The Fox Project: Advanced Development of Systems Software

Eric Cooper Robert Harper Peter Lee

August 1991
CMU-CS-91-178

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

The Fox project will use an advanced programming language to build software such as operating systems, network protocols, and distributed systems. The goals of the project are to improve the design and construction of real systems software and to further the development of advanced programming languages.

We will base our work on Standard ML, a modern functional programming language that provides polymorphism, first-class functions, exception handling, garbage collection, a parameterized module system, static typing, and formal semantics. The Fox project spans a wide range of research interests, from experimental systems building to type theory, and involves several faculty members.

This research was sponsored by the Air Force Systems Command and the Defense Advanced Research Projects Agency (DARPA) under Contract F19628-91-C-0128.

Keywords: programming languages, systems software, formal semantics, program analysis, compilers, operating systems, network protocols, parallel and distributed systems

1 Innovative Claims

The following claims summarize the expected outcomes of the proposed research. Further discussion and technical rationale is presented in Section 2.

Note that the term “advanced language” refers to a programming language with a mathematically rigorous semantics and featuring higher-order functions, polymorphic types, and an advanced system for manipulating modules. The language **Standard ML** has all of these properties, and thus is a suitable starting point for research in the design, implementation, and use of advanced languages for developing systems software.

- An advanced language can be used to build parallel and distributed systems, including a stand-alone operating system for a bare machine [Section 2.3]. The systems will be easier to understand and maintain and will not exhibit many kinds of bugs typically found in comparable systems written in C. Furthermore, the components will be portable and reusable, and will exhibit performance that is competitive with software systems written in C.

- More elegant and powerful operating system interfaces can be provided for applications written in an advanced language [Section 2.3.5]. In contrast, existing operating system interfaces for UNIX and Mach have been strongly influenced by the fact that almost all client applications are written in C.

- An advanced language can allow network protocols to be expressed as a composition of simple sub-protocols, each implemented as a module within the language [Section 2.3.4]. These sub-protocols are easier to understand, implement, and reuse than current monolithic protocols.

- An existing advanced language, Standard ML, provides a suitable platform for building advanced software. Extensions to support the needs of software development are required; the formal semantics of the language provides a framework for evaluating such extensions, thereby ensuring that the consistency and coherence of the language is preserved [Section 2.4.3].

- The widely used SML/NJ compiler can be modified and extended to allow for classical optimizations such as global register allocation, induction-variable elimination, copy propagation, and common subexpression removal [Section 2.4.2]. These optimizations can be based on semantic techniques, thereby greatly improving their effectiveness and reliability.

- The semantics of an advanced language allows the development of logical systems for specifying and deriving properties of programs [Section 2.4.3]. Such specifications are of value both to the programmer—as an extension of the type system—and to implementors—providing information to the optimizer. Examples of specifications include statements about side-effect behavior and patterns of storage allocation, properties that are crucial to managing the interaction between parallel threads of control.

2 Technical Rationale

In recent years there have been many advances in systems software and programming languages.

- **Advanced systems software:** advanced operating systems, network protocols, and reliable distributed computing systems.
- **Advanced programming languages:** languages with a mathematically rigorous semantics, and featuring higher-order functions, polymorphic types, first-class continuations, and concurrency.

Unfortunately, advanced programming languages have not been used to implement "real" systems software. Until now, advanced languages have been too inefficient and lacking in features for use in practical systems. Furthermore, although the rigorous foundations underlying advanced languages should in principle provide many benefits for software development, these benefits have rarely been achieved 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, faster development, and so forth—have not been realized for systems software. We believe that this has led to undesirable compromises in the design of present-day systems software.

The goal of our research is twofold:

1. To improve the design and construction of real systems software by developing advanced parallel and distributed systems in an advanced programming language and using advanced program-development techniques.
2. To further the development of advanced programming languages by using them to build medium-scale software systems.

These goals stress the vital interplay between programming language design and program development. The construction of software systems in an advanced language will almost certainly expose weaknesses in the language and its implementation, and will suggest changes to both. This will spur further fundamental research in programming languages, taking account of the valuable experience gained in the process. On the other hand, the use of advanced languages will encourage and support novel software construction techniques. The experience of building medium-scale software in such a language will make important contributions to the future of software development.

This research builds upon our extensive experience and knowledge in the areas of operating systems (specifically, the Mach operating system [1]), programming languages (the Standard ML programming language [25], the Types in Programming project [16]), formal program development (the Ergo and Reasoning about Programs projects [21]), reliable distributed computing (the Avalon [13] and Camelot [34] projects), and high-speed parallel and distributed computing (the Nectar [12] and Warp [7, 2] projects).

2.1 Conventional Systems Programming

Systems software for UNIX and Mach systems has been written almost exclusively in the C programming language, and with little regard for formal analysis based on the language semantics. By today's standards, C is a relatively low-level language, with a number of problems that effectively preclude advanced program-development techniques.

The most severe problem with C is its lack of type safety. The compiler provides no type checking among separately compiled files. There is no module system, only a programmer's convention about the use of header files included via a text-based preprocessor. As a result, trivial errors like omitting or interchanging parameters go uncaught until the program crashes at run time.

There is no run-time bounds checking on array or pointer operations in C, and even if there were, the lack of an exception-handling mechanism makes it infeasible for C programs to cope gracefully with these kinds of run-time errors.

The need for C programmers to manage storage explicitly leads to a number of common bugs. Failing to deallocate a block of memory causes a storage leak. Deallocating a block too soon, while there are still dangling references, is likely to lead to a mysterious crash.

C does not allow nested function definitions, so all values of functional type must be top-level procedures. The lack of nested functions makes it impossible to construct non-trivial closures without introducing extraneous global variables.

2.2 Systems Programming in an Advanced Language

In our research we seek to escape from problems associated with using a low-level language like C, and instead use what we call an *advanced programming language.* We claim that an advanced language not only allows one to avoid the problems of C without an inordinate loss in performance, but also has a fundamentally positive influence on the design and organization of systems software. The resulting systems will be easier to understand and maintain than current systems, and will not exhibit many kinds of bugs that are typically found in functionally equivalent C-based systems. Furthermore, the system components will be more portable and reusable.

Of course, these claims are not new. Researchers have long stated the virtues of programming languages that allow a more abstract level of expression. However, until recently the use of such languages to develop "nitty-gritty" systems software was not realistic, due in part to the lack of necessary features in existing advanced languages, and in part to the poor performance of programs written in such languages. Recent advances in programming language design and implementation have changed the situation dramatically. One of the most important aspects of current advanced languages is the strong basis on a mathematically rigorous semantics.

The semantics is critically important for developing reliable and maintainable software, as it provides the basis for making precise statements about properties of programs, and for expressing certain guarantees about program behavior. Such certainty about the language makes it possible to study new features and extensions in a rigorously defined framework, which in turn allows the language to evolve in a sound manner. The result is a language that provides a power and clarity of expression, and which actually helps guide the design of software systems. Furthermore, the semantic foundation makes possible highly optimizing compilers and powerful support tools, thereby making the increased expressive power of the language practical to use in real systems.

But the most important and far-reaching consequence of the formal semantics is that it is crucial to the long-term goal of formal program development. Specifications are assertions about programs that are verified with respect to a formal semantics for the language. While it is still unrealistic to consider "complete" specification and verification of large-scale programs, it is still possible to prove "small theorems about large programs." To do so it is often useful to develop *purpose-built logics* for specifying and deriving important properties of programs. These logics are restricted to specifying only very limited, but nonetheless useful, properties of a program, and are thus much more amenable to mechanization. The most well-known such "purpose-built" logic is the type inference system of Standard ML which allows for the specification and verification of assertions about the execution behavior of programs. In particular, it may be proved that certain classes of run-time errors such as commonly occur in comparable C or LISP programs cannot arise in ML. Thus *every* ML program has a "small theorem" proved about it before it is executed. Moreover, the compiler takes full advantage of this fact when generating code, leading to greater efficiency without sacrificing security.

Recent work in programming languages suggests that it may be possible to extend the paradigm of "purpose-built" logics to encompass more interesting properties than just type information. For example, Standard ML programs not only yield values when executed, but also have an effect on the store. Recent work [14] suggests that it may be possible to (conservatively) track the side effect behavior of ML programs at compile time to facilitate optimization and to provide an additional degree of security to the programmer. In related work Reynolds [30] has studied the problem of controlling interference between program phrases in Idealized Algol, the progenitor of his language Forsythe, currently under development. Here again we have a small, purpose-built logic for specifying properties of programs. Such analyses will provide information to the programmer and the compiler, which in turn will allow efficiency to be obtained without sacrificing reliability or maintainability of the system.

It is our contention that advanced languages, and advanced program analysis techniques, will play an increasingly important role in software development in the future. One goal of the project is to assess this claim by using one such language as a starting point for research in advanced systems software development: Standard ML. Although lacking a number of features that are important for systems software, Standard ML combines many important advances in programming languages. For example, Standard ML allows higher-order functions, which provide a great deal of expressive power, and polymorphic types, thereby allowing static type checking and more efficient execution. Standard ML's sophisticated module system provides great flexibility for large-scale software development, including type-safe separate compilation, data abstraction, and some object-oriented features. The formal semantics of Standard ML is published and well-understood [25]. Several good compilers for a variety of machines are available in the public domain [5]. With these compilers, Standard ML programs are often comparable to C programs in run-time efficiency.

2.2.1 Previous Work

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 [26]; the Topaz system, written in Modula-2+ at DEC SRC [24, 32]; the Symbolics and Xerox Lisp machines; the Swift system, written in CLU at MIT [9, 22]; and various real-time and embedded systems written in Ada [36].

Although several of these languages provide some of the same advantages over C that Standard ML does, none provides them all, as the following table indicates.

Language	First-class functions	Compile-time type checking	Polymorphism	Parameterized modules	Formal semantics
Lisp	•		•		
Scheme	•		•		•
CLU		•		•	
Ada		•		•	
Mesa		•			
Modula-2+		•			
Standard ML	•	•	•	•	•

The other languages (with the exception of Lisp) are all imperative; systems programs written in them look like "industrial strength" C programs. 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 will be qualitatively different from previous work.

2.2.2 The Standard ML Programming Language

From the viewpoint of software engineering, implementing systems software in Standard ML has numerous advantages over current practice. The type system allows the compiler to guarantee complete *type safety:* if a program passes the compiler's type checker, the program will never "dump core" at run time by applying a function to a value of the wrong type. This feature also sets Standard ML apart from other interactive languages, such as Lisp and Smalltalk, that use run-time rather than compile-time type checking. In these languages, type errors in rarely exercised portions of code can go uncaught for an arbitrarily long time, at which point the user rather than the developer must deal with them.

The fact that the Standard ML type system is *polymorphic* increases the reusability of code. Polymorphic functions in ML can be applied to arguments of infinitely many different types. As a simple example, the function

```
fun length [] = 0
  | length (head :: tail) = 1 + length tail
```

has type `'a list -> int`; in other words, it is a function from lists of any element type to integers. This means that one can apply `length` to lists of booleans, lists of strings, lists of other list types, and so on. Using polymorphism, high-quality implementations of queues, tables, trees, and so on can be written once, and then used with elements of any data type.

Standard ML's typechecking algorithm not only provides polymorphism and complete type safety, it also does so without requiring the programmer to declare the types of any variables. This technique, called *type reconstruction,* or *type inference,* gives ML programmers all the benefits of strong typing with none of the inconvenience of having to declare variables. Type declarations can optionally be provided, in which case they are treated as assertions that are checked at compile time.

Standard ML provides efficient heap allocation of objects together with automatic garbage collection. This makes Standard ML programs simpler, because the application program doesn't have to determine when to free objects that are no longer in use. Programs are more robust, because the possibility of storage leaks or dangling references is eliminated.

Finally, programs that use large amounts of memory perform better, because Standard ML's heap allocation utilizes memory more efficiently than a number of application-specific allocators, and its garbage collection reclaims unused objects more efficiently than explicit deallocation.

Since Standard ML is a functional programming language, functions are first-class values. Function definitions may be nested, and a function value (or *closure*) whose definition refers to values in enclosing lexical scopes continues to refer to those values, no matter when the closure is applied. *Higher-order* functions are common in Standard ML programming: a function may take other functions as arguments and/or produce functions as results. All the techniques of functional programming, such as delayed evaluation, lazy evaluation, streams, process networks, and "infinite" data structures, can be expressed naturally in Standard ML.

Standard ML has a polymorphic exception handling mechanism. Exception handling complements the compile-time type checking by allowing graceful, application-specific recovery from errors that occur at run-time. In addition to allowing more robust programs, exception handling increases the reusability of code. As an example, consider a table module that raises a "not found" exception when a lookup operation fails. By handling this exception, a client can easily build a new operation that inserts an item in a table only if it is not already present. Without an exception handling mechanism, the table module would not be reusable unless the implementor had foreseen this particular application.

Programming-in-the-large in Standard ML is accomplished by means of a parameterized module system. *Signatures* correspond to what are variously called interface modules, definition modules, or package specifications in other languages. *Structures* correspond to implementation modules or package bodies. *Functors* are (link-time) functions from structures to structures, similar to Ada's generic packages.

A functor allows an implementation to be parameterized by other modules. For example, a sorting utility can be implemented as a functor that takes an ordered domain (a structure consisting of an element type and a comparison function on that type) and produces a structure containing a sort function for lists of that element type. A generic request-response protocol can be implemented as a functor that takes a one-way message protocol as a parameter. As with polymorphism, parameterized modules enhance software reuse.

2.2.3 Evaluating Standard ML

A number of metrics will be used to verify the claims we have made for Standard ML as a systems programming language. We will measure the performance of the resulting software, examine the portability and reusability of the code, and compare the expressive power of Standard ML with the C programming language.

The performance of Standard ML programs is determined both by the quality of the code generated by the compiler, and by the quality of the run-time system (in particular, the garbage collector). To quantify the quality of the code generator, we will look at CPU time and memory usage. The performance of the garbage collector will be evaluated similarly, and compared to explicit deallocation. The variance in response time due to garbage collection will also be measured.

Portability can be judged by the amount of non-portable code needed to implement real systems. Examples of non-portable code include assembly language, conditional compilation based on machine, operating system, or site-specific features, and use of nonstandard language extensions.

Reusability can be quantified by the amount of library code that is usefully shared among multiple systems programs. Expressive power is the most difficult to quantify, but the number of lines of source code may provide a reasonable basis for comparison.

2.3 Building Advanced Systems Software in Standard ML

One of the goals of the Fox project is to use Standard ML to build operating systems, network protocols, and distributed systems. Since the Mach kernel plays a key role in our proposed research, we first describe some of its important features.

The development of Mach at Carnegie Mellon represents many advances in the design and implementation of systems software [1]. For instance, Mach separates the notion of address space from thread of control. Multiple threads of control can share an address space, all running in parallel on a multiprocessor. Mach also makes aggressive use of virtual memory, allowing flexible sharing of memory between address spaces. This leads to performance gains for many applications.

The portability of the operating system is greatly enhanced by its small kernel approach. The pure Mach kernel provides only the essential services: interprocess communication, threads, and virtual memory. Traditional UNIX services such as the file system are provided by server processes. There can be several such servers, and they can provide multiple operating system environments. The small size of the kernel makes it easier to port, analyze, and optimize.

2.3.1 Parallel Programming

There is no way to express explicit parallelism in Standard ML. A compiler may, of course, introduce parallelism implicitly, but the state of the art in this area is not sufficiently advanced for our purposes. Instead, we need a means of creating and manipulating multiple *threads of control* within a single Standard ML program.

Threads are needed for two main reasons:

1. To express the naturally concurrent structure of the software we intend to construct as part of this research effort: operating systems, network protocols, and client-server distributed systems.

2. To achieve parallelism on real multiprocessors.

Programs constructed with multiple threads can be simpler and more modular than programs using alternative mechanisms.

A system constructed from multiple threads can use simple synchronous interfaces that are easy to understand. Whether or not a module uses multiple threads in its implementation need not be visible in its interface. This improves modularity and allows programs to be developed by composition.

Implementors of distributed and interactive applications must deal with asynchronous events such as users' keystrokes, incoming network messages, device interrupts, expiration of timers, and so on. The principles of modularity and information hiding dictate that different events be detected and processed in different modules. The use of multiple threads, each waiting for the appropriate class of events, supports this programming methodology.

The alternatives to multiple threads in systems programming languages include software interrupts; non-blocking operations that permit polling; or an operation that allows a program to wait for any of a set of events. In UNIX, for example, all of these mechanisms are provided, adding considerable complexity to the system interface.

None of the alternatives to threads provides modularity. If polling or some form of **select** operation is used, the programmer must write one portion of code to detect and dispatch events appropriately to their handlers; this polling loop violates modularity by "knowing" about all the event-processing modules in the system. Similarly, if software interrupts (such as UNIX signals) are used, logically unrelated modules must share a common signal handler.

We therefore need to add multiple threads to Standard ML. We plan to design a thread module that can be used directly by parallel applications, as well as by higher-level constructs for parallel programming (such as Multilisp *futures* [15] or CSP communication channels [17]). The thread extensions will not require any modifications to the syntax of the language or to the compiler, but the formal semantics of the language will have to be expanded, and the run-time system will have to be modified for some implementations.

The simplest implementation will use *continuation passing*. Continuations are an extension of Standard ML provided by the SML/NJ compiler via the `callcc` and `throw` functions. Continuations allow a single process to be multiplexed among multiple logical threads of control by treating them as coroutines.

A more sophisticated implementation will introduce *preemptive scheduling*. This approach uses timer interrupts to schedule the execution of multiple threads within a single process. The SML/NJ run-time system turns interrupts into continuations that are passed to an interrupt-handling function. This approach fits in with the previous continuation-based implementation of coroutines in a natural way.

The most advanced implementation will execute Standard ML threads on multiprocessors running the Mach operating system. This necessitates changes to the run-time system, since its operations and data structures will be accessed in parallel. We will use at least one Mach kernel thread for each physical processor, and each Mach kernel thread will in turn be multiplexed among multiple Standard ML threads. The run-time system's heap allocation and garbage collection will have to be modified to be safe for multiple threads. The naive approach of serializing all allocations would result in intolerable performance, because heap allocation is used extremely frequently. A scheme that allows parallel allocation, with synchronization only at garbage collection time, will be developed instead.

Threads can be used directly by parallel applications, as evidenced by experience with similar facilities in Mesa at Xerox PARC [19], Modula-2+ at DEC SRC [32], and in C Threads for Mach at Carnegie Mellon [11]. But the functional style of programming supported by Standard ML allows higher-level mechanisms to be used as well. Examples of these mechanisms include *futures,* as in Multilisp [15], and communication channels, as in CSP [17].

2.3.2 Language and Operating System Interaction

Our extensions to the Standard ML programming language will make use of several advanced features of the Mach kernel interface. We expect this to result in the first programming environment that fully exploits Mach's capabilities while still providing complete type safety and the other desirable characteristics discussed in Section 2.2.2. In this section, we describe how we will take advantage of the Mach support for advanced virtual memory management, multiprocessor thread scheduling, and interprocess communication.

Memory Management Standard ML uses heap allocation for all data structures, and provides garbage collection to free the programmer from the burden of explicitly deallocating memory. In the SML/NJ implementation, the allocator and garbage collector use only the simplest UNIX operations to request more memory from the operating system and return it when no longer needed. This approach makes the implementation highly portable, but results in mediocre performance.

Mach's advanced capabilities offer the possibility of significantly more efficient memory management for Standard ML. The Mach virtual memory system supports large, sparse address spaces; flexible sharing of virtual memory (both read-write and copy-on-write) between address spaces; and user-defined memory management via *external pagers*.

The SML/NJ garbage collector uses a variant of the *stop-and-copy* algorithm. In this scheme, the heap is divided into two halves, called *from-space* and *to-space*. Objects are allocated from from-space until it is exhausted. Then garbage collection occurs, and all live data is copied from from-space to to-space. Once this has been done, the roles of from-space and to-space are reversed. The live data has been compacted, and the contents of old from-space are no longer needed.

The garbage collector thus has a great deal of knowledge about its use of memory, but cannot take advantage of it in the naive UNIX implementation. For example, the operating system does not know that the old from-space is now garbage, so it will carefully preserve the contents of those pages by writing them out to disk when they need to be reclaimed by the virtual memory system. In the current SML/NJ implementation, this results in a flurry of disk I/O after each garbage collection.

To avoid this, one would like to indicate to the virtual memory system that the pages in from-space are in a special state, with the following two properties. First, the pages are clean—they can be reclaimed at any time, without writing them to backing store. Second, the contents of the pages need not be preserved, since these pages will be written before they are next read. If the virtual memory system reclaims one of these pages, a page fault will occur the next time SML accesses the page. But since that access is guaranteed to be a write, the virtual memory system can provide any free physical page, no matter what it last contained.

In Mach, we can obtain a first approximation to this behavior by deallocating from-space immediately after the garbage collector has flipped semispaces, and reallocating it (as to-space) before the next garbage collection. But the newly allocated to-space pages are marked "zero fill," so that the first reference to each page causes a trap that results in the kernel zeroing the page. These page faults and zeroing operations are unnecessary overhead, although much less than the cost of writing a page to disk. This simple optimization should therefore make a significant improvement in performance.

During garbage collection, the collector knows which from-space pages have already been copied and which pages might still contain useful data. Unfortunately, the kernel doesn't have this knowledge, so it may choose to reclaim a page that has not yet been scanned, instead of one that contains garbage. We plan to explore the use of an ML-specific external pager, integrated with the garbage collector, to avoid this problem. An ancillary benefit of this approach is that the external pager can maintain the backing storage for the ML program in a form that makes the "save world" operation extremely efficient.

Thread Scheduling The multiprocessor implementation of Standard ML threads, described in Section 2.3.1, will use Mach kernel threads as *virtual processors,* and Mach *processor sets* to perform user-level scheduling. Using ideas due to Anderson and Bershad, all context switching among ML threads will be handled by the ML threads package, rather than the Mach kernel. Context switching between two ML threads on the same virtual processor will occur entirely in user mode, which is significantly more efficient than having the Mach kernel switch between two virtual processors.

Interprocess Communication Since we plan to use our extension of Standard ML to implement operating system servers for the Mach kernel, we must provide access to Mach's interprocess communication (IPC), based on sending and receiving messages to and from *ports.* We plan to add a low-level interface to Mach IPC, similar to the UNIX operating system interface currently in SML/NJ.

Using the Mach IPC interface, we will build higher-level forms of communication. Section 2.3.3 describes our plans for a type-safe remote procedure call system and an implementation of Linda tuple space.

2.3.3 Distributed Programming

We propose to support two mechanisms for constructing distributed programs in Standard ML. The first is remote procedure call (RPC), the most popular method of constructing client-server distributed systems [6]. For example, Mach uses RPC for all of its operating system services, and Camelot and Avalon use RPC in transaction-based reliable distributed systems. The second mechanism is Linda *tuple space,* a form of high-level shared memory that can be provided in both distributed and parallel systems [8].

Remote Procedure Call The client-server model is an extremely successful method of structuring distributed systems and fits naturally into ML. A server defines its exported interface via a signature, so that the unit of distribution is the ML module. Constructing distributed systems becomes no different than composing modules on a single machine.

We will develop an RPC stub generator for Standard ML that uses ML signatures as input, and produces the following three functors as output:

1. A stub implementation of the input signature, for use by the client. The stub functions handle the details of marshaling requests, invoking the server, and unmarshaling responses.

2. An implementation of a standard "listener" interface, for use by the server. This functor takes a structure matching the input signature (the actual implementation of the server functions), and takes care of unmarshaling requests, dispatching them to the appropriate server function, and marshaling responses.

3. Marshaling functions, for use by both of the functors described above. For each type mentioned in the signature, this module contains a marshaling and unmarshaling function that are responsible for translating values of that type to and from an external representation suitable for transmission over the network.

Much of the functionality needed by the client and server stubs is independent of the input signature. In particular, the communication protocols, marshaling strategies for basic types, and server control structures for handling requests may be modularized and provided as a library. The client and server stub functors are parameterized by these library modules and actual implementations are selected at link time. Hence, the same generated code can be used to create one client that uses an ASCII external representation over Mach IPC, and a second client that uses a binary external representation over UNIX sockets.

The rich type system of Standard ML poses a number of challenges that have not been faced in previous RPC systems. In particular, we must support abstract (opaque) types, polymorphism, first-class functions (closures), pointers, and exceptions. We propose to explore a novel *proxy object* approach to communicating these types.

For example, an object of abstract type never leaves the address space of the server in our scheme. Instead, we associate the object with a unique ID, and transmit the ID instead. The first time a unique ID arrives at a client, it is translated into a proxy object that is used as the client's version of the abstract type. Since the type is abstract, the Standard ML type system guarantees that there is no way the client code can observe any difference between the proxy and the original; to perform any non-trivial operation on the object, it must invoke the server. In doing so, the proxy's unique ID is sent to the server, which translates it back into the original object.

The same mechanism can be used, in the other direction, to support polymorphism. During an invocation of a server function with a polymorphic formal parameter, the actual parameter remains at the client, and the unique ID and proxy scheme is used instead. Since the formal parameter to the server function is polymorphic, the Standard ML type system guarantees that the function cannot perform any non-generic operation on it, and therefore cannot observe the difference between the proxy and the actual parameter. If the polymorphic function constructs a result that uses the proxy as a component and returns it to the client, then the occurrences of the proxy will be mapped back into the same unique ID when the result is marshaled, and the occurrences of that unique ID will be mapped back into the correct value at the client.

A similar approach supports the transmission of function values and pointers. These are turned into proxy objects that contain "back pointers" to the corresponding original object. When operated on (applied in the case of functions; dereferenced or assigned in the case of pointers), the proxy uses the back pointer to invoke the appropriate operation on the original object.

Linda Tuple Space We will implement the Linda model of shared distributed tuple space in Standard ML, using ML's flexible type system and pattern matching facilities to provide the basic Linda operations on tuples. We will use separate ML modules to implement the Linda interface, operations on tuple space, communication of tuples over the network, and replication of tuple spaces. Our approach allows different compositions of these modules to be used to configure a system with either local or remote access to tuple space, and with either a centralized or distributed implementation of tuple space. The communication of tuples over the network will use the RPC stub generator described above.

The resulting implementation of Linda in Standard ML will offer an attractive way to separate the functional and the imperative portions of a distributed system. Individual processes can be written in ML in a pure functional style and the Linda shared tuple space can be used to interconnect the processes and maintain the state of the system.

2.3.4 Implementing Network Protocols in Standard ML

Based on our experience with protocol implementation in the Mach and Nectar projects, we believe that all of the benefits listed in Section 2.2.2 will apply to network protocols written in Standard ML. We propose to implement such network protocols, including TCP, in order to demonstrate this claim.

Garbage Collection for Buffer Management Many lines of Nectar CAB code are devoted to managing buffers, and this accounts for much of the execution time. We expect that ML's garbage collection will greatly simplify buffer management, but that in its present form (as implemented in SML/NJ) it will introduce unacceptable delays during protocol processing. We will therefore add either incremental or concurrent garbage collection to guarantee predictable real-time response.

Using Modules to Express Protocol Structure Our preliminary research indicates that ML's parameterized modules offer a natural way to express the fine structure of protocols that is usually not apparent in current, monolithic implementations. For example, one functor can implement the notion of sequenced delivery, transforming a stream of duplicated or out-of-order messages into a stream in which each message appears once, in order.

From this perspective, a protocol is viewed as a composition of functors, each adding specific functionality to the overall protocol. For example, a request-response protocol for use with RPC might be constructed from a functor that provides sequenced delivery, composed with a functor that provides segmentation of messages larger than a single packet, composed with one for reliable delivery via acknowledgment and retransmission, composed with one that adds transaction numbers to pair requests with responses.

This approach allows the protocol designer to match the semantics of the protocol both to the needs of the application and to the characteristics of the underlying network. A library of protocol building blocks will be developed, so that protocol implementation is as simple as functor composition.

These ideas are similar to the notion of micro-protocols in the *x*-kernel at the University of Arizona [29]. Our approach, developed independently, is completely integrated with the Standard ML language and requires no extra-linguistic tools. Indeed, the idea simply "falls out" of good ML programming practice. In contrast, the *x*-kernel designers introduced a graphical module interconnection language, which is then translated into C, their implementation language.

2.3.5 Operating Systems

In this section, we describe our proposed research in the area of operating systems. We will explore the use of Standard ML to provide cleaner, more powerful interfaces to existing operating systems like UNIX and Mach. We will implement a production-quality name server in ML, for use with the Mach kernel. And we will use a language based on Standard ML to implement a stand-alone operating system for a bare machine.

Higher Level Operating System Interfaces The existing interfaces provided by UNIX and Mach are strongly constrained by the limited kinds of data types and control structures found in programming languages like C and Pascal. We can design operating system interfaces for use with applications written in Standard ML that are both simpler and more powerful.

For example, UNIX system call errors are reported using a special return value (-1) together with a global variable set to the specific cause of the error. This is undesirable for a number of reasons. In practice, C programs either ignore these errors, and are therefore unreliable, or else check the return value of every system call on the off chance that an error occurred, and are therefore cluttered and unreadable. In addition, the use of a global variable to report the cause of the error makes the interface impossible to use reliably in multithreaded programs. In a language based on Standard ML, it is much more natural to use the language's exception handling mechanism. This leads to cleaner code, since there can be a single exception handler at some enclosing level, and works correctly in the presence of multiple threads.

The Mach interfaces are similarly influenced by the expected client programming language (in this case, the influence is also from Pascal, the predominant application language of Accent, Mach's predecessor). Mach's interprocess communication mechanism, and the RPC interfaces built on top of it, are used to access all operating system services. But the RPC facility (called MIG) is based on a type system that is quite unnatural for Standard ML. The MIG representations of integers, strings, records, and so on map well into C and Pascal, but require nontrivial conversions to and from the corresponding ML data types. And the other types that ML can express, such as recursively defined data types, abstract types, first-class functions, polymorphism, and exceptions, cannot be used in MIG interfaces. The RPC system we propose to build (described in Section 2.3.3) overcomes these limitations.

The greatest opportunity for simpler, more powerful operating system interfaces is to exploit first-class functions in Standard ML. Using function values as parameters to system calls, for example, will allow us to dispense with the many variants of UNIX system calls that are required for non-blocking and/or asynchronous operations. Similarly, the handful of UNIX functions that are needed by C programs to iterate over file system directory contents can be replaced by a single ML mapping function, one that applies its functional argument to each directory element.

Operating System Services for the Mach Kernel Figure 1 shows the structure of the Standard ML environment that will be developed for Mach. The ML run-time system will be extended, as described in Section 2.3.2, to take full advantage of Mach's capabilities. At the highest level, operating system servers, including a name server, will be developed in Standard ML for use with the Mach kernel.

Analysis of these servers, including detailed performance evaluation, will provide us with early feedback about ML's suitability for systems programming. The servers will be usable by C programs as well as ML applications. In fact, users should be unaware of the fact that they are using servers written in Standard ML.

A Stand-alone Operating System in Standard ML We propose to use a language based on Standard ML to build an operating system for the Nectar communication processor (called the CAB) [12]. As shown in Figure 2, this system will run on the bare CAB (a SPARC-based CPU board, similar to many embedded controllers), rather than on top of Mach.

The current CAB software, written in C, consists of a simple real-time operating system designed especially to run network protocols such as TCP/IP, along with communication-related user tasks. We can therefore hope to develop an equivalent system in Standard ML in a reasonable time, without having to implement all the features that a complete time-sharing system would require. And since the tasks running on the CAB are assumed not to

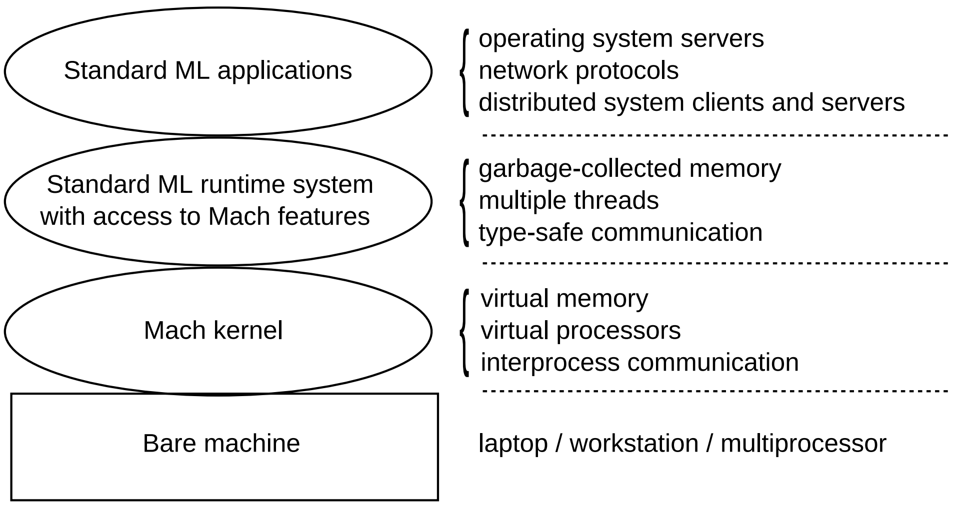

Figure 1: Standard ML System on Mach

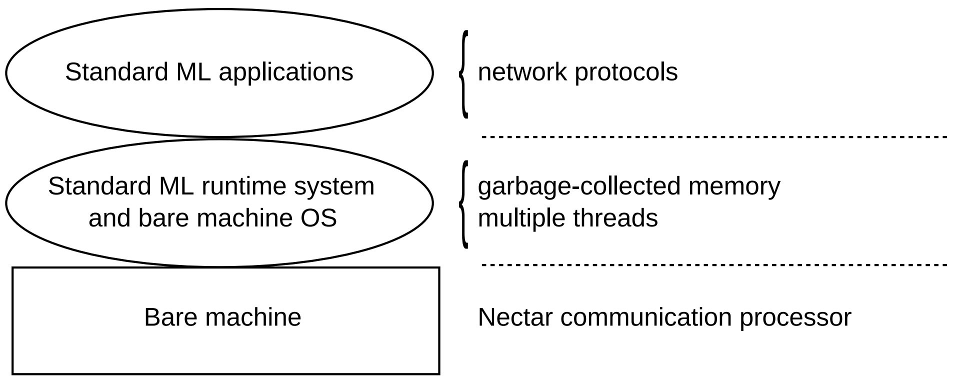

Figure 2: Standard ML System on the Nectar CAB

be arbitrary user programs, no firewalls are needed beyond those provided by the ML type system.

The CAB hardware is also simpler than typical workstations. For example, it has no memory management unit, so we will not be implementing a virtual memory system. Nevertheless, it provides sufficient memory *protection* (not *translation*) in hardware to support real-time concurrent garbage collection, using techniques described by Appel, Ellis, and Li [4]. As mentioned in Section 2.3.4, we expect some form of real-time garbage collection to be necessary for acceptable response time during protocol processing.

By comparing our ML implementation to the current CAB software written in C, we expect to validate many of the qualitative claims we made in Section 2.2.2, and demonstrate that the use of Standard ML for advanced development represents a significant improvement in the state of the art.

2.4 Design of Advanced Programming Languages for Systems

The Standard ML programming language has developed over many years with a remarkable amount of care and discipline, while still attracting significant use in a wide range of applications. As a result, the language has stayed relatively small yet achieving features and power for solving realistic problems. One of the most important products of Standard ML's long and careful development is the publication of its complete formal semantics [25]. The semantics is the touchstone for all future development of the language and its implementation.

There are a number of benefits to having a formally defined semantics for a programming language. The semantics provides the basis for making precise statements about the properties of programs written in that language. On the one hand, rather general statements can be made about the guarantees afforded by the language. For example, it is possible to prove that well-typed ML programs are free of many of the kinds of bugs that commonly occur in comparable C or Lisp programs [37]. It is also possible to prove that abstract types afford representation independence—the implementation of an abstract type may be changed without affecting the final results of client programs [27]. On the other hand, the semantics of a language provides the means for making precise assertions about particular programs. Specifications are assertions about the meanings of programs; the correctness of a program with respect to a given specification is determined by the semantics. Hence, the utility of a correctness proof is directly dependent on the extent to which the "real world" is consistent with the idealizations of the semantics. It is important to realize that "specification" should be read loosely, encompassing all manner of assertions about the semantics of a program, ranging from type information to detailed descriptions of a program's behavior.

A rigorous semantics is also an indispensable tool for evaluating modifications and extensions to the language. The practical utility of Standard ML, despite its great expressive power, can be attributed in large measure to the use of semantical methods to assure the coherence of the design. Time and again plausible-seeming features were discovered to have serious flaws that were only discovered by careful consideration of the formal definition of that feature, especially in the context of the rest of the language. During the design process a number of critical semantic invariants were isolated. These invariants summarize certain crucial properties of the language that must be preserved by any perturbation, lest coherence and implementability be lost. In the absence of a suitably well-developed semantics, only "seat of the pants" criteria are available, and experience has shown that such methods invariably lead to flawed designs [31].

Many new compilation and optimization techniques are semantically based. In particular, there is increasing stress on the use of purpose-built logical systems for specifying program properties such as types, effects, and data and control-flow information. The development of semantics-based compilation techniques is essential to the practicality of advanced programming languages. Indeed, it appears that the complexity of such languages can only be managed using semantics-based techniques, in contrast to traditional methods developed for simple imperative languages. For example, dataflow analysis for languages with higher-order functions is a very delicate matter; only recently have solutions been proposed, and these solutions are based entirely on techniques from denotational semantics [28, 33].

2.4.1 Evolution of Standard ML into a Systems Programming Language

Standard ML is a well-defined, advanced language in the sense that we have been describing. Although it has been applied in a number of situations, it has not thus far been used for the sort of systems-building applications that we are considering here. It is therefore to be expected that significant changes to the language will be required to support the needs of advanced systems software development. A proper understanding of these changes can only be made in terms of its formal semantics, and hence it is expected that the implementation effort will suggest a number of important problems for language design. This interplay between theoretical investigations and practical experimentation is an essential aspect of the project. This section describes the proposed research in the area of language design. Since we have not yet undertaken to construct the software systems proposed in the preceding section, it is difficult to anticipate the exact nature of the modifications that will be needed. The following description should therefore be understood as a sketch of likely areas of research. It may be expected that some of these topics will turn out to be less important than others, and, conversely, that heretofore unexpected changes will be required.

Extending the Standard ML Core Language Commonly occurring programming techniques such as the use of co-routines are most naturally expressed using first-class continuations. Continuations are a representation of the control state of the evaluator. Since all expressions in ML yield a value (in addition to possibly having effects on the store), continuations may be thought of as special kinds of functions that resume an evaluation context with a given argument value. To maximize flexibility in programming with continuations, it is important that they be "first-class" values that may be passed around at will. We will therefore study the addition of first-class continuations to ML. This will involve extending the type system, and establishing an appropriate soundness theorem for well-typed programs that use first-class continuations.

In order to take advantage of the parallelism afforded by the hardware and underlying operating system (if any), we will add support for concurrent execution of Standard ML programs to the language. Such changes will have a far-reaching effect on both the semantics and the implementation of the language. Achieving a proper understanding of their role in the language poses a tremendous challenge that may well resist solution in the early stages of this work. It is not yet clear, for instance, how difficult it will be to include such extensions in the formal semantics of Standard ML, nor to what extent the guarantees afforded by the ML type system can be extended to programs that make use of concurrency. The addition of concurrency primitives also promises to have a far-reaching effect on the implementation of Standard ML. Considerations such as the relationship between the "logical" threads of control in an ML program and the underlying "physical" threads of control afforded by the operating system and hardware will be investigated. The presence of multiple, concurrent threads of control will also have a far-reaching effect on the run-time system, in particular, the garbage collector.

Extending the Standard ML Modules System Many operating system services can be understood as arising by a process of elaboration on a basic service. For example, network services may be built up in layers, with each layer providing additional services in terms of those at the previous level. In order to support such incremental construction of software services, it may be necessary to extend the ML modules system to support a weak form of inheritance. As currently defined, the ML modules system does not support incremental extension of an unknown, or partially known, structure by a new set of operations. Instead, the structure is first coerced to conform to a given signature, and only then may it be extended. Consequently, the extension is of a restricted view of the structure, rather than the original structure itself. It appears, on preliminary investigation, that support for extensible structures may be necessary for building network services incrementally. We therefore plan to investigate the addition of such features to the Standard ML modules system.

Support for Large-scale Software Development in Standard ML The ML modules system provides the linguistic mechanisms for organizing systems into manageable units. However, it currently provides only limited support for separate compilation, and no support for building one system on top of another. We anticipate that it will be necessary to introduce a larger form of aggregate that allows for collecting together the modules that form a given system into a unit, and for building one system on top of another. This mechanism will also provide the basis for a more sophisticated approach to separate compilation that allows for altering the initial standard basis on a system-by-system basis, and provides a form of persistence at the module level. Separate compilation and configuration management utilities can then be built on top of these constructs.

2.4.2 Compiling and Analyzing Standard ML for Systems Software

Advanced languages have traditionally been far too inefficient for serious systems-level applications. Thus, significant new implementation techniques will be required to provide an adequate basis for advanced systems software development in an advanced language. For our language based on Standard ML, such improvements can be added to the widely used SML/NJ compiler [5]. Besides allowing us to more rapidly bring new implementation techniques into use in the project, this will also benefit a sizable community of Standard ML users. We have already had several good experiences with making such improvements to this system, for example in the Portable SML project [35].

In this section we describe our proposed research activities in the area of language implementation.

Optimizing Standard ML Programs Dataflow analysis has become the standard basis for many optimization techniques for conventional programming languages. Such optimizations include global register allocation, induction-variable elimination, copy propagation, and common-subexpression removal. In conventional low-level programming languages such as C, good-quality dataflow analysis is often possible because a program's control-flow graph can be computed at compile time. Unfortunately, the control-flow graph for a Standard ML program generally is not compile-time computable, due to the presence of higher-order functions. Thus, in practice the "classical" dataflow optimizations are not used by current compilers for Standard ML.

Semantics-based analysis techniques, however, provide a framework for developing algorithms that generate useful conservative approximations of the control-flow graph for ML programs [28, 33]. Such algorithms can then be used to implement the classical optimizations, which we believe will lead to substantial improvements in run-time efficiency. Besides the classical optimizations, other optimizations would also be enabled, such as compile-time garbage collection [18]. Eventually, such techniques might also form the basis of a semantics-based compiler generator, extending previous work [20]. In this case the semantics can lead the compiler to better decisions on representations for certain data types, for instance for mutable records and arrays, which currently are handled in an awkward and inefficient manner.

The approach we will take is as follows. Building upon our previous work on the semantics-based dataflow analysis of the continuation-passing style fragment of Scheme, we will first implement control-flow analysis for the SML/NJ compiler. This is a natural step to take since SML/NJ uses a continuation-passing style intermediate representation. Then, based on an analysis of the characteristics of the generated control-flow graphs, selected dataflow optimizations will be implemented.

Run-time Support Issues The traditional organization of run-time systems for advanced languages is not suitable for systems software development. For example, the run-time system for the SML/NJ compiler depends on services provided by an operating system [3], which is clearly not acceptable if one is implementing an operating system on a bare machine! Thus, we will develop a new run-time system with a more traditional organization that will allow selected services to be incorporated into compiled code. Under this organization, fundamental run-time services such as garbage collection will have to be made operating-system independent.

As work on purpose-built logics for inferring side-effect and garbage-collection behavior progresses (see Section 2.4.3), the balance between the run-time system and code compiled from application-level programs may be adjusted. For example, analysis of the storage allocation behavior of ML programs may lead to significant simplifications of the run-time systems for programs within an identifiable class. This will reduce the size of the run-time system, and hence the overhead involved in porting the system to a different machine architecture. This will also enhance the retargetability of the compiler.

2.4.3 Formal Semantics, Program Analysis, and Language Design

A crucial property of an advanced programming language (as we have used the phrase) is the existence of a precise formal semantics. A semantics serves three related purposes: as a language design tool, as a basis for compiler construction, and as a basis for program analysis. Each of these uses of formal semantics will be important in our work. We discuss each in turn, and summarize our expected activities in this area.

As a language design tool, the role of formal semantics is to provide a precise framework in which language features can be studied, both in isolation and in relation to one another. The formal semantics makes it possible to state precise properties of the language, and to rigorously establish properties of the language such as the guarantees afforded by the ML type system. The use of semantical techniques was critically important in the design of Standard ML [25], leading to the early detection of design flaws, and suggesting alterations to informally-presented language features. In this capacity a formal semantics is most useful as a means of avoiding pitfalls, and helping to ensure the overall coherence of the design. Semantical analysis cannot, of course, ensure that the resulting language is *useful* to practitioners—that can only be established empirically. It is one of the goals of the proposed research to test the utility of advanced languages in the setting of advanced software development.

It is to be expected that in the process of building systems in an advanced language such as ML it will be necessary to extend the language with features that have not heretofore been considered. Making such extensions raises problems of both design and implementation, and it is therefore to be expected that theoretical work on language design will go hand-in-hand with practical work on implementation of the proposed extension. The purpose of the semantical investigation is to uncover hidden flaws in the design, to say something about its properties, to suggest implementation techniques and trade-offs, and to suggest alternatives. Our experience indicates that such an approach is very fruitful, and has been used to good result in the design of Standard ML.

As a basis for compiler construction, the formal semantics of Standard ML proved critically important for its implementation. The semantics of ML is expressed in the "natural semantics" style [10], a form particularly well-suited for direct application in compiler construction. In fact, Kahn has developed the techniques to such an extent that it is possible to automatically generate prototype compilers from specifications written in natural semantics. Even at the level of hand-written compilers, the formal semantics is critically important: the data structures used in the Standard ML of New Jersey compiler front-end mirror the semantical structures used in the definition of Standard ML [23, 25]. The Edinburgh "kit compiler" for Standard ML, currently under development, is directly based on the formal semantics, and will therefore be an invaluable tool in experimenting with variations on the language.

More generally, semantics-based techniques are of increasing importance in compiler construction and the development of program analysis tools. Some of the features that distinguish advanced languages like Standard ML from languages like C make it impossible to apply the standard approaches to code optimization and program analysis. By using the semantical basis of the language, however, certain properties of programs, such as the control-flow structure [33] and binding-time information [28], can be precisely defined. Information about such properties has long been shown to be crucial for many kinds of compile-time optimizations and also for other kinds applications requiring automated analysis of programs. In most cases, then, the semantics-based definition of these properties directly suggests an algorithm for carrying out the analyses. In other cases, it provides a basis for proving the correctness of aspects of *ad hoc* algorithms for carrying out such optimizations and analyses.

Finally, a formal semantics is essential for proving properties of programs, ranging from typing assertions ("this program maps integers to reals") to detailed descriptions of a program's behavior ("specifications"). The importance of semantics in this setting stems from the fact that an assertion about a program—of whatever form—is a claim about what the program "means", *i.e.*, a claim about the semantics of that program. For example, the typing assertion "$e : int \to real$" in ML means that the meaning $||e||$ of the expression e is a member of the set $||int \to real||$ which is the meaning of the type $int \to real$.

The extent to which the truth of such a specification is of relevance to the user is directly related to the extent to which the properties of the "real world" are correctly modeled by the formal semantics, just as in any branch of applied mathematics. As simple and obvious as it seems, this point has often been overlooked, and is a constant source of confusion. To stress the point: *a formal verification that a program enjoys a particular property does not entail that the program will work correctly in practice*. Nonetheless, proving properties of programs is not a pointless exercise: such proofs can significantly increase our confidence in a program by establishing properties that hold in the semantic model.

In order to forestall any misunderstandings, we hasten to stress that we are most emphatically *do not* envision "verifying" or "proving correct" the systems that we plan to build. Although we believe, for the reasons discussed above, that systems constructed in advanced languages will be more reliable and maintainable than comparable programs written in conventional languages, we *do not* make any claims about the formal correctness of the resulting code. Such an enterprise remains beyond the reach of current methods, despite substantial developments both in the theory and practice of formal program development. What is clear, however, is that if the long-term goal is to be realized, it can only be through the use of well-defined languages with a precise semantics. By working with such languages we hope to help bridge the gap between current trends in programming languages and current practice in software development.

On the other hand there is evidence that short-term gains can be made by working with "purpose-built" logics for specifying and deriving narrowly-constrained properties of programs. The most well-developed example of such a "special-purpose" logic is the ML type system. The process of type checking is a weak form of theorem proving that establishes a "small theorem about a large program." Despite the relative weakness of types as specifications, they are nonetheless useful for both the programmer and the implementor. In particular, the run-time organization of ML is vastly simplified by the fact that only well-typed programs are compiled and executed. Recent work suggests that the methodology of type inference can be extended to encompass assertions about the side-effect behavior and storage usage of a program [14]. Similarly, Reynolds has devised a formal system for expressing "interference assertions" about Idealized Algol programs [30]. This assertions allow one to prove that the execution of two phrases do not interfere with each other on any common variables. It appears, on preliminary investigation, that such properties will be especially important with multi-threaded programs.

We therefore expect that a substantial amount of effort will be devoted to semantical investigations: to assess language features, to build compilers and optimization methods, and to build program analysis tools. A lively interplay between theory and practice is a characteristic feature of the proposed research.

Initial investigations suggest that a number of changes to Standard ML will be necessary in order to support software development of the kind that we propose. We therefore expect to study the following issues from both a theoretical and implementation point of view:

- Adding first-class continuations to allow for the natural expression of constructs such as co-routines.

- Extensions to the modules system to support incremental construction of systems software.

- Incremental separate compilation to admit more flexible construction of large-scale software.

These investigations will, in many cases, be conducted for small languages related to ML, chosen to emphasize the relevant issues. Some will take the form of direct extensions to the Definition of Standard ML, and its implementation. It should be emphasized that at it is too early to tell which of these extensions will be required, or whether our investigations will turn toward more pressing problems.

As a tool for developing optimization techniques, we plan to investigate the use of semantics in the development of logical systems for specifying and deriving properties of programs such as side-effect behavior and the use of storage. These properties appear to be particularly important in the conjunction with concurrency primitives since interference between parallel threads of execution is a common source of errors in such programs. We also expect that the development of such analysis tools will be important in the design of efficient run-time support for such programs. The optimization techniques that we envision developing for Standard ML programs are semantics-based.

3 Results, Products, and Transferable Technology

We expect that many of the results of the Fox Project will be useful to other researchers and practitioners. Included among these results are the software products listed in Section 1, as well as technical reports and papers, seminars, graduate courses, and participation in scientific conferences and symposia by the investigators. We now describe the relationship of these expected research results to the rest of the research community.

The most immediately useful results will be the software products. These products will be written in an extension of the Standard ML language, and will use an enhanced implementation based on the Standard ML of New Jersey system. Of the planned language extensions, the addition of concurrency will likely find the largest number of users. Already, within Carnegie Mellon, both Jeannette Wing's *Venari Project* and Ed Clarke's *Parthenon Project* have expressed strong interest in using the concurrent version of Standard ML, in the first case to support the development of reliable distributed systems in Standard ML, and in the second case to continue experimentation with techniques for parallel theorem proving. Researchers at other institutions, most notably John Reppy at Cornell University, are developing alternative approaches to concurrency in Standard ML. We expect a mutual benefit to arise from the close communication that we have already developed with him. Also at Cornell, Bob Constable's *NuPRL Project* is considering the use of a concurrent ML, and so we expect that our developments in this regard will have some impact on their work.

In addition to the concurrency features, other extensions will be made in order to make Standard ML practicable for systems software development. These extensions are likely to benefit the growing community of Standard ML users in the U.S. and Europe. This is also the case for other planned improvements to the SML/NJ system, particularly in the areas of code optimization and environment support for the development of large-scale programs. The techniques developed for optimizing ML programs are also likely to be applicable to other languages, for example Lisp and Scheme.

The initial development of ML took place mainly at Edinburgh in the U.K. and at INRIA in France. Most of the U.K. universities now use Standard ML in research and teaching, and there continues to be active related research in France, Denmark, Germany, and Sweden, to name but a few. Many of these projects are funded by the ESPRIT program and its successors; they range from basic research in programming language design and semantics, to programming environments, to applications such as CAD tools and network protocol specification and synthesis. We expect that our results will generate a good deal of interest in this community, and we expect to benefit greatly from their experience. In particular, we expect to interact closely with researchers in Edinburgh (on matters related to Standard ML) and in Paris (on matters related to CAML, their dialect of ML).

Less immediately, but still in the near term, we expect that the operating system developed by the Fox Project for the Nectar communication processor will be of benefit to other developers of high-performance embedded systems. We have hopes that the software itself will be fairly portable, so that it can be ported to other embedded systems without an inordinate amount of effort. More importantly, we expect that our use of an advanced language will lead to new ideas about the organization of operating systems, distributed systems, and networking software. This will be of interest and benefit to other researchers and practitioners in the area of software systems, such as the *Mach Project* at Carnegie Mellon, and research projects at DEC Systems Research Laboratory and AT&T Bell Laboratories.

In the longer term, we hope to see the realization of more fundamental benefits of the proposed research. We may see, for instance, that our insistence on developing language extensions in the framework of the formal semantics leads to new language design principles, especially with respect to module systems, side-effect analysis, and object-oriented programming. This would be directly related to current research being carried out at DEC Systems Research Center on the Modula-3 language, and Luca Cardelli's Quest language.

Future developments in compiling and hardware technology may also allow advanced languages to be used in a wider range of embedded real-time systems. Such systems must often be extremely robust, and it is here that the advantages of developing systems software in a programming language with a rigorously defined foundation will be especially attractive.

Finally, our developments in the area of purpose-built logics will likely have much wider applications in language design and formal program development. Our results are likely to benefit numerous research projects in these areas.

In the end, we hope that the results, products, and transferable technology of the Fox Project lead to new approaches to system software development. By providing a large-scale example of our proposed "language-centric" approach to systems software, we believe that the prospects for useful results are excellent.

References

[1] 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.

[2] Marco Annaratone, Emmanuel Arnould, Thomas Gross, H. T. Kung, Monica Lam, Onat Menzilcioğlu, and Jon A. Webb.
The Warp computer: Architecture, implementation, and performance.
IEEE Transactions on Computers, C–36(12):1523–1538, December 1987.

[3] Andrew W. Appel.
A runtime system.
Journal of Lisp and Symbolic Computation, 3(4):343–380, November 1990.

[4] Andrew W. Appel, John R. Ellis, and Kai Li.
Real-time concurrent collection on stock multiprocessors.
In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 11–20, June 1988.
Also published as *SIGPLAN Notices*, 23(7).

[5] Andrew W. Appel and David B. MacQueen.
A Standard ML compiler.
In *Functional Programming Languages and Computer Architecture*, pages 301–324. Springer-Verlag, 1987.
Volume 274 of *Lecture Notes in Computer Science*.

[6] Andrew D. Birrell and Bruce Jay Nelson.
Implementing remote procedure calls.
ACM Transactions on Computer Systems, 2(1):39–59, February 1984.

[7] Shekhar Borkar, Robert Cohn, George Cox, Sha Gleason, Thomas Gross, H. T. Kung, Monica Lam, Brian Moore, Craig Peterson, John Pieper, Linda Rankin, P. S. Tseng, Jim Sutton, John Urbanski, and Jon Webb.
iWarp: An integrated solution to high-speed parallel computing.
In *Proceedings of Supercomputing '88*, pages 330–339, Orlando, Florida, November 1988. IEEE Computer Society and ACM SIGARCH.

[8] Nicholas Carriero and David Gelernter.
The S/Net's Linda kernel.
ACM Transactions on Computer Systems, 4(2):110–129, May 1986.

[9] 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.

[10] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn.
A simple applicative language: Mini-ML.
In *Proceedings of the 1986 Conference on LISP and Functional Programming*, 1986.

[11] Eric C. Cooper and Richard P. Draves.
C Threads.
Technical Report CMU-CS-88-154, Computer Science Department, Carnegie Mellon University, June 1988.

[12] Eric C. Cooper, Peter A. Steenkiste, Robert D. Sansom, and Brian D. Zill.
Protocol implementation on the Nectar communication processor.
In *Proceedings of SIGCOMM '90 Symposium on Communications Architectures and Protocols*, pages 135–144, September 1990.
Also available as Technical Report CMU-CS-90-153, School of Computer Science, Carnegie Mellon University.

[13] David L. Detlefs, Maurice P. Herlihy, and Jeannette M. Wing.
Inheritance of synchronization and recovery properties in Avalon/C++.
Computer, pages 57–69, December 1988.

[14] David K. Gifford, Pierre Jouvelot, John M. Lucassen, and Mark A. Sheldon.
FX-87 reference manual.
Technical Report MIT/LCS/TR-407, MIT Laboratory for Computer Science, September 1987.

[15] Robert H. Halstead, Jr.
Multilisp: A language for concurrent symbolic computation.
ACM Transactions on Programming Languages and Systems, 7(4):501–538, October 1985.

[16] Robert Harper, Peter Lee, and Frank Pfenning.
Foundations of programming: Aspects of research in Ergo.
Carnegie Mellon 1988/1989 Computer Science Research Review, pages 29–38, 1989.

[17] C. A. R. Hoare.
Communicating sequential processes.
Communications of the ACM, 21(8):666–677, August 1978.

[18] Paul Hudak.
A semantic model for reference counting and its abstraction.
In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.

[19] Butler W. Lampson and David D. Redell.
Experience with processes and monitors in Mesa.
Communications of the ACM, 23(2):105–117, February 1980.

[20] Peter Lee.
Realistic Compiler Generation.
Series on Foundations in Computing. MIT Press, Cambridge, 1989.

[21] Peter Lee, Frank Pfenning, John Reynolds, Gene Rollins, and Dana Scott.
Research on semantically based program-design environments: The Ergo project in 1988.
Technical Report CMU-CS-88-118, School of Computer Science, Carnegie Mellon University, March 1988.

[22] 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.

[23] David B. MacQueen.
An implementation of standard ML modules.
In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, 1988.

[24] 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.

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

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

[27] John C. Mitchell.
Representation independence and data abstraction.
In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 263–276, January 1986.

[28] Flemming Nielson.
Towards a denotational theory of abstract interpretation.
In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.

[29] 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.

[30] John C. Reynolds.
Syntactic control of interference.
In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 39–46, January 1978.

[31] John C. Reynolds.
Using category theory to design implicit conversions and generic operators.
In Neil Jones, editor, *Proceedings of the Aarhus Workshop on Semantics-Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science*, pages 211–258. Springer-Verlag, 1980.

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

[33] Olin Shivers.
Control flow analysis in Scheme.
In *Proceedings of the SIGPLAN '88 Symposium on Language Design and Implementation*, pages 164–174, June 1988.

[34] Alfred Z. Spector, Joshua J. Bloch, Dean S. Daniels, Richard P. Draves, Daniel J. Duchamp, Jeffrey L. Eppinger, Sherri G. Menees, and Dean S. Thompson.
The Camelot project.
Database Engineering, 9(4), December 1986.
Also published as Technical Report CMU-CS-86-166, Computer Science Department, Carnegie Mellon University, November 1986.

[35] David Tarditi, Anurag Acharya, and Peter Lee.
No assembly required: Compiling Standard ML to C.
Technical Report CMU-CS-90-187, School of Computer Science, Carnegie Mellon University, November 1990.

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

[37] Philip Wadler.
Theorems for free!
In *The Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, September 1989.

Appendix: Schedule and Milestones

Year 1

Development of the necessary groundwork for systems software development in an advanced language.

1. Implement a multiprocessor version of Standard ML for the Mach operating system, with support for multiple threads of control.

2. Implement higher-level mechanisms for concurrency in Standard ML, such as Multilisp futures and Reppy's synchronous events.

3. Implement support for interprocess communication (Mach IPC) and remote procedure call (RPC) in Standard ML.

4. Implement Linda tuple space for parallel and distributed Standard ML.

5. Implement analysis phases for the SML/NJ compiler to support dataflow optimizations.

6. Study the addition of support for first-class continuations to Standard ML.

Year 2

Development of basic system components, environment support (modules, separate compilation, and run-time system), and theoretical groundwork for formal specification and development of systems software.

1. Implement operating system servers, including a name server, for the pure Mach kernel (Mach 3.0) in Standard ML.

2. Implement network protocols, including the TCP protocol, in Standard ML. Using Standard ML's parameterized modules (functors), apply and extend the "microprotocol" ideas introduced in the *x*-kernel.

3. Design extensions to the Standard ML modules system to support incremental construction of software components.

4. Design and implement support for incremental separate compilation of Standard ML programs.

5. Implement a run-time system for Standard ML for supporting systems-level software on a bare machine.

6. Study the use of logical systems for inferring properties of ML programs related to side-effects and patterns of storage usage, and evaluate the feasibility of using these systems in a practical compiler.

Year 3

Complete development of system components, and application of formal semantics to achieve reliability and performance.

1. Using Standard ML, implement a specialized operating system for a bare machine, with functionality and performance comparable to the current Nectar CAB.

2. Develop inference systems for inferring side-effect and storage allocation information in ML programs, and have the compiler exploit this information.

3. Implement dataflow optimizations for Standard ML to improve performance of systems software.