

Reflections on Standard ML

David B. MacQueen

AT&T Bell Laboratories
Murray Hill NJ 07974

Abstract. Standard ML is one of a number of new programming languages developed in the 1980s that are seen as suitable vehicles for serious systems and applications programming. It offers an excellent ratio of expressiveness to language complexity, and provides competitive efficiency. Because of its type and module system, Standard ML manages to combine safety, security, and robustness with much of the flexibility of dynamically typed languages like Lisp. It also has the most well-developed scientific foundation of any major language. Here I review the strengths and weaknesses of Standard ML and describe some of what we have learned through the design, implementation, and use of the language.

1 Introduction

The goal of programming language design is to maximize expressiveness while minimizing conceptual complexity, within the constraint that the language must have an efficient implementation. If a language is to be suitable for large-scale, serious programming (*i.e.*, implementing software that will have users and will therefore have to be robust and maintainable), then it must provide security and facilities for organizing large programs. These are some of the desiderata that guided the design of Standard ML.

This paper presents my personal views about language design and the lessons we have learned from the design and implementation of Standard ML. As the field of programming language design matures, it is essential that a more scientific approach should come into play. The history of ML provides an example of how theoretical insights and practical programming requirements can be successfully synthesized.

The first section summarizes some of the strengths of Standard ML. We then proceed to some remarks on the process of language design in general and on the design of Standard ML in particular. Next comes a review of some particular lessons learned from the experience of implementing and using Standard ML. Section 5 illustrates the continuing language research activity associated with ML by describing a number of experimental language extensions. These efforts can be seen as exploring options for the next generation of ML-like languages. Finally I discuss the relation between ML and object-oriented programming and finish with some projections about the future development of Standard ML and its successors.

I will not review the history of ML here, since that topic is well covered in Appendix E of the *Definition of Standard ML* ([35]) and in Robin Milner's Turing award interview ([17]). I will also assume some familiarity with ML and refer the reader to John Ophel's paper in this volume ([37]) for an overview of the language, and to Larry Paulson's book ([38]) for a more extensive introduction. I will often

use the shorter name “ML,” which may refer either to Standard ML in particular or to the generic family of ML languages according to the context.

2 Why Standard ML Works

Here is an enumeration of some of the major strengths of Standard ML from the view of a user of the language. Not surprisingly, many of these points parallel those mentioned in Appel’s critique ([3]).

A high-level programming model. A high-level model makes programming more efficient and more reliable by automating and suppressing many low level details such as memory management and data formatting. Standard ML’s programming model is based on the functional programming paradigm, which derives from the lambda calculus.

Safety. ML’s type system, memory management system, and exception facilities combine to guarantee that even buggy ML programs do not “crash” (*i.e.*, fail catastrophically due to some corruption of the computational state or inappropriate manipulation of data). It *is* possible for a program to fail because an ML exception is raised and never handled. In the interactive system this returns one to top-level, but in a stand-alone program it terminates the program abnormally. It is, of course, also possible for an ML program to produce the wrong result because of a bug, or to fail to terminate.

Security and robustness. Static type checking detects many errors at compile time. Error detection is enhanced by the use of pattern matching, which helps ensure coverage of all cases because the compiler can statically detect and report incomplete matches, and by the exception mechanism, which provides a type-secure, disciplined way to deal with potential runtime exceptions. The result is that unexpected runtime errors can be largely avoided or dealt with appropriately.

Expressiveness and conciseness. The ability to treat functions as first-class values, the use of higher-order functions to express a wide range of control structures, the use of pattern matching for concise analysis of data, and the availability of imperative constructs provide great expressive power within a simple and uniform conceptual framework. The combination of state and higher-order functions is very powerful, but can cause difficulties in specifying and reasoning about programs (as in object-oriented programming).

Flexible strong typing. The ML type system is simple, uniform,¹ flexible, and sound. The flexibility is achieved through the use of parametric polymorphism. New types can be defined to match precisely the requirements of a problem, and data abstraction (*i.e.*, hiding of data representations) is well supported. Automatic inference of most general type schemes streamlines interactive programming. Type checking provides security by detecting statically a large fraction of common mistakes. But perhaps the most valuable aspect of the type system is that it provides a language for designing and expressing data representations and program interfaces and then automatically enforces their proper use through type checking.

¹ Any type constructor can be applied to any types without restriction – all types are treated equally. Also, user-defined types are treated the same as primitive types.

Modularity. The ML module system is an organic extension of the underlying polymorphic type system ([31, 36]). It provides for separation of interface specification and implementation and supports abstraction and parameterization. These facilities are very effective in structuring large programs and defining generic, reusable software components.

Efficiency. Highly optimizing compilers ([4]) have been developed for ML that, for many applications, produce code whose speed is competitive with those for corresponding programs written in much lower-level languages such as C. However, space efficiency tends to be significantly worse than for low level languages like C. The cost of automatic memory management and a uniform, simple type system is more intensive use of memory, though advancing compiler technology is reducing this overhead.

Multiple modes of usage. In its original incarnation as the metalanguage of the LCF system ([19]), ML played two roles. It was used to program proof tactics and tacticals to automate theorem proving, and it was used as the command language of the interactive LCF system. In several more recent systems such as HOL ([42]), ML also serves as the underlying implementation language for the whole system (LCF itself was implemented in Lisp). These roles can be separated; in particular, it is possible to use an ML compiler in a traditional batch style to generate binary object files that are later linked to create a stand-alone application. Such stand-alone applications must incorporate the runtime system, but they are independent of the compiler and the interactive system. ML has attributes that support all these modes of usage: type inference simplifies use as an interactive command language, many characteristics mentioned above suit it to the role of a symbolic metalanguage, and the module and type systems support large-scale systems implementation.

3 Comments on Language Design Methodology

A principled and scientific language design should be guided by parallel efforts and insights coming from three perspectives: *semantics*, *implementation*, and last but not least, *pragmatics*, or how the language is to be used in practice. Ideally, the design of a language should consist of several iterations, each involving development of semantics, experimental implementation, and application experiments. Each iteration should use feedback from all three perspectives to adjust design decisions from the previous iteration.

Writing a formal semantic definition clearly provides a strong discipline and encourages simplicity and clarity in the language design. Ideally the development of a formal definition should be backed up by investigation of the metatheory of the language (*e.g.*, type soundness theorems, principal typing theorems), and of how to formally specify and reason about programs in the language (*i.e.*, the *logic* of the language). Unfortunately, the state of program logics is currently rather weak, so designing a language to have a tractable logic seems to be an impractically severe constraint (advocates of pure functional and logic languages may disagree with this point, but such languages have had limited practical impact so far).

Developing a prototype implementation of the language forces one to consider the cost of language features and their interactions in terms of implementation complexity, and this also favors simplification. The implementation perspective also makes

designers consider the consequences of their decisions on execution speed. A new language design may require new compilation technology, which is risky, but progress would be hindered if new languages were overly constrained by known compilation techniques.

How the language is to be used is obviously a central consideration in its design, but when novel features are included it is not easy to fully understand their utility. One excellent way of gaining experience with the use of a new language as it is being designed is to implement the language in itself. This works for general purpose languages, but may be impractical for special purpose languages that are not suitable for language implementations (*e.g.*, implementing Awk in Awk [1]).

The pragmatic perspective tends to lead designers to add features to the language to enhance its perceived expressive power or convenience, and hence there is a danger of excessive complexity and featurism, especially if the interaction of different features is not carefully considered. One of the hardest tasks in language design is judging what should be left out of the language, because every feature has its constituency. Another difficult and critical problem is to judge the expressiveness/complexity tradeoff for a given feature. This is especially tricky because the use of a particular feature in programming may involve a high mental overhead for programmers even though the feature's semantic description is relatively straightforward (*e.g.*, first-class continuations).

In the case of Standard ML, Robin Milner's leadership guaranteed that the semantic perspective would be emphasized, especially in the earlier stages of the language design. As is often the case, Standard ML was a new member of an existing family of languages (LCF/ML, Cardelli ML, Hope), which meant that there was already a body of pragmatic and implementation experience on which to draw. However, the design of Standard ML could have benefited from further iterations to feed back early implementation and usage experience, since a number of minor design problems surfaced fairly early in the implementation efforts, but after the relevant parts of the design had been frozen (*e.g.*, the syntactic ambiguity of nested matches).

A final piece of advice about concrete syntax design: do it last! Though it would be quite difficult to carry out a language design without using a concrete syntax, at least for testing the implementation and pragmatic aspects of the design in progress, it should be remembered that the abstract syntax is the primary embodiment of the structure of the language, and endless time and effort can be wasted on premature arguments about concrete syntax. During the course of the design a simple, provisional version of the concrete syntax should be used. Only when the language design has entered its final stages of refinement and the abstract syntax has stabilized should serious concrete syntax design begin, ensuring that the concrete syntax will faithfully represent the semantics of the language. The provisional concrete syntax should be discarded and an expert in grammars and parsing with proven good taste should be assigned the task of designing the final syntax. Syntax design is like architecture: a blend of engineering and artistry best not left to a committee.

In syntax design, beware the temptation to make a language more "palatable" to potential users by having it masquerade in the inappropriate syntax of a more familiar language (*e.g.*, by giving a functional language a Fortran-like syntax). On the other hand, it is quite appropriate to borrow from the linguistic tradition behind

a language design (*e.g.*, most Pascal family languages have similar, Pascal-style, syntax properly reflecting their semantic similarities).

4 Experience with SML

Implementations of Standard ML have now been available for about six years and have been steadily improving their capabilities. The *Definition of Standard ML* was published three years ago, followed by the *Commentary on Standard ML* ([34]), and there is now a substantial body of research publications relating to the Standard ML type and module systems and to implementation and optimization techniques. We have also accumulated considerable application experience, including the implementation of several compilers and interpreters (*e.g.*, [4]), and a large number of systems designed to support theorem proving, verification, and specification of software and hardware (*e.g.*, HOL, Isabelle ([39]), Lego ([8]), ALF (Chalmers), Jape (Oxford), Lambda Prolog (CMU and Bell Labs), Concurrency Workbench ([12, 11])). It has been used to implement a large computer-aided design system (at Abstract Hardware, Ltd.) and CASE specification tools (Design ML at Meta Software, Inc.). Standard ML has also been widely used for teaching programming at various levels and as a tool in teaching and research in programming language theory, applied logic, and formal methods.

This accumulated experience has generally validated the design of the language. The type and module system provide an extremely valuable organizational discipline and safety net that makes systems implemented in ML particularly easy to modify and maintain. The language supports the implementation of large systems very well, though the facilities to support separate compilation are still evolving.

Our experience also suggests that Standard ML is about the right size. A language should be as simple as possible, but no simpler, to paraphrase Einstein. One way of judging this is to see whether a typical programmer who is conversant with the language uses all its facilities in a typical (nontrivial) application. Standard ML meets this test. Nevertheless, we can see a number of ways in which the language could be further simplified without loss of expressiveness.

ML is also a very good prototyping language. There is a general but fallacious impression that dynamically typed languages like Lisp are more appropriate for prototyping because it is “easy” to modify Lisp programs. The reason this impression is false is that serious prototypes can be large and complex programs, and if one changes the representation of data in one part of the prototype it can be a long, error-prone, and labor intensive process to trace all the consequences of such changes when using a dynamically typed language. In contrast, our experience with the Standard ML of New Jersey compiler (about 60,000 lines of ML code) is that we can make radical changes in fundamental data representations and interfaces that affect dozens of modules and still have the compiler working again within a day. Based on personal experience, these kinds of changes in a correspondingly large Lisp program might lead to weeks of debugging, even with a careful and disciplined style of programming.

Syntax is always a contentious subject, because of the subjective, aesthetic aspect of syntax design. The concrete syntax of Standard ML has received its share of criticism, and I shall add some of my own below, but I would claim that it is

reasonably successful and no one has proposed a substantially better style of syntax. The rationale behind the Standard ML syntax is that ML is a language based on the lambda calculus, but the conventional lambda calculus notation is not well-engineered for large-scale programming, so some substantial amount of “syntactic sugar” is required. Landin proposed a sugared version of the lambda calculus called ISWIM ([26]), and this formed the basis of the original LCF/ML dialect. The ISWIM syntax was fairly light-weight, and was suited to small-scale and interactive programming. However, we felt that for large-scale programming readability becomes more important relative to convenience of writing code, so Standard ML was given a somewhat heavier syntax partly inspired by the Algol/Pascal syntactic tradition. This moved the language further from the underlying lambda calculus, and this compromise between lambda-calculus and Algol-style notation is workable though not entirely comfortable. The syntax was also influenced by the local POP-2 tradition ([7]) in Edinburgh, which in turn had ISWIM and CPL ([5]) in its linguistic heritage. The syntax is still reasonably light, as indicated very roughly by comparing the total number of keywords and special signs in ML with the corresponding numbers for other languages:

Standard ML	52
Scheme	33
Haskell	39
Pascal	59
Modula 2	68
C	76
Ada	88
C++	99
Common Lisp	100+

There were also a few features of the Standard ML syntax that could be viewed as mistakes. Here is my list (most of these problems are discussed at greater length in [3]).

- The parsing of nested matches (*i.e.*, nested case expressions, fn expressions, and exception handlers) is ambiguous. This ambiguity is similar to the nested if statement ambiguity in Pascal and is a common source of errors, especially for novice programmers; experienced SML programmers learn to parenthesize all nested matches. This could be easily be fixed by adding a terminator (probably end) to these expressions.
- The infix handle keyword used to attach an exception handler to an expression makes it difficult to determine the textual scope of the handler when it is a large expression. This could be corrected by having an initial keyword (say try as in Modula 3) delineating the scope of the handler, yielding an expression of the form “try *expression* handle *match* end.”
- There is a danger of confusing variable and constructor identifiers. It is a common form of bug to misspell a constructor name, which is then interpreted as a variable in a pattern match, silently matching anything. The obvious solution is to lexically distinguish constructors, as is done in Haskell. This solution would

force us to discard the dubious capability to implicitly coerce constructors to values.

- Lexically-scoped infix directives were an ill-advised legacy from Pop-2. They complicate parsing and they do not work well with the module system because infix directives cannot be exported by structures or specified in signatures (and it would *not* be a good idea to add such a feature). One solution would be to assign a single precedence to all members of a lexically distinguished set of infix symbols, except for a fixed set of distinguished infix operators like “+” with special precedences and associativities.
- The syntax of parameter patterns in clausal function declarations is more complex than necessary and could be simplified.

Correcting these problems would eliminate most of the substantive complaints about the syntax of Standard ML, while leaving the “look and feel” of the language essentially unchanged.

A fairly common question from programmers coming to ML from a Lisp background is whether ML should support some form of macros. The CAML dialect of ML ([9]) has actually provided a form of macros, so it can be done, but there are many well-founded objections to macros, especially in a statically typed language. Macros are an admission that the syntax of the language is incomplete or inadequate, but they are not a principled or disciplined approach to enriching a language because they have no semantics or logic of their own. The semantics of macros is normally defined in terms of their expansion into the base syntax, so all static analysis, such as type checking, would be insensitive to the existence of macros in the original program and would generate messages (*e.g.*, type error messages) that could be difficult to relate to the code as it was written.²

There are a number of other “features” of Standard ML that can cause difficulties or make the language less elegant than it might be. Solutions for these problems are the subject of continuing research.

- Equality types add complexity, especially in their interaction with modules, that does not appear to be justified by their utility. The treatment of equality properties (equality kinds) could be generalized and made more sophisticated ([20]), but it would probably be better to simplify the language by eliminating them. See also the comments on the problems that equality types cause for library design in [6].
- Overloading has some inconvenient interactions with polymorphism, as noted in [3], because both of them dilute the type information available in different ways. Another problem with overloading is that it is treated in an *ad hoc* manner in Standard ML. There are a few special identifiers (mainly arithmetic and relational operators like “+” and “<”) that are overloaded in the conventional manner in the base environment (or “basis” as it is called in the Definition of Standard ML), but user-defined overloading is not supported. But on the other hand, it would not be a good idea to permit user-defined overloading because of

² Actually, in Standard ML of New Jersey the “derived” syntax forms like `while`-expressions and `if`-expressions are essentially macros, and indeed they are not accurately presented in type error messages.

the complexities of interaction between overloading and the module system. One principle that is violated by the current treatment of overloading is that the base environment should not be special or privileged and should be constructed using the same declaration facilities available to the Standard ML programmer (except for its incorporation of certain primitive types and operators like “int” and “+”). Once again the language could be simplified at the cost of slight inconvenience by eliminating overloading.

- The “open” declaration has various drawbacks. It complicates certain kinds of static analysis of ML programs, such as determining the dependency graph over a set of modules without knowing *a priori* in what order the modules are defined, a task that is relevant to an ML “make” utility such as that provided by the SourceGroup separate compilation system ([41]). The open declaration also adds considerable complexity to the implementation of the module system, especially in its most recent versions supporting higher-order modules. It would be tempting to eliminate the open declaration from the language, but it is currently the only way to import a datatype and make it a component of another module.
- The interaction between polymorphism and imperative features is handled by the less than elegant device of weak or imperative type variables (see [3, Section 3]). The main problem with this treatment is that weak types tend to propagate excessively and “infect” the types of functions that should in principle have normal polymorphic types. Further refinement of the type checking algorithm may reduce this inconvenient effect, but at the cost of making type checking more mysterious than it ought to be. A truly satisfactory solution to this problem may not be possible within the framework of the current Standard ML type system and may have to await the design of a successor language, but much useful theoretical progress is being made ([29, 23, 24, 45]).
- The Definition of Standard ML specifies an *initial basis* or standard environment, but this initial basis is rather parsimonious and has been significantly extended in all implementations. Unfortunately, these extensions were not coordinated, so different ML implementations have incompatible standard environments (*e.g.*, Standard ML of New Jersey, Poly ML, and Poplog ML all provide a substring function, but they are all different!). A library devised by Dave Berry at Edinburgh can be used to provide some commonality between implementations but the ML community still faces a difficult process of convergence to a common definition of a standard environment as a basis for portability among ML compilers. Designing a good standard environment is a time-consuming and difficult job, and it depends even more than the core language design on programming experience. In retrospect, the failure to provide an adequate standard environment was a major problem with the definition.

These design flaws are all fairly minor and do not materially degrade the utility of the language. The main limitations of Standard ML have to do with its automatic memory management and the uniform, high-level view of data that automatic memory management and polymorphism entail. The memory management system involves space overheads (*e.g.*, multiple heap spaces in a copy garbage collector), and the data representation is often less efficient than in a low-level language where the programmer has detailed control over the formatting of data representations.

Garbage collectors also involve latency – pauses in computation that are inconvenient for interactive applications and may be unacceptable in applications with real-time constraints.

ML shares these limitations with other garbage collected languages like Lisp and Scheme, but as compiler technology improves ML’s type system will provide better opportunities for optimizing data representations ([28]). The fact that most ML values are immutable also can be exploited in various ways in memory management. With regard to latency, low-latency concurrent garbage collectors have been implemented ([25]), showing that this problem can be overcome.

5 Experiments and Extensions

The Definition of Standard ML provides a solid, stable foundation for the language, but a number of significant experimental extensions to ML have been studied, designed, and implemented. It is a testament to the flexibility of the basic language that most of these extensions can be implemented essentially as library modules, without adding new syntactic constructs to the language. These library modules may require some “magic” in the form of new implementation primitives, but the magic is not externally visible. On the other hand, some of these extensions, such as first-class continuations and the concurrency features built on them, have a profound affect on the semantics of the language and would require considerable work to incorporate into the formal definition. The discussion below is biased toward Standard ML of New Jersey, which has been the basis for most experimentation because of its open nature.

5.1 Control Constructs

The basic control constructs of ML consist of function application, recursion, case analysis, and exception transmission. Another very powerful control construct that has been extensively studied and exploited in the context of (call-by-value) functional languages is that of first-class continuations. It turned out to be possible to define a typed version of first class continuations ([16, 23]), and an efficient dynamic representation of continuations was already available in the SML of New Jersey compiler, which is based on a continuation-passing-style (CPS) representation of ML code. The only visible change required to support first-class continuations was the addition of one type constructor, `'a cont`, and two associated functions: `callcc` (call-with-current-continuation) for capturing a continuation, and `throw` for “invoking” a continuation.

First-class continuations are a very powerful, but very dangerous construct. They represent a functional version of the “go to” statement of imperative languages, and can be misused to produce higher-order versions of spaghetti code that are extremely difficult to understand. Their main benefit is their use as a wizard’s tool for implementing higher-level and more disciplined control constructs. In the case of Standard ML of New Jersey, they have been used to implement several concurrency libraries, notably Concurrent ML (described in this volume, [40]) and ML Threads ([13]). These libraries also depend on a signal handling module for the implementation

of preemptive scheduling, and the interface of this signal handling module is also effectively expressed in terms of continuations.

These concurrency libraries support multiple threads running within a single address space, either on single or multiple processors. The work on distributed computing with Standard ML being pursued at Cornell ([14]), the Facile group at ECRC ([18]), and Edinburgh ([33]) represents a further step, dealing with the difficult problems of asynchronous communication and transmission of ML values between address spaces in a multicomputer.

5.2 Modules

The basic module system described in the Definition was strictly first-order. One could form functors by abstracting a structure expression with respect to other structures that it depended on, but not with respect to functors that it used. But a long-term goal of the design was to provide a truly higher-order module system in which structures and functors would be treated similarly. The semantic problems involved in such a generalization have been largely solved now ([43]), and a higher-order version of the module system has been implemented ([15]) and is part of the current release of Standard ML of New Jersey. Opportunities to exploit higher-order functors arise frequently, and the new facilities increase both the symmetry and expressive power of the module system.

The implementation of higher-order functors required that some new techniques and representations be developed, but it also opened up additional possibilities to exploit the generalized mechanisms that are the subject of ongoing research. This work is aiming toward a new level of polymorphism in the higher-order module system allowing one to parameterize with respect to an unknown signature.

Another extension of the module system that is under investigation is support for type abbreviations in signatures. Many ML programmers have noted the usefulness of such abbreviations, but the interaction between type abbreviations and sharing constraints has been problematical. Dealing with sharing constraints in the presence of type abbreviations with complete generality involves solving a special form of second-order unification problem. The general problem of second-order unification is undecidable, and it may also be the case that this specialized form is also undecidable. However, it is clear that in cases of practical interest, the unification problem is almost always trivial, so it should suffice to restrict the problem to a very limited form of second-order unification that would be tractable.

5.3 Metaprogramming and separate compilation

One concept we have been exploring recently is “turning the compiler inside-out.” By this I mean externalizing various internal compiler representations and processes to make them available to ML applications. The facilities that have been externalized so far include symbols, environments (static and dynamic), syntax trees, object code, and a set of compilation functions (parsing, elaborating, translating to the lambda intermediate representation, code generation, and execution). These interfaces support separate compilation, the interactive system, and other applications that need to manipulate and then execute ML code.

One currently missing element in these externalized interfaces is type dynamic ([30]), which we expect to add in future versions. Type dynamic has applications in metaprogramming and also in distributed programming and in supporting persistence of data.

One amusing possibility opened up by these metaprogramming facilities is that those who are unhappy with the current concrete syntax can easily define their own syntax and provide an alternate parser to support it.

ML's interfaces and static environments are rich and highly structured. Because of generative declarations and the resulting sharing relations, separate compilation for ML has to be fairly sophisticated. The SourceGroup library developed by Gene Rollins at CMU ([41]) is able to determine dependencies among a set of ML source files and minimize recompilation when changes are made. The SourceGroup system is implemented using the metaprogramming interfaces (particularly environments and the compilation interface) made available by the compiler.

5.4 Miscellaneous experiments

Other features that have been investigated and experimentally implemented are “or” patterns and vector literals and patterns. “Or” patterns allow several alternative patterns in a single match, as in `fn ((nil,x::y) | (x::y,nil)) => (x,y)`. Vector literals and patterns make it easier to construct and destruct vectors, putting them more on a par with lists. A more elaborate extension of pattern matching has been investigated by Aitken and Reppy ([2]). At the simplest level this extension supports symbolic names for constants that can be used in place of those constants in patterns. Beyond this, it supports a kind of pattern macro that could be defined in structures and specified in signatures (and therefore abstracted over by functors). This proposal has been implemented and the semantics have been fully developed.

A couple of intriguing and potentially useful variations on datatype definitions have been suggested. One is a notion of existential datatypes, described in [27]), and the other is the idea of *extensible* datatypes, which generalizes the existing type `exn`. Laufer and Odersky have implemented their version of existential datatypes, and implementing extensible datatypes is mainly a matter of adding appropriate declaration syntax.

The idea of an extensible datatype is that one declares the datatype separately from its data constructors, and an unlimited number of constructors can be introduced by constructor declarations. This is analogous to the way that exception constructors are declared. An example would be

```
datatype any;    (* data constructors to be declared separately *)
...
con (any) Int of int;
...
con (any) Real of real;
...
datatype foo = ...;
con (any) Foo of foo;
fun f(Int i) = ...
  | f(Real r) = ...
  | f(Foo x) = ...;
```

As in the case of exception constructors, all constructors that are in scope may appear in a pattern match over type any. Extensible datatypes are unboundedly extensible tagged unions, while type dynamic is an infinite tagged union using types as tags. Extensible datatypes can perform some of the functions that type dynamic might be used for.

6 ML and Object Oriented Programming

A common question these days is “Is ML an object-oriented language?” or “Can you do object-oriented programming in ML?” As usual, the answer depends on what the questioner means by “object-oriented.” In any higher-order language with state, such as ML, a function closure is a rudimentary form of object. To model more sophisticated objects one can use records of functions sharing common elements of their closure environments (corresponding to *instance variables*). There is also a useful analogy viewing structures as objects and functors as classes.

If by “object-oriented programming” one means encapsulization, data abstraction, and information hiding, it can be argued that ML supports these goals more straightforwardly and more elegantly than conventional object-oriented languages, and in addition provides better mechanisms for parameterization via higher-order functions, parametric polymorphism, and functors.

What about inheritance? Many uses of inheritance are covered by the simpler notion of subtyping. Subtyping in Standard ML can often be approximated by classical variant record techniques,³ and signature matching provides another version of subtyping at the level of modules. Consequently, many algorithms involving simple uses of subtyping can be naturally expressed in Standard ML as it is. There have been many investigations of extensions to the ML type system based on record subtyping ([10]) or polymorphism over “row variables” ([44, 22]), and it is reasonable to assume that future versions of ML or its successors will provide some such extension.⁴ But really serious applications of inheritance involving method specialization or delegation are not so easy to approximate directly in ML (but see [32]). However, these more “advanced” features of object-oriented programming languages have proven to be semantically very complex, and there is a good deal of evidence that programmers also have difficulty coping with them. My claim is that even without enhancements like record subtyping or row variables, Standard ML provides structuring and abstraction mechanisms that are at least as expressive as inheritance and considerably less complex from both the semantic and pragmatic perspective. Substantiating such claims regarding relative expressiveness is difficult, and will require extensive empirical investigations through comparative programming case studies.

³ The use of record variants as an approximation to subtyping would be strengthened by adding extensible datatypes.

⁴ See [21] for extensive coverage of the semantics and type theory of object-oriented programming languages.

7 The Future of ML

Experimentation with extensions of Standard ML as described in Section 5 above will probably taper off in the next year or so. Considerable progress is still being made in the base technologies of code generation and optimization, garbage collection and the runtime system. Active topics of investigation include representation specialization, dataflow analysis, low-level and architecture-dependent optimizations, and floating point performance. A continuing task is the development of generic libraries and tools to support ML programming. The existing set of tools for Standard ML of New Jersey is fairly extensive, including a profiler, debugger, ml-yacc and ml-lex, the SourceGroup separate compilation system, Concurrent ML, and the eXene X-window interface library, but more tools are needed. Facilities are required to enable implementations to mix ML and foreign code (*e.g.*, C), and ML application development tools must interact gracefully with conventional software tools.

New text books will be published to support the teaching of ML, and the documentation of ML implementations and programming tools must increase in quality and quantity.

In the longer term, discussions have begun on a research program leading to the design of a successor to Standard ML. This hypothetical language, known as “ML 2000” or “Millenium,”⁵ would be a completely new design, not upward compatible with Standard ML, but representing the next step in the evolution of the family and remaining faithful to the “essence” of ML. This language should be based on the call-by-value lambda calculus augmented with imperative features and providing a more uniform static type system (probably stratified and predicative like that of Standard ML). The type system should gracefully accommodate polymorphism and mutable references, and it may well include either a form of subtyping or “record polymorphism” based on row variables.

8 Conclusions

How well has Standard ML met its design goals? It provides safety, security, and good structuring tools for large-scale programming. It has a relatively high expressiveness to complexity ratio and seems to be the right “size” in that programmers tend to use the whole language. It has proven to be implementable, and it is possible to generate competitive code for ML programs. On the negative side there are the relatively minor design problems noted above and in [3], and the language probably shows its history more than necessary (*e.g.*, in the distinction between Core ML and the module system). ML has been and continues to be the subject of intensive research and the future promises further advances leading to new (and, one hopes, improved) languages in the ML family.

References

1. Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language*. Addison-Wesley, Reading, Massachusetts, 1988.

⁵ As suggested by Dana Scott.

2. William E. Aitken and John H. Reppy. Abstract value constructors: symbolic constants for Standard ML. Technical Report TR 92-1290, Dept. of Computer Science, Cornell Univ., June 1992.
3. Andrew W. Appel. A critique of Standard ML. *Journal of Functional Programming*, 1993. In press.
4. Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Martin Wirsing, editor, *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag.
5. D. W. Barron, J. N. Buxton, D. F. Hartley, F. Nixon, and C. S. Strachey. The main features of CPL. *Computer Journal*, 6:134–143, 1963.
6. Dave Berry. Lessons from the design of a Standard ML library. *Journal of Functional Programming*, 1993. In press.
7. R. M. Burstall, J. S. Collins, and R. J. Popplestone. *Programming in POP-2*. Edinburgh University Press, 1977.
8. Rod Burstall. Computer assisted proof for mathematics: an introduction, using the LEGO proof system. In *Proc. of the Institute of Applied Math. Conf. on the Revolution in Mathematics caused by Computing*, 1989.
9. The CAML Reference Manual (Version 2.6). Projet Formel, INRIA-ENS, March 1989.
10. Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.
11. Rance Cleaveland. The concurrency workbench, 1993. In this volume.
12. Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The concurrency workbench: a semantics-based tool for the verification of concurrent systems. *ACM TOPLAS*, 15(1):36–72, January 1993.
13. 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, Pittsburgh, PA, December 1990.
14. Robert Cooper and Clifford Krumvieda. Distributed programming with asynchronous ordered channels in Distributed ML. In Peter Lee, editor, *Proc. of ACM SIGPLAN Workshop on ML and its Applications*, pages 134–148, June 1992.
15. Pierre Crégut. Extensions to the SML module system. Rapport de Stage d'Ingenieur Eleve des Telecommunications, November 1992.
16. Bruce Duba, Robert Harper, and David MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 1993. In press.
17. Karen Frenkel. An interview with Robin Milner. *Comm. of the ACM*, 36(1):90–97, January 1993.
18. Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad. FACILE: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, April 1989.
19. M. J. C. Gordon, A. J. R. G. Milner, L. Morris, M. C. Newey, and C. P. Wadsworth. A metalanguage for interactive proof in LCF. In *Proc. 5th ACM Symp. on Principles of Programming Languages*, New York, 1978. ACM Press.
20. C. A. Gunter, E. L. Gunter, and D. B. MacQueen. Using abstract interpretation to compute ML equality kinds. *Information and Computation*, 1993. In press.
21. C. A. Gunter and J. C. Mitchell. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. The MIT Press, 1993.
22. C. A. Gunter and R. Rémy. A proof-theoretic assessment of runtime type errors. unpublished, 1992.
23. Robert Harper and Mark Lillibridge. Explicit polymorphism and CPS conversion. In *Proc. 13th ACM Symp. on Principles of Programming Languages*, 1993.

24. My Hoang, John Mitchell, and Ramesh Viswanathan. Standard ML weak polymorphism and imperative constructs. Ftp: ftp/pub/jcm/weak-poly.dvi on theory.stanford.edu, 1993.
25. Lorenz Huelsbergen and James R. Larus. A concurrent copying garbage collector for languages that distinguish (im)mutable data. In *ACM Symp. on Principles and Practice of Parallel Programming*, 1993.
26. P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–66, 1966.
27. Konstantin Laufer and Martin Odersky. An extension of ML with first-class abstract types. In Peter Lee, editor, *Proc. of ACM SIGPLAN Workshop on ML and its Applications*, pages 78–91, June 1992.
28. Xavier Leroy. Unboxed objects and polymorphic typing. In *Nineteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 177–188, New York, January 1992. ACM Press.
29. Xavier Leroy. Polymorphism by name for references and continuations. In *Proc. 13th ACM Symp. on Principles of Programming Languages*, 1993.
30. Xavier Leroy and Michel Mauny. Dynamics in ML. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture: 5th ACM Conference (LNCS 523)*, pages 406–426. Springer-Verlag, 1991. To appear in *Journal of Functional Programming*, 1993.
31. David B. MacQueen. Using dependent types to express modular structure. In *Proc. 13th ACM Symp. on Principles of Programming Languages*, pages 277–286, 1986.
32. Luigi V. Mancini. A technique for subclassing and its implementation exploiting polymorphic procedures. *Software—Practice and Experience*, 18(4):287–300, April 1988.
33. David C. J. Matthews. A distributed concurrent implementation of Standard ML. Technical Report ECS-LFCS-91-174, Laboratory for Foundations of Computer Science, Edinburgh University, August 1991.
34. Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, Cambridge, Massachusetts, 1991.
35. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1989.
36. John C. Mitchell and Robert Harper. The essence of ML. In *Proc. 15th ACM Symp. on Principles of Prog. Languages*, pages 28–46, New York, 1988. ACM Press.
37. John Ophel. An introduction to the high-level language Standard ML. In this volume.
38. Laurence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, Cambridge, 1992.
39. Lawrence C. Paulson and Tobias Nipkow. Isabelle tutorial and user’s manual. Technical Report Technical Report No. 19, University of Cambridge Computer Laboratory, January 1990.
40. John H. Reppy. Concurrent ML: design, application, and semantics, 1993. In this volume.
41. Eugene J. Rollins. SourceGroup: a selective-recompilation system. In Robert Harper, editor, *Third International Workshop on Standard ML*, September 1991.
42. Konrad Slind. HOL90: not just an ugly rumor! In *Proc. of the 1991 Int’l. Workshop on the HOL Theorem Proving System and its Applications*, 1991.
43. Mads Tofte. Principle signatures for higher-order program modules. In *Proc. 19th ACM Symp. on Principles of Programming Languages*, pages 189–199, 1992.
44. Mitchell Wand. Type inference for record concatenations and multiple inheritance. In *Proc. of the Symp. on Logic in Computer Science*, pages 92–97. IEEE, June 1989.
45. Andrew K. Wright. Polymorphism for imperative languages without imperative types. Ftp: public/languages/tr93-200.[dvi,ps].Z on titan.cs.rice.edu, 1993.

This article was processed using the L^AT_EX macro package with LLNCS style