The Edinburgh Standard ML Compiler

Kevin Mitchell
Dept of Computer Science
Edinburgh University

Alan Mycroft
Computer Laboratory
Cambridge University

Draft Jan 1985

Abstract

We describe the abstract machine (FAM) used by the Edinburgh Standard ML system and its compiler phase. The reasoning behind the design choices is given.

Note

The parser and typechecker phases of the ML system are not described here. The former is intended to be the subject of another paper and the theory behind the latter given in [Mil 77] and [Damas].

1 Introduction

ML started life as the Metalanguage for the Edinburgh LCF system [GMW], an interactive theorem prover. However, it became apparent that the features of the language were of more general interest. (ML had a strong, but flexible, type system [Mil 77] often referred to as "polymorphic", encouraged programming in a functional style although it allowed assignment and had an exception mechanism (failures) which many found convenient and easy to use). HOPE [BMS] later used the same type system in a functional language together with a new type definition facility and expressive pattern matching notation.

Cardelli [Car] then re-implemented the ML language, adding new features, but probably his most important contribution was the definition of a functional abstract machine (FAM) which he regarded as an intermediate code between ML and VAX machine code.

Milner [Mil 84] unified these approaches by defining a new language, Standard ML, incorporating many of the best features of ML, HOPE and Cardelli's dialect. It is intended that Standard ML should replace these languages.

This paper describes part of the Mitchell, Mycroft and Scott Standard ML system developed at Edinburgh in 1983-4. In particular we describe our FAM which we treat as a real machine to be emulated (or preferably microcoded) on current hardware, together with the compiler from Standard ML abstract syntax into FAM code. We regard our FAM as an improvement on Cardelli's in a number of significant areas. It also bears resemblance to an abstract machine previously designed by one of the authors for the language PAL [Evans].

At this point we should explain certain policy decisions. Probably most importantly, our decision to emulate rather than compile FAM code on (in our case) a VAX. The motivation was that Cardelli's compiled VAX code tended to be very large (many primitive ML functions require quite a few bytes of code or a procedure call). Whilst this produced very acceptable results for one-page benchmarks we soon found that large systems behaved very badly. (Our ML system consists of some 10 000 lines of ML code.) We put the bad behaviour down to the increased paging caused by VAX code over our more compact FAM (byte stream) code. (We are talking about a factor of 10 in size.) Moreover an emulator for compact code produces a much better cache performance than the larger VAX code which tend to duplicate code which would have been part of an emulator. A further point is that the emulator overhead on a VAX is only two VAX instructions per FAM instruction and is less than the cost of a procedure call. Tests (as yet inconclusive) actually show a speed advantage to the emulator of a factor of 2 on a sizeable program. We attribute most of this to compiler optimisations.

Certainly, we feel that, regardless of the ever-reducing cost of memory that program and data should still be stored as compactly as possible since applications will also continue to grow. This view is shared by Clocksin [Clo] who makes similar arguments concerning an emulated prolog machine, ZIP.

An as-yet-unrealised intention is that we might compile time-critical parts of an ML program into native machine code and emulate FAM code for the rest. This would have the (small-scale) time advantage of native code together with acceptable large-scale behaviour.

One very beneficial advantage of having a defined FAM code is that it inhibited us from "bit-twiddling" VAX machine code optimisations and ensured that our ML to FAM compiler incorporated higher-level (and thus potentially more profitable) optimisations.

Thus we incorporated recursion-to-iteration and inline expansion optimisations which are easy to postpone when one has a native compiler.

The structure of this paper is to describe in subsequent sections, the FAM, the ML compiler and its assembler.

2 The Abstract Machine

The abstract machine is of the style of Cardelli's functional machine [Car] and we acknowledge this by giving it the same name, FAM. It is basically a zero-address instruction stack machine although many of the control instructions are followed by 1 or 2 items of immediate data (offsets for example). Cardelli's intention was that his FAM was not a real machine at all, but merely an intermediate code form which was to be translated into the native machine code.

Our attitude is the total opposite in that we actually emulate the FAM instructions (which are stored in a form of byte vector called a TEXT) with a simple interpreter, which we have available in C or VAX machine code. The instructions are also oriented to micro-coding.

We now discuss the effects and consequences of this difference in more detail.

Firstly, because our FAM instructions are emulated, we strive towards simplicity rather than generality. For example (see later), our instruction "DestTuple n" takes an object on the top of it with n new stack elements derived from its components. Cardelli's "DestRecord d_0 d_1 ... d_n" instruction is similar, but finds the tuple at offset d_0 from the top of stack and places its components in offsets d_1 ... d_n from the top of stack. The size, (n) is determined from the object.

Moreover, because we intend to emulate the FAM code, the <u>implementation</u> of the FAM (or perhaps we should refer to it as the Functional Concrete Machine) can have specialised versions of certain general instructions available. Thus, for example, the instruction "GetLoc n" fetches the local (stack) variable n from the top of stack and pushes it on the stack. It transpires that "GetLoc 0" and "GetLoc 1" are very common special cases, and we give them separate op-codes. (In fact the GetLoc 0 instruction is still the most common byte code!)

Doing this further reduces code size and increases execution speed (as no stack offset fetch must be performed).

Similarly, we are not adverse to later incorporating generalised versions of opcodes if necessary, but we do not need the DestRecord generalisation of DestTuple to implement ML. Many of our motivations are those which drive research for "Reduced instruction set computers" (RISC) - see, for example [ ].

As we mentioned earlier, we really do believe that code size is an important factor in building real systems (rather than one page benchmarks) in that it has drastic effects both on paging and caching. The same attitude is also seen in the design of the data structures for ML.

Note that these arguments apply to both large virtual memory machines (to enhance caching and paging performance) and to machines with physical memory (to avoid it becoming prematurely exhausted) and improve garbage collection performance.

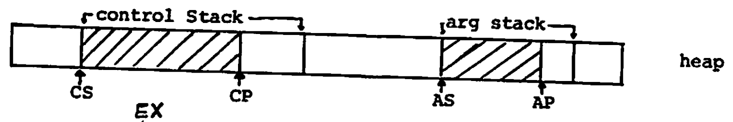
The FAM Architecture

Our FAM has 2 stacks and a heap. The stacks cannot grow and shrink independently, but we keep them separate to avoid permuting the top elements of a single stack (or leaving holes) on procedure call/return.

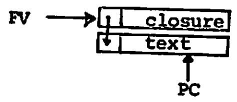
In the following we will draw the stacks as occupying increasing addresses and with the stack pointer pointing to the item most recently pushed.

One stack is the local variable stack (AS) which is used to hold parameters to procedures and tempories as well as supporting zero-instruction stack-based operations. The other is the Control Stack (CS) which contains activation records (. . procedure return information) and details of currently active exception handlers. The stacks are referenced by two machine registers which we will call AP and CP holding the current top of stack pointer. The heap is used for allocating all (non-immediate) data and is assumed to be garbage collected when full below the level of this description. Suffice it to say that garbage collection is autonomous and updates all machine registers transparently to the FAM operations. We currently allocate the stacks as elements in the heap thus simplifying the machine design and permitting dynamic change of stack size on overflow.

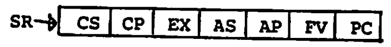
So far the machine looks like this:



There are in principle only three more registers on the FAM: the exception register EX which points to the most recent exception handler frame within CS, the closure pointer FV which points to the data structure representing the currently active routine closure and the program counter PC itself which points to the FAM instruction next to be executed. This will be given in more detail later, but may be visualised



For many purposes, including the possibilities of coroutines, multi-processing, garbage collection and the like, it is convenient to view the machine as having a single register, the state register SR. A state object is then a pointer to the heap which contains (essentially)



In principle, executing a FAM instruction updates the fields addressed by SR to reflect its effect. Of course, this is not an efficient way to organise a machine (either emulated or microcoded) and so the fields addressed by SR are slaved into emulator registers and updated only on task switch (simply updating of SR), garbage collection and the like.

Similarly, the instruction descriptions below will refer to machine register LIT which slaves a pointer to the base of literals which is accessible from FV, and which is transparently updated whenever FV is. See the description of TEXT objects.

<u>Data Representation</u>

In the FAM, due to the needs of garbage collection (a stack element may point to any object), objects must be tagged with their type. Note that this type is almost certainly not the type in the high level language sense - for example many different high-level language types may have the same machine representation.

There is a choice as to whether to place this tag information in the pointers themselves (the so-called rich pointer sheme often used in LISP) (See for example [NF]) or whether to store it within the object pointed to (a natural approach for (say) Algol 68). We choose the latter method but without any dogmatic attitude. The reasoning is that the FAM instructions either operate on a single type of object, or on all objects - there are no operations like PLUS in LISP which perform different operations according to the tag bits. Therefore the ubiquitous removal of tags from rich pointers merely gets in the way. Significantly Clocksin's Prolog-X sytem [Clo] adopts the former approach - thus prompting the thought that the two methods' efficacy depends on whether the source language favours dynamic (Prolog, LISP) or static (Algol 68, ML) typing.

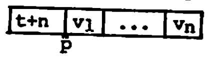
However, we do not wish every object to be a pointer and so reserve the top bit(S) in every value, to specify whether the value is to be treated as a pointer (S=0) or an immediate object (S=1). We accept the criticism that this is somewhat of a halfway house, but justify it on the grounds of compactness and of simple emulation on current machine architectures.

There is also the possibility, realised in earlier versions of the FAM including Cardelli's, of using "caged" data structures in which the type of a structure is determinable from the memory page in which it resides via a type table. However, we (and [NF]) reject this as being essentially incompatible with compacting garbage collection.

We now describe the various data structures which the FAM uses to represent ML - however, due to our search for minimality it is hard to envisage a language which does not require most of these objects.

(1) Small integers. On an n-bit machine these range from $[-2^{n-2}, 2^{n-2} - 1]$ and are represented by immediate objects by scaling with 3.2^{n-2}.

(2) Tuple of n components.
   (a) for n=0 we use the value 0 which is also regarded as an immediate value. () the single value of type <u>unit</u> (<u>void</u> in Algol (68) and is the result of the assignment operator etc.
   (b) For n>0 a pointer p tc a block of n+1 cells



The tag t for tuple and the length n are stored at a negative offset from p (they are only required for garbage collection) and the components $v_1, \ldots, v_n$ stored successively from p.

(3) Array of size n.

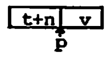
Arrays are stored identically to tuples, but with a different type tag to reflect that their components are updatable - this can help data sharing on a shared ML in a multi-user machine and some garbage collectors.

(4) Reference object.

Reference cells model assignable variables and are represented as arrays of length 1.

(5) Variant object

Variants are used to represent disjoint sums in ML and (excepting special cases where the compiler can determine that summands are already disjoint) are represented by a pointer to a 2-cell object.

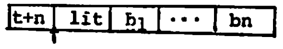


The value t+n stored in a negative offset shows that the object is the result of the nth injection function applied to v. The tag t again is only used for garbage collection.

To save space and time the common case of v=0 (ie()) is represented as an immediate object (top bit set) corresponding to the small integer n. This covers common cases such as <u>type</u> colour = <u>data</u> red <u>of</u> unit| green <u>of</u> unit| blue <u>of</u> unit.

(6) Text object

A text object is the representation of the body of a routine. Notionally it consists of a single object containing FAM (or other) machine code. However in general machine code has to refer to literals, and the garbage collector has to be able to determine the position of these. Hence we follow Cardelli and store the literals in a tuple and store the address of this as the first cell in the text.



there t is the tag for text, n the number of bytes of code and lit the literal pointer.

(7) Bignum, String, Floating point objects

These are stored identically, as for tuples but with tags corresponding to their type (the tag informs the garbage collector that the components are not structures to be followed). We use different type tags for Bignum and String so that we may transport binary files of "heap images" without incurring problems concerned with byte "sex" and differing floating point representations.

Representation of ML objects in the FAM

Most ML objects have a direct correspondent in FAM objects. One exception is closure objects. A closure of free variables whose values are $v_1 \ldots v_n$ and whose text is c represented as the (n+1) tuple (c, $v_1 \ldots v_n$).

Another is the representation of "exception" objects. We must satisfy the ML requirements that an exception declaration must generate a new exception each time it is elaborated and that the name (a string representing the declaring identifier) must be accessible from an exception. Our solution is to treat a declaration such as

```
  exception foo;
```

as equivalent to

```
  val foo = ref ("foo")
```

We describe this as a reference cell so that each elaboration of the declaration requires construction of a new cell containing the string "foo". However we never update the contents of this ref cell. Exception equality is now pointer equality. For further details consult the definition of the FAM opcodes for dealing with exception.

Machine Operations

The description of all the FAM opcodes is not built into the compiler because many of them are simple stack operations like integer addition which pops two integers from AS adds them and pushes the result. These can be declared within ML by syntax of the form

```
  primitive    + (x:int, y:int):int opcode   n;
```

where n is the above mentioned opcode.

We will accordingly describe these opcodes in a second list (see appendix) since they are in general specific to ML.

Firstly, however, we will describe the basic machine code operators which it seems necessary for any compiler to know about. The following lists the opcodes and parameters.

Data movement

GetLoc n: Push (on AS) the object (which was) nth from top of AS

GetFree n: Push the nth component of FV

GetLit n: Push the nth literal, ie (((FV])+n)

GetClos: Push FV on AS (used for certain forms of recursion).

Slide n k: Lose k items from the stack - those in positions [n, n+k-1] from the top - by sliding down the top n items by k positions. Useful special cases are n=0 (pop) and n=1 (lose lcoal declarations, saving result). which have special case opcodes. The general case is used for tail recursive calls of more than one parameter.

Jumps

Jump l: Jump to label l. Labels are two byte relative addresses from the current PC.

JTrue l: Pop AS. The value popped should be an immediate object representing $in_0()$ or $in_1()$. Jump iff the latter.

JFalse l: As for JTrue but with "former" for "latter".

See also the case instruction amongst "Variant operations".

Tuple operations

Tuple n: Pop n values $v_1, \ldots v_n$ from AS and push the value ($v_1, \ldots, v_n$).

DestTuple n k: The kth element of AS must be a tuple of size $n \geq 1$. Store its first component in poisition k on AS and Push its subsequent components. The special case k=0 is common (and often referred to, as earlier, as DestTuple n), is an inverse to Tuple n and is given a special case opcode.

The general case $k \neq 0$ is needed for (non-wasteful) compilation of patterns like ((a,b),(c,d)).

Dot k: The top of stack is a tuple of size $n \geq k$. Replace it with its kth component.

Variant operations

Inject k: Replace v, the top of AS, with the variant object (k,v) (which is stored immediately if v=0).

Case n $l_1 \ldots l_n$: Pop AS. The value popped must be a variant (k,v) (or its immediate form). Push v and jump to l_k.

Ord : Replace the variant (k,v) on top of AS with k.

Functions

The breaking of "apply a function" into "Saveframe, Apply, Restframe" is due to Sussman & Steele [SS]. It allows optimisation of curried applications such as $f(e_1)(e_2)$.

SaveFrame:	Push FV on CS
Apply:	Push PC+1 on CS, pop AS into FV and then load PC with (FV)+1
Return:	Pop PC from CS
RestFrame:	Pop FV from CS
TailApply:	Pop AS into FV, then load PC with (FV)+1.

TailApply has the effect of SaveFrame; Apply; ReturnFrame; Return but less (code and) stack space.

NB since sequences like "Slide 1 n; Return" are so common there is an optimised opcode "Return n" which has the same effect - see later under optimised opcodes.

Exceptions

Handle n $l_1 \ldots l_n$: (see case). Pop AS which must be a tuple of size n containing the exception names the handler handles. 0 represents a wildcard. Save it in temporary R. Push on to CS the following exception frame:
AS, FV, PC, EX, R
Set EX to point to the new CS top and execute next instruction.

Raise: Pop from AS the exception name and parameter value. Search the chained frames headed by EX for one which matches (pointer equality of wildcard match). On a match (which must occur) with the k^{th} component of an exception name tuple, restore CS from EX and then pop 5 items restoring AS, EX. PC is set to the label l_k accessed from the stored PC in the EX frame. Push the exception parameter value, and also the exception name if the exception was caught by a wildcard.

This scheme was chosen rather than a scheme such as that in VAX hardware where each function call sets up a (potential) exception handler because we wanted function call to be efficient. Moreover, the above scheme is quite efficient itself since exception frames are dynamically allocated and deallocated on CS without producing garbage.

Optimised opcodes

Certain sequences of such opcodes are common in ML and we have provided single opcodes for such sequences - this saves both FAM code size and time in emulation. (As mentioned before we also have special case opcodes for specialised froms of parameterized opcodes - such as GetLoc 0

and Slide 1 n).

Num n: (for n a small integer) abbreviates GetLit where Literal k would contain n.
There are special forms for n=0, n=1, n<256.
False and True are conveniently represented by the first two forms.

Is k: Pop AS, replace it with true or false according to whether it was variant k or not. This is shorthand for a certain case.

DestVariant k n: Examine the k^{th} element of AS, which must be a variant object. If it is a variant (k,v) replace it with v otherwise push the exception name "bind" and () on AS and perform raise. The general case is used in pattern matching - see the description of DestTuple.
Outject k $\equiv$ DestVariant k 0 - this abbreviates a certain case.

ML primitive opcodes

The machine has many other operations which pop k objects from AS operate and push a result. These are not built into the ML compiler but are introduced by special syntax. They are listed in the appendix but examples are

Iplus:	pop two integers off AS and push their sum.
Assign:	pop a value and a reference cell off AS, update the reference cell with the value and push () on AS.

3 The Compiler

In this section we describe the ML compiler. This is written in ML and has a type SynDecl→FAMCode list, ie it accepts an abstract syntax tree for an ML declaration (ML top level expressions, e, are considered as a shorthand for the declaration "val it=e") and returns a list of FAM instructions possibly interspersed with labels. Labels definitions are considered as a FAM pseudo-opcode in the usual manner and are resolved by the assembler described in the next section.

Let us first say a little more about the ML abstract syntax used. Firstly, the ML "derived forms" such as the while e do e^1 construct are by and large represented by their equivalent "basic forms" in this case
let val rec f()= if e then (e^1; f()) else () in f().

This translation is performed on the basis of a simple macro expansion facility by the parser, which merely constructs one (slightly more complicated) piece of abstract syntax tree instead of another. We do not suffer a performance penalty for the above
nisation" "~~permission~~" since compiler algorithms described later (self-recursion removal and inline expansion) will convert the second piece of code into the obvious naive piece of code which one would expect from a while-loop, but with more generality and without penalising the user who wishes to use a functional style.

Secondly, given that typechecking (which in our system is a separate phase performed on the abstract syntax tree before compilation) requires determination of which use of a variable is tied to which definition, the compiler assumes that typechecker inserts a reference (pointer) back to the definition of a variable from each use. We consider this to be a good technique for two reasons. On one hand it allows simple coding of the compiler which therefore needs to perform no environment analysis. On the other it satisfies our quest for simplicity, efficiency and reliability - if a complicated operation is performed twice then it leads to increased possiblity of error, or checking code to ensure that the two algorithms actually yield the same result.

(In actuality, a consequence (which we regard as unfortunate) of the definition of Standard ML requires certain knowledge about the binding of names to definitions in a program to enable it to be parsed and so,,at some later stage, this name/definition association code probably ought to be moved into the parser rather than the semantic analysis phase where it naturally belongs.)

The basic form of the compiler is to traverse the declaration to be compiled by recursive descent.

Somewhat surprisingly at first, we perform this recurseive descent in a right-left fashion. The reasons for this are
1) ML only has constructs for forward branching.
2) The compiled code representing "the program to the right" has a natural interpretation as a continuation - this enables coding in a functional style.

The interest in 1) from the standpoint of compilation is that the optimisation to elide jumpt to jumps is trivial to implement - simply test whether the first real instruction to the target code sequence is a jump instruction. Eliding such jumps is necessary to make full use of tail recursion optimisations. Thus, to give an example, conditional constructs are compiled in the following way:

```
compile (if e then e' else e", RhtProg]=
     let ContProg = label (RhtProg) in
     let l"  = label (compile (e" , ContProg]] in
     let l' = compile (e', jump (ContProg, l" []] in
     compile (e, Falsejump (l" ,l'))
```

where the routinesJump and Falsejump take 2 arguments - the first a target to the jump (this is always a terminal sublist of the continuation for forward jumps) and the second the continuation if the branch is not taken.

In general all compilation routines are continuation transformers - ie of the form

τ x FAMCode list $\longrightarrow$ FAMCode list.

Optimisations

In addition to the "peephole" optimisations such as jump elision the compiler supports the following optimisations:

1) Constant folding

Any constant structure, built from pure structure building operations constants, tupling, injection (and consequently closure formation) but not occurrences of "ref e" for obvious reasons, is folded into a single literal which is accessed via a getliteral instruction. Thus, for example, supposing a and b are global (top level) variables, then the following is a constant expression

((λx.X), [1,2,3,4], "abc", (λx.a+x), inl (b))

2) Tuple optimisation,

In ML all functions take one argument which may be a tuple. If the value to a function is always a tuple whose value is never referenced (only its components) then the function is compiled as a function of several arguments taken from the stack thus saving CONS^{1}ing into a tuple and subsequent decomposition.

That this is possible is determined by the definition of the function. In general we still need the single argument version of the function - consider

map (+) [(1,2), (3,4), (5,6)].

Our solution is to have a "general entry point" to a function such as +, which spreads its (guaranteed by type checking) tuple argument onto the stack, and then tail-applies to the "fast entry point".

Note In our tuple optimisation, we have chosen only to optimise "top-level" patterns tuples - thus the definition
f((a,b), (c,d),x) = e
would receive three parameters and itself be responsible for further decomposition of the first and second into a,b and c,d respectively. Similarly, for functions defined by cases, the case analysis is performed within the called function. We did this because it conforms to (our) view of common programming

style and the above probably produces the most efficient code for this style. However we do not adopt this standpoint dogmatically and welcome other views.

3) Tail Application

If an Apply instruction is followed by a Return instruction (possibly separated by stack adjustments (slide) or jumps which are previously optimised by optimisation 1) then it (and the Return instruction is replaced by a TailApply instruction. Moreover, if the previous instruction was a load current function (GetClos) the TailApply is replaced by a Slide and a (backward) Jump to the beginning of the current function - note that recursion is the only source of backward jumps in functional languages. This is called tail self-recursion removal.

4) Inline expansion

Whenever a function is compiled and its compiled FAM code is "small" then the FAM code is saved along with the function value. Currently we expand in-line any function whose calling sequence would be longer than the function expansion. In general this achieves our aim of not penalising the use of small functions (used, for example, to define abstract data types).

Also, any function which is only used once is also inline-expanded - this, together with optimisation (3) ensures that "while" loops are compiled into the usual code.

Inline expansion is more complicated than it might first appear due to the need to change some free variables (of the expanded function) into local variables.

4. The Assembler

As we indicated in the previous section, the result of the compiler pass is a list of FAM opcodes, interspersed with labels. This is assembled by a standard two pass assembler into a byte stream stored in a text object, together with its literals. The assembler is responsible for selection of an appropriate form of opcode. Thus the compiler might emit

```
   LoadLit  (int 3)
```

and the assembler chooses an optimal instruction - here

```
   ByteNum 3
```

which constructs a small integer (immediate object) by loading the byte 3 from the instruction stream and adding 3.2^{n-2} (n is here the

word length).

This has the advantage that the compiler can restrict itself to higher level optimisations and the assembler performs the machine dependencies.

We note that our assembler takes less than 10°/° of the total (non-run) time - the majority being consumed in parsing and type checking and so there has been no motivation to change to a 1-pass assembler.

The first pass of the assembler collects a literal pool and determines the code size. The literal pool is then formed into a tuple and an empty text object of appropriate size containing the literal tuple is obtained (there is a machine operation "Maketext" which pops these values from AS and replaces them with the empty text).

On the second pass the assembler inserts an appropriate sequence of bytecodes and immediate literals into the text.

5. Statistics

Our ML compiler is written in ML (about 10 000 lines) which produces about 300kbytes of code of which about 80kbytes are FAM byte code instructions.

The FAM emulator on the VAX together with C's I/O library and the garbage collector consumes 64kbytes of which the central interpreter loop is about 1kbytes.

The compiler takes about 24 minutes to compile itself in a VAX/780.

References

[BMS] Burstall, R M, MacQueen, D B, Sannella, D T, "HOPE - Internal Report, Dept of Computer Science, Edinburgh University 1980. CSR-62-80

[Car] Cardelli, L, "The ML functional abstract machine", unpublished draft 1984.

[Clo] Clocksin, W F, "Design and Simulation of a Sequential Prolog Machine", New Generation Computing, 3 (1985)

[Damas] Damas, L, PhD thesis, Dept of Computer Science, Edinburgh University, 1984.

[Evans] Evans, "PAL- Pedagogic Algorithmic Language", MIT report 1970.

[FN] Ffitch, J, Norman, A C, "Implementing LISP in a high-level language", Software Practice and Experience, 1977.

[GMW] Gordon, M, Milner, R, Wadworth, C, "Edinburgh LCF", Springer-Verlag LNCS, Vol 78.

[Mil 77] Milner, R, "A theory of polymorphism in programming" JCSS, 1977.

[Mil 84] Milner, R, "The Standard ML core language", Internal Report CSR-168-84, Dept of Computer Science, Edinburgh University, 1984.