for all types). This type-based approach seems promising, and sheds considerable light on a number of issues in representation analysis and efficient compilation of SML programs. In addition to developing the type-theoretic and semantic basis for this new approach to compilation, there remain important language design questions concerning how best to program low-level manipulations such as arise in building network protocols or operating systems services.

The need for subtyping of some form has arisen in various contexts in our investigations. For example, the type of expressions whose value will be known at compile time can be considered a subtype of arbitrary expressions. Another example comes from the work on explicit data representations (flat types), where one might consider the type of small integers to be a subtype of the type of arbitrary integers. Associated with this subtyping relationship is a coercion function that changes the representation of a small integer to that of a large integer. ML does not currently provide a notion of subtyping; two common problems with subtyping mechanisms are the non-existence of *principal types* and the undecidability of type-checking. For example, the negation function on integers maps small integers to small integers and large integers to large integers, but ordinarily there is no single type expressing both properties. In general terms, these difficulties are addressed by *refinement types* which combine the semantic elegance of subtyping and intersection types with the polymorphism of ML while preserving the decidability of type inference and the existence of principal types. Unlike general program analysis for the purpose of compilation, refinement types also provide a language for communicating complex program properties across module boundaries. We will investigate particular instances of the refinement types idea for the example applications mentioned above. We further hope to demonstrate the practicality of refinement type inference by a realistic implementation within the SML of New Jersey compiler; currently only a small prototype exists.

It can be readily seen from the foregoing discussion that there is a tight interplay between language design (including semantics and type theory), language implementation, and systems building. This interplay is a vital aspect of the project that we expect will continue to play a central role.

Long Term

The most important long-term direction is the design and implementation of the successor language to Standard ML, dubbed "ML2000" or "Millennium", a joint effort with several other researchers in the ML community (see the Related Work section for more information on this collaborative effort). The goal of the ML2000 design is to consolidate the substantial advances in type theory and semantics that have been made over the last 15 years in the design of a powerful new programming language that will exceed Standard ML in expressive power. ML2000 will serve as a vehicle for experimentation with new language features, implementation techniques, and applications software. Research conducted under the present contract has already contributed substantially to the ML2000 design, and we expect that our work will be of central importance to the development of the language.

The first phase of the ML2000 design consists of establishing the overall structure of the language, which amounts to settling on the type theory of the language. At the time of this writing there are two extant proposals, one which preserves the existing separation between the module and core level, and one which eliminates this distinction. The advantage of the stratified approach is primarily a matter of conceptual simplicity — large programs are viewed as amalgams of small programs which are combined using the constructs of the modules language. The advantage of the integrated approach is that it eliminates certain redundancies that are found in the stratified approach, notably the distinction between a record and a structure with no type components. The integrated approach is substantially more powerful than the stratified approach because it admits the possibility of choosing the implementation of an abstract type based on run-time, rather than compile-time, considerations. Until recently it was thought that a first-class modules system would necessarily violate the "phase distinction", namely the ability to type check programs prior to evaluating them. However, recent advances by the proposers establishes that this is not the case, and a first-class modules system supporting compile-time type checking is indeed feasible.

An important area for investigation is to develop the type-theoretic basis for higher-order modules with SML-style sharing constraints. The notion of a sharing constraint is unique to the SML modules system, and has proved itself to be of vital importance to software development in SML. (In particular we make heavy use of sharing constraints in the signatures for the FoxNet implementation of the TCP/IP network protocol.) This feature of SML has until recently defied adequate type-theoretic analysis. However, we have made significant progress toward understanding the role of sharing constraints in higher-order modules systems through the notion of a *translucent sum* type, a hybrid between the well-known *opaque* and *transparent* sum types (also known as existential and sigma types, respectively) of type theory. Translucent sums are the key to understanding not only explicit sharing constraints, but also to supporting separate compilation and providing fine-grained control over the "degree" of abstractness of a module. Initial steps in this direction have been taken, but much more work remains to be done. In particular it has been established that the type checking problem for an integrated calculus of higher-order modules with sharing is undecidable, raising the question of how best to build a practical type checker for this language.

Another important area of investigation is the relationship between modularity and object-oriented programming. Since the bulk of object-oriented methodology can be viewed as desiderata for modular programming (see the Related Work section for further discussion of this point), it is natural to expect an integration between modules and objects. We are currently investigating this topic in the ML2000 design. Here again the question of a stratified versus an integrated module language is of central importance. In a stratified system we are able to separate the treatment of objects from the treatment of modules, at the cost of a certain degree of redundancy. This approach has the virtue of not confusing what may well turn out to be two distinct features of programming in the large. In the integrated system we achieve a uniform treatment of modules, objects, and records, an appealing parsimony of concepts. The type-theoretic basis for both of these approaches is far from clear, but it is safe to say that the fully integrated approach is substantially more complex than the stratified view. Achieving a full understanding of the issues and trade-offs involved is an important issue for the design of ML2000.

Once the initial design of ML2000 is in place, the next step will be to build an implementation of it so that we can gain the necessary experience with it. This aspect of our efforts ties in with our long-term goals on language implementation which are discussed in more detail below.

In addition to the design of ML2000 itself we plan to experiment with the formalization of its semantics in the LF logical framework. Briefly, LF is a formalism for specifying formal systems such as the ML2000 static and dynamic semantics. A formal system is presented as an LF signature (related to, but not the same as an SML signature). This signature may then be presented to the Elf implementation of LF (developed by Pfenning and his students) which provides support for building type checkers, compilers, evaluators, and proving properties of the language. We plan to formalize the semantics of ML2000 in LF and to use Elf to experiment with the language and to provide an executable specification of the ML2000 semantics.

Compiler Technology

Near Term

The preceding discussion illustrates how our approach to research in language design also involves the development of new techniques for analyzing, optimizing, and compiling programs into efficient object code. This integrated approach, in which the formal semantics is used not only to study properties of the language but also to guide the development of its implementation, promotes the development of semantically well-grounded compilation techniques. At the same time, it allows implementability and efficiency considerations to influence the language design process at its earliest stages. Furthermore, we have found that this approach provides an effective way to address thoroughly the requirements on both expressiveness and efficiency that are raised by our systems-building experiences.

As mentioned above, a specific example is the development of "flat types". Our experience in the construction of network protocols in Standard ML exposed the inability to control precisely the layout of data structures in memory. This led to the development of flat types and a compilation strategy for this feature. Significantly, the compilation technique is also likely to be useful in optimizing polymorphic functions and register usage in general, thereby largely reducing or even eliminating the costs normally associated with polymorphic types.

In addition to the development of compiler technology for specific language features and extensions to Standard ML, we will also continue work on general semantics-based methods for analyzing and optimizing SML programs. One major focus will be the development of semantics-based alternatives to dataflow analysis and traditional optimization techniques. In contrast to dataflow analysis, semantics-based techniques are firmly grounded in the semantics of the programming language, and in particular they can be proven to be sound and effective. One of the primary technical problems, however, is that control flow is not a static property of Standard ML programs (this is generally true for languages with higher-order functions), as it is for most conventional programs written in C or FORTRAN. To address this issue, we have developed several semantics-based analyses that compute conservative approximations of the control flow of SML programs. If these analysis techniques are found to be practical, they would enable classical global optimizations, which to date have not been attempted in a general way for functional programs, as well as many other standard program transformations. In our earlier studies of the code generated by the Standard ML of New Jersey compiler, we have determined that such global optimizations could lead to dramatic improvements in performance of SML programs, and thus we plan to implement these analyses and a conduct a thorough study of their effectiveness.

We also plan to continue work on another semantics-based analysis method called "set-based analysis". Traditional analysis techniques are based on domains of approximate values, and thus the information inferred by them is fundamentally limited by the choice of these domains. In contrast, a set-based analysis computes constraints on the values that each variable in a program can have. A separate constraint solver can then be used to infer (conservatively) the set of actual values that each program variable could possibly have during any execution of the program. Thus, no approximation of values is made, and the precision of the analysis is determined completely by the sophistication and effectiveness of the constraint solver. This method has a number of advantages over previous semantics-based analysis techniques, perhaps the most important being that it can be implemented efficiently without sacrificing the precision of the analysis. A key application of set-based analysis will be to provide a so-called "binding-time analysis", which will provide crucial information for the various late compilation and code specialization techniques that we plan to explore in the longer term. Another anticipated application is a form of range analysis, which in addition to improving the performance of tight loops (such as checksum routines in our network code), and would also be valuable in optimizing programs involving flat data structures.

Finally, as we have done in the past, we plan to make large-scale quantitative measurements of many of the systems we build. This is so that we can understand the behavior of the systems at a detailed level, and also to study the effectiveness of the various analysis, optimization, and code generation techniques. In analogy with work in compilers for C and FORTRAN, our goal is to fully exploit the low-level features of the target machine in order to maximize the potential performance.

Long Term

In the long term, we plan to fully exploit the formal foundation of the programming language by exploring new methods for code generation and optimization based on semantically well-grounded analysis and transformation techniques. In particular we plan to explore the use of "late" or "delayed" compilation, including link-time and run-time code generation and specialization, to improve the performance of systems software. The main goal is to achieve the performance of an integrated software system without sacrificing the software engineering benefits of a clean, modular structure. For example, by delaying aspects of compilation to link-time, the code of a parameterized module (called a "functor" in Standard ML) can be specialized to the specific types and values that become available when the module is instantiated at link time. This has the effect of eliminating a significant portion of "boundary" between separately compiled modules, thereby greatly reducing the overhead associated with modular structure. Indeed, we believe that in the case of network communications, this will result in a form of automatic integrated layer processing (ILP). Already, our current work on developing a modular implementation of the TCP/IP network protocol suite provides strong evidence that modularity need not compromise performance, and will serve as a "test-bed" for evaluating the importance of advanced optimization techniques for systems software.

We have found that similar kinds of gains are also possible by delaying low-level optimizations and code generation to run-time. Many compiler optimization techniques depend on static analysis to determine invariants about a program's run-time behavior. As a result, a great deal of research has gone into developing various approaches to static program analysis. Despite good progress, the analyses tend to be excessively conservative in practice, thus making it difficult for a compiler to achieve a thorough optimization of programs. This is, of course, a fundamental problem since most aspects of program run-time behavior are undecidable. Also, as a practical matter, further compromises in accuracy must be made in order to cope with the complexity and inefficiency of many analysis algorithms. An alternative approach is to defer at least some of the analysis and optimization (and therefore also code generation) to run time. The challenge here is to limit the overhead of performing run-time compilation so that overall performance gains can actually be realized in practice. For this, our approach will be to adapt techniques from partial evaluation, in a sense "specializing" the code generation process for particular pieces of code. As with link-time delayed compilation, we can see many opportunities in our network implementation for run-time compilation. We thus plan to explore how practically these opportunities can be exploited.

Another long-term plan is to develop a prototype implementation of ML2000, including both the front end, which provides type checking and static elaboration, and the back end, which translates programs into code. Unlike the near-term effort, this prototype development will be free from the basic design constraints imposed by the Standard ML of New Jersey system. In particular, we will be free to develop our own intermediate language, along the lines discussed earlier, to best take advantage of our work on language design and semantics-based analysis techniques, as well as our experience in building real systems in an advanced programming language. In addition to this we plan to experiment with an implementation of the ML2000 semantics in the LF logical framework. This will not only allow rapid prototyping of type checkers, evaluators, and compilers, but also permit representing formal proofs of their correctness and, more generally, the formalization of the meta-theory of ML2000. The environment for this undertaking will be provided by the Elf implementation of LF (developed by Pfenning), which has shown itself to be practical for related, but simpler pilot studies in theory of ML-like languages.

Related Work

Software Development in Advanced Languages

A number of previous projects have used high-level languages to implement real systems. These include the Cedar system, written in Mesa at Xerox PARC [94]; the Topaz system, written in Modula-2+ at DEC SRC [87, 121]; the Trestle window system, written in Modula-3 at DEC SRC [19, 98, 82]; the Symbolics and Xerox Lisp machines; the Swift system, written in CLU at MIT [24, 77]; and various real-time and embedded systems written in Ada [132].

Each of these languages provides some advantages over lower-level languages such as C, but none provides all of the features found in Standard ML, as the following table summarizes. (A solid bullet (●) indicates that the language supports the indicated feature; an open bullet (○) indicates that it supports a weak or limited form of that feature.)

Language	First-class functions	Compile-time type checking	Poly-morphism	Parameterized modules	Formal semantics	Automatic storage mgmt
C		○				
C^{++}		○	●			
Lisp	●					●
Scheme	●				●	●
CLU		●		●		
Ada		●		●		
Mesa		●				
Modula-2+		●				●
Modula-3		●	○	●		●
Eiffel		●	●	●		
Standard ML	●	●	●	●	●	●

A few comments are in order. Although C lacks higher-order functions, it has a limited ability to pass functions as parameters and results through the use of a *code pointer*. However, all such functions are statically defined; one cannot create functions with dynamically-determined parameters at run time. A limited form of compile-time type checking is available in C, but is extremely weak due to the unrestricted cast mechanism. All of the problems with C are inherited by the C^{++} design. In contrast to C, the C^{++} language provides a form of modularity through the class mechanism, and a form of polymorphism that is only distantly related to that found in ML. Modula-3 provides a form of polymorphism through the **REFANY** type, but again this is only distantly related to the kind of polymorphism available in ML. Modula-3 provides parameterized modules through the generic mechanism. Similarly, Eiffel provides subtype polymorphism based on inheritance hierarchies and a form of modularity through the generic class construct.

The other languages (with the exception of Lisp) are all thoroughly imperative; in practice Common Lisp programs are written in an imperative style, even though higher-order functions are available. Programs written in Standard ML, on the other hand, are mostly functional: they use few assignment statements and make liberal use of higher-order functions. In this respect, systems software implemented in Standard ML is qualitatively different from that written in the other languages mentioned above.

Omitted from the above diagram is the (ill-defined) property of being "object-oriented", a subject of considerable current interest. The languages C^{++}, Common Lisp, and Modula-3 are all considered to be object oriented, but the sense in which this is meant differs in each case. The main properties associated with the term "object oriented" are these:

1. Data abstraction. Ability to ascribe explicit interfaces to bodies of code, and to enforce abstraction boundaries associated with these interfaces.

2. Localization of state. Ability to control access to mutable storage through the use of controlled interfaces and scoping mechanisms.

3. Subtyping and subtype polymorphism. Ability to handle objects based on particular "behaviors" without regard to what other behaviors they may support.

4. Inheritance. Ability to define the code of one object by a systematic modification or extension to that of another object.

Although not normally regarded as an "object-oriented" language, Standard ML supports many of the above features to varying degrees. Most importantly, the sophisticated modules language of

SML strongly supports data abstraction, and strongly encourages the use of modular programming with explicit interfaces. The presence of higher-order functions and the modules system together provide strong support for localization of state; in particular, we may always localize assignable variables to the operations that use them, and we may easily create fresh mutable cells at run time to avoid unwanted interference between instances. However, SML provides only limited support for subtyping, a decided weakness that is to be rectified in the design of ML2000. (It does provide a form of subtyping at the modules level that has proved to be extremely useful in practice.) The importance of inheritance is subject to debate. ML provides a very limited form of code and specification inheritance through the **open** and **include** constructs at the modules level, but does not support method specialization. These limited mechanisms play an important role in the FoxNet implementation, indicating that further work on understanding the use of inheritance is needed; we plan to undertake such an investigation in the ML2000 design.

We have not included in the above discussion any mention of "lazy" functional languages [56] such as Miranda [128, 130], Haskell [55], or FL [11]. Although some experimental work has been done on building "purely functional" operating systems [51, 129, 125], these languages have by and large not yet been used in realistic systems programming. One thesis of the present work is that "lazy" languages are not appropriate vehicles for building large systems, particularly systems programs, because they cannot deal adequately with external notions of state or temporal progress of events. Of course, this is a working hypothesis rather than an established fact. More work needs to be done to assess the viability of lazy functional programming, a task that we do not undertake here.

A closely-related effort to ours is the work of the Standard ML group at AT&T Bell Laboratories, led by David MacQueen. The overall charter of their group is to improve the quality of the software written by AT&T in support of its corporate activities. Their effort focuses on the use of Standard ML and, more broadly, the role of mathematical logic and semantics in the design, implementation, and verification of software and hardware systems. The Standard ML of New Jersey compiler [9] (on which this project relies) was written jointly by MacQueen and Andrew Appel of Princeton, along with their co-workers and students. An important development is the extension of SML with concurrency primitives, called CML, by John Reppy [113, 111, 112] and its application to windowing systems [38]. The message-passing approach of CML provides a counterpoint to our own ML-Threads system [27], which stresses a more imperative style of programming. The Fox Project has a established a close interaction with AT&T that we expect to continue in the future.

The work on Distributed ML at Cornell [66, 28] and the Venari Project at CMU[136, 100] are of particular relevance to the Fox Project. Both of these efforts are concerned with developing linguistic support for distributed computing based on Standard ML. The DML language is an extension of CML [113] to support an ISIS-like [13] approach to reliable distributed computing [67]. The Venari project focuses on implementing atomic transactions [78, 97] in SML, with the emphasis being on exploiting the modules system of SML to provide a "mix and match" approach to transaction functionality [135, 40]. These efforts are very similar in spirit to the Fox Project, and provide an important source of experience and technical contributions for our work. In particular we expect the work of the Venari and DML projects to be potential applications for the network and kernel technology that is currently under development.

The design of the FoxNet implementation of TCP/IP has drawn heavily on the design of the University of Arizona x-kernel [102, 101]. The design principles underlying the x-kernel served as a model for our implementation of the FoxNet protocol stack. In contrast to the x-kernel, which is written in C, we are able to give full expression in Standard ML to the informal structuring principles behind the x-kernel. For example, a fundamental concept in the x-kernel is the idea of ***composable microprotocols.*** The idea is to decompose monolithic protocols such as IP into a number of discrete components that are composed to build the full IP protocol and which may also be used in a "mix and match" fashion to build more efficient special-purpose protocols. In a C implementation the idea of a microprotocol is entirely a design principle, and cannot be given linguistic expression. However, the SML modules system allows us to represent composable microprotocols as "functors", or module constructors, that precisely capture the idea of composability. In particular, the SML compiler ensures that protocols are combined sensibly, and that a microprotocol is indeed independent of the details of any underlying layer. To take another example, the x-kernel exploits the notion of an "upcall" to avoid copying and context-switch overhead when passing between protocol layers. This idea can be elegantly expressed in SML using higher-order functions: the receive operation of a protocol is passed an "upcall handler" as a higher-order function which is applied to the incoming data upon its arrival. Since functions can be dynamically created in ML, it is possible to define specialized handlers based on run-time information. We may also exploit higher-order functions in SML to perform such operations as specialization of the send code on a particular connection, avoiding the interpretation overhead that would otherwise be required for call to send. As a final example, we note that one focus of the x-kernel project has been to improve the efficiency of multi-layer protocol implementations through the technique of Integrated Layer Processing [3]. One important area for future exploration is to achieve the efficiency of ILP without sacrificing the software engineering benefits of a layered, modular design. The SML modules language presents significant opportunities in this area; in particular, it seems promising to consider the possibility of generating specialized code for each instance of a parameterized module, rather than compiling them uniformly for all instances.

In our project we have made use of some of the facilities of the Mach 3.0 microkernel [4]. In addition, we regularly compare the performance of our implementation of TCP/IP to that of the regular Mach and of a highly optimized implementation developed by Maeda and Bershad [81] (who are now at the University of Washington). As we proceed to implement progressively more of the functions of Mach, Mach itself will be less and less important to us; we expect the embedded operating system we are planning to design to be substantially more modular than Mach, for example. One operating system project that has really focused on modularity is the Spring operating system project [62, 63, 99] underway at Sun. Unlike Spring, we propose to use SML instead of C^{++}, and we plan to focus on modularity more than on having an "object oriented" operating system (see above). Our focus is an operating system which can be assembled out of different modules at compile time with assurance that all errors introduced by incorrect composition will be detected by the compiler. Unlike the Spring project, we do not believe we have the resources to produce a complete unix-like production operating system, instead concentrating on small, prototype operating systems for embedded systems.

Other operating systems efforts that in some ways resemble ours include Wahbe's sandboxing [133], used to safely execute user code in kernel space. We do plan to safely execute user code in kernel space, but we expect our approach, unlike theirs, to have a strong theoretical foundation based on language semantics and linguistic theory, as well as advanced compilation technology.

Language Design

Standard ML has its origins in the pioneering work of Robin Milner on the design of the metalanguage, ML, for a proof system, LCF, for reasoning about programs [39, 89]. These ideas were further developed by Rod Burstall and David MacQueen [15, 79], culminating in the design of Standard ML [93, 92]. (For a more complete description of the history of ML, see Milner's article [90] and the historical appendix of *The Definition of Standard ML* [93].)

The design of ML is strongly influenced by the work of Dana Scott on domain theory [122, 123] and, in particular, the *Logic for Computable Functions (LCF)* [91]. Also of central importance is the work of Landin [68, 69] and Reynolds [114, 115]. Subsequent work on ML and its derivatives has been strongly influenced by Plotkin's work on operational [106, 108] and denotational semantics [107, 109], by Reynolds's work on polymorphism [117, 118, 119], and Martin-Löf's and Constable's work on constructive type theory [83, 84, 26].

Notable among the off-shoots of the original work on ML is the development of CAML, a dialect of ML [36]. The CAML effort has been an important source of ideas in the design and implementation of type functional languages (see [29, 110, 76, 73] to name but a few). The CAML effort is being carried forward in the development of CAML-Light [71, 75], a "lightweight" implementation of CAML which is currently being extended with a new modules system [74].

Cardelli's dialect of ML [16] was a direct precursor to Standard ML, and the beginning of an important program of research into the type-theoretic foundations of programming languages [21], leading to the design of Quest [17] and Modula-3 [19, 20]. Quest represents the current state-of-the-art in type-ful programming languages, and is an important reference point for the design of ML2000. Though less ambitious, Modula-3 [19, 98] has been used to build "serious" systems software, notably the Trestle window system [82].

Recent work by Mitchell and his co-workers [96] has focused on extending SML with object-oriented features, in particular subtyping and inheritance. The initial motivation for this work was to address the problem of rapid prototyping, in particular bridging the gap between experimental and production software. This effort is the current state-of-the-art in combining object-oriented programming with the modular programming style exemplified by Standard ML.

Fundamental studies on the type theory of languages with sophisticated modules systems and object-oriented features is a subject of considerable current research. The type-theoretic basis for the SML **sharing** construct has recently been clarified by Leroy [74] and Harper and Lillibridge [48]. The work of Cardelli [17, 2, 1, 18], Pierce [105], and Bruce [14] on type systems for object-oriented languages is of central importance to assessing the role of objects in advanced ' ɔe-theoretic languages.

The bulk of our effort on ML2000 will be conducted in collaboration with the other members of the ML2000 design team: Andrew Appel (Princeton), Luca Cardelli (DEC SRC), Carl Gunter (Pennsylvania), Xavier Leroy (INRIA Rocquencourt), David MacQueen (AT&T), John Mitchell (Stanford), Jon Riecke (AT&T), and Mads Tofte (Copenhagen).

Compiler Technology

Much of our work in compiler technology is based on the Standard ML of New Jersey system (SML/NJ), developed by Andrew Appel and David MacQueen [8, 80]. This compiler, which is publically available from AT&T Bell Laboratories, represents the culmination of over 20 years of research on compilers for functional programming languages. As a research vehicle, SML/NJ is convenient because it is written in a modular style, thus making it relatively straightforward to modify and extend. Also, the system supports code generators for a wide variety of CPU architectures, including all of the major 32-bit RISC machines. Work on retargeting the system to newer machines is ongoing at many institutions. Our improvements and extensions to the SML/NJ are carefully coordinated with AT&T and regularly incorporated into the standard distribution of the system.

A key feature of the SML/NJ is its use of an intermediate language based on lambda terms written in continuation-passing style (CPS). The application of CPS in the implementation of programming languages was described by Reynolds [116] and used in the original compiler for the Scheme programming language by Steele [124]. Later, Kranz *et al.* attempted the construction of a production-quality Scheme compiler based on CPS [65, 64], and this compiler directly inspired the development of the SML/NJ system, as well as an extension of the CPS concept to "closure-passing style" [7, 6]. Danvy and Filinski have recently developed a thorough account of transformations from various lambda calculi into CPS [31]. The main advantage of CPS as an intermediate representation is that it is semantically well-understood while at the same time allowing low-level details such as register usage and closure representations to be represented explicitly. Thus, CPS is amenable to analysis by semantics-based methods while still supporting optimizations at the machine level (such as register allocation). Much of our research in compiler technology has centered around CPS and the transformation of typed programming languages into CPS, and in the future we expect to use CPS as a starting point in exploring alternative intermediate representations.

Work on other compilers for ML and functional programming languages also has an affect on our work. The implementation of CAML is not based on CPS but rather a translation of programs to an abstract machine [126]. This abstract machine, called the Categorical Abstract Machine (CAM) [29], along with an associated translation scheme for source programs, are systematically derived from the operational semantics of CAML, thereby ensuring the correctness of a substantial portion of the compiler. The style of operational semantics is very similar to that used for Standard ML, and hence the implementation techniques are relevant to our work. Also, work on the compilation of lazy functional programming languages [10, 57, 103] such as Haskell [55] is also related to our research, particularly in recent years as the Haskell community has added more and more features for simulating imperative programming and strict function application.

There is a growing body of research in types-based analyses, which are related to our own work on typed intermediate languages, flat types, analysis of data representations, and set-based analyses. These include the work on types-based "unboxing" optimizations by Leroy [72], tagging optimizations by Henglein [53], and tag-free garbage collection by Tolmach [127]. In the sense that unboxing and tagging optimizations can be viewed as type coercions, the notion of "soft typing" [5, 22] is also related. There is also a strong connection here to set-based analysis, particularly in the case of Henglein's approach to analysis [52], which also uses constraint-solving in the implementation of the analysis. Much of our other work on semantics-based analyses is based on the large body of recent research on abstract interpretation, the formal framework for which is usually credited to the Cousots [30]. The more recent work on operational frameworks for abstract interpretation by Deutsch [32, 33] forms the basis for some of our current developments.

A considerable amount of related research has been conducted in the area of quantitative analysis of program behavior, particularly with respect to the memory subsystem. Our current experimental framework for measuring the memory behavior of the SML/NJ system is based on an adaptation of Larus' quick profiling tool [70] and the Dinero III cache simulator by Hill [54]. Our analysis of garbage collection and memory management costs was partially inspired by previous work by Zorn [137] and Wilson *et al.* [134].

Finally, our long-term exploration into delayed compilation is based in part on the large body of research in partial evaluation, much of which was pioneered by Jones *et al.* [59, 58]. The potential for applications of partial evaluation ideas to systems programming have been pointed out by Consel, Pu, and Walpole [25] and implemented in *ad hoc* ways by Massalin and Pu in the Synthesis Kernel [86, 85]. Further experimentation with *ad hoc* forms of delayed compilation have been explored by Keppel *et al.* for C [60, 61], Deutsch and Schiffman for Smalltalk-80 [34], and

Chambers and Ungar for SELF [23].

1 Key Research Personnel

The research personnel of the Fox project span the areas of systems building, language design, and compiler technology. These include full-time and part-time faculty involvement, post-doctoral researchers, research programmers, and doctoral students. Below, we list the faculty and staff involved in the project research, along with their main area of work. It should be noted that many others, particularly graduate students, are also making contributions, but these are too numerous to list here.

Edoardo Biagioni. Post-doctoral researcher. System networking and kernel design and implementation.

Kenneth Cline. Research programmer. System networking implementation and applications.

Nicholas Haines. Research programmer (on leave). System networking implementation, compiler and run-time system development.

Robert Harper. Assistant Professor. Language design and implementation.

Nevin Heintze. Post-doctoral researcher. Static program analysis and optimization, compiler technology.

Peter Lee. Assistant Profess r. Compiler technology.

Brian Milnes. Research programmer. System networking implementation, testing and performance analysis.

References

[1] Martin Abadi and Luca Cardelli. A theory of primitive objects: Second-order systems. In *European Symposium on Programming*, 1994. (To appear).

[2] Martin Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. In *Theoretical Aspects of Computer Science '94*, 1994. (To appear.).

[3] Mark B. Abbott and Larry L. Peterson. A language-based approach to protocol implementation. *IEEE/ACM Transactions on Networking*, 1(1), February 1993.

[4] Michael J. Accetta, Robert V. Baron, William Bolosky, David B. Golub, Richard F. Rashid. Avadis Tevanian, Jr., and Michael W. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–113, July 1986.

[5] A. Aiken, E. L. Wimmers, and T. Lakshman. Soft typing with conditional types. In *Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland*, January 1994.

[6] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[7] Andrew W. Appel and Trevor Y. Jim. Continuation-Passing, Closure-Passing Style. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages,* pages 293–302, Austin, Texas, January 1989. ACM.

[8] Andrew W. Appel and David B. MacQueen. A Standard ML Compiler. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274, pages 301–324. Springer-Verlag, 1987.

[9] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In J. Maluszynski and M. Wirsing, editors, *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming,* pages 1–13, New York, August 1991. Springer-Verlag.

[10] Lennart Augustsson. *Compiling Lazy Functional Languages, Part II.* PhD thesis, Department of Computer Sciences, Chalmers University of Technology, 1987.

[11] John Backus, John H. Williams, and Edward L. Wimmers. The programming language FL. In Turner [131], pages 219–247.

[12] Edoardo Biagioni, Nicholas Haines, Robert Harper, Peter Lee, Brian G. Milnes, and Eliot B. Moss. Standard ML signatures for a protocol stack. Fox Memorandum CMU-CS-93-170, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, July 1993. (Also published as Fox Memorandum CMU-CS-FOX-93-01).

[13] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems,* 5(1):47–76, February 1987.

[14] Kim Bruce. Safe type checking in a statically typed object-oriented programming language. In *Twentieth ACM Symposium on Principles of Programming Languages*, Charleston, SC, January 1993.

[15] Rod Burstall, David MacQueen, and Donald Sannella. HOPE: An experimental applicative language. In *Proceedings of the 1980 LISP Conference*, pages 136–143, Stanford, California, 1980. Stanford University.

[16] Luca Cardelli. Compiling a functional language. In *1984 ACM Conference on LISP and Functional Programming,* Austin, Texas, August 1984.

[17] Luca Cardelli. Typeful programming. Technical Report 45, DEC Systems Research Center, 1989.

[18] Luca Cardelli. Obliq: A lightweight language for network objects. (To appear), 1994.

[19] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report (revised). Technical Report 52, DEC Systems Research Center, Palo Alto, CA, November 1989.

[20] Luca Cardelli, Jim Donahue, Mick Jordan, Bill Kalso, and Greg Nelson. The modula-3 type system. In *Sixteenth ACM Symposium on Principles of Programming Languages,* pages 202–212, Austin, TX, January 1989.

[21] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 18(4), December 1986.

[22] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, Toronto*, pages 278–292, June 1991.

[23] Craig Chambers and David Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, Portland*, pages 146–160, June 1989.

[24] David D. Clark. The structuring of systems using upcalls. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 171–180. ACM, December 1985.

[25] Charles Consel, Calton Pu, and Jonathan Walpole. Incremental partial evaluation: The key to high performance, modularity and portability in operating systems. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 44–46, June 1993.

[26] Robert L. Constable, *et. al. Implementing Mathematics with the NuPRL Proof Development System*. Prentice-Hall, 1986.

[27] Eric C. Cooper and J. Gregory Morrisett. Adding threads to Standard ML. Technical Report CMU-CS-90-186, School of Computer Science, Carnegie Mellon University, December 1990.

[28] Robert Cooper and Clifford Krumvieda. Distributed programming with asynchronous ordered channels in distributed ML. In *ACM SIGPLAN Workshop on ML and Its Applications*, pages 134–150, San Francisco, CA, June 1992.

[29] Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The categorical abstract machine. *Science of Computer Programming*, 8, May 1987.

[30] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Annual ACM Symposium on Principles of Programming Languages, New York*, pages 238–252, January 1977.

[31] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.

[32] Alain Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco*, pages 157–168, January 1990.

[33] Alain Deutsch. An operational model of strictness properties and its abstractions. In R. Heldal, C. K. Holst, and P. Wadler, editors, *Proceedings of the 1991 Glasgow Workshop on Functional Programming*, pages 82–99, August 1991.

[34] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages, Salt Lake City*, pages 297–302, January 1984.

[35] Bruce Duba, Robert Harper, and David MacQueen. Typing first-class continuations in ML. In *Eighteenth ACM Symposium on Principles of Programming Languages*. January 1991.

[36] Projet Formel. CAML: The reference manual. Technical report, INRIA-ENS, June 1987.

[37] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation, Toronto, Ontario*, pages 268–277. ACM Press, June 1991.

[38] Emden Gansner and John Reppy. eXene. In Robert Harper, editor, *Third International Workshop on Standard ML*, Pittsburgh, PA, September 1991. School of Computer Science, Carnegie Mellon University.

[39] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[40] Nicholas Haines, Darrell Kindred, J. Gregory Morrissett, Scott M. Nettles, and Jeannette M. Wing. Tinkertoy transactions. Technical Report CMU-CS-93-202, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, December 1993.

[41] John Hannan and Frank Pfenning. Compiler verification in LF. In Andre Scedrov, editor, *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418, Santa Cruz, California, June 1992. IEEE Computer Society Press.

[42] Robert Harper, Bruce Duba, and David MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, ?(?):?–?, ? 1993. (To appear. See also [35].).

[43] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Second Symposium on Logic in Computer Science*, pages 194–204, Ithaca, New York, June 1987.

[44] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993. (See also [43].).

[45] Robert Harper and Mark Lillibridge. Polymorphic type assignment and CPS conversion. In Olivier Danvy and Carolyn Talcott, editors, *Proceedings of the ACM SIGPLAN Workshop on Continuations CW92*, pages 13–22, Stanford, CA 94305, June 1992. Department of Computer Science, Stanford University. Published as technical report STAN-CS-92-1426.

[46] Robert Harper and Mark Lillibridge. Explicit polymorphism and CPS conversion. In *Twentieth ACM Symposium on Principles of Programming Languages*, pages 206–219, Charleston, SC, January 1993. ACM, ACM.

[47] Robert Harper and Mark Lillibridge. Polymorphic type assignment and CPS conversion. *LISP and Symbolic Computation*, ?(?):?–?, ? 1993. (To appear. See also [45].).

[48] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Twenty-first ACM Symposium on Principles of Programming Languages*, pages ?–?, Portland, OR, January 1994. (To appear.).

[49] Robert Harper, David MacQueen, and Robin Milner. Standard ML. Technical Report ECS-LFCS-86-2, Laboratory for the Foundations of Computer Science, Edinburgh University, March 1986.

[50] Robert Harper and John C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, April 1993. (See also [95].).

[51] Peter Henderson. Purely functional operating systems. In *Functional Programming and Its Applications*. Cambridge University Press, 1982.

[52] Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In *Proceedings of the 1991 Conference on Functional Programming and Computer Architecture*, pages 448–472, June 1991.

[53] Fritz Henglein. Global tagging optimization by type inference. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming, San Francisco*, pages 205–215, June 1992.

[54] M. D. Hill. Experimental evaluation of on-chip microprocessor cache memories. In *Proceedings of the 11th International Symposium on Computer Architecture, Ann Arbor*, pages 158–166, June 1984.

[55] Paul Hudak and Philip Wadler. Report on the programming language Haskell, version 1.0. Technical Report Research Report YALEU/DCS/RR-777, Yale University, April 1990.

[56] John Hughes. Why functional programming matters. In Turner [131], pages 17–42.

[57] Thomas Johnsson. *Compiling Lazy Functional Languages*. PhD thesis, Department of Computer Sciences, Chalmers University of Technology, 1987.

[58] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[59] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, 2(1):9–50, 1989.

[60] David Keppel, Susan J. Eggers, and Robert R. Henry. A case for runtime code generation. Technical Report 91-11-04, Department of Computer Science and Engineering, University of Washington, November 1991.

[61] David Keppel, Susan J. Eggers, and Robert R. Henry. Evaluating runtime-compiled value-specific optimizations. Technical Report 93-11-02, Department of Computer Science and Engineering, University of Washington, November 1993.

[62] Yousef A. Khalidi and Michael N. Nelson. An implementation of UNIX on an object-oriented operating system. In *Proceedings of Winter '93 USENIX Conference*, January 1993.

[63] Yousef A. Khalidi and Michael N. Nelson. The spring virtual memory system. Technical Report SMLI TR-93-09, Sun Microsystems Laboratories, Inc., February 1993.

[64] David Kranz. *ORBIT: An Optimizing Compiler for Scheme*. PhD thesis, Department of Computer Science, Yale University, New Haven, Connecticut, February 1988.

[65] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. ORBIT: An Optimizing Compiler for Scheme. In *Proceedings of the SIGPLAN '86 Conference Symposium on Compiler Construction*, pages 219–233, Palo Alto, California, June 1986. ACM.

[66] Clifford Krumvieda. DML: Packaging high-level distributed abstractions in sml. In *Proceedings of the Third International Workshop on Standard ML*, Pittsburgh, PA, September 1991. School of Computer Science, Carnegie Mellon University.

[67] Clifford Krumvieda. Expressing fault-tolerant and consistency-preserving programs in distributed ml. In *ACM SIGPLAN Workshop on ML and its Applications*, pages 157–168, San Francisco, CA, June 1992.

[68] Peter J. Landin. A correspondence between ALGOL-60 and Church's lambda notation. *Communications of the ACM*, 8:89–101, 1965.

[69] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6:308–320, 1966.

[70] James R. Larus. Efficient program tracing. *Computer*, 26(5):52–61, May 1993.

[71] Xavier Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report 117, INRIA, 1990.

[72] Xavier Leroy. Unboxed objects and polymorphic typing. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque*, pages 177–188. ACM Press, January 1992.

[73] Xavier leroy. Polymorphism by name. In *Twentieth ACM Symposium on Principles of Programming Languages*, January 1993.

[74] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proceedings of the Twenty-first Annual ACM Symposium on Principles of Programming Languages, Portland*. ACM, January 1994.

[75] Xavier Leroy and Michel Mauny. The caml light system, version 0.5 — documentation and user's guide. Technical Report L-5, INRIA, 1992.

[76] Xavier Leroy and Pierre Weis. Polymorphic type inference and assignment. In *Eighteenth ACM Symposium on Principles of Programming Languages*, pages 291–302, Orlando, FL, January 1991. ACM SIGACT/SIGPLAN.

[77] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*, volume 114 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[78] Barbara Liskov and Robert Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):382–404, July 1983.

[79] David MacQueen. Modules for Standard ML. In *1984 ACM Conference on LISP and Functional Programming*, pages 198–207, 1984. Revised version appears in [49].

[80] David B. MacQueen. The implementation of Standard ML modules. In *ACM Conference on Lisp and Functional Programming*, pages 212–223. ACM, 1988.

[81] Chris Maeda and Brian N. Bershad. Protocol service decomposition for high-performance networking. In *14th ACM Symposium on Operating Systems Principles*, December 5–8 1993.

[82] Mark Manasse and Greg Nelson. Trestle reference manual. Technical Report 68, DEC Systems Research Center, Palo Alto, CA, December 1991.

[83] Per Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175. North-Holland, 1982.

[84] Per Martin-Löf. *Intuitionistic Type Theory*, volume 1 of *Studies in Proof Theory*. Bibliopolis, Naples, 1984.

[85] Henry Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Department of Computer Science, Columbia University, 1992.

[86] Henry Massalin and Calton Pu. Threads and input/output in the synthesis kernel. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 191–201, December 1989.

[87] Paul R. McJones and Garret F. Swart. Evolving the UNIX system interface to support multithreaded programs. Research Report 21, DEC Systems Research Center, September 1987.

[88] Spiro Michaylov and Frank Pfenning. Natural semantics and some of its meta-theory in Elf. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Proceedings of the Second International Workshop on Extensions of Logic Programming*, pages 299–344, Stockholm, Sweden, January 1991. Springer-Verlag LNAI 596.

[89] Robin Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[90] Robin Milner. How ML evolved. *Polymorphism: The ML/LCF/HOPE Newsletter*, 1, 1983.

[91] Robin Milner, Lockwood Morris, and Malcolm Newey. A logic for computable functions with reflexive and polymorphic types. In *Proceedings of the Conference on Proving and Improving Programs*, Arc-et-Senans, France, 1975.

[92] Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, 1991.

[93] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[94] James G. Mitchell, William Maybury, and Richard Sweet. Mesa language manual, version 5.0. Technical Report CSL-79-3, Xerox PARC, April 1979.

[95] John Mitchell and Robert Harper. The essence of ML. In *Fifteenth ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988.

[96] John Mitchell, Sigurd Meldal, and Neel Madhav. An extension of Standard ML modules with subtyping and inheritance. In *Eighteenth ACM Symposium on Principles of Programming Languages*, January 1991.

[97] J. Eliot B. Moss. Nested transactions: An approach to reliable distributed computing. Technical Report MIT/LCS/TR-260, MIT Laboratory for Computer Science, Cambridge, MA, April 1981.

[98] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[99] Michael N. Nelson, Yousef A. Khalidi, and Peter W. Madany. The spring file system. Technical Report SMLI TR-92-389, Sun Microsystems Laboratories, Inc., February 1993.

[100] Scott M. Nettles and Jeannette M. Wing. Persistence + undoability = transactions. In *Proceedings of Hawaii International Conference on Systems Science 25*, January 1992. Also published as technical report CMU-CS-91-173, August 1991.

[101] Sean W. O'Malley and Larry L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2), May 1992.

[102] Larry Peterson, Norman Hutchinson, Sean O'Malley, and Herman Rao. The x-kernel: A platform for accessing Internet resources. *Computer*, 23(5):23–33, May 1990.

[103] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

[104] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[105] Benjamin C. Pierce and David N. Turner. Object-oriented programming without recursive types. In *Twentieth ACM Symposium on Principles of Programming Languages*, Charleston, SC, January 1993.

[106] Gordon Plotkin. Call-by-name, call-by-value, and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[107] Gordon Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–257, 1977.

[108] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Computer Science Department, Aarhus University, 1981.

[109] Gordon Plotkin. Notes for a post-graduate course in semantics. Available from the Computer Science Department, University of Edinburgh, 1983.

[110] Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 77–87, Austin, Texas, January 1989. ACM.

[111] John Reppy. First-class synchronous operations in standard ML. Technical Report TR 89-1068, Computer Science Department, Cornell University, Ithaca, NY, December 1989.

[112] John Reppy. Asynchronous signals in Standard ML. (Unpublished manuscript.), August 1990.

[113] John H. Reppy. CML: A higher-order concurrent language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 293–305, June 1991.

[114] John C. Reynolds. GEDANKEN — a simple typeless languages based on the principle of completeness and the reference concept. *Communications of the ACM*, 13(5):308–319, May 1970.

[115] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Conference Record of the 25th National ACM Conference*, pages 717–740, Boston, August 1972. ACM.

[116] John C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the 25th ACM National Conference, New York*, 1972.

[117] John C. Reynolds. Towards a theory of type structure. In *Colloq. sur la Programmation*. volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer-Verlag, 1974.

[118] John C. Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing '83*, pages 513–523. Elsevier Science Publishers B. V., 1983.

[119] John C. Reynolds. Preliminary design of the programming language forsythe. Technical Report CMU-CS-88-159, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, June 1988.

[120] John C. Reynolds. The coherence of languages with intersection types. In T. Ito and A. R. Meyer, editors, *International Conference on Theoretical Aspects of Computer Software*, pages 675–700, Sendai, Japan, September 1991. Springer-Verlag LNCS 526.

[121] Paul Rovner. Extending Modula-2 to build large, integrated systems. *IEEE Software*, 3(6):46–57, November 1986.

[122] Dana Scott. Outline of a mathematical theory of computation. In *Proceedings of the Fourth Annual Princeton Conference on Information Systems and Sciences*, pages 169–176. Princeton University, 1970.

[123] Dana Scott. Domains for denotational semantics. In M. Nielson and E. M. Schmidt, editors, *Ninth International Colloquium on Automata, Languages, and Programming*, volume 140 of *Lecture Notes in Computer Science*, pages 577–613. Springer-Verlag, 1982.

[124] Guy L. Steele Jr. RABBIT: A Compiler for Scheme (A Study in Compiler Optimization). Master's thesis, AI Laboratory Technical Report AI-TR-474, Massachusetts Institute of Technology, May 1978.

[125] W. R. Stoye. A new scheme for writing functional operating systems. Technical Report 56, Cambridge University Computing Laboratory, Cambridge, England, 1984.

[126] Ascánder Suárez. Compiling ML into CAM. In Gérard Huet, editor, *Logical Foundations of Functional Programming*, University of Texas at Austin Year of Programming Series, pages 47–73. Addison-Wesley, 1990.

[127] Andrew Tolmach. Tag-free garbage collection using explicit type parameters. Submitted to the 1993 ACM Conference on Lisp and Functional Programming, November 1993.

[128] David A. Turner. Miranda: A non-strict functional language with polymorphic types. In *1985 Conference on Functional Programming and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[129] David A. Turner. An approach to functional operating systems. In *Research Topics in Functional Programming* [131], pages 199–217.

[130] David A. Turner. Overview of Miranda. In *Research Topics in Functional Programming* [131], pages 1–16.

[131] David A. Turner, editor. *Research Topics in Functional Programming*. Addison-Wesley, 1990.

[132] United States Department of Defense. *Reference Manual for the Ada Programming Language*. February 1983. U.S. Government Printing Office, ANSI/MIL-STD-1815A-1983.

[133] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of SOSP '93*, 1993.

[134] P. R. Wilson, M. S. Lam, and T. G. Moher. Effective 'static-graph' reorganization to improve locality in garbage-collected systems. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, Toronto*, pages 177–191, June 1991.

[135] J. M. Wing, M. Faehndrich, N. Haines, K. Kietzke, D. Kindred, J. G. Morrissett, and S. Nettles. Venari/ML interfaces and examples. Technical Report CMU-CS-93-123, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, March 1993.

[136] Jeannette Wing, Manuel Faehndrich, J. Gregory Morrissett, and Scott Nettles. Extensions to standard ML to support transactions. In *ACM SIGPLAN Workshop on ML and its Applications*, pages 104–118, San Francisco, CA, June 1992. Also published as techical report CMU-CS-92-132, April, 1992.

[137] Benjamin Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice, France*, pages 87–125, June 1990.