A Standard ML Compiler

Andrew W. Appel *

Dept. of Computer Science
Princeton University
Princeton, NJ 08544

David B. MacQueen †

AT&T Bell Laboratories
Murray Hill, NJ 07974

ABSTRACT

Standard ML is a major revision of earlier dialects of the functional language ML. We describe the first compiler written for Standard ML in Standard ML. The compiler incorporates a number of novel features and techniques, and is probably the largest system written to date in Standard ML.

Great attention was paid to modularity in the construction of the compiler, leading to a successful large-scale test of the modular capabilities of Standard ML. The front end is useful for purposes other than compilation, and the back end is easily retargetable (we have code generators for the VAX and MC68020). The module facilities of Standard ML were taken into account early in the design of the compiler, and they particularly influenced the environment management component of the front end. For example, the symbol table structure is designed for fast access to opened structures.

The front end of the compiler is a single phase that integrates parsing, environment management, and type checking. The middle end uses a sophisticated decision tree scheme to produce efficient pattern matching code for functions and case expressions. The abstract syntax produced by the front end is translated into a simple lambda-calculus-based intermediate representation that lends itself to easy case analysis and optimization in the code generator. Special care was taken in designing the runtime data structures for fast allocation and garbage collection.

We describe the overall organization of the compiler and present some of the data representations and algorithms used in its various phases. We conclude with some lessons learned about the ML language itself and about compilers for modern functional languages.

---

* Supported by NSF Grant DCR-8603453 and by a Digital Equipment Corporation Faculty Incentive Grant.

† Part of this author’s work was done while an SERC Senior Visiting Fellow at the University of Edinburgh.

1. Introduction

The ML language is a typed functional language roughly based on Landin's ISWIM [1]. It was originally designed in the mid-1970s as the metalanguage of Edinburgh LCF, a machine-assisted reasoning system, and its features were in part motivated by its intended use to express proof tactics. However, these features made it an attractive vehicle for general purpose symbolic programming, and it wasn't long before free-standing implementations appeared, such as Cardelli's compiler[2], which was written in Pascal. In 1983 a group of interested parties began work on an extensive revision of the language design that lead to Standard ML [3, 4, 5, 6]. Standard ML extended the earlier versions in certain ways and incorporated ideas from Hope[7], another language developed in Edinburgh in the late 1970s. It also included a module facility that significantly extends the basic polymorphic type system of earlier ML versions and supports large-scale program development.

Several implementations of Standard ML have been under development in recent years. Luca Cardelli's compiler was modified to be partially compatible with Standard ML. Meanwhile, Cardelli's original compiler had been reimplemented in its own variant of ML by Kevin Mitchell and Alan Mycroft at Edinburgh, and this compiler was also modified with the help of Robert Harper to make it conform fairly closely to the Standard ML definition. A new compiler was developed in Cambridge by David Matthews, using his Poly language [8, 9] as the implementation language, and sharing the Poly back end. At INRIA, a group headed by Gerard Huet and Guy Cousineau have been implementing an ML variant called CAML[10] that is intermediate between the LCF version and Standard ML.

Yet there was still justification for another Standard ML compiler. One of the main motivations was to build a compiler that would itself be implemented in Standard ML. Such a compiler would have several advantages. It would be a good basis for building meta-level tools (pretty-printers, program analyzers, debuggers, etc.) that could share data types and code with the compiler itself and might be used in ML application programs or in an ML programming environment. It would also serve as a large-scale test of the Standard ML design, particularly the module facility, which was the most novel and untested part of the language. Finally, a new compiler provides an opportunity to try out new implementation strategies (for code generation, optimization, type checking, and modules, for instance). A simple lambda-calculus intermediate representation makes the compiler much cleaner, and could also provide a base for research in evaluation algorithms (call-by-value, normal-order, lazy, combinator-based, parallel, and others) for lambda-calculus.

As a call-by-value, higher-order functional language, many of the conventional compilation techniques for functional languages apply to Standard ML. The distinguishing features of ML are its polymorphic type system, the notion of concrete data types for expressing disjoint unions and recursion, the use of pattern matching based on the constructor-functions associated with data types, the use of typed, value-carrying exceptions, the fact that all updateable variables are introduced explicitly by application of the **ref** constructor, and modular programming based on signatures (interface specifications), structures (implementations of interfaces) and functors (functions from structures to structures). These features, singly or in combination, produce novel challenges in compiler design. They also, it turns out, can be effectively exploited in the writing of a compiler.

The compiler was written using the Edinburgh Standard ML compiler and then bootstrapped to compile itself. However, the compiler shares no code with previous compilers; and the runtime system, written in C with the help of Peter Weinberger, is also completely new.

2. Architecture

The ML compiler described here consists of three phases. The front end consists of a recursive descent parser into which environment management (determining scopes, associating binding and applied occurrences of variables) and type checking are integrated. The output of the front end is a fully typed abstract syntax tree. The front end makes no decisions about runtime locations of variables. It simply assigns them unique identifiers, and the back end makes representation decisions.

The middle end performs a simple translation of this abstract tree into an untyped lambda calculus representation, and produces an optimized representation of pattern-matching as one-dimensional switch expressions.

The back end transforms the lambda form into target machine code using a largely machine-independent code generator based on an ‘‘abstract machine’’ interface. The expressions given to the back end never contain free variables; this simplifies the interface between the code generator and the runtime system.

3. The Front End

The front end of the compiler consists of a conventional lexical analyzer, abstract syntax datatypes, environment management machinery, a type checker, and a recursive descent parser that drives the lexical analysis, environment management and type checking.

3.1. Abstract syntax

The abstract syntax is implemented in two layers. There is a collection of data types defining the ‘‘bare’’ abstract syntax that forms a minimal kernel of the language. For instance, the *expression* data type is defined as follows:

```
datatype exp = VARexp of var ref
             | CONexp of datacon
             | INTexp of int
             | REALexp of real
             | STRINGexp of string
             | RECORDexp of (numberedLabel * exp) list
             | SEQexp of exp list
             | APPexp of exp * exp
             | CONSTRAINTexp of exp * ty
             | HANDLEexp of exp * handler
             | RAISEXexp of exp
             | LETexp of dec * exp
             | CASEexp of exp * rule list
             | FNexp of rule list
```

The full abstract syntax consists of these bare syntax types augmented with a small number of derived forms, such as clausal function definitions. These derived forms are implemented as functions that generate the appropriate expansion into bare syntax forms.

3.2. Environment management

We separate the idea of symbol-table manipulation from the details of the kinds of bindings found in ML. There is a generic environment mechanism that performs binding, lookup, and scope management functions; it is implemented as a functor that takes the binding type as a parameter. The binding type is a disjoint union of the specific categories of identifiers (variables, constructors, type constructors, etc.):

```
datatype binding
  = VARbind of var            (* variables *)
   | CONbind of datacon       (* data constructors *)
   | EXNbind of datacon       (* exceptions *)
   | TYCbind of tycon ref     (* type constructors (patchable) *)
   | TYVbind of tyvar         (* type variables *)
   | SIGbind of signatureVar  (* signatures *)
   | STRbind of structureVar  (* structures *)
   | FCTbind of functorVar    (* functors *)
   | FIXbind of fixity        (* infix attributes of variables *)
```

The generic environment uses hash tables, which we call binding tables or simply tables, to map symbols to bindings. The environment consists roughly of a stack of tables to hold the actual bindings and a stack of ‘‘remarks’’ that records information about bindings and scopes. Only the stack of tables is used when looking up symbols, while the remarks are used when entering and leaving scopes and when binding symbols. There are two types of tables in the stack of tables: those representing the bindings in a previously defined structure that has been ‘‘opened’’, and those representing bindings in currently open scopes (i.e. top-level, let-bound, and function parameter bindings).

As indicated above, the binding type is a union of all the various kinds of bindings a symbol may have. The tables therefore hold bindings of all kinds, and the specialized access functions for variables, constructors, etc. search only for bindings of the desired kind. Thus it is possible for a symbol to simultaneously have several bindings of different kinds, e.g. as a variable, a type constructor, and an exception. However, variables and constructors share the same name space, so it is not possible for a symbol to be simultaneously bound as a variable and a constructor, despite the fact that these are two disjoint kinds of bindings. The information associated with a symbol, such as the type of a variable or the signature of a structure, is contained in the projected value of its binding (e.g. the var or structureVar).

3.3. Parsing

The lexical analyzer is conventional, turning the input character stream into a stream of tokens, recognizing keywords, identifiers, and string and numeric literals. The parser reads tokens from the lexical analyzer and produces an abstract syntax tree containing type and binding information. Parsing is done by recursive descent, with a precedence parser for infix operators in expressions. Unlike previous ML compilers, but like most compilers for Algol-like languages, management of the compile-time environment is done at the same time as parsing: identifiers are entered into the environment as they are defined, and looked up in the environment as they are used.

Two particular problems crop up in parsing ML: identifiers may be declared infix, or have their precedence changed, in lexically-scoped declarations; and constructors cannot be syntactically distinguished from identifiers. To handle the infix-operator problem, the lexical analyzer makes no distinction between ordinary identifiers and identifiers that are declared as infix operators. Instead, the operator-precedence expression-parser looks up each identifier in the symbol table to determine if it has a fixity binding, and if it does, what its precedence is. Recognizing constructors can be done similarly by looking up identifiers in the environment. This scheme works nicely in the presence of modules and **open** declarations: if a module *A* is opened in a scope (even a local scope), and an infix identifier *i* is used from *A*, the precedence parser will automatically find the precedence declaration of *i* in the environment.

Forward reference to identifiers is legal in mutually recursive definitions. For example, in the function declaration:

```
fun f(a,b) = if a=0 then b else g(a-1)
and g(x) = f(x,x)
```

the first occurrence of the identifier `g` is in the body of `f`, which is before the function `g` is declared. To handle this problem, all free identifiers found in the body of a recursive function definition are kept on a ‘‘backpatch list;’’ when the mutually recursive declarations are completely parsed, any references to identifiers defined in those declarations are patched. This extends straightforwardly to nested mutually recursive definitions. Note that constructors are not treated this way; constructors cannot be redefined by **fun** or **val rec** definitions, so they are not put on the backpatch list. This means, of course, that all identifiers must be looked up just to determine if they are constructors.

3.4. Type Checking

The conventional polymorphic type checking algorithm[11, 12] is used at present. The basis of the algorithm is the unification of type terms by destructively instantiating type variables so that they become indirections to other types. The administration of generic or bound type variables is done more systematically than in the Cardelli and Edinburgh compilers, but generic types are still copied for each applied occurrence of a variable. A new type representation and type checking algorithm based on structure sharing (in the Boyer-Moore sense [13]) has been prototyped with the help of Nick Rothwell. In this new scheme, a polymorphic type is represented as a scheme whose bound variables are indices into an environment vector; different instantiations share the scheme but have different environment vectors.

Overloading is accommodated to a limited extent. Currently only certain predefined primitives such as ‘‘+’’ are overloaded, and there are no facilities provided by the language for defining new overloaded identifiers. Overloading is implemented as in the Edinburgh compiler, where a type scheme is associated with the overloaded identifier and each overloading (variant) has a type that is an instance of that scheme. The contextual type of an occurrence of the identifier is matched against the scheme and the resulting instantiations of the scheme type variables is used to choose the appropriate variant. This is simpler though more restrictive than the technique used in Hope[7], where variants could have arbitrary types, as long as they were incomparable (i.e. one was not an instance of another).

3.5. Reference and exception types

The treatment of references and exceptions with ‘‘open’’ types is new, and is based on the fact that the contents of a reference cell cannot be constrained to be polymorphic, and so must be considered to be monomorphic. The following example illustrates the problem.

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

If s were given the polymorphic type ALL $\alpha.(\alpha\rightarrow\alpha)$ ref, then this expression would type check, permitting an obvious type error. To prevent this, we insist that the type of an applied occurrence of the ref constructor should always be given a ‘‘ground’’ type (one with no locally-bound type variables).

However, functions whose application can create reference variables can still have polymorphic types of a restricted kind. Consider the declaration

```
val F = fn x => let val r = ref x
                 in !r
                end
```

Here the function F can be given polymorphic type ALL $\alpha^1.\alpha^1\rightarrow\alpha^1$ where α^1 is a special kind of type variable called a *weak* type variable (the superscript ‘‘1’’ indicates that there is one lambda abstraction suspending the creation of the ref cell). When F is applied to an argument, a reference value of type α^1 is created, and hence this weak type variable must be instantiated to a ground type. This means that an expression like (F nil) would not be properly typed. In contrast, the type α^1 ref assigned to r is permissible because α^1 is implicitly bound in an outer scope and within the scope of its binding is treated as a constant type.

Exception declarations raise similar problems, which are handled by an analogous use of weak type variables.

The Cardelli and Edinburgh compilers used an earlier form of this treatment proposed by Damas[14]. This earlier version was looser in one respect (it allowed unbound weak type variables in types) and more restrictive in another (multiple levels of lambda abstraction were not allowed), and it had some rather counterintuitive effects.

3.6. Modules

The simplest kind of module in Standard ML is called a structure. It is most simply defined as an encapsulated set of declarations, as in

```
structure S =
  struct
     type t = int * int
     val x = (13, 18)
  end
```

The compile-time symbol table representation of such a structure is a binding table containing a binding for each identifier declared in the structure. There is also an abstract syntax tree for the structure, from which code is generated.

A signature is an interface specification for a structure. For instance, the structure above will match the following signature:

```
signature SS =
  sig
     type t
     val x : t
  end
```

Signatures are also represented by binding tables, but an abstract syntax tree is not needed since there is no code generated for signatures.

A functor is a structure abstracted with respect to one or more structure parameters, which are characterized by signatures. It follows that the representation of a functor consists of a structure representation (the body) and a parameter specification. As part of the parameter specification one can impose sharing constraints among the parameters; these constraints are represented by transforming the parameter specification into a directed acyclic graph of parameter structures.

Functors introduce some additional forms of structures, namely formal parameter structures and structures defined by functor applications. For formal parameters, the parameter signature serves as a virtual template defining the components of the structure. In the case of functor applications, the functor body (which may itself be a functor application) is closed with respect to the environment formed by binding actual parameters to the formals.

Functor application in principle involves a beta-reduction in which the actual structure parameters are substituted for the formal parameters in the functor body to create the result structure. This beta-reduction acts at two levels—the static level, involving the type aspects of the structures involved, and the dynamic level, involving the value and exception components. The dynamic aspect of the reduction is realized by generating code that performs an ordinary function application. The static aspect is performed at compile time, producing a static representation of the result structure by ‘‘instantiating’’ the body of the functor using the formal/actual parameter binding environment.

3.6.1. Naive copying

This static instantiation of functor bodies can be implemented in several ways. The simplest and most direct is to perform a straightforward nondestructive substitution, copying the representation of the functor body in the process. This is roughly the approach followed by Harper and Matthews in their implementations, but the space consumed has been found to be excessive.

3.6.2. Functor application closures

A second alternative initially employed in our compiler was to represent structures formed by functor applications as closures, consisting of the unmodified functor body structure together with the parameter binding environment. This approach saves some copying, but it turns out to be rather unwieldy because functor applications can appear within the body of a functor definition, and consequently the actual parameters may be expressions that themselves refer to other formal parameter variables. For instance, consider

```
functor F(X:sigX) = bodyF

functor G(Y:sigY) =
  struct
    structure S = F(Y)
    ...
  end

structure B = G(A)
```

In this example, the interpretation of `B.S` is given in terms of the closure of `F(X)`, which is a pair $<bodyF,\ \{X \mapsto Y\}>$. Since the binding of `X` involves `Y`, this must be interpreted in the additional context of the binding environment $\{Y \mapsto A\}$ produced by the application `G(A)`. In general, a functor application closure may need several layers of such contexts to be properly defined.

Another drawback of this approach is that it still performs the copying that is inherent in the naive implementation of signature matching, where a declaration such as

```
structure S′: sig1 = S
```

may cause a partial copy of `S` to be constructed and bound to `S′`.

3.6.3. Structure-sharing

The final approach is inspired, like Harper's prototype implementation, by the semantic model developed by Harper, Milner, and Tofte[15]. We return to the straightforward idea of actually performing the static reductions to obtain instantiated copies of the functor body, but a structure-sharing representation is used to minimize the amount of copying. In this representation, which is similar in principle to the structure-sharing representation of polytypes alluded to above, the statically ‘‘interesting’’ components of each structure (i.e. types and substructures) are represented by indices into an environment vector that is associated with the binding table in another, simpler form of closure. The copying of structures that is entailed by functor application and signature matching can then be reduced to copying the closure objects and their associated environment vectors, leaving the binding tables themselves unaffected.

Each type or structure component also has an identifying ‘‘name’’, which is basically just a unique number or time stamp. These names serve a couple of purposes: representing sharing constraints, and identifying two sorts of ‘‘bound’’ components. To capture sharing specifications in signatures, components that are required to ‘‘share’’ (i.e. represent views of the same structure) are given the same name. There are two forms of bound components: those incorporated in functor parameter structures, and those representing the components of a functor body that are created each time the functor is applied. All other names are free, and represent actual structures. The first sort of bound names are mapped to components by the process of signature matching between formal and actual functor parameters. The second sort are replaced by new unique names during the static elaboration of functor applications.

This third approach is currently under development and will eventually replace the implementation based on functor application closures.

4. Translating abstract syntax into lambda calculus

The middle end of the compiler translates type-checked abstract syntax trees into lambda-calculus trees. Because all of the environment and scope manipulation has been done by the front end, and all of the abstract-machine manipulation will be done by the back end, the translator is simple, small, and fast.

Though most optimization is purposely left for the back end, some simplification is done in the middle end. Formally recursive **fun** definitions are examined to see if they really contain references to themselves; if not they are replaced by (simpler) nonrecursive definitions. The composition of structure-creation and structure-thinning is evaluated at compile-time rather than run-time. The ML equality predicate is a special function that can be applied to any concrete data type (one built up from primitive types and reference types using record and datatype constructors). The algorithm for testing equality is a recursive tree traversal that varies in its details for each type to which it is applied. For each such occurrence, the translator builds an equality predicate appropriate to the instance.

Recent changes to Standard ML will require the use of an ‘‘equality interpreter’’ which traverses arbitrary structures in the runtime system, distinguishing certain kinds of cells (like **ref** cells) by special descriptor tags. The present implementation can still be used as an optimized version in those cases where enough is known about the type at compile time.

4.1. Translation of pattern-matching

One important and nontrivial job of the middle end is to select optimal comparison sequences for the compilation of pattern-matching. A *match* in ML is a sequence of pattern-expression pairs, called *rules*. When a match is applied to an argument, the argument is matched against the patterns, and the first rule with a matching pattern is selected and its expression is evaluated. A pattern is either a constant, which must match the argument exactly; a variable, which matches any argument (and is bound to it for the purposes of evaluating the expression); a tuple of patterns, which matches a corresponding tuple argument whose components match the components of the pattern-tuple; or a constructor applied to a pattern, which matches an argument built using that constructor if the rest of the pattern matches.

As an example, consider the case statement:

```
case a
 of (false, nil)   =>  nil
  | (true, W)    =>   W
  | (false, cons(X, nil))  =>  cons(X, cons(X, nil))
  | (false, cons(Y, Z))  =>  Z
```

The argument (false, cons(4,nil)) matches the third pattern, while the argument (true, cons(4,nil)) matches the second pattern.

One could imagine a naive compilation of matches just by testing the rules in turn as called for by the semantics. Our approach is to transform a sequence of patterns into a decision tree[16]. Each internal node of the decision tree corresponds to a matching test and each branch is labeled with one of the possible results of the matching test and with a list of the patterns that remain potential candidates in that case. It is then straightforward to translate the decision tree into code for pattern matching. During the construction of the decision tree it is also easy to determine whether the pattern set is ‘‘exhaustive,’’ meaning that every possible argument value matches at least one pattern; and whether there are any ‘‘redundant’’ patterns that only match arguments covered by previous rules. Nonexhaustive and redundant patterns result in warning messages by the compiler.

Our goal in constructing the decision tree is simply to minimize the total number of test-nodes. This minimizes the size of the generated code and also generally reduces the number of tests performed on value terms. However, finding the decision tree with the minimum number of nodes is an NP-complete problem[16]; so a set of efficient heuristics is used that in practice produces an optimal decision tree in almost all cases.

In the example above, testing the first component of the pair for truth or falsity suffices to distinguish the second rule from the others; then testing the second component to see whether it is `cons` or `nil` distinguishes the first rule from the last two; one more test suffices to separate the last two rules. Thus, in just two or three tests, the appropriate rule can be selected; instead of two or three tests *per rule* that the naive algorithm would use.

The details of this algorithm were originally worked out by Marianne Baudinet and have been implemented in our compiler by Trevor Jim.

4.2. The Lambda Language

The front end of the compiler translates ML source into lambda calculus; the back end translates lambda-calculus into machine code for the VAX or MC68020. A significant advantage of having a very simple lambda-calculus as the intermediate representation is that many compiler optimizations can be cleanly described. This is the approach successfully taken in the Rabbit[17] and Orbit[18] compilers for Scheme.

The ‘‘lambda language’’ is simply an ML datatype, as follows:

```
datatype lexp
  = VAR of lvar
  | FN of lvar * lexp
  | FIX of lvar list * lexp list * lexp
  | APP of lexp * lexp
  | CON of con * lexp
  | DECON of con * lexp
  | SWITCH of lexp * (con*lexp) list * lexp Option
  | RECORD of lexp list
  | SELECT of int * lexp
  | RAISE of lexp
  | HANDLE of lexp * lexp
```

The elements of the lambda language are variables (VAR), abstraction (FN), simultaneous recursive definition (FIX), application (APP), constructors (CON, DECON, and SWITCH), tuples (RECORD and SELECT), and exception handling (HANDLE and RAISE). Exception names, and integer, real, and string literals, are represented as constructors.

There are no built-in library functions, at least from the point of view of the compiler; instead, the library module is lambda-bound at top level to some variable *v*, and functions from the library are just components of the structure *v*. Certain functions that are compiled in-line are represented as fields of a special structure bound to a distinguished variable v_0.

Constructor-expressions are explicitly represented in the lambda language using $\mathrm{CON}(c,e)$ representing the application of the constructor *c* to the expression *e*, and $\mathrm{DECON}(c,e)$ representing the removal of *c* from the constructed expression *e*. The lambda language could be simplified by representing constructor-expressions as pairs of (tag,value), and removing the CON and

DECON operators. The advantage of using CON and DECON is that the lambda language can be a typed lambda calculus; the disadvantage is that the code generator is somewhat more complicated. Since it's not clear why the low-level representation needs to be typable, we may change the representation of constructors in the lambda language.

A structure resembles a record of components; in the lambda language, it is treated exactly as a RECORD. There are no special forms in the lambda language for structures and functors. Functors are treated as functions from structures (RECORDs) to structures. Actually, a structure definition is not just a record; it is an expression that *evaluates* to a record.

Structure *A* may refer to an element of structure *B*; in the lambda language this will mean that *B* is a free variable of the expression representing *A*. To keep things simple, it is important to eliminate all free variables from an expression before code is generated for it; so the variable *B* will be lambda-bound at top level in the definition of *A*, and the runtime system will be instructed to apply the resulting pseudo-functor to the structure *B* to initialize it. This is a form of automatic, compiler-controlled link-loading. For a link-loading scheme it is extremely simple; our runtime structure-manager and link-loader is less than a page of ML code.

5. Code generation

The back end of the compiler is organized as the composition of functors; these functors are most naturally described starting at the end and working toward the front. The last phase is the backpatching of jumps and other relative addresses in a machine-language program. Relative jump instructions on many machines are of different sizes depending on the distance jumped, and several iterations of estimating jump sizes may be required before a fixed point is found[19]. This is handled in a machine-independent way by the Backpatch functor:

```
signature RelativeAddresses =
  sig  type JumpKind
       val sizeJump : JumpKind -> int
       val emitJump : (int -> unit) -> JumpKind -> unit
  end
```

```
signature BackPatch =
  sig
    type Label
    val newlabel : unit -> Label
    type JumpKind
    val emitbyte : int -> unit
    val align : unit -> unit
    val define : Label -> unit
    val jump : JumpKind*Label -> unit
     . . .
  end

functor Backpatch( Rel : RelativeAddresses ) : BackPatch
  = struct
        . . .
    end
```

For a given machine with a particular kind of relative address, one builds a small structure explaining what size of relative address can reach what distance, and how to emit code for that sort of relative address. For example, on the Vax, a short conditional branch takes 2 bytes, but a medium-size conditional branch requires 6 bytes (a short conditional branch around a longer unconditional jump), and a longer conditional jump takes 8 bytes.

When the Backpatch functor is applied to a structure of type RelativeAddresses, a new structure is created that understands how to backpatch the code for a particular machine. A code generator for that machine can make use of the *emit* and *label* primitives provided by the specialized Backpatch structure.

The signature VaxCode (not shown here) provides an interface to a useful subset of the Vax instruction set. The various instructions and addressing modes are defined in this signature. There are two structures that implement the VaxCode signature in different ways: one that generates machine code when the instructions in the signature are called for, and one that generates assembly code. The VaxMcode structure matches the VaxCode signature, and makes use of the Backpatch functor:

```
structure VaxMcode : VaxCode =
struct
   structure R : RelativeAddresses =
    struct
       . . . functions to compile relative jumps, etc.
    end
   structure B = Backpatch(R)
   datatype AddressMode = . . .
   fun movl (src : AddressMode, dst: AddressMode) = . . .
end
```

There is a peephole optimization module that operates on abstract Vax instructions. It is implemented as a functor taking any VaxCode structure into an ‘‘optimized’’ VaxCode structure:

```
functor Peephole ( V : VaxCode ) : VaxCode = struct . . . end
structure OptVaxMcode = Peephole(VaxMcode)
```

If some other structure asks the module OptVaxMcode to produce a particular instruction sequence, it may be that some optimized version of the sequence will be generated instead.

The code generator translates programs from lambda language into machine code by means of an abstract machine intermediate form. The abstract machine is similar in spirit to Cardelli’s[20]. The abstract machine interface is written as a signature in ML:

```
signature machine =
  sig type Label
      val select : int -> unit
      val apply : unit -> unit
      val tailrecur : int -> unit
      val startrecord : int -> unit
      val endrecord : unit -> unit
       . . .
 end;
```

Each of these functions, when called, generates machine code for the corresponding operation.

There is a functor Vax that transforms a VaxCode structure into a Machine structure:

```
functor Vax(C : VaxCode) : Machine =
struct
  fun select j = C.movl(Displace(r0,4*j),Direct(r0))
      .
      .
      .
end

structure VaxM = Vax(OptVaxMcode)
structure VaxA = Vax(OptVaxAcode)
```

This Vax functor can be applied to the structure OptVaxMcode, which will result in an abstract machine that generates machine code for the Vax; or to the structure OptVaxAcode, which generates assembly language. We have a similar set of structures and functors that generate code for the Motorola MC68020 architecture.

Finally we can take this implementation of the abstract Machine, and apply the Codegen functor to it:

```
functor Codegen(M : Machine) =
struct
   fun codegen(APP(a,b)) = (codegen a; codegen b; M.apply())
     | codegen(FN(v,b)) = . . .
```

(The codegen function is not quite as simple as this, of course.)

```
end

structure VaxMCodegen = Codegen(VaxM)
structure M68MCodegen = Codegen(M68M)
```

Now the function VaxMCodegen.codegen, when given a lambda expression, generates optimized, backpatched machine code for the Vax.

This arrangement of functors is quite satisfactory. However, the details of the Machine signature might be changed to make it less like a stack machine. This might improve the generated code, especially for architectures that are not naturally stack-oriented. We are considering a re-implementation of the code generator using continuation-passing style[17] and Orbit[18].

5.1. Pattern-matching in code generation

Many modern code generators are written in the form of tree-pattern matchers[21, 22]. By writing the lambda-calculus tree data structure as an ML datatype (with constructors), the tree pattern matching can be directly specified as an ML function. There is one pattern for each constructor, e.g.

```
| APP(f,a) => (gen(f); gen(a); machine.apply())
```

but there are also cases that match certain ‘‘idioms;’’ like **let** expressions, which are represented as the application of a lambda-function:

`| APP(FN(w,b), a) =>` *generate code for* **let** *w*=*a* **in** *b*

There are about a dozen simple cases that handle small patterns with just one constructor, and a dozen more complicated cases that recognize idioms.

Some of these cases are applicable in only certain situations. For example, the pattern `APP(VAR w, a)` can be compiled without a closure only if *w* refers to a ‘‘known’’ function. One might like to write the clause for this case as:

```
| APP(VAR w, a) => if knownfunc(w) then applyknown(w,a) else ???
```

The problem is that if *w* is not a known function, then this pattern-match is not useful, and the pattern `APP(f,a)` should be matched. By the time the else clause is reached, it is impossible to transfer control to the `APP(f,a)` clause.

One solution to this problem is to attach boolean conditions to patterns, as is done in

Miranda[23]:

```
| APP(VAR w, a) when knownfunc(w) =>
```

Though would add some ‘‘syntactic sludge’’ to the language, it is worth considering as a future extension.

5.2. Flat versus linked environments

An *environment* is a mapping from some sort of *identifiers* to some sort of *bindings*, on which the fundamental operations are

$$\textit{update} \;:\; \text{Env} \times \text{Ide} \times \text{Bdg} \;\rightarrow\; \text{Env}$$

$$\textit{overlay} \;:\; \text{Env} \times \text{Env} \;\rightarrow\; \text{Env}$$

$$\textit{access} \;:\; \text{Env} \times \text{Ide} \;\rightarrow\; \text{Bdg}$$

The update function adds an identifier and binding to an environment; the access function looks up an identifier; and the overlay function adds all of the bindings of one environment to another. The overlay function might be used to implement the ML **open** primitive, for example.

Environments occur both in the compiler, where the identifiers are those of the source text, and the bindings are types and other information; and at runtime, in the form of function closures. A function closure, in implementations of lambda calculus, is a pair consisting of a function-code and the values to be associated with all of its free variables. Though these two kinds of environments are used in different ways, they share many of the same problems of implementation.

We know of no data structure that implements environments with both a very fast *access* function and a very fast *overlay* function. For compile-time environments, a fast *access* can be achieved by using a hash table; but then to overlay one hash table onto another takes time proportional to the size of the smaller table. A fast *overlay* can be achieved by representing environments as trees of *overlay* operations, but then *access* requires a tree search. In the compiled code, closures can be represented as flat or linked structures. A flat closure[2] is just a vector with one slot for each free variable, containing the value to which it is bound. A linked closure[17] has the binding of one free variable at the innermost level, along with a pointer to the closure for the enclosing scope. Flat closures have a fast *access*, but to build the vector takes time proportional to the number of free variables in the function. Linked closures can be built quickly: since the enclosing linked closure has already been built, just one *cons* operation is required. However, linked closures have a slower *access* function.

The question of whether to make *access* or *overlay* fast is a difficult one, since examples can be found to support either decision. Consequently, we have adopted a compromise: a mixed representation will be used, so that the time required for a typical sequence of *access* and *overlay* operations will probably be smaller than if either the flat (hash-table) or linked (tree) representation were used.

In the front end, environments are represented as linked lists of hash tables. The operation of **open**ing a structure just adds the (already-built) hash-table for that structure onto the list, but a typical *update* operation just adds a binding to an existing hash table. In the generated code, closures are represented as trees— a compromise between flat and linked closures. For each free variable in a closure, the back end heuristically decides whether to add another slot to the closure for this variable, or to access this variable through the (already existing) linked closure. The former is more expensive to build, but cheaper to use. When we have more experience with this compiler, we hope to measure the relative efficiencies of flat and linked closures in practice.

5.3. Data structures in the run-time system

The representations of ML structures in the compiled code are designed to be simple and general. We took care to avoid arbitrary restrictions on the size of the address space, or of individual objects. Except for **ref** cells, the graph of runtime objects is acyclic— even mutually recursive closures are represented without cycles.

Because of ML's polymorphic type system, all values must be the same size. We take this size to be one 32-bit word (this discussion of runtime data structures is specific to implementations on 32-bit computers). Any value that is naturally represented in more than one word will be manipulated as a (one-word) pointer to a (multi-word) object; this is known as a *boxed* value. An *unboxed* value is one which fits in one word and is not a pointer. The garbage collector must be able to tell which words are boxed (pointers) and which words are not. We use the low-order bit of the word as the indicator. Since all of our pointers (on byte-addressable machines) point to word boundaries, the low-order bit of any pointer is zero. Unboxed values are represented with their low-order bits turned on. Unboxed values include constructor tags, one-character strings, and integers (the integer k is represented as $2k+1$).

All records in the runtime system contain an integral number of 32-bit words, and are prefaced by a descriptor describing their size and type. There are two types of records: those that contain pointers (and unboxed values), and those that contain no pointers. All ML tuples and closures are of the former variety (Figure 1); strings and machine-code fragments are of the latter. The fact that machine-code fragments contain no pointers is a consequence of the fact that the back-end code generator is never applied to any lambda-expression with free variables.

There are several different representations used for constructors, just as in Cardelli's compiler[2].

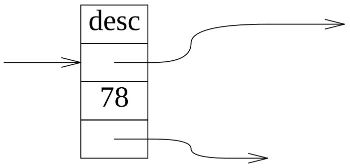

Figure 1. A triple of two pointers and an integer

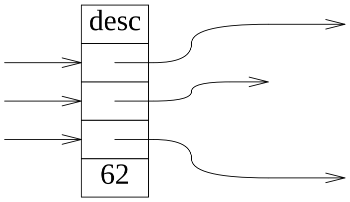

Figure 2. Three mutually recursive functions with one free variable

Constructors that carry values are usually represented as pairs of (tag, value). Constant constructors (link **nil**) are represented as small unboxed integers. Value-carrying constructors can be represented completely transparently if they carry an always-boxed value, and there are no other value-carrying constructors in the datatype (like **cons**), or if there is only one constructor in the datatype. Finally, **ref** cells look like one-word records.

Earlier ML compilers[20] represented mutually recursive functions as a set of closures that point to each other in a cyclic data structure. We wanted to avoid gratuitous circularity, because many garbage collection algorithms are more efficient on acyclic data structures[24]. We represent mutually recursive functions with just one closure record that has several function-part entries (Figure 2). To represent function *i*, one just uses a pointer to the i^{th} field. If the i^{th} function wants to refer to the closure for the j^{th} function, it just adds $j-i$ to the pointer value for its own closure. This creates no cycles of pointers; although there is an implicit cycle because each segment of machine code can refer to the machine's registers, which in turn point to the closure.

Several machine-code fragments may be stored in the same record. Because the record may be moved by the garbage collector, references within the block must be by relative addresses. Many computers have PC-relative addressing modes to facilitate this; for other machines, a pointer to the currently-executing block would have to be kept in a register. A typical record contains all of the function and string-literal fragments for an entire ML structure (module). In order not to confuse the garbage collector, just before the start of each fragment within the record is a (relative) pointer back to the beginning of the record, where the record's descriptor may be found; and all fragments must begin on even byte addresses, so that pointers to fragments within the record won't look like unboxed values.

5.4. Fast consing

The operation of allocating and initializing a record— known in Lisp as *cons*— is a fundamental operation in ML. The creation of a tuple value, the application of a constructor, and the creation of a function-closure all rely on this operation. Our implementation strives to make this operation as fast as possible.

We use a copying (and compacting) garbage collection algorithm. One consequence of this is that the free space available for storage allocation is a contiguous region of memory. We reserve a register to point at the lowest address in the free region. To allocate an *n*-word initialized record, the n^{th} word is stored first, followed by the other words and the descriptor. Then the free-space register is incremented by *n* words.

At some point the free space is exhausted. We arrange to have an inaccessible page of the address space immediately beyond the free space region. When there is an attempt to allocate a record that crosses into this page, a page fault occurs; and since the page is marked as inaccessible, the operating system transfers control to the ML runtime system instead of handling the page fault itself. Because the last word of a record is stored first, the creation of any record that crosses the boundary into the inaccessible region will fault at the beginning (where it is easy to patch things up and recover) rather than halfway through its initialization. (A special case is required for those records larger than a page; but these are rare and identifiable at compile-time.) Thus, the creation of an *n*-word record takes just *n* stores (plus one for the descriptor) and an add instruction; the *n* stores are required in any case to do the initialization. This is an extremely low-overhead *cons* operation.

A fast *cons* is not helpful if garbage collection is slow. When physical memory is large, then the cost of copying garbage collection, amortized over the *cons* operations, can come to less than one instruction per *cons* [25]. We are using a variant of generational garbage collection[24, 26, 27] that promises to be very fast even in moderate-size memories.

In writing our compiler, we have aimed for clarity and straightforwardness; we have eschewed the coding tricks that enable programmers to avoid using dynamically allocated memory. As a result, our compiler probably *cons*es a lot more than other compilers; it is fortunate that we have made *cons* so fast.

5.5. Exception Handling

Exception names in Standard ML behave a bit like constructors, except that they are generatively (dynamically) defined. The statements

```
exception E : int and R : real
```

and

```
 ... handle E with 0 => 10
                 | i => 2*i
         || R with 0.0 => 10
                 | r => 2*floor(r)
```

are, if one considers *E* and *R* to be constructors, a bit like the (hypothetical) statements

```
datatype exception =  . . . E of int | R of real . . .
```
and
```
 ... handle E(0) => 10
          | E(i) => 2*i
          | R(0.0) => 10
          | R(r) => 2*floor(r)
```

Generative data constructors cannot be given small integer tags like ordinary constructors. The tag for the constructor `E` must be constructed at run time. Its representation will be that of a **ref** cell: `ref("E")`. Any match applied to the exception type can compare the address of the ref cell against the tag of the exception object; and if the exception is propagated out of the user program into the bootstrap system, then the name of the exception can be determined (for debugging) by dereferencing the tag.

The ML statement `raise e with v` is compiled into the lambda language as `RAISE(CON(e,v))`; and the handle statement applies a match (with exceptions as constructors) to the exception raised. That is, `HANDLE(e,h)` executes expression `e`; if any exception is raised in `e`, then the handler `h` (which is a *match* as above) is applied to that exception. If `e` has type *t*, then `h` must have type *exception* → *t*.

The implementation of `HANDLE(e,h)` in the code generator is straightforward. An exception handler is pushed on the stack; expression `e` is executed; if no exception is raised, the exception handler is popped from the stack. An exception handler has two components: the address of the machine-code for the match `h`, and the address (on the stack) of the enclosing exception handler. Thus, entering the scope of a **handle** takes about two instructions, as does leaving its scope.

To raise an exception `E(v)`, first the constructor-expression `E(v)` is evaluated and put into a register; then the stack pointer is reset to the position of the current exception handler; the current handler is removed; and control passes to the text address found in the current handler. The match `h` found at this address is thereby applied to the exception `E(v)` previously evaluated. Raising an exception therefore takes just three or four instructions, not including the match that may have to be done upon arrival at the handler to determine which exception was raised.

The constructor view of exception names is not just a convenience in implementation. After some discussion in the Standard ML community, exceptions-as-constructors have been incorporated into the language. We no longer need an awkward notation (using `|` and `||`) for exception matches; and such things as re-raising an arbitrary expression are now possible:

```
handle e => (clean up; raise e)
```
(e is a variable of type ‘‘exception’’)

6. The runtime system

The ML standard library is implemented as a single module in the runtime system. The library functions are mostly written in ML, with some references to primitive functions written in assembly language. All references to library functions are treated as references to fields of this structure. Thus, a typical expression might have only one free variable— the library structure itself. In order to ensure that the code generator is given only closed expressions, this variable is lambda-bound; the machine code resulting from this closed expression can then be treated as a function whose argument will be the standard library. Access to user-defined structures is handled similarly.

This general plan is used both for the ‘‘bootstrap’’ system (in which the machine code for each structure is written to an external file), and the ‘‘interactive’’ system (in which the code for each structure or expression is put in memory and executed). In the interactive system, each expression when compiled is represented as a function whose argument is the current top-level environment, and whose result is the new binding. Thus, the interface between the compiler and the runtime system is very narrow and clean.

7. Conclusion

Standard ML is a complicated language. We manage the complexity by using a sequence of well-defined intermediate representations: tokens, abstract syntax, lambda-calculus, abstract machine, symbolic machine instructions. The datatypes and signatures of ML allow these interfaces to be cleanly specified, which is a great help.

Lexical analysis— the translation from source programs to tokens— we make as simple as possible (760 lines). All complications of recognizing constructors and infix operators are handled in the parser, which has better access to the environment mechanisms.

The translation from tokens to fully typed abstract syntax is the most complicated phase of our compiler (4500 lines). Because compile-time environments have some effect on parsing (constructors and infix operators), we merged the compilation of syntax, scopes, and types. This is where most of the complexity of the module features appear. But since structures and functors impinge on scopes and types in fundamental ways, we decided to build them into this phase from the beginning.

Since the abstract syntax is so well-digested and complete, the next phase— translation into lambda-calculus— is relatively simple (1480 lines). The major task of this phase is the compilation of patterns into decision trees.

A simple lambda-calculus representation of programs allows a simple and consistent code generator that can recognize idioms in lambda calculus and optimize them (750 lines). The lambda-calculus can also be useful for other kinds of analysis, like in-line function expansion. The back end needs to know very little about front-end data structures; thus, we found it simplest to have an environment manager for the back end that is completely separate from the front-end environment manager. Back-end environments map applied occurrences of variables in the lambda-calculus to their bindings, just as the front-end environment does that job for abstract syntax trees.

The last two phases— translation from abstract machine instructions into symbolic (Vax or MC68020) instructions, and the translation of those into backpatched byte sequences— are the only machine-dependent phases. These phases have a largely straightforward structure (950 lines per machine). In the current implementation, our code generators do not make adequate use of registers, and we may redesign the abstract machine interface.

Finally, the runtime system is kept very simple (450 lines of C, including the garbage collector; 300 lines of assembly language). As a statically-typed language, ML by its nature is oriented to compile-time analysis rather than complicated runtime systems; and we push this as far as possible. Our standard library is mostly implemented in ML (750 lines), and those parts implemented in assembly language are arranged to match a Standard ML signature. We pay particular attention to fast allocation of dynamic storage, since the features of ML (constructors, function closures) necessitate an efficient memory allocator and garbage collector.

Our compiler is concise, efficient, and (we think) well-designed. It should prove to be a useful tool for the functional programming community.

1. P. J. Landin, ‘‘The next 700 programming languages,’’ *Comm. ACM* **9**(3), pp. 157-166 (1966).
2. Luca Cardelli, ‘‘Compiling a functional language,’’ *1984 Symp. on LISP and Functional Programming*, pp. 208-217, ACM (1984).
3. Robin Milner, ‘‘A proposal for Standard ML,’’ *ACM Symposium on LISP and Functional Programming*, pp. 184-197, ACM (1984).
4. Robin Milner, ‘‘The Standard ML Core Language,’’ *Polymorphism* **2**(2) (October 1985).
5. David MacQueen, ‘‘Modules for Standard ML,’’ *Proc. 1984 ACM Conf. on LISP and Functional Programming*, Austin, Texas, pp. 198-207, ACM (1984).
6. David MacQueen, ‘‘Modules for Standard ML,’’ *Polymorphism* **2**(2) (October 1985).
7. R. Burstall, D. MacQueen, and D. Sannella, ‘‘Hope: an Experimental Applicative Language,’’ *Proceedings of the 1980 LISP Conference*, Stanford, pp. 136-143 (1980).
8. David C. J. Matthews, ‘‘The Poly manual,’’ *SIGPLAN Notices* (September 1985).
9. David C. J. Matthews, *An implementation of Standard ML in Poly*, May 1986.
10. G. Cousineau, P. L. Curien, and M. Mauny, ‘‘The Categorical Abstract Machine,’’ pp. 50-64 in

Functional Programming Languages and Computer Architecture, LNCS Vol 201, ed. J. P. Jouannaud, Springer-Verlag (1985).

11. Robin Milner, ‘‘A Theory of Type Polymorphism in Programming,’’ *J. CSS* **17**, pp. 348-375 (1978).
12. Luca Cardelli, ‘‘Basic polymorphic typechecking,’’ *Polymorphism* **2**(1) (January 1985).
13. R. S. Boyer and J Moore, ‘‘The sharing of structure in theorem-proving programs,’’ in *Machine Intelligence 7*, ed. D. Michie, Edinburgh University Press (1972).
14. Luis Damas, ‘‘Type Assignment in Programming Languages,’’ PhD Thesis, Department of Computer Science, University of Edinburgh (1985).
15. Robert Harper, Robin Milner, and Mads Tofte, ‘‘A type discipline for program modules,’’ ECS-LFCS-87-28, Univ. of Edinburgh (1987).
16. Marianne Baudinet and David MacQueen, *Tree Pattern Matching for ML*, 1986.
17. Guy L. Steele, ‘‘Rabbit: a compiler for Scheme,’’ AI-TR-474, MIT (1978).
18. D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams, ‘‘ORBIT: An optimizing compiler for Scheme,’’ *Proc. Sigplan ’86 Symp. on Compiler Construction* **21 (Sigplan Notices)**(7), pp. 219-233 (July 1986).
19. W. Wulf, R. K. Johnsson, C. B. Weinstock, C. B. Hobbs, and C. M. Geschke, *Design of an Optimizing Compiler*, Elsevier North-Holland, New York (1975).
20. Luca Cardelli, ‘‘The functional abstract machine,’’ *Polymorphism* **1**(1) (January 1983).
21. R. G. G. Cattell, ‘‘Formalization and automatic derivation of code generators,’’ Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, PA (April 1978).
22. A. V. Aho, M. Ganapathi, and S. W. K. Tjiang, *Code generation using tree matching and dynamic programming*, 1986.
23. D. A. Turner, ‘‘Miranda: a non-strict functional language with polymorphic types,’’ pp. 1-16 in *Functional Programming Languages and Computer Architecture, LNCS Vol 201*, ed. J. P. Jouannaud, Springer-Verlag (1985).
24. Henry Lieberman and Carl Hewitt, ‘‘A real-time garbage collector based on the lifetimes of objects,’’ *Communications of the ACM* **23**(6), pp. 419-429, ACM (1983).
25. A. W. Appel, ‘‘Garbage collection can be faster than stack allocation,’’ *Information Processing Letters* **(to appear)** (1987).
26. David A. Moon, ‘‘Garbage collection in a large LISP system,’’ *ACM Symposium on LISP and Functional Programming*, pp. 235-246, ACM (1984).
27. David Ungar, ‘‘Generation scavenging: a non-disruptive high performance storage reclamation algorithm,’’ *SIGPLAN Notices (Proc. ACM SIGSOFT/SIGPLAN Software Eng. Symp. on Practical Software Development Environments)* **19**(5), pp. 157-167, ACM (1984).