A Critique of Standard ML

Andrew W. Appel*
Princeton University

revised version of CS-TR-364-92

November 12, 1992

Abstract

Standard ML is an excellent language for many kinds of programming. It is safe, efficient, suitably abstract, and concise. There are many aspects of the language that work well.

However, nothing is perfect: Standard ML has a few shortcomings. In some cases there are obvious solutions, and in other cases further research is required.

The **M**eta-**L**anguage of the Edinburgh LCF theorem-proving system [12] evolved into a free-standing programming environment [7] and then into *Standard ML* [29, 26]. After further evolution the language is fairly stable [31].

This is a critique of the language from two perspectives: the user's and the implementor's. The first part of this paper describes why ML is a pleasant language to use, and the second shows how some of these language features are interesting to compile. Then the third and fourth parts of the paper point out some of the annoyances ML programmers and implementors have to deal with.

*Supported in part by NSF grant CCR-9002786.

1 Why I like ML

In this section I list the reasons why I like programming in ML, in decreasing order of importance. Some features of the language for which ML is especially known fall surprisingly far down the list.

Safety

> Certain programming errors cannot always be detected [by a compiler], and must be cheaply detectable at run time; in no case can they be allowed to give rise to machine- or implementation-dependent effects, which are inexplicable in terms of the language itself. This is a criterion to which I give the name *security*.
>
> C. A. R. Hoare, 1973. [13]

One of the most pleasant things about ML is that it is *safe*: programs cannot corrupt the runtime system so that further execution of the program is not faithful to the language semantics.[1] Nelson [32] divides programming languages into three geneological categories: The BCPL family, including C and C++, which are not safe; the Algol family, including Pascal and Ada, which are almost safe; and the

[1]Thanks to the Modula-3 manual [32] for this phrasing

"mathematically derived" family, including Lisp, ML, Smalltalk, and CLU, which are safe—except when Lisp programmers disable runtime type checking because it's too expensive. (There are, of course, languages such as FORTRAN and COBOL that do not fall into these categories.)

In a safe language, all errors that could "derail" the program (cause behavior not explainable in terms of the source language) are detected either at compile time or at run time. [2] This makes it much easier to reason about program behavior: if an expression uses the first element a of a list l, we can be assured that l is really a list and not a misunderstood integer. Furthermore, a large class of storage-allocation mistakes common to unsafe languages are simply not possible in ML.

When fallible humans attempt to write large programs to do complicated things, safety is very important. Of course, safety is not the same thing as freedom from bugs. But at least the bugs can be understood in the framework of the language semantics (formal or informal). There is no behavior that cannot, in principle, be predicted from the program text.

In an unsafe language, program bugs that corrupt the runtime system are usually the most difficult to diagnose and have the most disastrous effects. But in a safe language, even buggy programs stay within the "semantic model" of the language, which makes program development much easier.

Garbage collection

Garbage collection frees the programmer from calculating the lifetime of every object in order to deallocate it. With automatic storage management it is possible to write programs more concisely, elegantly, and abstractly; one can manipulate *values*, instead of *objects* whose addresses must be remembered so they can be freed.

Even with a garbage collector, the programmer should avoid keeping unnecessary pointers to useless objects lest the program use too much space; occasionally it may be necessary to analyze and rewrite parts of the program to avoid keeping data structures live [37][4, chapter 12]. But this performance tuning is preferable to the "correctness tuning" necessary in a language with explicit *dispose*.

Without garbage collection, it is difficult to make a safe language that does interesting things. All modern languages, from all three of the families mentioned above, have dynamic storage allocation. But, in general, only languages of the "mathematical" category have automatic garbage collection. In the BCPL and Algol families, dynamic storage that is no longer active must be explicitly freed by the program if it is to be re-used. It is practically impossible (i.e., no one knows how) to make a *safe* language with explicit storage deallocation. This is the main (though not the only) reason that languages of the Algol family are not completely safe.

In some C or Pascal programs it is obvious where to put the *free* or *dispose* statements. But when data structures get just a bit more complicated, it's harder to predict when to dispose of things. Programmers often resort to explicit reference counts, or even to special-purpose mark-and-sweep garbage collectors implemented anew for each class of record.

The problem becomes worse across module boundaries. If a "server" module implements an abstraction using dynamic storage, then the "client" module won't know the format of the records to dispose of them. But the server won't know when the client is finished with the abstract objects. A typical solution is to add new operators to the abstract interface for freeing of abstract objects. This quickly becomes tedious.

Storage allocation bugs can corrupt the runtime system, or go undetected until millions of program statements have been executed after the error. Thus they are particularly nasty to diagnose. Safe languages of the "mathematical" family, including Standard ML, have automatic garbage collection and avoid this kind of bug entirely.

Compile-time type checking

Programmers make mistakes. Even when they have proved their algorithms correct in some formal or informal sense, it's difficult to avoid all errors when translating into the concrete formal notation of a programming language. Since I am particularly slapdash in my programming, perhaps I make even more mistakes than the average programmer.

So I must find my mistakes and fix them. Any help that the programming environment can give me in finding mistakes is most welcome. As a practical matter, I have found that the vast majority of my mistakes are found at compile-time by the ML type checker. These mistakes are particularly easy to fix, because:

- Compiling something takes less time than compiling and running it.

[2] In ML, anything detected at run time is considered to be an "exception," not an "error;" exceptions include such events as arithmetic overflow, array-bounds errors, and taking the head of an empty list.

- One compilation can find many compile-time errors; it's harder to find several bugs with one run (or even one compile and several runs) of a program.
- Compile-time errors are caught regardless of the input data; run-time type errors may not be caught until the program is exercised on many inputs.
- Compile-time errors often come with helpful explanations; run-time errors can be harder to diagnose.

Finally, compile-type types (especially the elegant type system of ML) help me to understand my program in a consistent way, so that perhaps I make fewer mistakes in the first place.

It is interesting to note that most languages of the "mathematical" family have had run-time type systems (in Lisp, Scheme, Smalltalk, APL, etc.), while the Algol-like languages have had compile-time type checking. Perhaps this is because the "mathematical" languages have garbage collectors; garbage collectors require some run-time type information to trace reachable objects; as long as the type information is in the run-time data there is a temptation to use it; or perhaps no one knew how to do good "mathematical" compile-time type-checking before ML's type system [28] was invented. Of course, run-time type checking can be slow; but the "mathematical" languages have not had raw speed as a primary design concern. In ML, the absence of run-time checking does make for more efficient implementation; this will be discussed below.

The module system

ML has a module system supporting abstract data types, hiding of representations, and type-checked interfaces. Modules are very important in structuring large software systems.

Much has been written about the advantages of modules and abstract data types. The "classes" of Object-Oriented programming are a kind of module, and support abstraction nicely; as are the "modules" of Modula and Ada. It is not controversial to say that modules with enforced interfaces and representation-hiding are an essential feature of a modern programming language.

ML's module system is particularly nice, in that it allows one module to be *parameterized* by the interface of another. Ada[1] and Modula-3[32] also support "generic" modules that are parameterized in this way. However, ML is unusual in that its parameterized modules—*functors*—can be compiled (with code generation) before any actual parameter is presented. The same arguments in favor of compile-time type checking also favor the checking of functors when they are parsed, independently of the arguments to which they might be applied.

In a language with parameterized modules *and* abstract data types, it's necessary to check that a given abstract type always refers to the same concrete representation—but at the same time, without "giving away" the representation. In Ada and Modula-3 such checking is possible because "compilation" (and type checking) of the parameterized module body is done for each application to actual parameters. ML uses the `sharing` spec[3] to require that two functor parameters must use the same representation for a shared abstract data type.

For example, suppose the signature (interface) `HASH` specifies a module to map strings to unique tokens. There are certainly different ways to implement this signature; and even the same implementation might exist in multiple instances, maintaining different hash tables. Now, if a parser module `Parse` with signature `PARSE` produces parse trees containing tokens, and a type checking module `Typecheck` (with signature `TYPECHECK`) also deals with tokens, they can be combined using a paremeterized module `Compiler`:

```
functor Compiler( structure P : PARSE
                  structure T : TYPECHECK
                  sharing P.Hash = T.Hash
                ) = . . .
```

The advantage of parameterizing `Compile` is that it can be applied to different parsers or different type-checking algorithms later on. But the program will be meaningless unless the particular `Parse` module we use relies on the same `Hash` table as the `Typecheck` module does. And—even worse—if the internal representation of the unique tokens is sufficiently different, then the program is not even safe from mistaking pointers from integers, etc. ML's module system may be unique in safely combining *compiled* parameterized modules with abstract data types.

Immutable values

In a functional language one describes the relationships between *values*, not *objects*. I will illustrate with a silly example. Consider the statements (in some programming language),

[3]Henceforth I will use *spec* to mean the syntactic construct in ML signatures, and *specification* in a more general, informal sense.

```
x := 1+6
y := 2+5
```

Now, to reason about the relationship between x and y, one might ask the following questions:

- Is x the same 7 as y?
- If we modify x, does y change?
- Need we make a copy of 7 to implement `z:=x`?
- When we're done with x how do we dispose of the 7?

If these questions seem silly, consider the analogous case for this program fragment:

```
x := cons(a,b)
y := cons(a,b)
```

Now, is x the same list cell as y? If we modify car(x), does y change? When should we make a copy of the cons cell? How do we dispose of it?

The disposal question is adequately handled in languages with garbage collection, of course. But the update and identity questions are not. It is very distracting, when writing and understanding a program, to worry about sharing of substructures, side-effects, and aliasing. (An optimizing compiler is distracted by these problems too!)

These questions are all silly for integers because we treat integers as values, not objects. If we considered integers as objects, perhaps with a command to "update" some of the bits of an integer object, then the complexities listed above would have to be considered by anyone programming with integers.

Values have many advantages over objects. Sharing of the substructures of values never leads to problems if the substructures can't be modified. One doesn't need to reason about equal versus identical values—and to ensure that this is true, ML does not permit testing address equality on immutable types. One can perform induction over structure to prove useful things about values; for objects one has to do induction over their histories, which complicates reasoning about them.

Mutable objects

Even though values have many nice properties, the notion of mutable objects should not be discarded. Only an extremist would say that updateable cells are always too hard to use and understand. The extremists might yet be proved right: it is certainly true that any algorithm on objects can be simulated on values, and recent work has made such algorithms ever more readable and understandable [43]. But there are millions of programmers who have sufficiently comprehended the notion of assignment and updateable data structures to write successful programs. Of course, the same argument could be made for bringing back the GOTO and the 64-kilobyte address space. But it is true that programming with updates is a proven technology, and programming *entirely* without them is still "research."

Now, other languages have combined a functional style with the capability to do updates—Scheme, for example. But the question is, how can these two styles be combined without losing the benefits of the immutable values? Once updates are permitted, the "silly" questions posed in the previous section begin to have complicated answers.

ML solves this problem by carefully segregating the mutable and the immutable types. An integer values has type `int`, and a mutable cell containing an integer has type `int ref`; these types are not the same. One can fetch the (immutable) value out of an `int ref` and bind it to a variable of type `int`; one can store a different (immutable) value in the `int ref`. Reference values are the only ones for which questions of sharing and identity are important.

Reference cells can be components of data structures. For example, `tree` shown below is the type of immutable trees with integer leaves; elements of `tree1` are trees whose leaves may be modified but whose structure is immutable. On the contrary, the leaves of `tree2` are immutable but the structure can be re-arranged (and entirely new leaves can be inserted):

```
datatype tree
       = LEAF of int
       | NODE of tree * tree

datatype tree1
       = LEAF of int ref
       | NODE of tree1 * tree1

datatype tree2
       = LEAF of int
       | NODE of tree2 ref * tree2 ref
```

Mutable reference cells, which are carefully identified in advance to the compiler and the human reader of the program, have turned out to be a very good compromise. They allow value-based reasoning about non-references, and the use of updates where necessary.

Polymorphic types

The implicit parametric polymorphism of ML is a great convenience. In writing a C or Pascal program that deals in linked lists of several different types of objects, for example, it is bothersome to have to copy almost verbatim the definitions of functions to create lists, map functions over lists, reverse lists, calculate lengths of lists, and so on. In ML, as in Lisp, the same `map` function can operate on a list of anything, and similarly for `length`, `reverse`, and `cons`. The `length` function is *polymorphic*: it has the type *int list* $\rightarrow$ *int* and the type *string list* $\rightarrow$ *int* and many others besides. In object-oriented languages with inheritance, polymorphism can be achieved without much difficulty (depending on the language). But in C, polymorphism can be accomplished only by using *cast* to avoid the type-checker, and in Pascal only by clumsy use of variant records.

Type inference

In ML it is never[4] necessary to declare types for variables or for functions and their formal parameters. The compiler can infer types for these identifiers, and it checks that the variables are used consistently. Thus ML achieves the advantages of compile-time type-checking with the conciseness of undeclared types.

This is a convenience, but of course it doesn't shorten programs by an enormous factor: in languages with explicitly declared types, the type declarations don't overwhelm the program. A big advantage of type inference is that the compiler infers the most general (polymorphic) type for each function. Then the programmer doesn't tend to prematurely over-specify the types of functions.

For example, consider writing a `length` function to compute the number of integers in a list:[5]

```
fun length (head::rest) = 1 + length(rest)
  | length (nil) = 0
```

Because the programmer needn't specify the type of the list element `head`, there is no temptation to overspecify it as `int`. So the length function, just as written, has type α *list* $\rightarrow$ *int* for any α, and can be applied to lists of strings, lists of reals, lists of lists, and so on.

Complete formal definition

The programming language Pascal was an advance in language design, and became very popular, for several reasons. It supported clean and useful control structures and data structures. It is a small enough language, and was specified precisely enough (in informal prose) [16] that people could understand what Pascal programs should do.

But Pascal still has "ambiguities and insecurities" [46]. That is, the language definition is *ambiguous* about the meaning of certain constructs (and different compilers give different results on the same program); and the language is *insecure*: it is not safe in the sense described by Hoare.

ML is not only secure, it is also unambiguously defined. *The Definition of Standard ML* [31] is a complete operational semantics for the entire language. One can use the *Definition* to calculate exactly which programs should be accepted by a compiler, and what their result will be.

Furthermore, the *Definition* (with accompanying commentary [30]) is readable—as formal semantic definitions go. This does not mean that the definition is suitable as a manual for the programmer; there is too much formal notation and not enough worked examples for that. But the student of language design, or the serious compiler-writer, can use the *Definition* as a reference to understand the meaning of any construct that might be in doubt. This leads to portability between implementations, provability of programs (in principle), and confidence in the safety and security of ML programs.

The Definition has, over time, proved to be tractable enough to serve as the basis for useful technical discussion among its many readers. Even when there have turned out to be holes in the Definition, they can be discussed and repaired with confidence and agreement over what the changes mean.

A formal definition is merely a complicated good-luck charm unless it can be used to prove important properties of the language. The *Definition* is mathematically tractable enough to prove, for example, that programs that type-check will execute "safely," that there can be no "dangling references" (invalid pointers), that the type inference algorithm always finds the most general type for an expression, and many other theorems that inspire confidence in the

---

[4] well, hardly ever; see the section on *Overloading.*

[5] A list in ML can be empty, or *nil*, or can be "cons" cell containing a "head" (first element) and a "tail" (the rest of the list). Thus, *list* is a disjoint union type, or `datatype`, of the following form:

```
datatype 'a list = nil | :: of 'a * 'a list
```

The constructors of this datatype are `nil` and `::` (pronounced "cons"). All the elements of a list must be of the same type; if this type is, e.g., α then the list is called an α *list*. Because keyboards don't have Greek letters, we write α as `'a`. It is convenient to make `::` infix and right-associative by default, so that `1::2::3::nil` is the list of the first three positive integers.

semantics of the language[6] [30].

The proponents of formal specifications of programming languages have long claimed that semantics should be used as a tool for language design, not just for writing down the semantics of existing languages. The conciseness and completeness of the ML *Definition* stem, in part, from the reluctance of the Standard ML design committee to admit features into the language for which they didn't understand how to write a provably sound semantics.

Higher-order functions

In ML, as in Scheme and other languages derived from the λ-calculus, functions are first-class values that may be passed as arguments, returned as results, and put into data structures.

Of course, the C programming language has "first-class" functions, too; but there is an important difference between the functional values of ML and those of C. ML has nested function definitions with lexical scope; the inner functions can refer to local variables and formal parameters of the outer functions. Thus, each time an outer function is invoked with different actual parameters, a "new" version of the inner function is built. A simple example:

```
fun add(x: int) =
   let fun f(y) = x+y
    in f
   end

val smallinc = add(1)
val biginc   = add(10)

val twelve = smallinc(biginc(1))
```

The `fun` keyword introduces a function declaration. The `let` *dec* `in` *exp* `end` syntax introduces a local declaration *dec* visible only in the expression *exp*. Thus, when `add` is applied to 1, the function $f_1(y) = 1 + y$ is created and returned as a result. When `add 10` is computed, the function $f_{10}(y) = 10 + y$ is the result.[7]

Imagine, for a moment, a programming language in which character-string values can be stored in variables, passed as arguments, returned as results; suppose there are character-string literals, and it's possible to extract the individual characters from string values. But suppose *there are no operators (such as* concatenate*) that can create new character-string values at run time*! Then the character-string type would be of limited utility; one might use it for printing interactive prompts defined at compile time, and so on. Any data type in which one can only pass around compile-time literals, is hardly "first-class."

But this is exactly the situation for function pointers in C! The only function values are those created at compile time; one cannot make "new" functions like f_1 and f_{10} shown in the example above. This is because C does not allow nested functions with lexical scope. Similarly, even though Modula-3 has nested functions and lexical scope, *only functions at the outermost level of nesting can be passed as arguments.*

On the other hand, Pascal allows nested functions (with lexical scope) to be passed as arguments, *but not to be returned as results or stored in data structures.* This restriction limits the utility of function values. Both the C restriction and the Pascal restriction are motivated by the desire to avoid the need for garbage collection: first-class functions with nested scope cannot be implemented with a conventional stack of activation records. But when the system has a garbage collector already, first-class nested functional values don't add great complexity to the implementation of the language.

Perhaps one must write some programs with higher-order functions to really appreciate their expressiveness. However, I will present some examples of their use:

Reduction functions on lists: Take a binary operator (like $+$ or $\times$), and apply it to an entire sequence of values, thus:

$$a_1 \times a_2 \times ... \times a_n \times 1$$

(Append the term $\times 1$ in order to appropriately handle the case where $n = 0$.) This notion can be easily generalized: given an operator *opr* and an identity I for that operator, $reduce(opr, I)$ is the function that applies the operator to an entire list of values. Thus, the function *sum* that totals the elements of a list is just $reduce(+, 0)$ and *product* is $reduce(\times, 1)$. In ML one might write:

```
fun reduce(opr,I) =
  let fun f(nil) = I
        | f(a::rest) = opr(a, f(rest))
   in f
  end
```

[6] Some of the theorems mentioned have actually been proved only for subsets of Standard ML.

[7] This `add` function can be written more concisely as
`fun add x y = x+y:int`
where the type constraint `:int` is necessary because of overloading; see section 3.

```
val sum     = reduce(op +, 0)
val product = reduce(op *, 1)
fun min(a, b:int) = if a<b then a else b
val infinity = 1000000000
val minlist = reduce(min,infinity)
val fifteen = sum(1::2::3::4::5::nil)
```

The `op` keyword allows an infix operator like `*` to be used as an ordinary identifier.

Window manager: One could organize a window interface so that an application running in a window is represented by its *keyboard* and *mouse*.[8] To hand the application characters typed into its window, one calls its *keyboard* function; to give it mouse-clicks, one calls its *mouse* function. Thus:

```
type window_app =
     {keyboard: string->unit,
      mouse: int*int->unit}
```

This says that `window_app` is a record type containing two fields, `keyboard` and `mouse`. `keyboard` is a function that takes a string parameter and returns "unit" (which is a placeholder like "void" in C), and `mouse` takes a coordinate-pair as an argument. Now, the window manager can pass keypresses and mouse-clicks to the application by calling these functions. This has an "object-oriented" flavor; the private data of the application (i.e., "self" in OOP terminology) is hidden in the free variables of the two functions. In C it would be necessary to include an explicit "self" field in the `window_app` record, and pass this as an extra argument to `keyboard` and `mouse`.

Most of the interesting uses of first-class functions combine the use of nested lexical scope (where inner functions' free variables are bound in outer functions) with functions returned as results or stored in data structures. Thus, the very combination that is left out of C and Pascal because it is difficult to implement (it requires a garbage collector for activation records) is the most useful.

Efficiency

An elegant language will have few applications if programs written in it always run too slowly. So it is important that ML can be compiled to run efficiently. There are many reasons to believe that it can. ML has compile-time type checking, which means that type tags need not be carried around at run time, and operators need not check the types of their arguments at run time. ML does not have the "dynamic method lookup" required of many object-oriented languages.

ML does do array-bounds checking, which is not present in C and which slows things down unless safely removed by a good optimizing compiler. ML does check pointers for **nil** before dereferencing; but the way this is incorporated in pattern-matching feature of the language, these tests will be part of the ordinary control flow written by the programmer. (Unfortunately, sometimes the programmer knows that a list can't be **nil**, but the check must be done anyway except by an impossibly intelligent compiler.) And ML checks for overflow of arithmetic expressions, but on most computers this is handled by the hardware without the need to issue extra instructions.

But can ML be as efficient as C? To some extent, this is still a research question (one that interests me very much). It's a difficult question to answer, because it requires that "the same" program be written both in C and in Standard ML. And what does it mean to say that a program written in idiomatic C is "the same" as one written in idiomatic ML?

One might make a good attempt at a quantitative measurement by rewriting some C programs in idiomatic ML, and vice versa, and running the results with "good" compilers on the same hardware. This is a sufficiently unrewarding job that few people have done it on "realistic" programs.

On the other hand, there are many good Scheme compilers. While Scheme does not run as efficiently as C on all problems, Scheme and Common Lisp are sufficiently efficient that many real applications are written in them. It should be possible to get ML to run at least as efficiently as Scheme, since the languages are similar in many ways but ML doesn't require the run-time type checking that Scheme does.

In any case, there is at least one reasonably efficient implementation of ML [6]. This and other implementations [9] have many users, for whom they

[8] An interesting and useful windowing library has been implemented in ML by Gansner and Reppy[36] as a very elegant interface to an X server. The example here does not describe their system.

[9] Several Standard ML implementations are available:

- Standard ML of New Jersey, from Princeton University and AT&T Bell Laboratories (contact appel@princeton.edu)
- Poly/ML, from Abstract Hardware Ltd. (contact bob@ahl.co.uk)
- Poplog ML, from the University of Sussex (isl@integ.uucp, pop@cs.umass.edu)

are adequately efficient; this might not be the case if they were too slow by an order of magnitude.

ML programs (run under some compilers) have used much more space than comparable C programs. This is a serious problem, but recent research [4, chapter 12] has hinted at solutions. At present, it appears that ML is efficient enough to use for a wide variety of applications. C programs are faster probably by no more than a factor of two, and often less than that. For many purposes, ML's advantages in safety, elegance, ease of storage management, and so on may outweigh this difference in performance. And programs that require complicated and expensive storage management in C may run *faster* in an ML implementation with a good garbage collector [8].

Why some people don't like ML

An (anonymous) early reviewer of this paper complained about ML's "lack of dynamic types, mutation (and lack thereof), lack of access to machine (as in C), restrictive type system, small changes usually require complete recompilation, bizarre syntax, lack of macros, etc."

These criticisms merit some discussion.

Lack of dynamic types: There are some things that are easier to do in a dynamically-typed language. For example, subtyping is easy to do in Lisp, since *list-of-real* is automatically a subtype of *list-of-(real-or-string)*; and ML doesn't have a subtyping mechanism. But such examples are not very compelling; an ML program might have a few more injection and projection functions than a Lisp program.

A more interesting use of dynamic types is for programs that wish to do type-safe, structured input/output, which is problematic in Standard ML. Within the ML community, the type *dynamic* has been proposed as a solution to this problem[22]: values of type *dynamic* would carry full ML-style types as part of their runtime representation, and could be coerced into ordinary statically-typed values with a runtime check.

- Edinburgh ML 4.0, from the University of Edinburgh (lfcs@ed.ac.uk)
- ANU ML, from the Australian National University (mcn@anucsd.anu.oz.au)
- MicroML, from the University of Umea, Sweden (olof@cs.umu.se)

Restrictive type system: ML's type system is less restrictive than that of most statically-typed languages (except those, like C, that allow evasion of the type system). In return for obeying the type rules, the programmer is rewarded with compile-time error messages instead of run-time bugs.

Mutation (and lack thereof): ML makes it incovenient (but not extremely so) to modify fields of data structures: such fields must be declared in advance. This is just enough to encourage a functional style of programming (which is good) with an escape hatch where necessary (which is also good).

Lack of access to machine: ML succeeds all too well in abstracting away from the machine. This makes it difficult to implement those programs that must do machine-level things, with memory words, pages, protections, signals, etc. It is possible to make interfaces to these things in ML; but it must still be admitted that a typical ML system has a large runtime system written in C to handle the things that couldn't be implemented in ML.

Recompilation: Separate compilation is essential in a programming environment. In statically-typed languages such as C or Modula, a system like **make** can recompile just those files that may need it; in dynamically-typed languages such as Lisp, only files actually modified need recompilation (in the absence of macro definitions, of course).

Implementations of Standard ML have not usually had very good separate compilation systems. This is partly a problem with the language, as elaborated in section 4, but mostly a problem with the individual implementations. In any case, it appears to be a problem that can be solved without modifying the language definition.

Bizarre syntax: Lisp syntax has a wonderful consistency, but is an acquired taste. Standard ML syntax is a mediocre example of the Algol school, in which keywords are used instead of some of the parentheses, and in which infix operators are used where it makes sense to do so. Some of the obvious "bugs" in the grammar are reported later in this paper; but in general, don't we have better things to argue about than syntax?

Lack of macros: This is clearly an advantage, not a disadvantage. For the programmer to have to calculate a string-to-string rewrite of the program before any semantic analysis invites problems of the worst kind. Where macros are used to attain the effect of in-line expansion of functions, they are doing something that should be done by an optimizing compiler. Where macros are used to attain call-by-name, the effect can be obtained by passing a suspension as an argument; in ML this is written with the syntax `fn()=>` which though admittedly ugly is fairly concise, and is better than tolerating the semantic havoc wrought by macros.

2 ML is fun to compile

Some of ML's characteristics enable compilers to use interesting techniques that are applicable to few other languages. On the other hand, many aspects of the language are best attacked by quite conventional techniques. And there are features of ML that might be considered an annoyance (or a "challenge") by compiler writers; these are described in section 4.

Safety

Compilers for safe languages, in which every compileable program has a well-defined result, can perform certain transformations that compilers for unsafe languages may not. For example, if the programmer cannot access data structures except through the "official" operators, then the compiler is free to choose arbitrary representations—even different representations for the same data structure in different parts of the same program. In an unsafe language, the programmer can access the underlying bit pattern of a data type; this tends in practice (and by convention) to force the compiler into predictable choices.

Another example of the use of safety is given below under the heading "Accurate control dependence." Essentially, the input program is the representation of a computable program, and the compiler may use "extensional equality" to substitute any other representation of the same function. On the other hand, in an unsafe language, some aspects of the program can be represented only by an operational semantics specifying a sequence of operations whose order cannot be rearranged.

Compile-time type checking

Compilers for languages with run-time type checking, such as Lisp and Smalltalk, must work very hard to minimize the execution cost of type checking. An advantage of ML (and all languages of the Algol and BCPL families) is that all type checking is done at compile time, and does not slow the execution of the program.

Representation analysis

The types of variables in ML are known sufficiently at compile time to guarantee, as in Algol-like languages, that primitive operators will never be applied to values of the wrong type. However, because of ML's parametric polymorphism, there are other contexts (such as inside the *cons* function) in which the types of (polymorphic) variables are not completely known. In such cases, the program always manipulates values without inspecting their internal representation. But in order to manipulate them (pass them as arguments, store them in data structures, etc.) it is necessary to know their size. The solution is to represent all polymorphic variables by bit-patterns of the same size (e.g., one word). Then polymorphism will work: at run time, polymorphic variables will be passed from one place to another by machine code that is oblivious of its actual type. This is exactly the strategy used in implementing Lisp: the *cons* function needs to know that the size of every object is the same, but does not need to know the internal representation of the objects it is *cons*ing.

This has been interpreted to mean that every variable, every function closure, and every argument of a function, must be represented in exactly one word. Where the natural representation of a value does not fit into one word (as with a list, a floating-point number, etc.), then a *pointer* to a heap-allocated object is used instead. This is a source of great inefficiency.

Parametric polymorphism is a useful kind of abstraction; abstraction often leads to inefficiency. ML programmers have always had to face this tradeoff, which the language has resolved in favor of abstraction. But perhaps it is possible to *pay for the abstraction only where abstraction is actually used.*

Xavier Leroy has recently pointed out that it is not necessary to represent *every* variable in one word, just *polymorphic* variables [21]. The type-checker can identify those places where non-polymorphic values are passed to polymorphic variables, and vice versa. Then the compiler can choose specialized representations, just as languages of the Algol family do, for nonpolymorphic variables. Then, to the extent that an ML program uses nonpolymorphic variables (as a Pascal program does), it will be as efficient as a Pascal program. This could be a very significant savings, as Leroy's measurements show. And it is a kind of optimization that would be impossible in Lisp (because the types cannot be safely analyzed at compile time).

Separation of static and dynamic semantics

In an ML compiler the static semantics (type checking) and dynamic semantics (evaluation) can be evaluated independently of each other, and in either order. In a compiler, dynamic semantics determines the machine code to be generated.

This may have interesting consequences for the implementation of a separate compilation facility. It should be possible to generate machine code for a module *in vacuo*; that is, without knowing the types of the module's free identifiers. Then, *at link time* the module can be type-checked, since the types of free identifiers then become known. Since code generation is much more expensive than type checking, we might gain significant benefit from this approach. The algorithms for *in vacuo* separate compilation have been worked out [38], and are now being implemented.

A more mundane advantage of the separation of static and dynamic semantics is that a simple, untyped intermediate representation can be used; and the translation of ML into this intermediate representation need not pay attention to types. This somewhat simplifies a compiler.

Of course, the *representation analysis* described above makes the implementation of dynamic semantics dependent on static semantics. So a compiler that uses link-time type checking, or a simpler translation to intermediate representation, could not take use representation analysis.

Immutable records

A common problem that plagues optimizing compilers is *aliasing*. It is often very difficult to determine when two pointers point to the same thing; this inhibits certain kinds of optimizing transformations. For example (in Pascal):

```
a := p^.x;
q^.x := b;
c := p^.x;
```

or, similarly,

```
a := p^.x;
f(x);
c := p^.x;
```

we might like to replace the statement `c := p^.x`, which involves a fetch, by `c := a`, which might be a register-register move. However, if there is a possibility that q points to the same record as p, (i.e., is *aliased*); or if $f(x)$ might modify `p^.x`, then this transformation is invalid.

It's no easier to solve aliasing problems in ML than in any other language. However, they don't need to be solved! Fetches from immutable objects cannot possibly be affected by any store instructions. And the vast majority of objects created are immutable (over 99% in a variety of real applications). Thus, most fetches can be moved past stores and procedure calls, and common subexpressions involving fetches from immutable objects can be eliminated. It is very pleasant to exploit this freedom in writing an optimizing compiler.

Mutable cells

In ML the updateable parts of data structures (`ref` cells) are identified at compile time. This could be useful to a garbage collector. Generational garbage collectors[24, 42] segregate heap-allocated records by age. Because records are initialized (to point to already-existing records) when they are created, newer records usually point to older records. The only way that an older record can point to a newer record is by an update to the older record after the newer one has been created. Generational collectors need to efficiently identify all those cells in an older generation that have been updated to point into a newer generation.

There are many ways to keep track of updated cells. A software approach is to have the compiler generate code after each assignment statement to keep a list of all cells updated [42]. It's not necessary to put newly-allocated cells on this list, of course. So all the compiler needs to do is distinguish initializing *store* instructions from updating stores. This is easy to do in ML, as it is in Lisp and any other language where records are initialized as they are allocated. It is more difficult in Algol-like languages where records are created uninitialized and are then stored into afterwards to initialize them.

An alternate approach to updates is to use the virtual-memory hardware of the computer [39]. By making older generations read-only, an updating store will cause a page fault. This fault can be handled by making the page writeable, and marking all the objects on that page as possibly updated. Then future updates to the same object, or to nearby objects, will not incur the cost of a fault.

The page-based technique will work best if there is locality of reference among the updates. It would be best, for example, to put all the mutable objects close to each other on a small set of pages, so that fewer updating page faults occur. This is possible if the runtime system can guess which objects can be or will be updated. Fortunately, in ML the `ref` cells can be distinguished from immutable records, data constructors, and closures, as they are created. The compiler can mark `ref` cells as they are allocated, or allocate them in a different area of memory, and the runtime system can rely on this marking. Such a technique is not possible in Lisp, since any object can in principle be updated (even though few objects are actually updated in practice).

It is interesting to compare ML (which allows programmers to execute updating side effects) with lazy functional languages such as Haskell [14], from the garbage collector's point of view. Since generational garbage collectors *hate* updates to existing objects, it would seem at first glance that a purely functional language with no assignment statement would be easier to garbage-collect. But lazy languages are constantly updating lazy closures ("thunks") with the results of evaluating them. Paradoxically, from collector's viewpoint ML has many *fewer* assignments than Haskell, and garbage collection in ML is likely to be more efficient.

Accurate control dependence

A statement guarded by a conditional is said to be *control dependent* on the conditional. However, this definition can be refined for safe languages such as ML.

Consider these two ML fragments and a C fragment:

```
a)  if i>0 then case q of u::v => u
                        | nil => ...
            else ...

b)  case q of u::v => if i>0 then u
                                else ...
             | nil => ...

c)  if (i>0)  if (j>0)  s = p->link;
```

In each case there is a fetch guarded by a two conditionals. The compiler might wish to hoist the fetch above the inner conditional, perhaps to improve instruction scheduling or register allocation.

In case (a) this is impermissible, since q might be `nil`—a fetch from `nil` might be illegal on the target machine. The pattern `u::v` ensures that q is a cons cell. In case (b) it is clearly permissible to hoist the fetch, since the validity of the pointer q cannot be affected by the value of i.

But in example (c) we cannot tell anything about the relationship between j and p. The programmer might know that j is the length of the linked list p, so that the fetch cannot be hoisted; or the value of j could be unrelated to whether p is *nil*, so the fetch can be hoisted. ML provides more precise information to the compiler than C does about the true control-dependences of fetches.

In summary, the safety of the language gives us a tool for reasoning accurately about control dependencies.

No pointer equality

Pointers in ML cannot be tested for identity. That is, except for `ref` cells, the program cannot determine if two similar objects are located at the same address. Since non-reference objects cannot be updated, the program cannot even perform the experiment of modifying one object and seeing if the other changes. This unusual feature leads to several interesting consequences.

Compilers can perform common subexpression elimination on record expressions. That is, in the program

```
val t = (a,b)
val s = f(x)
val u = (a,b)
```

the last line can be implemented as `val u = t` by the compiler. This transformation would not work in Lisp, Pascal, or almost any other language because the program would be able to test whether u and t pointed to the same address.

Compilers and garbage-collectors can do "hash-consing." That is, if the record `(a,b)` is to be created, and a similar record already exists (and can be found using a special hash table), then a pointer to the existing record is used instead of making a new one. In systems that allow address comparisons, hash-consing would entail an observable semantic change to the program; in ML it would not. Now, hash-consing may be intolerably slow. But consider a variation in which a generational garbage collector does hash-merging of objects that survive into the second generation. Then it's only necessary to hash a very small percentage of the objects that get allocated (since only a few objects survive a garbage collection). This idea has been implemented by Marcelo Gonçalves at Princeton University.

Garbage collectors like to move an object from one place to another; but then they need to update all the pointers to the object. A concurrent garbage collector might have trouble finding all these pointers quickly. In that case, it might be desireable to have two usable copies of the object—old and new—until all the pointers can be "forwarded" [33].

Distributed systems can copy objects without worrying about identity. Suppose we want to make the distributed nature of a system transparent to the programmer. If several processors want to look at a data structure at the same time, to obtain adequate performance it is necessary to copy pieces of the data structure onto the different processors. With a conventional programming language we now have to worry about address identity and making updates visible to all the processors. These problems are usually solved in hardware (e.g., with snoopy caches). In ML, worries about updates disappear for all but reference values, which are rare enough that conventional synchronization and message passing would be adequately efficient.

The module system

Run-time aspects of the module system turn out to be very simple [5]. A structure that exports n types and m values can be implemented as an ordinary m-tuple (types are needed only at compile time). Functors can be implemented as functions that take structures (tuples) as arguments and return structures as results. Since all inter-module linkage can be expressed this way, a conventional link-loader is not even necessary—which is particularly convenient in an interactive system that can load and execute programs and modules on the fly.

First-class continuations

A very interesting and powerful feature of Scheme[34] is the *call-with-current-continuation* mechanism, whereby the dynamic calling context of a function can be abstracted as another function. Standard ML does not have such *first-class continuations*; but it turns out that they can easily be introduced, and they fit very nicely into the ML type system[11].

First-class continuations make it easy to implement coroutines, or their generalization, lightweight processes [45]. Low-level details that must ordinarily be confronted in such implementations—such as the allocation of new activation stacks, the garbage-collector interface, and the mechanisms for saving registers to invoke a new thread—are all neatly encapsulated in the continuation mechanism.

Thread scheduling is much more efficient when done in the client process, without requiring hardware- and operating-system context switches when synchronizing or interleaving thread executions. Recent operating-system research [2] has shown how to let the operating system schedule *processors* while the client programs manage *processes* to take advantage of the efficiency of user-mode schedulers. In ML extended with first-class continuations, the scheduler can be a *source-language program* that manipulates continuations directly. This approach is very elegant and robust, and has proved successful in *Concurrent ML* [35] and *ML-Threads* [9], two quite different concurrent programming environments for ML.

3 ML traps and pitfalls

The syntactic and semantic pitfalls that an ML programmer encounters are much less severe and less numerous than those described in languages such as C [20], which is an egregious example.

Misspelled constructors

A well-known and most dangerous pitfall awaiting the ML programmer is the misspelling of a constant data constructor in a pattern. Because there is no syntactic distinction between constructors and variables, any identifier declared as a constructor is understood by the compiler as a constructor, and any other identifier is interpreted as a variable (which matches anything). Thus, a misspelled constructor looks like a variable, and is accepted by the compiler. For example, the misspelling of `nil` in this implementation of `length` causes the function always to return zero:

```
fun length (nill) = 0
  | length (head::rest) = 1 + length rest
```

In many cases (as in this one), the pattern-match will have redundant rules as a result of the programmer's mistake. Since the compiler warns about redundant rules, perhaps the error can be detected that way. But not in all cases. And warning messages are easily ignored by the programmer.

The approach Prolog takes to solve the same problem is to make constructors syntactically different from variables: Prolog constructors begin with lower-case, variables with upper-case. The same solution would not quite work in ML, for two reasons: ML allows "symbolic" identifiers such as

`::` and `+` that don't begin with a letter (and for which an upper/lower-case rule wouldn't apply); and ML allows data-constructors to be "thinned" to identically-named value bindings at module interfaces, so that what is seen as a constructor in one module is seen as a function (variable) in another module. These are both small things; they are cute but minimally useful, and programmers could easily work around their absence.

Some variation of the Prolog approach would solve this problem without significantly altering the nature of Standard ML. The Haskell language [14] uses such an approach.

Overloading

Most languages support some kind of overloading of operators, also known as *ad hoc polymorphism.* In its simplest form, this means that an operator such as `+` can be applied to integer arguments (yielding an integer result) or to real arguments (yielding a real result). This is not the same as the *parametric* polymorphism of ML or Lisp functions such as *cons* or *map*: The algorithm used to implement `+` is different for integers and reals, but the implementation of *cons* is the same for all types.

Languages of the Algol and BCPL families have always had overloaded operators built in, with overloading resolution (the determination of argument types, and therefore of what implementation function to use) at compile-time. Languages of the "mathematical" family have typically had overloading resolution at run-time.

Several languages in all three families have allowed programmers to define new overloaded identifiers, and to specify the implementation function to use for each argument type. Object-oriented languages, especially, have sophisticated support for user-defined overloading.

Compile-time overloading resolution and ML-style polymorphic type inference do not work well together [10]. In processing a function definition such as

```
fun double(x) = x+x
```

it is impossible to know at compile-time whether `+` is to be implemented as integer or floating-point addition.

This is not a dangerous "trap" for the programmer, since any ambiguous function such as `double` will be caught at compile-time as a type-checking error; the programmer will fix the problem (presumably) by inserting a type constraint, e.g.

```
fun double (x: real) = x+x
```

But it's a frequent annoyance; when writing a program on the integers I am just not thinking about real numbers, and I am constantly surprised to see the overloading-resolution failures. And in teaching the language, I must always qualify statements such as "The ML type inference algorithm can always derive a most-general type for any expression" with technicalities about a half-dozen built-in operators.

One way to solve this problem is to allow run-time resolution of overloading, as in the language Haskell [44, 15] and in other extensions of typed lambda calculus [18]. In these languages, class operators are passed (at runtime) as implicit extra arguments to functions that take polymorphic overloaded types as arguments.

But this mechanism makes dynamic semantics dependent on static semantics, which precludes certain kinds of separate compilation schemes. And Haskell uses a rather heavyweight mechanism for an apparently small gain. After all, making do with non-overloaded identifiers wouldn't make programs any bigger—one would just have to make up different names for different operations.

I am often asked whether I seriously mean that floating point addition should not be represented by the `+` symbol. That is exactly what I mean: Standard ML provides only a half-dozen overloaded operators anyway, and the use of `+'` or some such admittedly ugly symbol would be a reasonable price to pay for the deletion of overloading from the language. The designers of Standard ML considered the problem carefully and came to the opposite conclusion—so it must be a matter of taste.

Weak type variables

The ML type system, and type inference algorithm, works very effectively on programs without side effects. Particularly important is that the types are "intuitive:" the inferred types seem very natural and obvious to most programmers in most cases.

It has long been known that this algorithm does not work for polymorphic references. To illustrate with an oft-used example, consider

```
let val f = fn x=>x in f 1; f true end
```

The function f has the type $\forall \alpha. \alpha \rightarrow \alpha$, and can correctly be applied to an *int* and a *bool.*

But let f be a *reference* to a polymorphic function and the type inference algorithm cannot be naively applied. It seems natural to give polymorphic types to the `ref`, `:=`, and `!` operators:

$$\texttt{ref} \; : \; \forall \alpha. \alpha \rightarrow (\alpha \; ref)$$

$$:= \;:\; \forall\alpha.(\alpha\ \mathit{ref} \times \alpha) \rightarrow \mathit{unit}$$
$$! \;:\; \forall\alpha.\,\alpha\ \mathit{ref} \rightarrow \alpha$$

Now try to type-check the expression

```
let val f = ref(fn x=>x)
 in f := (fn x=>x+1);
    (!f) true
end
```

If f had type $\forall\alpha.((\alpha \rightarrow \alpha)\mathit{ref})$, then the program would (inappropriately) type-check, and would "go wrong" at run time by incrementing a boolean. So the naive polymorphic type checker has proved inadequate to handle reference cells. A more appropriate type for f might be $(\forall\alpha.\,\alpha \rightarrow \alpha)\mathit{ref})$, with the quantifier nested inside the *ref* constructor; but the ML type inference system cannot cope with "inner" quantifiers.

Cardelli's ML compiler [7], and the initial proposal for Standard ML [29], required that reference cells be completely monomorphic; that is, the compiler must be able to infer a type without type variables for any argument of the `ref` constructor. This is certainly safe, but insufficiently flexible. Tofte[41] generalized this idea, introducing "weakly polymorphic" references and "imperative types." These allow a function that creates references to be applied to more than one type, as long as each such type is itself monomorphic. Tofte's imperative types are a substantial improvement, and make for a usable language; they have been adopted as part of the Standard ML *Definition*.

However, Tofte's scheme can be made more flexible. In particular, it does not seem to work very naturally with higher-order functions; currying a function of imperative type can lead to a function that is rejected by Tofte's algorithm. MacQueen solved this problem by assigning numerical weakness indices to the type variables[27]. MacQueen's scheme is strictly more powerful than Tofte's, and has been implemented in Standard ML of New Jersey.

However, MacQueen's weak types aren't very easy for programmers to understand. It's difficult for the uninitiated to infer types for functions that make *ref* cells; typically I write the expression and get the compiler to print out the type, which I can then use in writing module signatures, etc. This approach to interface design is the opposite of that usually recommended!

The most annoying thing about Tofte's and MacQueen's imperative types is the "visibility" of locally-used references in interface descriptions. Consider a function

```
sort: (int * 'a) list -> (int * 'a) list
```

which is given a list of pairs; the first element of each pair is an integer key and the second element is of arbitrary type (though, of course, the same type for each element of the list). The `sort` function returns the list sorted by key. It is easy to write a purely functional quicksort or merge sort to solve this problem efficiently.

But suppose one expects all the integers to be in the range 1–1000, and the list contain thousands of elements. Then a bucket sort is faster, using an array of 1000 elements. But even though the array is not returned from `sort`, or retained way after `sort` returns, the type of this bucket-sort program would now be

```
sort: (int * '_a) list -> (int * '_a) list
```

indicating that the non-key elements of the list cannot be polymorphic values. It is too bad that this purely internal data structure must be "mentioned" in the interface.

Many researchers have recently been engaged in devising better type inference systems for polymorphic programs with references[25, 23, 17, 40, 48], which indicates that the problem of type-checking references is not yet regarded as "solved;" some of these systems address the problem of internal, temporary references described above.

The ML Grammar

The designers of Standard ML worked very hard to get the semantics right, and to define the semantics as completely and as formally as possible. Unfortunately, the same attention was not paid to syntax. Thirty years after Algol, and fifteen years after Yacc, *The Definition of Standard ML* does not contain an unambiguous context-free grammar for the syntax of the language.

As presented, the grammar is ambiguous for two reasons: The parser must "guess" whether an identifier in a pattern is a variable or a constructor; and it must "guess" whether an identifier is defined as `infix`, and if so, at what precedence and associativity.

These problems are not very difficult to solve semantically. For example, one might think the expression `a b c d e f` has to be parsed very differently if `b` is an infix operator than if `c` is. The solution is to parse such an expression as a sequence of atoms, and implement a simple precedence parser (37 lines of code in SML/NJ) as a "semantic action" for infix operators.

So the problem is not that ML has no context-free grammar; it's that the grammar is not clearly specified in the *Definition*. One immediately runs into problems when one wants to implement a parser for ML. A good language definition should include a complete LR(1) grammar with no reduce/reduce conflicts and as few shift/reduce conflicts as possible. Even if the implementor intends to parse using a different strategy (e.g., LL(1) or recursive descent), the LR(1) grammar is a useful starting point. The *Standard ML of New Jersey* implementation [6] uses such a grammar (with 68 terminals, 76 nonterminals, 231 productions, 452 LALR(1) states).

Most languages have a shift/reduce conflict with `else`. In the expression

```
if A then if B then C else D
```

it's not clear whether the `else` is supposed to match the first `then` or the second. This is customarily resolved by saying that the innermost (in this case, the second) `then` is matched; that is, an LR parser should resolve the conflict by shifting.

ML cleverly avoids this problem by requiring that every `if` have both a `then` and an `else` clause. But a similar problem occurs in `case` expressions:

```
case  A
 of X => case  B
           of  Y =>  C
  | Z => D
```

Now, is the `Z` pattern part of `case A` or `case B`? The *Definition* says that it's the latter; and this corresponds to resolving a shift/reduce conflict in favor of the shift. This is the only shift/reduce conflict in the *Standard ML of New Jersey* grammar.

Programmers have grown accustomed to the behavior of `if-then-else`. But as an ML programmer I often fall into the `case` trap: I often write pattern-matches like the one above. The solution is to enclose the inner `case` expression in parentheses, but I would rather the problem didn't occur in the first place.

These extra parentheses are ugly. In fact, having a shift/reduce conflict in the grammar is ugly. A better solution might be to require that `case` and `fn` expressions end with `end`, so the example above would be written:

```
case  A
 of X => case  B
           of  Y =>  C
          end
  | Z => D
end
```

Now there is no ambiguity. It is, however, a matter of taste whether the `end` is uglier than the extra parentheses.

There are some other syntactic glitches. It was clearly the intent of the designers to make semicolons optional after declarations. Thus, the declaration

```
val a = 5;
val b = 6;
```

would have the same meaning without the semicolons.[10]

This is a good thing; I'd rather not have semicolons cluttering up my programs (my prose is another matter). But it turns out that between a `structure` declaration and a `functor` declaration a semicolon is required (though not between two `structure` declarations or two `functors`). The only apparent reason for this discrepancy is that the syntax of module declarations was not carefully thought out.

Finally, I will remark that I have heard from many different people that they find ML syntax confusing, ugly, and difficult to learn. As a longtime ML programmer, I am quite comfortable with ML syntax; but perhaps the frequency of these complaints might serve as a hint that there is an opportunity for a syntax designer of rare taste and genius.

Infix operators

Programmers may define new infix operators in Standard ML, and may give them a precedence (between 0 and 9, where a higher number indicates tighter binding) and left or right associativity. If the programmer wants to define an exponentiation operator `**` and make it right-associative and tighter-binding than multiplication, the declaration `infixr 8 **` works quite well.

The *Definition* states

> `infix` and `infixr` dictate left and right associativity respectively; association is always to the left for different operators of the same precedence.

This is not as good a rule as it could be. Consider the list-like datatype

```
datatype 'a list2 = NIL
                  | $$ of 'a * 'a list2
                  | && of 'a * 'a list2
infixr 5 $$ &&
```

Here there are two flavors of cons cells. Then the expression

```
1 $$ 2 $$ 3 && 4 $$ NIL
```

is intended to be a "list2" of integers, some of which are marked with `$$` and others with `&&`, just as `1::2::3::4::nil` is an ordinary list of integers.

[10] The ML "top level" (read-eval-print loop) adds some twists of its own; these are discussed elsewhere in the paper.

In both cases, the cons operators (`::`, `$$`, `&&`) are meant to associate to the right. But the ML *Definition* requires that the "list2" expression above should associate to the left because different operators of the same precedence are used. Perhaps the *Definition* "meant" to say that "operators of the same precedence but opposite associativity associate to the left." But an even better rule would be that left- and right-associative operators of the same precedence don't mix without parentheses; this is the rule in Haskell [14].

Infix vs. Modules

Infix declarations are not exported from modules, and cannot be specified in signatures. This makes them significantly less useful.

For example, if one implements a module `Vector` to implement random-access, integer-keyed tables, one might want a signature like

```
signature VECTOR =
   sig  type 'a vector
        val vector: 'a list -> 'a vector
        val sub: 'a vector * int -> 'a
   end
structure Vector: VECTOR =  . . .
```

One might then want to make `sub` an infix operator, so that expressions like `V sub i` could be used for getting the ith element of a vector.

To use vectors in another module B, one could refer to the vector-creation function `Vector.vector` and the subscript function `Vector.sub`. But it is more convenient to write `open Vector` inside B, so that `vector` and `sub` can be used without prefix within B.

However, one cannot write `infix sub` in the signature `VECTOR`; within B the `sub` operator won't be infix unless there is a separate `infix sub` declaration in B.

The idea behind the module system is that an arbitrary piece of static environment can be "encapsulated;" then `open` will reconstitute that environment in another scope. By prohibiting this encapsulation of the "fixity" portion of the static environment, the *Definition* makes `infix` declarations second-class.

The only good argument against allowing `open` to reconstitute fixity declarations is that it might make programs hard to understand; the interpretation (i.e., fixity) of an operator cannot be understood by looking lexically upwards in the text of the program for a declaration of that identifier, because one might not notice the `open` of a module identifier (e.g., `Vector`). But this argument applies to all declarations implicitly introduced by `open`, not just fixity declarations. The semantics (i.e., type, value, etc.) of an operator can't be determined lexically because of the use of `open`; the programmer who can parse the operators but doesn't know what they do is almost as badly off as the one who isn't sure about operator precedence.

The *Definition* [31, page 10] states that "a more liberal scheme (which is under consideration)" would allow infix specs in signatures, and then an `open` declaration would re-install fixities of operators. Such a scheme has been implemented in Standard ML of New Jersey[6], and is quite convenient to use.

Separate compilation

The ML language definition is purposely quite vague about the pragmatics of putting programs together. The *Definition* chooses to pretend that all programs are typed into an interactive "top level" read-eval-print loop, and vaguely alludes to the fact that programs might be compiled from files.[11]

This is reasonable: there is nothing wrong with defining a programming language in the abstract, without tying it to the concrete details of operating systems and file systems. It is far better to underspecify this aspect of a language than to get it wrong.

However, modern languages with module facilities (including C, Modula, Ada) usually specify quite clearly which parts of a program can be compiled separately from the rest of the program: in C, a `.c` file generally requires some `.h` files for compilation, but not other `.c` files[19]; the Modula-2 definition[47] is even more specific about the organization of compilation units.

Since ML has a rather elaborate module system, it would seem that each module should be a separately compileable unit. But this is not necessarily the case; structures with free structure identifiers do not sufficiently specify what they are importing. The *Commentary* suggests some (severe) restrictions on the module system that would allow separate compilation. But on the whole, the relationship between structures, modules, and separate compilation could use some further work.

[11] In fact, most implementations have a function called `use` that allows files to be compiled; but they disagree on the semantics of nested `use`s.

Abstract structures

When a structure definition in ML is constrained by a signature, the representations of types are not hidden; they "show through." Thus, the declaration of a module implementing complex numbers,

```
signature COMPLEX =
   sig type complex
       val * : complex*complex -> complex
   end

structure Complex : COMPLEX =
 struct
   type complex = real * real
   val op * = fn ((r1,theta1):complex,
                  (r2,theta2):complex) =>
                  (r1*r2, theta1+theta2)
 end
```

does not hide the fact that the polar representation is used: **structure** declarations, even when constrained by signatures, allow type and sharing information to "show through" the constraint. Other modules that make use of the Complex structure will be able to access the components of a complex number, unless they import Complex as the parameter of a functor. I have found that most people learning ML are surprised by this, because the signature declaration itself makes no mention of the representation.

In some cases the transparency of signatures is necessary and useful; but in many cases it would be useful to use the module system to implement *abstract data types*. MacQueen's original module proposal[26] provided for **abstraction**, a special kind of structure declaration in which all type representation and sharing information not specified in the signature constraint would be hidden. Giving programmers the choice between **structure** and **abstraction** would better support programming with abstract data types. Abstract datatypes with hidden representations are the apple pie and motherhood of modern software engineering, and rightly so.

Of course, there exist other mechanisms for abstract data types in Standard ML (**abstype** and **functor**). But it is particularly convenient to use abstract data types at the module level, where **abstraction** is more straightforward than **abstype**. And functors can be a clumsy mechanism for structuring programs.

The *Commentary* to the definition shows that **abstraction** is not semantically problematical [30, page 85], and even gives a useful generalization of MacQueen's proposal. It's a pity that this feature was omitted from the *Definition*.

open in signatures

It is customary, in writing modular software, to specify the interfaces between modules and to implement the modules to meet those interfaces. Even when the programmer develops the implementation first, it is good practice to pretend otherwise by writing the interface signature and cleaning up the implementation as necessary to meet the signature. Then the reader of the program can first understand the interfaces (which are generally more concise than the implementations), and then proceed to learn about the implementation of one module at a time. The *signatures* of the Standard ML module system support the writing of clear interface specifications.

Now imagine an interface definition that says, in effect, "the signature S is defined to be whatever interface happens to be met by the implementation module M." Then to understand S, one must read through the entire implementation M, inferring types for all the values, keeping track of which identifiers are visible in the outermost scope. A right-thinking software engineer should certainly frown at such a method of defining interfaces.

But this is exactly what is provided by **open** specs in signatures! The signature

```
signature S = sig open M end
```

specifies that the interface S is just whatever (largest) interface is obtained by elaborating the structure M.

The **open** spec may be pleasingly symmetrical to the theoretician; it may be technically useful in defining the semantics of the rest of the module system. But it has no place in a real programming language.[12]

A related problem has to do with overlapping **open** (or **include**) specs. Since **open M** or **include S** has the effect of including many identifiers, it is easy for the programmer to inadvertantly (or even purposefully) include two different signatures containing the same type, value, or structure identifier. Though there is no ambiguity in the semantics (the later spec takes precedence), multiple definitions make the scope of specs much more complicated to follow, and make the implementation of semantic analysis for signatures and sharing much more difficult.

The scope rules for ML expressions, while simple, are not completely trivial; and that is appropriate: programs are complicated things. But it seems

[12] A **sharing** constraint can also relate a signature interface to a free structure. But this is not so problematical for the reader of the program, since it has no effect on the visibility of names.

worthwhile to strive for extreme simplicity in interface (signature) definitions: scope rules for signatures should be trivial. A clear understanding of the interfaces of a program is a prerequisite to understanding the program. Removing `open`, `local`, and `include` specs[13] from Standard ML would result in much cleaner interfaces, without causing great inconvenience.

One of the arguments for `include` is that it helps in writing concisely a signature for modules that satisfy several different specifications. Consider a signature `HASH` of hashable values, and a signature `GROUP` for mathematical group structures:

```
signature HASH =
 sig type value
     val hash : value -> int
 end

signature GROUP =
 sig type elem
     val id : elem
     val * : elem * elem -> elem
     val inverse: elem -> elem
 end
```

How can these be combined to make a signature for hashable groups? With `include`, one could write

```
signature HASHGROUP =
 sig include HASH
     include GROUP
     sharing type value = elem
 end
```

But substructures serve almost as concisely, without using `include`:

```
signature HASHGROUP =
 sig structure H: HASH
     structure G: GROUP
     sharing type H.value = G.elem
 end
```

In fact, the latter approach is more robust, since unfortunate naming coincidences between the two signatures can be distinguished by qualified identifiers (imagine that the HASH signature also had an identity function `id: value->value`). The only disadvantage is that the client of `HASHGROUP` must either `open G` and `H`, or use qualified identifiers such as `G.id` instead of `id`.

4 Problems in compiling ML

ML is designed to be compiled: many things can be evaluated at compile time. ML has static types, static (lexical) scope, statically-checked modules. However, some aspects of the language design are hard to compile efficiently.

Polymorphic equality

ML has an operator `=` to test the equality of two values (which must have the same type). Values of any of the primive types (integer, real, string, etc.) may be tested for equality, but values of function type may not. Abstract types, of which the programmer has purposely hidden the representation, also do not "admit equality;" they are not "equality types."

Any values of a record type or datatype built only from "equality types" may be compared for equality. Equality of records, lists, and so on is *structural*: the record (x_1, y_1) is equal to (x_2, y_2) if $x_1 = x_2$ and $y_1 = y_2$; there is no way to tell if the two records are at the same address.

This is all very well, but now there is a complication. Consider the program

```
fun alleq(a,b,c) = a=b andalso b=c

val t = alleq(3,3,3)

val x = alleq(fn x=>x+1, (* ILLEGAL! *)
              fn x=>1+x,
              fn x=>x+1)
```

The function `alleq` should have a type resembling $\forall\alpha.\, \alpha \times \alpha \times \alpha \rightarrow \mathit{bool}$, so that we can pass three integers to it, or three strings, or three lists of real numbers. But we cannot pass any values of a type (such as $\mathit{int} \rightarrow \mathit{int}$) that does not admit equality; thus the last declaration must be illegal. (After all, to tell whether two functions are "equal" the compiler must be able to tell whether they give the same results on all inputs, which is rather difficult.)

In Standard ML the problem is resolved by introducing "equality type variables," which can be instantiated only by types that admit equality. Thus, the type of `alleq` is something like

$$\forall\alpha^=.\ \alpha^= \times \alpha^= \times \alpha^= \rightarrow \mathit{bool}$$

where we can substitute int for $\alpha^=$, but not $\mathit{int} \rightarrow \mathit{int}$. In an (ASCII) ML program, equality type variables are written starting with two apostrophes instead of just one.

This seems like a clever solution, but it introduces three kinds of problems into the ML language:

1. The static semantics of the language become very complicated;

[13] I am not proposing to remove `open` declarations from expressions, just `open` specs from signatures.

2. code generation and the runtime system require unpleasant special cases;

3. and perhaps programming with equality types isn't a good idea anyhow.

Static semantics: Now the language designers must worry about type constructors that admit equality, specs in signatures of types that admit equality, propagation of the equality property through sharing constraints and functors, and so on. In *The Definition of Standard ML*, no fewer than *twenty-two pages* mention some syntactic or semantic aspect of equality types; this is approximately *one out of every four pages of the Definition.* The ramifications of equality similarly metastasize throughout a Standard ML compiler. Equality types add significant complexity to the language and its implementation.

Dynamic semantics: In almost every respect the type checking of an ML program is distinct from the evaluation of the program. Thus, type checking can be done at compile time, and type tags need not be carried on runtime objects. This saves considerable space and time, and is one of the most important features of the language.

But a function (such as `alleq`, above) must be able to test variables for equality, even though the type of these variables is polymorphic and not known until run time. There are two ways that this might be accomplished:

1. The runtime representation of each object can have sufficient tag information to determine whether the object is a pointer, and if so, how many fields are in the pointed-to record, and whether the record is a `ref` cell. Then an "equality interpreter" can recursively traverse data structures to test bitwise equality on non-pointers, and structural equality on pointers. I believe this is the solution chosen in all existing ML compilers.

2. The representation of any formal parameter whose type is a polymorphic equality type variable could be a pair, whose first field is the value itself and whose second field is a function for testing equality on values of that type. Then a function such as `alleq` could use these implicit parameters to perform equality testing. This is the solution adopted in Haskell[44], which generalizes the notion of equality types to include other kinds of overloading.

There are disadvantages to either solution. The first requires runtime tags which are otherwise not necessary for ordinary execution. The argument is often made that these tags are there to allow the garbage collector to traverse pointers and records. But it's possible to devise a garbage collector that relies on the *static* type information computed at compile time [3], without any runtime tags on data. Furthermore, even a conventional garbage collector might use a BIBOP (Big Bag of Pages) scheme that groups many objects of similar type on the same page, so that one tag suffices for all of them. Then the runtime "equality interpreter" faces a very complex task in understanding the structure of objects.

As to the provision of implicit arguments to functions, this is workable but inelegant. As the *Commentary on Standard ML* states, "the static and dynamic semantics can be studied independently of one another." [30, preface] In structuring a compiler, it is very convenient that translation of expressions into machine language is independent of the types of the expressions. Requiring that some expressions must be treated specially depending on their types corrupts the interface between the components of the compiler.

Programming with equality types: An oft-used example of the utility of equality types is the implementation of *sets* (with union, intersection, etc.) as lists. Thus,

```
fun set(x) = x::nil
fun member(x, nil) = false
  | member(x, a::r) = x=a orelse
                        member(x,r)
fun union(a::r,b) =
              if member(a,b)
                    then union(r,b)
                    else a::union(r,b)
  | union(nil,b) = b
```

Then these functions can be used to make sets whose elements are any type α, as long as α admits equality (i.e., doesn't contain components of functional or abstract type). And the programmer doesn't even have to provide an explicit equality function—the compiler figures it all out.

But there are two very significant problems with this program, and these problems are sufficiently general that they may affect any program that makes much use of equality type variables. First, the set union function takes quadratic time. Any realistic program that deals with sets will want to make set union take linear time; and this can only be done if there is some sort of ordering (less-than) comparison operator available on the elements, or some way to hash the elements to integer values. Thus, a "production quality" set abstraction will be parameterised by more than just an equality function.

Second, consider what happens with sets of sets. As an example,

```
val a = union(set(1),set(2))
val b = union(set(2),set(1))
val x = set(a)
val t = member(b,x)
```

The set x has a single element that is the set $\{1,2\}$; the last line tests the set $\{2,1\}$ for membership. Of course, the program will tell us that $b \notin \{a\}$, which violates the set abstraction. The problem is that structural equality is the wrong equality to use on sets; the programmer should really provide an `eq_set` function that tests whether two sets have the same elements.

Thus, implicit structural equality is often bad programming practice. The programmer should provide an explicit equality function because (1) the explicit function will likely be more efficient to use, and (2) the explicit function will have the right semantics for the application.

A reasonable compromise would be to allow a kind of *statically* overloaded equality function, of the kind found in earlier versions of Standard ML [29]. This equality operator worked on any non-functional *monomorphic* type. Such an operator is quite convenient to the programmer, and does not unduly complicate the language semantics, compiler, or runtime system. (Half as many pages of the Definition[14] would mention equality; equality attributes would cease to interact with the type checker or the module system; no "equality interpreter" would be needed in the runtime system.) It must be admitted that with this solution (as with ML overloading) we are left without principal types in some cases.

Datatype representations

Recursive data types are declared in ML using `datatype`, which defines the constructors (and associated types) of a disjoint union type. Linked lists are just a special case of this more general notion.

The runtime representation of a typical datatype element consists of a *constructor* and an associated value. A straightforward implementation of this representation would be as a two-element record, with one field containing a small integer tag (standing for the constructor) and the other containing the value (since ML has polymorphic types, every value must be the same size—one word in a typical implementation).

This scheme, if applied to a datatype like *list*, would require that the representation of `a::b` be a pointer to a two-element record containing a constructor and a value; the value would be a pointer to another pair containing a and b. Each element of the list, then, requires not one "cons cell" but two!

Cardelli's ML compiler[7] avoided this extravagance by taking advantage of the fact that in the runtime representation of values, pointers could be distinguished from small integers. Thus, the compiled code could tell which constructor (`nil` or `::`) had been applied by seeing if the value was a small integer (`nil`) or a pointer (`::`). The pointer would then point directly at a record containing a and b. Thus the representation of lists in Cardelli's compiler (and in every subsequent ML compiler) is just like the representation used in Lisp.

In fact, all these compilers generalize the idea slightly: in any datatype with just one non-constant constructor (and any number of constant constructors), if the non-constant constructor carries a value that is always represented by a pointer, then an extra indirection to carry the constructor is not necessary.

Now, consider the following perfectly legal Standard ML program:

```
functor F(type 'a t
          datatype 'a list =
              nil |  :: of 'a t
          ) = struct . . . end;

structure S =
  F(datatype 'a list =
         nil |  :: of 'a * 'a list
    type 'a t = 'a * 'a list
   );
```

In compiling the functor F, the compiler does not know whether the representation of `'a t` is always a pointer; so an explicit indirection (a record for the constructor) must be used in the representation of *list*.

But in compiling the structure S, the actual parameter has a datatype *list* in which the value carried by `::` is a record, and thus always a pointer. So the representation chosen by the compiler will use Cardelli's optimization.

Then when lists created outside of F are passed to functions inside F, the program will go wrong: different compilation units will disagree about the representation of lists.

Thus, Standard ML does not permit Cardelli's

[14] Pages 4, 18, 19, 21, 22, 25, 26, 74, 75, 77, 79 of the Definition[31] would still mention equality; pages 13, 16, 33, 35, 36, 39, 40, 41, 43, 44, 57 would no longer need to.

optimization;[15] but all the implementations use it because the alternative is too expensive.

The problem is a bit more general. There are many other possible generalizations of Cardelli's technique, all with the aim of making the representations of datatypes more compact and efficient. None of these techniques work across functor boundaries.

Cardelli's technique is a variant of the idea, *pay for abstraction only where things are abstract.* Leroy's representation analysis applies to functions; Cardelli's to data structures. But it appears that this idea cannot be made to apply to recursive datatypes in Standard ML; this is extremely unfortunate. I believe the problem lies in the partial abstraction of datatypes. In the example above, the programmer has abstracted $\alpha \times \alpha\, list$ into $\alpha\, t$, but has not abstracted the datatype *list*. This is an unusual program. The whole point of a concrete datatype is that it is *not* abstract; if the programmer wanted an abstract type in the interface then the parameter of F wouldn't have mentioned a datatype at all.

Thus, a solution to this problem might be to change very slightly the notion of a datatype. Instead of saying that a datatype is the disjoint sum of several types, let us say that it is the disjoint sum of several *product* types. That is, the value carried by a constructor is not just a type, it is a record type. Note that this is exactly the way that a variant record works in Pascal.

Then the problematic program above would not be legal. The functor definition would be allowed, but the datatype in the actual parameter would not match the datatype in the formal parameter.

This slight restriction would allow compilers to use much more efficient representations of concrete datatypes in ML. At present we are experimenting with an implementation of this representation (and consequent language restriction) to explore this tradeoff.

One might think that a compiler should also represent each element of an $(int \times int)list$ as a triple $(int, int, tail - pointer)$. But here the product type $(int \times int)$ is not part of the datatype itself, but part of the type parameter of the *list* constructor. This would lead to problems when polymorphic functions on list types are applied to a specially-represented lists. Thus, such an optimization has problems not only at functor boundaries but at function boundaries.

[15] Cardelli, of course, was not compiling a language with `functors`.

The initial basis

The *Definition* specifies an *initial basis*, that is, a set of predefined types, values, and exceptions that are the "built-in functions" (etc.) of any ML system. These include the arithmetic operators on integers and reals, string concatenation, a few operators on lists, and so on.

The initial basis is not large enough to write real programs that use nontrivial input/output, or that interact much with the operating system. That's perfectly acceptable; this is a language definition, not a library module. The type and module systems of Standard ML are adequate to describe appropriate libraries, and that's what is important.

But the initial basis, such as it is, has some rough edges:

- There are functions for reading and writing strings of characters, for converting integers into single-character strings (and back), and for concatenating strings, and for "exploding" strings into lists of single-character strings, and "imploding" (concatenating a list of strings together). But there is no way to access the ith character of a string in constant time—there is no *substring* operator! The only way to extract an internal character of a string is to *explode* the string and then to traverse the resulting list; this takes time linear in the length of the string.

- There is no way to make updateable arrays with constant-time access to arbitrary elements. Arrays can be simulated by lists (or trees) of *ref* cells, but access and update operations will then take linear (or logarithmic) time. Updateable arrays are certainly not out of place in a language with updateable *refs*.

- The arithmetic operators may overflow, in which case the *Definition* prescribes that `+` will raise the `Sum` exception, `*` will raise the `Prod` exception, and so on. It is extremely inconvenient for the implementor to have distinct exceptions for the different operators; most computers don't raise separate hardware exceptions for different kinds of overflow. And the programmer would almost always be served just as well by a single exception called `Overflow`.

- There is no bit string type, and there are no bitwise logical operators on the integer type. There are many applications of bitwise operators in graphics, number theory, cryptography, and other areas. On the other hand, it is worth noting that ML's `div` and `mod` have rounding behavior (towards negative infinity, not towards zero) that allow shifts and masks to be defined using powers of two; compilers could optimize this case, in principle.

- Upon an input/output error, the `Io` exception is raised with a string argument. The format of the argument is specified in the *Definition*, and this format does not provide enough information for serious applications. It would have been preferable to leave the contents of the string unspecified rather than prematurely settling on an inadequate standard.

- To finish on a trivial note: the list concatenation operator `@` is declared infix, associating to the left. Programs would compute the same result under right associativity, but would run faster, since `@` must copy its left argument but not its right one.

It is worth noting that every implementation of ML since Cardelli's has had a constant time array subscript and an efficient substring function; the *Definition* could have provided a helpful standardization.

5 Conclusion

The popularity of ML seems to be increasing, both as a language for writing real programs and as a starting point for theoretical investigations of type theory and language design. Programmers should note that the good points of ML discussed in this paper are all rather general and important; the criticisms tend to be narrow, technical, and not always important.

Theorists should note that, even though some of the criticisms are minor and not of much theoretical interest, they all affect the usability of the language. Those theorists who anticipate designing a language themselves someday might want to remember this critique, along with the classics of the genre[13, 46].

Acknowledgment

I would like to thank Doug McIlroy and an anonymous referee for many valuable comments.

References

[1] Military standard: Ada programming language. Technical Report MIL-STD-1815, Department of Defense, Naval Publications and Forms Center, Philadelphia, PA, 1980.

[2] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Trans. on Computer Systems*, 10(1):53–79, February 1992.

[3] Andrew W. Appel. Runtime tags aren't necessary. *Lisp and Symbolic Computation*, 2:153–62, 1989.

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

[5] Andrew W. Appel and David B. MacQueen. A Standard ML compiler. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture (LNCS 274)*, pages 301–24, New York, 1987. Springer-Verlag.

[6] 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.

[7] Luca Cardelli. Compiling a functional language. In *1984 Symp. on LISP and Functional Programming*, pages 208–17, New York, 1984. ACM Press.

[8] Will Clinger and Lars Thomas Hansen. Is explicit deallocation really faster than garbage collection? unpublished manuscript, 1992.

[9] 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.

[10] Luis Damas. *Type Assignment in Programming Languages.* PhD thesis, Department of Computer Science, University of Edinburgh, 1985.

[11] Bruce Duba, Robert Harper, and David MacQueen. Typing first-class continuations in ML. In *Eighteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 163–73, New York, Jan 1991. ACM Press.

[12] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF.* Springer-Verlag, New York, 1979.

[13] C. A. R. Hoare. *Hints on Programming-Language Design*, pages 193–216. Prentice Hall, 1989. keynote address to the ACM SIGPLAN conference in 1973.

[14] Report on the programming language Haskell: Version 1.1. Technical Report Technical Report, Yale University and Glasgow University, August 1991.

[15] Report on the programming language Haskell, a non-strict, purely functional language, version 1.2. *SIGPLAN Notices*, 27(5), May 1992.

[16] Kathleen Jensen and Niklaus Wirth. *Pascal: User Manual and Report.* Springer-Verlag, New York, 1974.

[17] Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *Eighteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 303–310, New York, Jan 1991. ACM Press.

[18] Stefan Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *Proc. 1992 ACM Conf. on Lisp and Functional Programming*, pages 193–204. ACM Press, 1992.

[19] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1978.

[20] Andrew Koenig. *C Traps and Pitfalls*. Addison-Wesley, Reading, Mass, 1989.

[21] 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.

[22] 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.

[23] Xavier Leroy and Pierre Weis. Polymorphic type inference and assignment. In *Eighteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 291–302, New York, Jan 1991. ACM Press.

[24] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–29, 1983.

[25] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Fifteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 47–57, New York, Jan 1988. ACM Press.

[26] David MacQueen. Modules for Standard ML. In *Proc. 1984 ACM Conf. on LISP and Functional Programming*, pages 198–207, New York, 1984. ACM Press.

[27] David B. MacQueen. `weak-types`. Distributed with Standard ML of New Jersey, 1988.

[28] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–75, 1978.

[29] Robin Milner. A proposal for Standard ML. In *ACM Symposium on LISP and Functional Programming*, pages 184–97, New York, 1984. ACM Press.

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

[31] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1989.

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

[33] Scott Nettles and James W. O'Toole. Carnegie Mellon Univ., Pittsburgh, PA, personal communication from Scott Nettles, 1990.

[34] J. Rees and W. Clinger. Revised report on the algorithmic language Scheme. *SIGPLAN Notices*, 21(12):37–79, 1986.

[35] John H. Reppy. Concurrent programming with events. Technical report, Cornell University, Dept. of Computer Science, Ithaca, NY, 1990.

[36] John H. Reppy and Emden R. Gansner. The eXene library manual. Cornell Univ. Dept. of Computer Science, March 1991.

[37] Colin Runciman and David Wakeling. Heap profiling of lazy functional programs. Technical Report 172, University of York, Dept. of Computer Science, Heslington, York Y01 5DD, United Kingdom, April 1992.

[38] Zhong Shao and Andrew W. Appel. Smartest recompilation. In *Proc. Twentieth Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, page to appear. ACM Press, 1993.

[39] Robert A. Shaw. Improving garbage collector performance in virtual memory. Technical Report CSL-TR-87-323, Stanford University, Palo Alto, CA, 1987.

[40] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region, and effect inference. Technical Report EMP-CRI E/150, Ecole des Mines de Paris, February 1991.

[41] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, November 1990.

[42] David M. Ungar. *The Design and Evaluation of a High Performance Smalltalk System*. MIT Press, Cambridge, MA, 1986.

[43] Philip Wadler. The essence of functional programming (invited talk). In *Nineteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 1–14, New York, Jan 1992. ACM Press.

[44] Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *Sixteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 60–76, New York, January 1989. ACM Press.

[45] Mitchell Wand. Continuation-based multiprocessing. In *Conf. Record of the 1980 Lisp Conf.*, pages 19–28, New York, August 1980. ACM Press.

[46] J. Welsh, W. J. Sneeringer, and C. A. R. Hoare. Ambiguities and insecurities in Pascal. *Software—Practice and Experience*, 7(6):685–96, 1977.

[47] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, New York, NY, 1981.

[48] Andrew K. Wright. Polymorphic references for mere mortals. In *Proceedings of the European Symposium on Programming*, 1992.