Compiling a Functional Language

Luca Cardelli

AT&T Bell Laboratories

Murray Hill, New Jersey 07974

1. Introduction

This paper summarizes my experience in implementing a compiler for a functional language. The language is ML[1] [Milner 84] and the compiler was first implemented in 1980 as a personal project when I was a postgraduate student at the University of Edinburgh[2]. In this paper, “the” ML compiler refers to my VAX implementation.

At the time, I was familiar with programming language semantics but knew very little about compiler technology; interpreters had been my main programming concern. Major influences in the design of this compiler have been [Steele 77] [Steele 78] and the implementation folklore for statically and dynamically scoped dialects of Lisp [Allen 78]. As a result, the internal structure of the compiler is fairly unorthodox, if compared for example with [Aho 78].

Anyway, a compiler for a language like ML *has* to be different. ML is interactive, statically scoped, strongly typed, polymorphic, and has first class higher-order functions, type inference and dynamic allocation. These features preclude many well-known implementation styles, particularly the ones used for Lisp (because of static scoping), the Algol family (because of functional values) and C (because of nested scoping and strong typing). The interaction of these features is what gives ML its “character”, and makes compilation challenging.

The compiler has been recently partially converted to the new ML standard. The major points of interest which are discussed in this paper are: (a) the interactive interpreter-like usage; (b) the polymorphic type inference algorithm; (c) the compilation of pattern matching; (d) the optimization of the representation of user defined data types; (e) the compilation of functional closures, function application and variable access; (f) the intermediate abstract machine and its formal operational semantics; (g) modules and type-safe separate compilation.

2. Potential efficiency

One should first realize that, at least potentially, ML can be compiled very efficiently. Because of static scoping, it is possible to compute at compile-time the location of variables. Because of strong typing, no type information is needed at run time[3]. Because of the regularity of parameter passing, one-argument function application is trivial, and many-argument function application can be optimized in many situations to use the stack, as in most languages, instead of using intermediate dynamic struc-

[1] A superficial knowledge of ML is assumed.
[2] A compiler for ML was already available as part of the LCF system [Gordon 79]
[3] Except some used by the garbage collector, which does not affect normal execution speed; this can be encoded in the address of a datum so that no extra space is used.

tures, as it would appear necessary. Because of pattern matching, the destructuring and discrimination of arguments can be compiled more efficiently than compiling explicit destructuring and discrimination code. Because of data type definitions, the compiler can optimize data representations somewhat; for example, on totally general principles the abstract ML definition of “list” type (which seems to require extra pointers) produces the normal representation of lists as pairs of memory words with “0” for nil.

On the other hand, some inefficiencies are essentially embedded in the structure of the language. The most noticeable one is probably polymorphism. Polymorphism is, in a sense, space-efficient because it facilitates writing smaller programs whose code can be reused in many different situations, where non-polymorphic typed languages require writing and/or compiling several copies of essentially identical procedures. However, for polymorphism to work, every piece of data must have the same identical format: a pointer. This implies space and run-time inefficiencies. Packed records, arrays of records, character arrays and bitmaps cannot be handled in their space-efficient form, unless (as in the case of strings in ML) they are primitive or predefined abstract types.

However, the abstract type mechanism in ML provides a uniform way of extending the compiler to fit special applications. For example, bitmap graphics could be embedded by introducing a primitive abstract type *bitmap*, with primitive operations like *bitblt*. This new type would fit perfectly well in the language, which already supports user-defined abstract types. An impractical but semantically correct definition of the bitmap type and operations could be given in ML as a reference.

3. Compiler architecture

The compiler is organized in several phases, more for programming convenience than from necessity. The first phase is scanning and parsing, which produces an abstract syntax tree. The second phase is analysis, which includes typechecking and the computation of stack displacements and free variables. The third phase is compilation into an intermediate stack machine language, and peephole optimizations (e.g. tail recursion removal). The fourth is VAX-code generation. Finally the compiled program is executed in an interactive run-time system which performs garbage collection.

This process is repeated for every definition or expression typed by the user (e.g. **3;**) or fetched from an external file. Because of the interactive use of the compiler, the compilation of small phrases must be virtually instantaneous. This does not mean that the compiler has to be particularly fast in itself, but it must have very little startup overhead. These requirements pretty much preclude the use of external intermediate files and of system tools like assemblers. All the compilation is done in main memory, and the code generator directly produces VAX binary code. Large programs may require unrealistically large amounts of memory, but separate compilation allows one to break programs into manageable pieces.

The compiler is incremental, in the sense the compilation of a top level phrase takes place in the environment created by all the previous compilations. ML is a statically scoped language *even at the top level*. A top level expression **E** is compiled like **let D in E end** where **D** is the current top level environment. Top level and local environments all live on the same stack. After every successful top level declaration, the top level environment grows.

The parsing phase does nothing but parsing. The parser is recursive descent with an association stack for infix operators. Recursive descent allows one to beautifully structure the parser and break a few rules where one has to. For example the parsing of ML declarations is organized as a two-level grammar.

The polymorphic type system is the major feature of ML. The typechecker is also one of the major components of the compiler. The typechecking algorithm used here is due to Robin Milner [Milner 78], for the basic algorithm, and Luis Damas (unpublished) for the extension to updatable data. Typechecking, performed during the analysis phase, is based on the unification algorithm. Typechecking requires a compiler pass of its own, due to its sophistication, but other tasks are performed during the typechecking pass to gather information useful to the succeeding translation pass. One of these tasks is the computation of the free variables of every function and the consequent computation of stack and closure displacements for program variables.

Conventional compilers are organized around a symbol table which contains all sorts of information. The ML compiler has a symbol table, but this is used solely to speed-up string equality. The compiler is instead organized around a linked-list environment, very much like the ones used in interpreters. This was done to keep the structure of the compiler in close correspondence with the denotational semantics of the language, which was well understood. ML has a very sophisticated treatment of environments, and encoding that into a flat symbol table seemed dangerous.

Scoping of primitive identifiers in ML is uniform with respect to user-defined identifiers. Some care is needed to deal with this property of the language. For example, **+** in **1+2;** is compiled as a primitive *operator*, but **+** in **f(op +);** is a primitive *function*. Moreover **fun(op +). 1 + 2;** is not necessarily addition. The compiler is smart enough to recognize **(op +)(1,2);** as an occurrence of the operator **+** and **(op +)(x);** as an occurrence of the function **+**, and to generate the appropriate code.

The translation phase produces an intermediate stack machine code, described in detail in one of the following sections. The initial reasons for using an intermediate language were portability and to facilitate code optimization. The intermediate language later turned out to be a tremendous aid in debugging the compiler: this fact alone amply justifies its use.

A primary goal was to compile very fast function applications; this task was facilitated by the extreme regularity of function application in ML. I now believe I gave a bit too much importance to this problem ignoring other efficiency considerations, but this single goal shaped the whole compiler. For example: (a) the abstract machine uses two stacks to avoid stack shuffling on function call; (b) tail recursion is removed; (c) in the abstract machine, function application is split into three operations so that when one application immediately follows another application, intermediate context saving and restoring can be optimized away; (d) the escape-trap mechanism uses a third stack so that it never interferes with function call.

The peephole optimization phase operates on the intermediate code and does some jump optimization and tail recursion removal. I now believe that it is a mistake to remove tail recursion at this late stage: many complex auxiliary peephole optimizations are needed to really eliminate tail recursion in combination with conditionals and local declarations. Rather, the analysis phase should recognize tail recursive situations so that the translation phase can immediately generate the proper code.

The code generation phase is very naive: the abstract machine instructions are translated one by one to operate on stacks and *no* optimization is made, except some detection of special cases. Although techniques like register allocation are believed to be a mistake in functional languages because of frequent context switching, the ML execution time could greatly benefit from a VAX-code peephole optimizer. This was never introduced because the execution time was found to be already satisfactory for typical applications (but typical applications are now becoming more demanding).

Finally, the run-time system provides dynamic memory allocation, garbage collection and system routines (e.g. input-output). Garbage collection is done by a recursive copying-compacting two-space algorithm with data caged in different pages depending on data format. The two spaces grow incrementally, and a simple adaptive algorithm changes the frequency of collections based on the amount of garbage reclaimed in previous collections.

A major drawback of the compiler is the absence of debugging tools. This problem is not as serious as it may seem because typechecking, combined with the clean semantic structure of ML, eliminates a large proportion of bugs at compile-time, while allowing much of the flexibility of untyped languages (when I now occasionally program in untyped languages, I find myself thinking, over and over, “the ML typechecker would have trapped this bug...”). Moreover, the simple fact that ML is interactive already provides a debugging environment far superior to any debugging system for batch-compiled languages, as any Lisp programmer well knows. For these reasons I have never felt an unbearable pressure to introduce debugging tools. However, debugging is unquestionably desirable, especially for new users, and its introduction may require some rethinking of the compiler architecture.

4. Fetching variables

Having claimed that ML can be compiled efficiently, how do we do it? The primary decision to be made concerns the way functional objects are represented in terms of data structures. This in turn affects the way variables are fetched and applications are performed. Variable fetching and function application are among the most important factors in the performance of a functional language.

The first difficulty derives from the fact that functions can be returned as values of other functions. Lisp just ignores this problem, to reintroduce it later in the form of funargs, and fails to implement higher-order functions correctly. Algol-like languages forbid this situation (with the partial exception of Simula67 and Algol68).

To correctly solve this problem, the value of a function text must be a closure of the function text in an environment which defines the value of the variables which are free in the text. Two solutions are known from the implementation of statically scoped Lisp dialects. The simplest one is to implement run-time stacks as linked lists, so that a closure can simply contain a pointer to the relevant environment (if stacks were real stacks, some information might be overwritten by successive uses of the stack). This solution is usually too inefficient, although Simula uses it through clever optimizations.

The second solution is to let closures contain copies of (pointers to) the values of the relevant global variables, and to use a real stack for argument passing. This has the advantage that local variables can be directly accessed at some known depth on the stack, and global variables can be directly accessed at some known depth in the closure (no *static link* or *display* techniques are necessary). The disadvantage is that closures have to be built, and this can be moderately expensive in some situations. Also, closures cannot contain assignable variables, but only pointers to assignable objects, in order to preserve the sharing of side-effects.

The ML compiler uses the second solution: closures are bundles of global variables associated to a function text. The problem with sharing side effects is solved very simply by the fact that ML does not have assignable program variables: all the assignable structures live in the heap, and are shared by the pointers contained in the closures.

As I said, no display techniques are necessary. On the other hand the compiler must compute the set of global variables of every function. The naive way of doing this, corresponding to the logical definition of free variables, requires expensive set manipulations. The current ML compiler uses a tricky and supposedly clever stack discipline. However I am going to describe here a much simpler and reasonably efficient solution, which I have used in other prototype compilers.

Global variables are accumulated in a *var-set* data structure, which can be realized as a hash table or an association list. Three primitive operations are available on var-sets: (1) obtaining a new empty var-set, (2) inserting a variable in a var-set, and (3) applying a function to all the elements of a var-set. The insertion is done as follows. If the variable is not already in the set, it is inserted and associated with the old cardinality of the set; this number is then returned and is used as the closure displacement of that variable. If the variable is already in the set, the corresponding displacement is returned.

The analysis of a lambda expression E starts with a list of local variables L, initialized to the list of formal parameters of the lambda expression, and a set of global variables G, initialized to a new var-set. Local declarations make L grow. Whenever a variable is found in the body of the lambda the following *lookup* procedure is applied to it. The variable is first searched for in L; if it is found there, we have its stack dispacement. If it is not in L, it must be a global variable and it is inserted in G, obtaining a closure displacement for that global variable.

Suppose now that the lambda expression E contains another lambda expression E'. The procedure is applied recursively to E', and at the end of the process we obtain the var-set V' for E'. The variables in V' (which are global to E', and can be local or global to E) have to be fetched to create the closure for E': to do this we simply apply *lookup* to all the variables in V'.

An arbitrary top-level expression is processed with L being the top-level environment, and G being a new var-set. At the end of the analysis G should be empty; if not, G is the set of undefined variables.

5. Pattern Matching

ML functions can be defined by specifying a set of patterns to be matched in order against the arguments, and a corresponding set of actions to execute (this style of function definition has been inherited from Hope [Burstall 80]). Pattern matching communicates highly structured information about the input parameters. As a consequence, a compiler can produce better code from a pattern matching description than from a corresponding sequence of discrimination and selection operations.

ML pattern matching is less powerful than, for example, Prolog pattern matching, because every variable can occur only once in an ML pattern. Hence, ML pattern matching is just an abbreviation for nested if-then-else, case statements and selection functions.

The naive way of compiling pattern matching is: try to match the input against the first pattern, if it matches then execute the corresponding action, else backtrack and try the next pattern. A lot of useless work can be generated by this strategy. The ML compiler does a much better job by compiling the patterns into a discrimination tree which does an almost optimal number of tests on the datum in order to select the appropriate action (this technique is due to Dave MacQueen and Gilles Kahn, and was first implemented in the Hope compiler).

The pattern matching compilation algorithm is actually fairly complex, and it will be illustrated by a simple example. Consider the following Shuffle function, which converts a pair of lists of the same length into a list of pairs. The first definition is the conventional one, while the second definition uses pattern matching (where '**::**' is infix cons):

```
val rec Shuffle (List, List´) =
    if (null List) and (null List´) then nil
    else    if (null List) or (null List´) then escape "Shuffle"
            else (hd List, hd List´) :: (Shuffle (tl List, tl List´));

val rec
    Shuffle (nil, nil) = nil |
    Shuffle (Hd::Tl, Hd´::Tl´) = (Hd,Hd´)::(Shuffle(Tl,Tl´)) |
    Shuffle (_) = escape "Shuffle";
```

Pattern matching based programs are organized into *rows* of pattern-action pairs. To produce a discrimination tree for a given set of rows, we must analyze the patterns according to their kind: a pattern can be a variable (like **Hd** or **Tl**), a default ('**_**' which behaves like an anonymous variable) a data constant (like **nil**), a data constructor applied to more patterns (like **::**), or a tuple of patterns (like **nil,nil**). Patterns can be nested to any depth.

To generate the discrimination tree for tuples, we first split the patterns vertically into *columns*. In the example we obtain (here '**::**' is used in prefix form) Col-1=[**nil**; **::(Hd,Tl)**; **_**] and Col-2=[**nil**; **::(Hd´,Tl´)**; **_**]. Notice that we had to split '**_**' into '**_ , _**' before producing the columns. Variables may also need to be split into tuples of '**_**'.

We now make a heuristic choice to determine which column is best for discrimination. A reasonable heuristic is to choose the column with the largest number of distinct constants or constructors, because this is likely (but not guaranteed) to produce a shallower discrimination tree.

In this simple case we can choose any column, say Col-1. Col-1 exhausts all the possible list constructors, (**nil** and **::**); if this were not the case we would automatically introduce a new row '**_ = escape"match"**' to fail when none of the original patterns matches the input.

Hence, we can generate a test for null list or non-null list on the first column, and ignore the '_' case which is redundant. In either case we have to analyze the second column, based on the choice we have made in the first column. It the first choice is **nil**, Col-2 reduces to Col-2-1=[**nil**; _] (i.e. we eliminate all the rows whose first column does not match **nil**); if the first choice is **::**, Col-2 reduces to Col-2-2=[**::(Hd´,Tl´)**; _]. In Col-2-1 we simply generate a test for nil, to distinguish the two cases. In Col-2-2 we generate a test for non-nil.

At this point we have exhausted all the possible inputs. In more complex situations we may have to carry on the analysis on **(Hd´,Tl´)**, split those into columns, etc.

The code produced by this process closely resembles the code generated for the following definition of Shuffle:

```
val rec Shuffle (List, List´) =
    if null List
    then if null List´ then nil else escape"Shuffle"
    else    if null List´ then escape"Shuffle"
            else    let   val Hd = hd List and Tl = tl List;
                          val Hd´ = hd List´ and Tl´ = tl List´
                    in (Hd, Hd´) :: (Shuffle (Tl, Tl´)) end;
```

The destructuring of **List** into **Hd** and **Tl** is actually done more efficiently than explicit calls to **hd** and **tl** (which have to check for null list), because at that point we know that **List** is not null. This is one of the reasons why pattern matching can be more efficient than hand-written if-then-else code.[4]

6. Representation of user-defined types

The compiler chooses efficient representations of user-defined data types. For example, the ML definition of a polymorphic binary tree is:

```
type ´a tree = leaf | node of ((´a tree) * ´a * (´a tree));
```

The naive representation for this type is the following: every tree is a tagged pointer, where the tag is used to discriminate between the leaf and the node case. The tagged pointer is '0' in the leaf case, and points to a triple in the node case. The triple then contains the left and right subtree, and the node information. This kind of representation is guaranteed to work for any user-defined data type, but it is far from optimal in several frequently occurring cases. In the above example, we want a leaf to be represented as the number '0' and a node as a triple, with no extra tagged pointers.

ML objects are represented in memory as *boxed* or *unboxed* structures. An unboxed structure (e.g. booleans, enumerations and small integers) has a numeric value less then 64K. A boxed structure (e.g. tuples and closures) is a pointer to a memory area where the contents of the structure are allocated; a pointer is always bigger then 64K. Hence it is always possible to distinguish between boxed and unboxed structures (this fact is also used by the garbage collector to follow pointers).

For every data type, the compiler knows or computes an *always-boxed* property, which is true if all the objects of that type are boxed. For example, booleans are not always-boxed (they are never boxed), lists are not always-boxed (they are boxed only if they are non-null, as nil is represented as an unboxed '0') and tuples are always-boxed. Given a type like **´a tree**, and given the always-boxed properties of

[4] Currently, the if-then-else's are actually compiled as case instructions (for uniformity with the general case of user-defined data types), but special cases where branches suffice could be easily detected.

the types on which **´a tree** is built (i.e. **leaf** is built on an unboxed type, and **node** is built on always-boxed triples), the following optimizations are possible.

First of all, constants like **leaf** are represented as unboxed numbers, (different numbers if there are many constants) instead of tagged null pointers. Second, suppose there is now a single constructor, which furthermore contains always-boxed data: then we can eliminate the tagged pointer, leaving the naked always-boxed datum which can be distinguished from all the other constants (if any). Finally, the always-boxed property is computed for this data type, based on the representations which have just been selected. Some recursive and polymorphic type definitions can introduce indeterminacy in the computation of the always-boxed property; in that case it is enough to behave conservatively.

According to these optimizations the type **´a tree** is represented in the efficient form mentioned above; **type bool = false | true** is represented as '0' (false) and '1' (true); **type age = age of int** is represented exactly like **int**; **type ´a list = nil | cons of (´a * (´a list))** is represented as '0' (nil) and naked pairs (cons cells); etc.

These representation optimizations are not too complicated, but pattern matching has to know about them, and must use different discrimination tests according to the different representations. All this is possible because ML encapsulates the definition of new data types and their constructors and selectors into a single construct, so that they can be simultaneously optimized.

7. Typechecking

A type can be either a type variable **´a**, **´b**, etc., standing for an arbitrary type, or a type operator. Operators like **int** (integer type) and **bool** (boolean type) are nullary type operators. Parametric type operators like → (function type) or * (cartesian product type) take one or more types as arguments. The most general forms of the above operators are **´a → ´b** (the type of any function) and **´a * ´b**, (the type of any pair of values); **´a** and **´b** can be replaced by arbitrary types to give more specialized function and pair types. Types containing type variables are called *polymorphic*, while types not containing type variables are *monomorphic*. All the types found in conventional programming languages, like Pascal, Algol68 etc. are monomorphic.

Expressions containing several occurrences of the same type variable, like in **´a → ´a**, express contextual dependences, in this case between the domain and the codomain of a function type. The typechecking process consists in matching type operators and instantiating type variables. Whenever an occurrence of a type variable is instantiated, all the other occurrences of the same variable must be instantiated to the same value: legal instantiations of **´a → ´a** are **int → int**, **bool → bool**, **(´b * ´c) → (´b * ´c)**, etc. This contextual instantiation process is performed by *unification*, [Robinson 1]. Unification fails when trying to match two different type operators (like **int** and **bool**) or when trying to instantiate a variable to a term containing that variable (like **´a** and **´a → ´b**, where a circular structure would be built). The latter situation arises in typechecking self-application (e.g. **fun x. (x x)**), which is therefore considered illegal.

Here is a trivial example of typechecking. The identity function **I = fun x. x** has type **´a → ´a** because it maps any type onto itself. In the expression **(I 0)** the type of **0** (i.e. **int**) is matched to the domain of the type of **I**, yielding **int → int** as the specialized type of **I** in that context. Hence the type of **(I 0)** is the codomain of the type of **I**, which is **int** in this context.

The basic algorithm can be described as follows.

1. When a new variable **x** is introduced by a lambda binder, it is assigned a new type variable **´a** meaning that its type must be further determined by the context of its occurrences. The pair **<x,´a>** is stored in an environment which is searched every time an occurrence of **x** is found, yielding **´a** (or any intervening instantiation of it) as the type of that occurrence.

2. In a conditional, the **if** component is matched to **bool**, and the **then** and **else** branches are unified in order to determine a unique type for the whole expression.

3. In an abstraction **fun x. e** the type of **e** is inferred in a context where **x** is associated to a new type variable. A functional type is then returned, whose domain is **´a** (or any intervening instantiation), and whose codomain is the type of **e**.

4. In an application **f a**, the type of **f** is unified against a type **A** → **´b**, where **A** is the type of **a** and **´b** is a new type variable. This implies that the type of **f** must be a function type whose domain is unifiable to **A**; **´b** (or any instantiation of it) is returned as the type of the whole application.

In order to describe the typechecking of let expressions, and of variables introduced by let binders, we need to introduce the notion of *generic* type variables. Consider the following expression (where **pair** creates a pair of two objects)

F = fun f. pair (f 3) (f true) **[Ex1]**

In Milner's type system this expression cannot be typed, and the algorithm described above will produce a type error. In fact the first occurrence of **f** determines a type **int** → **´b** for **f**, and the second occurrence determines a type **bool** → **´b** for **f**, which cannot be unified with the first one.

Type variables appearing in the type of a lambda-bound identifier like **f**

are called *non-generic* because, as in this example, they are shared among all the occurrences of **f** and their instantiations may conflict.

Note that for some functional arguments, **F** is a perfectly well behaved function, e.g. **(F (fun a. 0))** which produces **(pair 0 0)**, or **(F (fun a. a))** which produces **(pair 3 true)**. One could try to find a typing for **F**, for example by somehow assigning it **(´a** → **´b)** → **(´b * ´b)** This would typecheck correctly in the above situations, because both **(´c** → **int)** (the type of **(fun a. 0)**) and **(´d** → **´d)** (the type of **(fun a. a)**) match **(´a** → **´b)** (the domain of **F**). However this typing is unsound in general: for example **succ: int** → **int** has a type that matches **´a** → **´b**, and it would be accepted as an argument to **F** and wrongly applied to **true**. There are sound extensions of Milner's type system which can type **F**, but they are beyond the scope of this discussion.

Hence there is a basic problem in typing heterogeneous occurrences of lambda-bound identifiers. This turns out to be tolerable in practice, because expressions like **F** are not extremely useful or necessary, and because a different mechanism is provided. We are going to try and do better in typing heterogeneous occurrences of let-bound identifiers. Consider:

let val f = fun a. a **[Ex2]**
in pair (f 3) (f true) end

It is essential to be able to type the previous expression, otherwise no polymorphic function could be applied to distinct types in the same context, making polymorphism quite useless. Here we are in a better position than **Ex1**, because we know exactly what **f** is, and we can use this information to deal separately with its occurrences.

In this case **f** has type **´a** → **´a**; type variables which, like **´a**, occur in the type of let-bound identifiers (and that moreover do not occur in the type of *enclosing* lambda-bound identifiers) are called *generic*, and they have the property of being able to assume different values for different instantiations of the let-bound identifier. This is achieved operationally by making a copy of the type of **f** for every distinct occurrence of **f**.

In making a copy of a type, however, we must be careful not to make a copy of non-generic variables, which must be shared. The following expression for example is as illegal as **Ex1**, as **g** has a non-generic type which propagates to **f**:

fun g. let val f = g **[Ex3]**
in pair (f 3) (f true) end

Again, it would be unsound to accept this expressions with a type like **(´a → ´b) → (´b * ´b)** (consider applying **succ** so that it is bound to **g**).

The definition of generic variables is as follows:

A type variable occurring in the type of an expression ***e*** *is generic iff it does not occur in the type of the binder of any lambda-expression enclosing* ***e****.*

Note that a type variable which is non-generic while typechecking within a lambda expressions, may become generic outside the lambda expression. This is the case in **Ex2** where **a** is assigned a non-generic **´a**, and f is assigned **´a → ´a** where **´a** is now generic.

To determine when a variable is generic we maintain a list of the non-generic variables at any point in the program: when a type variable is not in the list it is generic. The list is augmented when entering a lambda; when leaving the lambda the old list automatically becomes the current one, so that that type variable becomes generic. In copying a type, we must only copy the generic variables, while the nongeneric variables must be shared. In unifying a non-generic variable to a term, all the type variables contained in that term become non-generic.

Finally we have to consider recursive declarations:

let rec val f = ... f ...
in ... f ... end

which are treated as if the **rec** were expanded using a fixpoint operator **Y** (of type **(´a → ´a) → ´a**):

let val f = Y fun f. ... f ...
in ... f ... end

it is now evident that the instances of (the type variables in the type of) **f** in the recursive definition must be non-generic, while the instances following **in** are generic.

5. Hence, to typecheck a **let** we typecheck its declaration part, obtaining an environment of identifiers and types which is used in the typechecking of the body of the **let**.

6. A declaration is treated by checking all its definitions $\mathbf{x_i = t_i}$, each of which introduces a pair $\mathbf{\langle x_i,T_i \rangle}$ in the environment, where $\mathbf{T_i}$ is the type of $\mathbf{t_i}$. In case of (mutually) recursive declarations $\mathbf{x_i = t_i}$ we first create an environment containing pairs $\mathbf{\langle x_i,\acute{}a_i \rangle}$ for all the $\mathbf{x_i}$ being defined, and where the $\mathbf{\acute{}a_i}$ are new non-generic type variables (they are inserted in the list of non-generic variables for the scope of the declaration). Then all the $\mathbf{t_i}$'s are typechecked in that environment, and their types $\mathbf{T_i}$are matched again against the $\mathbf{\acute{}a_i}$ (or their instantiations).

Side-effects do not interact nicely with polymorphism, and require a special treatment (see [Gordon 79], page 52, for a discussion of the problems). The simplest solution consists in requiring all the assignments to act on monomorphic values; this is the position taken in Standard ML. The ML compiler uses a more flexible discipline, devised by Luis Damas, which however will not be discussed here.

8. The Functional Abstract Machine

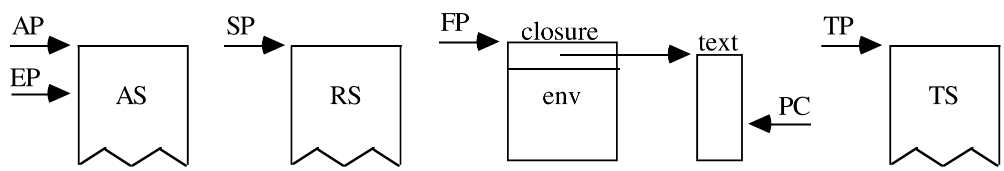
The Functional Abstract Machine (Fam) is a stack machine designed to support functional languages on large address space computers. It can be considered an SECD machine [Landin 64] which has been optimized to allow very fast function application and the use of true stacks (as opposed to linked lists). This section contains a brief overview of the Fam; which is fully described in [Cardelli 83].

The machine supports functional objects (closures, which are dynamically allocated and garbage collected). All the optimization and support techniques which make application slower are strictly avoided, while tail recursion and pattern-matching calls are supported. Restricted side effects and arrays are provided.

The machine is intended to make compilation from ML and other functional languages easy and regular, by providing a rich set of operations and an open-ended collection of data types. The instructions of the machine are not supposed to be interpreted, but assembled into machine code and then executed. Hence, no optimized special-case operations are provided because special cases can be easily detected at assembly time.

For efficiency considerations, the abstract machine is not supposed to perform run-time type checking and hence is not type-safe. Moreover, as a matter of principle, there is no primitive to test the type of an object; the correct application of machine operations should be guaranteed by typechecking in the source language. Where needed, the effect of run-time typechecking can be achieved by the use of *variant* (i.e. tagged) data types.

The state of the abstract machine is determined by six pointers, together with their denotations, and a set of memory locations. The pointers are the Argument Pointer, the Frame Pointer, the Stack Pointer, the Trap Pointer, the Program Counter and the Environment Pointer. They point to three independent stacks and, directly or indirectly, to the data heap. The memory takes care of side-effects in the heap and includes the file system.



The *Argument Pointer* (AP) points to the top of the *Argument Stack* (AS), where arguments are loaded to be passed to functions and results of functions are delivered. This stack is also used to store local and temporary values. In all machine operations which take displacements on AS, the first object on AS is at displacement zero.

The *Frame Pointer* (FP) points to the current closure (or frame) (FR) consisting of the text of the currently executed program and an environment for the free variables of the program.

The *Program Counter* (PC) points to the program to be executed (PR) (which is part of the current closure).

The *Stack Pointer* (SP) points to the top of the *Return Stack* (RS), where program counter and frame pointer are saved during function calls.

The *Trap Pointer* (TP) points to the *Trap Stack* (TS), where *trap frames* are stored. A trap frame is a record of the state of the machine, which can be used to resume a previous machine state (side effects in the heap are not reverted).

The *Environment Pointer* (EP) also points to AS, and defines the top level environment of execution for use in interactive systems. At the beginning of execution, EP is the same as AP, but it normally grows because of top level definitions.

The semantics of the abstract machine is given operationally by state transitions [Plotkin 81, Henderson 80]. A machine state is represented by a tuple:

(AS, RS, FR, PR, TS, ES, M)

For any stack S (i.e. AS, RS, TS or ES) we write S.x:t for the operation of pushing a cell x of type t on the top of S (t may contain type variables α, β, etc.). The empty stack is <> and S.x:t is a stack iff

S is a stack. Moreover S[n]x:t is a stack which is the same as S, except that the n-th cell from the top contains x of type t; the top cell of S has displacement 0. In case of conflicting substitutions, like S[n]x:t[n]x':t', the rightmost substitution is the valid one.

The frame FR (pointed to by FP) has the form:

$\mathrm{clos}(\mathrm{text}(c,l_0,..,l_n),x_0,..,x_m)\colon \alpha \rightarrow \beta$

where c is a sequence of machine operations, l_i are *literals* of c and x_j are values for the free variables of (the source program whose translation is) c. The literals of c are "big constants" like strings and inner lambda expressions, which occur in (the program whose translation is) c; they are taken out of the code so that the garbage collector can access them easily. The code c together with its literals is a *text* (and a literal can be a text). A text together with its free variables is a *clos* (closure). Every closure implements a function having some type $\alpha \rightarrow \beta$.

The program PR (pointed to by PC) is a string of abstract machine operations. The empty program is <> and the initial instruction of PR is singled out by writing $\mathrm{op}(x_1{:}\alpha_1,..,x_n{:}\alpha_n)$.PR', where x_i are the parameters of op.

The *memory* M is a pair of functions:

$M = L,F\colon (\mathrm{address} \rightarrow \mathrm{value}) \times (\mathrm{streamname} \rightarrow \mathrm{stream})$

where L are the *locations* and F is the *file system* (which is not discussed here).

Every machine operation op(..) implements a state transition, denoted by:

(AS, RS, FR, $\mathrm{op}(x_1{:}\alpha_1,..,x_n{:}\alpha_n)$.PR, TS, ES, M)
=> (AS', RS', FR', PR', TS', ES', M')

In order the make the operation more visible, we normally use the following equivalent notation:

$\mathrm{op}(x_1{:}\alpha_1,..,x_n{:}\alpha_n)$
(AS, RS, FR, *.PR, TS, ES, M)
=> (AS', RS', FR', PR', TS', ES', M')

Data operations transfer data back and forth between the Argument Stack and data cells. In general they take n arguments ($n \geq 0$) from the top of AS popping the stack n times, and push back m results ($m \geq 1$).

The set of abstract machine data types is open-ended, and so is the set of data operations. Some data types are used as arguments to basic machine operations, and hence must be present in every implementation. They include unit, bool, (short) int, string, list and closure.

Data operations may *fail*, and these failures can be trapped. Abstract machine failures and user defined failures behave uniformly.

Stack operations manipulate the argument stack. Here are some examples. *GetLocal(n)* copies the n-th cell from the top of AS onto AS. *Inflate(n,m)* inserts n null cells above the m-th cell from the top of AS. *Deflate(n,m)* deletes n cells starting from the m-th cell from the top of AS; cells above and below the deleted area are recompacted. An Inflate operation with m=0 pushes n cells on top of AS; similarly Deflate with m=0 pops n cells from the top of AS.

Closure operations manipulate closures. A closure is a data object representing a function; it contains the text of the function and the value of its free variables. The text of a function is in itself a rather complex structure; it contains a sequence of instructions in some suitable machine language, and a set of *literals* which may be strings or other text cells.

Closures are created by placing the values for free variables and the text of the function on AS, and then storing this information in a newly allocated closure cell. Closures for (mutually) recursive functions may contain loops, and are allocated in two steps: dummy closures for a set of mutually recursive functions are first allocated in the heap, and later on recursive closures are built by filling the dummy closures. This way the closures may mutually contain pointers to the other (dummy) closures.

The operations on closures are *Closure* (which creates a closure with arguments on AS), *DumClosure* (which allocates an empty closure), *RecClosure* (which fills in dummy closures), *GetGlobal* (which retrieves the value of a free (global) variable) and *GetLiteral* (which retrieves a literal from the text of the closure).

Closure (n: int≥0)
(AS.x_1:α_1..x_n:α_n.t:text, RS, FR, *.PR, TS, ES, M)
=> (AS.clos(t,x_1,..,x_n), RS, FR, PR, TS, ES, M)

DumClosure (n: int≥0)
(AS, RS, FR, *.PR, TS, ES, M)
=> (AS.clos(unity,unity_1,..,unity_n), RS, FR, PR, TS, ES, M)

RecClosure (n: int≥0, m: int>n)
(AS.x_n:α_n..x_1:α_1.t:text[m]clos(unity,unity_1,..,unity_n),
RS, FR, *.PR, TS, ES, M)
=> (AS[m–n–1]clos(t,x_1,..,x_n), RS, FR, PR, TS, ES, M)

GetGlobal (n: int≥0≤p)
(AS, RS, clos(t:text,x_0:α_0,..,x_p:α_p), *.PR, TS, ES, M)
=> (AS.x_n, RS, clos(t,x_0,..,x_p), PR, TS, ES, M)

GetLiteral (n: int≥0≤p)
(AS, RS, clos(text(c:code,x_0:α_0,..,x_p:α_p),..), *.PR, TS, ES, M)
=> (AS.x_n, RS, clos(text(c,x_0,..,x_p),..), PR, TS, ES, M)

Control operations affect the Program Counter or the Stack Pointer. *Jump* is an unconditional branch to another point in the same program text. *FalseJump* is a conditional branch which jumps when the top of AS is false; otherwise the normal execution flow continues. Function application is split into three operations: *SaveFrame* (which saves the calling closure on RS), *ApplFrame* (which saves the calling program counter on RS, and activates the called closure sitting on the top of AS by making it the one pointed by FP and by setting PC at its entry point) and *RestFrame* (which restores the calling closure from RS). This means that SaveFrame and RestFrame are inverses and can be canceled out in multiple (curried) applications. The called closure uses Return to restore the calling program counter and return to the calling function (where a RestFrame is normally executed). The sequence SaveFrame, ApplFrame, RestFrame, Return can be optimized to *TailApply*, which uses a jump to pass control to the called function. The advantage of TailApply is that the control stack does not grow, hence iteration can be programmed by (tail) recursion. Return and TailApply also incorporate a Deflate operation, and hence take two arguments for deflating n cells below the m cell from the top of AS.

Jump (c: code)
(AS, RS, FR, *.PR, TS, ES, M)
=> (AS, RS, FR, c, TS, ES, M)

FalseJump (c: code)
(AS.false, RS, FR, *.PR, TS, ES, M)
=> (AS, RS, FR, c, TS, ES, M)
(AS.true, RS, FR, *.PR, TS, ES, M)
=> (AS, RS, FR, PR, TS, ES, M)

SaveFrame ( )
(AS, RS, FR, *.PR, TS, ES, M)
=> (AS, RS.FR, FR, PR, TS, ES, M)

ApplFrame ( )
(AS.x:α.clos(text(c:code,..),..):$\alpha \rightarrow \beta$, RS, FR, *.PR, TS, ES, M)
=> (AS.x, RS.PR, clos(text(c,..),..), c, TS, ES, M)

RestFrame ( )
(AS, RS.FR', FR, *.PR, TS, ES, M)
=> (AS, RS, FR', PR, TS, ES, M)

Return (n: int≥0, m: int≥0)
(AS.x_{n+m-1}:α_{n+m-1}..x_0:α_0, RS.c:code, FR, *.PR, TS, ES, M)
=> (AS.x_{m-1}..x_0, RS, FR, c, TS, ES, M)

TailApply (n: int≥0, m: int≥0)
(AS.x_{n+m-1}:α_{n+m-1}..x_0:α_0.clos(text(c:code,..),..):$\alpha \rightarrow \beta$,
RS, FR, *.PR, TS, ES, M)
=> (AS.x_{m-1}..x_0, RS, clos(text(c,..),..), c, TS, ES, M)

Exception operations deal with the exception stack. There is no interaction between the trap stack and the other stacks, except during exceptions. Hence, function call in not made any slower by exceptions and traps which do not fire.

The FailWith operation takes a string and generates a failure with that string as failure reason. The failure can be trapped by a previously executed Trap or TrapList instruction, which saved the state of the machine (except the heap) at a failure recovery point. *Trap* saves AP, FP, SP and a PC, corresponding to the failure handler, on the trap stack TS together with a flag meaning that all failures will be trapped. *TrapList* takes a list of strings and saves AP, FP, SP and the PC of the handler on TS, together with the list of strings which is used to selectively trap failures. *UnTrap* reverts the effect of the most recent Trap or TrapList. *FailWith* takes a string s and searches the trap stack from the top for a Trap block or a TrapList block with a list of strings containing s. If one is found, the corresponding state of the machine (AP, FP, SP, PC) is restored and the Trap or TrapList block and all the ones above it are removed. If no matching trap is found, the message 'Failure: ' followed by the failure string is printed on the standard output stream, and the machine stops.

Trap (c: code)
(AS, RS, FR, *.PR, TS, ES, M)
=> (AS, RS, FR, PR, TS.(all,c,RS,FR,AS), ES, M)

TrapList (c:code)
(AS.sl:string list, RS, FR, *.PR, TS, ES, M)
=> (AS, RS, FR, PR, TS.(only(sl),c,RS,FR,AS), ES, M)

UnTrap (c: code)
(AS, RS, FR, *.PR, TS.(all,c,RS',FR',AS'), ES, M)
=> (AS, RS, FR, c, TS, ES, M)
(AS, RS, FR, *.PR, TS.(only(sl),c,RS',FR',AS'), ES, M)
=> (AS, RS, FR, c, TS, ES, M)

FailWith
(AS.s:string, RS, FR, *.PR, TS.(x,PR’,RS’,FR’,AS’).., ES, M)
=> (AS’.s, RS’, FR’, PR’, TS, ES, M)
where (x,..) is the first trap block from the top of TS
such that x=all or x=only(sl) and s is contained in sl.
(AS.s:string, RS, FR, *.PR, TS.(x,PR’,RS’,FR’,AS’).., ES, M)
=> (AS.printfailure(s), RS, FR, <>, <>, ES, M)
if there is no trap block satisfying the above condition.

Here are some suggestions about how to compile high-level language expressions into Fam operations. There is a translation function ‘⟦ ⟧’ from expressions to Fam programs, for example ‘⟦3⟧ => Int(3)’ means that the expression ‘3’ is translated into the Fam operation ‘Int(3)’.

Primitive operations (like ‘+’) which have a corresponding Fam machine operation are translated by translating their arguments left to right, and then suffixing the appropriate Fam operation:

⟦op(arg_1,..,arg_n)⟧ =>
⟦arg_1⟧ .. ⟦arg_n⟧ ⟦op⟧

Variables are converted to a GetLocal or GetGlobal operation, depending on where they are defined; strings are converted to GetLiteral:

⟦ .. x .. y .. "string" .. ⟧ =>
.. GetLocal(n) .. GetGlobal(m) .. GetLiteral(p) ..

Function applications are translated by translating the argument, the function and then appending the three parts of the apply operation:

⟦f(a)⟧ =>
⟦a⟧ ⟦f⟧ SaveFrame ApplFrame RestFrame

Functions are compiled into sequences of operations which, at run time, build closures. First all the global variables of the function are collected from the appropriate environments (we informally use Get(x) for GetLocal or GetGlobal with the appropriate displacement), then the text of the function is fetched by GetLiteral, and finally a Closure is generated.

⟦fun x. .. x .. y .. z .. "s" .. ⟧ =>
Get(y) Get(z)
GetLiteral(text(⟦ .. x .. y .. z .. "s" .. ⟧Return(1,1),"s"))
Closure(2)

Recursive functions involve DumClosure and RecClosure. Here is the compilation of two mutually recursive functions f and g:

⟦let rec val f = .. g .. and g = .. f .. ⟧ =>
DumClosure(1) DumClosure(1)
GetLocal(0) GetLiteral(text(⟦ .. g .. ⟧Return(1,1))) RecClosure(1,1)
GetLocal(0) GetLiteral(text(⟦ .. f .. ⟧Return(1,1))) RecClosure(1,0)

Modules and type-safe separate compilation.

While a complete module mechanism is being designed [MacQueen 84], the current ML compiler includes a preliminary implementation of separately compiled modules, which loosely resembles the facilities provided by Modula-2 [Wirth 83]. Current ML modules are parameterless and do not support multiple instances. The implementation techniques currently used for modules are briefly described here, also because they seem to be relevant for more ambitious proposals.

When a module is compiled, the compiler writes out an *implementation* file, containing the program code for the module, and a *specification* file, containing type information. A module can import other previously compiled modules, and the specification files are used to typecheck across module boundaries. Version numbers are automatically maintained to keep track of obsolete modules which must be recompiled.

The importation of modules is dynamic: it can happen conditionally, and can be hidden in functions imported by other modules. Every module, although it can be imported several times, is actually loaded only once. This is not only desirable, but necessary in this framework because of type declarations. If two modules A and B import the same module C, it is important that the type declarations of C are processed only once, otherwise A and B would have different version of the same type declarations, and the typechecker would complain if these had to interact (e.g. because of a module D importing A and B).

We can think of a module as a closure with no global variables (as modules are self-contained), whose program text defines an set of bindings, or *environment*. An environment is implemented as a record of n fields in the heap, as opposed to a piece of stack. The following example describes a module with parameters (to show how parameters may be treated), which also imports other modules.

module A[B,C]
includes D E
body
    ... import F G ...
end

The *text* of this module defines an environment A; the environments B and C will be found on the top of the argument stack before the execution of the text (just like after a function application); the environments D, E, F and G will also be found on the argument stack during the execution of the text (just like in a let definition).

The access to a variable in B, C, D, E, F or G, requires a GetLocal operation (to access the environment record) and a Field operations (record selection). This corresponds to two MOVE instructions on VAX and 68000 (one instruction on 16032). This is a bit more expensive than an internal variable access, but probably not noticeably so. To access variables in modules imported by B, C, etc, (but not imported by A) requires even more indirections.

Sharing of imported modules is achieved simply by sharing an environment record across several modules. Hence, reimporting an already imported module happens essentially for free.

Separate compilation is achieved by producing the text of a module without its closure (which is empty, anyway), and exporting that text to a file, which becomes the implementation file for that module. This text is compiled assuming that it will be called during linking with all the necessary parameters (i.e. environments) on the argument stack. The result of that call will be an environment with one field for every identifier directly exported by the module, and one field for every exported submodule.

Every **includes** or **import** declaration is carried out either by dynamic linking, or by simply fetching an already imported environment. To dynamically link a precompiled module we first produce an (empty) closure for it. If the module is parametric, before calling its closure we have to supply (i.e. link or fetch) the parameter environments. Then the closure is called, producing an environment which can be used to link further modules. The result is left on the argument stack and is inserted in a global module pool for later sharing.

Conclusions

This paper reviewed some ML implementation techniques which are, in various degrees, original in isolation or in combination with other techniques. These techniques have historical roots in Lisp and λ-calculus dialects, and have been developed by various people during the implementation of the ML/LCF system, the Hope language, and two further generations of ML.

References

[Aho 78] A.V.Aho, J.D.Ullman: "Principles of Compiler Design", Addison-Wesley, 1978.

[Allen 78] J.R.Allen, "Anatomy of LISP", McGraw-Hill 1978.

[Burstall 80] R.Burstall, D.MacQueen, D.Sannella: "Hope: an Experimental Applicative Language", 1980 LISP Conference, Stanford, August 1980, pp. 136-143.

[Cardelli 83] L.Cardelli. "The Functional Abstract Machine", Bell Labs Technical Report TR-107, 1983.

[Gordon 79] M.Gordon, R.Milner, C.Wadsworth. "Edinburgh LCF", Lecture Notes in Computer Science, No. 78, Springer-Verlag, 1979.

[Henderson 80] P.Henderson: "Functional Programming, Application and Implementation", Prentice-Hall, 1980.

[Kowalski 79] R.Kowalski: "Logic for Problem Solving", North-Holland 1979.

[Landin 64] P.J.Landin: "The Mechanical Evaluation of Expressions", Computer J., Vol. 6, No. 4, 1964, pp. 308-320.

[MacQueen 84] D.B.MacQueen: "Modules for Standard ML", this conference.

[Milner 78] R.Milner: "A theory of type polymorphism in programming", Journal of Computer and System Science 17, pp. 348-375, 1978.

[Milner 84] R.Milner: "A proposal for Standard ML" Internal Report CSR-157-83, Dept of Computer Science, University of Edinburgh.

[Plotkin 81] G.D.Plotkin: "A Structural Approach to Operational Semantics", Internal Report DAIMI FN-19, Computer Science Dept, Aarhus University, 1981.

[Robinson 65] J.A.Robinson: "A machine-oriented logic based on the resolution principle", Journal of the ACM, Vol 12, No. 1, Jan 1965, pp. 23-49.

[Steele 77] G.L.Steele: "RABBIT: A compile for SCHEME", AI-TR-474, MIT, May 1978.

[Steele 78] G.L.Steele: "Debunking the 'Expensive Procedure Call' Myth", Proc AMC National Conference (Seattle, Oct 1977) 133-162.

[Wirth 83] N.Wirth: "Programming in Modula-2", Springer-Verlag 1983.