A Compiler for Lazy ML

Lennart Augustsson

Programming Methodology Group
Department of Computer Science
Chalmers University of Technology
S-412 96 Göteborg, Sweden

Abstract

LML is a strongly typed, statically scoped functional language with lazy evaluation. It is compiled trough a number of program transformations which makes the code generation easier. Code is generated in two steps, first code for an abstract graph manipulation machine, the G-machine. From this code machine code is generated. Some benchmark tests are also presented.

1. Introduction

The LML compiler project is an attempt to produce efficient code for a functional language with lazy evaluation for an ordinary von Neumann machine. When we started we knew of no other attempt to do this, but since then some similiar things have appeared like [Huda84], and [Fair82]. There are several compilers for non-lazy functional languages, eg. [Card84].

The LML compiler is written in LML and it produces (as an intermediate step) G-code, code for an abstract graph manipulation machine, from which machine code generation is fairly easy. This makes the LML compiler easy to port to other machines.

The approach used in the compiler is to perform many transformation on the program ("source to source" transformations) to get a program for which generation of efficient code is less complicated.

The execution of the program is based on graph-reduction. The graph represents the expression that is evaluated, and it is transformed until it is in a printable form. During execution there is also an ordinary stack on which computations are performed (as in ordinary languages) when this can be done without violating the lazy evaluation semantics.

2. LML Language Description

Lazy ML, or LML for short, is a lazy and completely functional variant of ML, [Miln84] and [Gord79]. The syntax used here is slightly different from Standard ML. The main differences between LML and ML is that LML has lazy evaluation, there are no references type nor assignments and no exception-mechanism; strings are lists of characters, and there are no explicit input/output procedures.

An LML program is an expression whose value is printed when the program is executed.

Function definitions may use pattern matching, which makes programs both easier to write and understand. Such a definition contains a number of equations for a function separated by ||, e.g.

```
let rec fac 0 = 1
     || fac n = n * fac(n-1)
in fac 10
```

Pattern matching can also be used to bind multiple values like `let (a,b) = f x in'. There is also a case expression to do pattern matching.

An important concept in LML is local definitions. A local definition has the form* `let D in e' where D is a declarator**. In a let expression the declarator defines the meaning of a number of identifiers that can be used in the expression part of the let.

* Appendix A describes the terminology used in the following descriptions.

** Declarator syntax is described in appendix B.

2.1 Pattern matching semantics

All pattern matching is translated into case expressions (as described below), so this explanation need only concern the semantics of case expressions.

The case expression is evaluated by finding the first pattern that matches. The patterns are check from top to bottom, and each patters is checked from left to right. The checking is stopped as soon as a subpart fails to match. This rather explicit top-down, left-right ordering is perhaps unfortunate, but some ordering must be imposed to avoid the necessity of parallel evaluation of the subparts of the expression that is to be matched.

2.2 Type definitions

It is possible to define new types in LML, the mechanism for this is very similar to the one proposed for Standard ML (SML). The type definitions resemble those of Hope, [Burs80]. It would be possible to have no predefined types (except the function type) and instead let the user define all types. This would give the same performance as having them predefined, except for the integers which are implemented with the machine arithmetic.

A new type is introduced by

`let type T in e'

where T is a type declarator, which is similar to an ordinary declarator, ie. it can be

`T_1 and T_2' mutual definition

`rec T' recursive definition.

`$i(v_1, \ldots v_n)=C_1(t_{11}, \ldots t_{n1})+\ldots C_m(\ldots)$' where i is the name of the new type and $v_1 \ldots$ are type variables, $C_1 \ldots C_n$ are the new constructors, t_k are type expression possibly containing v_k.

Using this the booleans could be defined by

```
let type bool = true + false in
```

3-tuples

```
let type tuple3(*a, *b, *c) =
    T3(*a, *b, *c)
```

and lists by

```
let type rec list(*a) =
    nil + cons(*a, list(*a)) in
```

(To make things more readable nullary constructors are written without `()'.) All of these are predefined are of course predefined.

There is a difference in type definitions between LML and Standard ML. In SML all constructors take either zero or one argument. To get more arguments a tuple must be used instead; so the list definition would be

```
let type rec list(*a) =
    nil + cons(*a # list(*a)) in
```

(`#' is used for cartesian product.) This means that a list on cons form may be formed by `cons e' where `e' is any expression of the right type. We have not adopted this for two reasons:

- it requires tupels to be predefined, we can define them in the language.
- since we have lazy evaluation the domain for eg. lists would be different from the intuitive lazy list domain. It would be possible to have lists of the form `cons ⊥', ie. a list known to be on cons form but nevertheless without head and tail part, since there is a difference between ⊥ and (⊥,⊥) using our case semantics.

It is of course still possible to define the types as in SML.

When a type is introduced the new constructors, ie. the names introduced on the right, may be used to form expressions of that type. An expression `$C_k(e_1,\ldots)$' is a canonical value in the new type. A canonical value is something which yields itself as value when evaluated.

The new constructors may also be used to form patterns. The pattern matching is the only way to take apart an expression of the new type.

Eg.

```
let type rec tree =
      leaf(int) + node(tree, tree) in
let rec sumleafs (leaf(n)) = n
     || sumleafs (node(t1, t2)) =
              sumleafs(t1) + sumleafs(t2)
in ...
```

3. Compilation

This section describes the different passes of the compiler.

3.1 Parsing

The parsing is done with a ordinary recursive descent parser which builds an abstract syntax tree. This representation is then used in the compiler until the code generation.

3.2 Scope analysis

The scope analysis (or renaming) assign unique names to all identifiers in the program and checks that thee scope rules are obeyed. Giving unique names to the identifiers simplifies subsequent transformations of the program since parts of the program can now be moved around without the risk for name clashes.

Some rewriting of the syntax tree are also made, because some syntactic constructions are ambigous without further information about the symbols involved. They can only be resolved when the meaning of the identifiers are known, eg. `let s = $e_1$ in e'. If `s' is defined as a constructor in this scope then `s' is a pattern otherwise this is a normal variable binding.

The scope analysis traverses the tree and keeps a symbol table of the identifiers in the current scope, the tree is rebuilt where necessary.

3.3 Type checking

The type checking is the based on the algorithm described in [Miln78], but extended to handle the pattern matching. The typechecker (or typededucer) deduces the most general type of each subexpression and checks that they are used consistently. The deduced types are also used later on in the code generation.

The type checking eliminates the need to do runtime type checking, since at that run time we know that the program is type correct.

3.4 Pattern matching transformation

The purpose of the transformations described here is to reduce all pattern matching to case expressions containing only simple patters. A simple pattern has the form `$C(i_1\ \ldots\ i_n)$' (where i_k are variables), it's called simple because this is the basic pattern form and it's also easy to generate code for.

All constants in the predefined types (such as true and 5) are treated as constructors in their corresponding types and they are now written with parenthesis to indicate their special status (not to confuse them with variables). This means that a pattern is built up only from constructors and variables.

There are three different kind of pattern matching beside the case expression and they are all transformed into case. First, the declarator for function definitions is transformed from

```
   f p_11 ... p_1n = e_1
    ...
|| f p_m1 ... p_mn = e_m
```

to

```
f I_1 ... I_n = case tuple_n(I_1, ... I_n) in
      tuple_n(p_11, ... p_1n) : e_1
      ...
      tuple_n(p_m1, ... p_mn) : e_m
end
```

where $tuple_n$ is the n-tuple constructor.

In the the second kind of matching, the value binding, there is a declarator `p = e'. This declarator is replaced by

```
        local I = e in (
            i_1 = case I in
                      p : i_1
                end
        and i_2 = case I in
                      p : i_2
                end
        ...
        and i_n = case I in
                      p : i_n
        end)
```

where i_1, ... and i_n are the variables in the pattern p.

This declarator binds the same variables as the original one, but uses only case-matching. The reason behind this seemingly complex transformation is to preserve all properties of the declarator, it may for instance be prefixed by `rec' to make it recursive. This works for the transformed declarator as well. Eg. the expression `let rec (a,b) = f a in b' would be transformed to

```
let rec
        local I = f a in (
            a = case I in
                  (a,b) : a
                end
        and b = case I in
                  (a,b) : b
        end)
in b
```

The introduced case expressions can be viewed as selector functions to select the different parts of the expression and the recursion is still possible since bindings are "lazy".

Third, in the case of a lambda pattern (ie. `\p.e' where p is not a variable) it is transformed into `\I.let p = I in e'.

When all these transformations have been applied, there is only case pattern matching left. These must now be changed into simple patterns, ie. into expressions of the form

```
case e in
        C_1(i_11, ... i_1n_1) : e_1
        ...
        C_1(i_m1, ... i_mn_m) : e_m
        i : e_d
end
```

where the last entry may be missing if all constructors are present in the other entries. This last entry, with a single variable as the pattern, will be called the <u>default</u> entry.

The algorithm to transform complex patterns into simple patterns is as follows:

a. Sort the patterns by the outermost constructor and group those with the same constructor together.

b. Each of the groups is now expanded if it is not a single simple pattern. New variables are introduced for the subparts of the constructor and an expression with nested casing of those variables (with the corresponding parts of the original patterns as the new patterns) is used as the right hand side.

c. This process is now repeated for all the new case expressions until only simple patterns are left.

This description is simplified as it does not state how to handle failures to match. In fact every new case expression must also have a default entry to handle this. To make compilation to G-code efficient we have to introduce two new constructions used in expressions (they are only used internally by the compiler): IDEF and ODEF. IDEF stands for the same value as the default entry of the nearest enclosing case (it's only used in non-default entries), ODEF means the same value as the default entry of the second nearest enclosing case (used only in default entries).

An example:

```
case e in
     [true; x] : e1
  || [false] : e2
  || [] : e3
  || h.t : e4
end
```

Or with the uniform notation

```
case e in
    cons(true(), cons(x, nil())) : e1
 || cons(false(), nil()) : e2
 || nil() : e3
 || cons(h,t) : e4
end
```

Sort, group, introduce new variables, and expand cases

```
case e in
    cons(I1, I2) :
        case I1 in
            true() :
                case I2 in
                    cons(x, nil()) : e1
                end
            false() :
                case I2 in
                    nil() : e2
                end
            h : case I2 in
                    t : e4
                end
        end
    nil() : e3
end
```

Since the inner of the introduced cases is nonexhaustive a default entry is introduced. The degenerate case (only a default entry) is changed to eliminate the case.

```
case e in
    cons(I1, I2) :
        case I1 in
            true() :
                case I2 in
                    cons(x, nil()) : e1
                    I3 : ODEF
                end
            false() :
                case I2 in
                    nil() : e2
                    I4 : ODEF
                end
            h : e4[I2/t]
        end
    nil() : e3
end
```

($e_4[I_2/t]$ means e_4 where each free occurence of t has been changed to I_2.)

Repeat with innermost case.

```
case e in
    cons(I1, I2) :
        case I1 in
            true() :
                case I2 in
                    cons(I5, I6) :
                        case I6 in
                            nil() : e1[I5/x]
                            I7 : ODEF
                        end
                    I3 : ODEF
                end
            false() :
                case I2 in
                    nil() : e2
                    I4 : ODEF
                end
            h : e4[I2/t]
        end
    nil() : e3
end
```

As in the example, complex patterns may be transformed into quite large simple patterns. This does not seem to be much of a problem in practice, since most patterns used in ordinary programs are simple or almost simple. Other transformations can be used to keep the size down, eg. if large constants (ie. patterns with only constructors) are matched this may be transformed into ordinary equality comparison, thus saving space (but not execution time).

3.5 Other transformations

The transformations described here operates on declarators (and makes them simpler).

All let-expressions are transformed into one of two forms

`let D_1 and D_2 ... and D_n in e'
or
`let rec D_1 and D_2 ... and D_n in e'

where each D_k has the form `i = e'.

Function definitions are transformed from

`i i_1 i_2 ... i_n = e'
to
`i = i_1.i_2....i_n.e'.

Local declarators can now removed by:
`let local D_1 in D_2 in e' becomes
`let D_1 in let D_2 in e'
because of the renaming which made all identifiers unique. Having unique identifiers means that pulling D_1 out does not violate any scope rules (any names defined in D_1 are not used in e).

All let bindings can then be flattened out to the form above.

3.6 Lambda lifting

The input to the code generator should be an expression of the form

```
let rec f1=e1
    and f2=e2
    ...
in e
```

where each e_k may begin with a number of lambdas, but must otherwise contain no lambda-expressions. The only free variables in each e_k must be the predefined functions or one of the f_k:s. Furthermore the expression `e' must not contain any lambdas. The expression may still contain `let' and `let rec' -expressions.

The purpose of the lambda lifting is to lift out all lambda expression to the outermost level, which is the only place were thay may occur in the final expression. Any lambda expression not containing any free variables can simply be moved to the global level, so the main work of the lambda lifting is to remove free variables. This elimination of free variables is analogous the the abstraction used with combinators in [Turn79] and supercombinators in [Hugh82] except we do not enforce "fully lazyness" (see [Hugh82] for details). The elimination is done by passing each free variable as an argument, and adding the corresponding parameter, to the function in which it is used. A simple example of lambda lifting is shown below.

```
(a)     let x = 5
        in let f = \y.x
           in f 3

(b)     let x = 5
        in let f = \y.\x.x
           in f x 3

(c)
global definition: f = \y.\x.x
expression:          let x = 5
                     in f x 3
```

3.7 Code generation

The abstract machine and the code generation is described in detail in [John84] and will only be briefly outlined here.

The code generation is performed in two steps. First code for an abstract machine, the G-machine, is generated and then from this code, the G-code, machine code for the specific machine (in our case the VAX11) is generated.

Code is generated separately for each definition in the global definition list. The purpose of the code for a function is that when given a graph, corresponding to the left hand side of the definition, transform this to the canonical value corresponding to the right hand side. This is in contrast to most other approaches to lazy evaluation with graph reduction where the returned value may be non-canonical and the evaluation must proceed after a function has returned (cf. [Turn79]).

f x y = x + 2*y

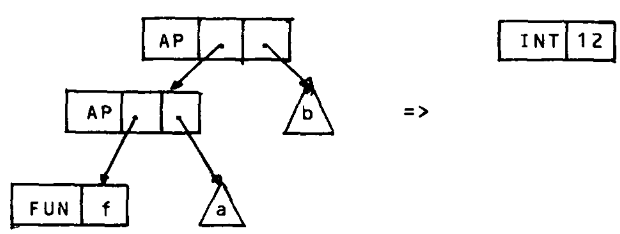

Fig 1.
the graph a has the canonical value 2
and b the value 5.

As shown in figure 1, the code for f will, when given the first graph, transform it into the second graph.

A program is executed by doing EVAL (and printing) of the expression part of the program. EVAL always searches down the spine of the graph until a function node is found, while going down pointers to the spine nodes are pushed so that they are easily accessible to the function (this is called unwinding). The function code is then invoked and when it returns (to the caller of EVAL) it has transformed a piece of the graph to canonical form.

As said above purpose of the code for the function is to produce the canonical value of the right hand side, and so there must be a code generation scheme to produce code which computes the value. But often the unevaluated expression must be used, eg. when passing parameters or as arguments to constructors, so there must also be a code generation scheme that will produce the graph for an expression. These schemes are called E and C respectively. The compilation is rule based, there are rules for how to compile the different variants of the syntax tree for each of the code generation schemes, see figure 2 for some (simplified) rules.

```
C[i]              = PUSHINT i
C[x]              = PUSH n
C[cons(x,y)]      = C[y]; C[x]; CONS
C[x y]            = C[y]; C[x]; MKAP

E[x]              = PUSH n; EVAL
E[mul x y]        = E[x]; GET; E[y]; GET;
                    MUL; MKINT
E[hd x]           = E[x]; HD; EVAL
E[x y]            = C[x y]; EVAL
E[if x then y else z] =
        E[x]; GET; JFALSE L1;
        E[y]; JUMP L2; L1: E[z]; L2:
```

Figure 2

By propagating the E scheme in the right way a lot of the produced code is for computing a value and not building a graph. The if expression, for instance, for which the value is wanted generates code that first evaluates the condition and then jumps to the code for the 'then' or 'else' branch. The code there will produce the value of the corresponding expressions.

An example:

```
        fac n = if n = 0 then 1
                else n*fac(n-1)
```

would give the code

```
fac:PUSH 1; EVAL; PUSHINT 0; GET; EQ;
    JFALSE L1; PUSHINT 1; JUMP L2;
L1: PUSH 1; EVAL; GET;
    PUSHINT 1; PUSH 2; PUSHFUN sub; MKAP;
    MKAP; PUSHFUN fac; MKAP; EVAL;
    GET; MUL; MKINT; JUMP L3;
L2: UPDATE 2; RET 1;
```

The PUSH instruction pushes a value from the pointer stack on top of the pointer stack, the number indicates the offset from the current stack pointer. EVAL causes the graph pointed to by the pointer stack top to be evaluated. PUSHBASIC pushes a constant of the value stack. PUSHINT pushes a reference to a constant on the pointer stack. GET transfers the value pointed out by the pointer stack top to the value stack. MKAP pushes a reference to application node containing two entries from the pointer stack. EQ, MUL etc. operate on the value stack. UPDATE updates a node in the graph.

3.8 Improvements

There are a number of thing that can are done to improve the code. Many of these imprivements are made by instroducing several different code generation schemes used in different contexts (eg. for return value computation, compataion of values on the value stack).

- By keeping track of what variables have already been evaluated in a certain context unnecessary EVAL instructions are avoided thereby saving time.

- When all arguments to an integer operation, for which a graph building code should be produced, are already evaluated the operation is performed in line instead. This is safe except for overflow and division by zero, but we do not handle those anyway.

- By using the type information some calculations can be simplified, eg. if two integers are compared the code for this can be emitted in line instead of calling a general value comparator.

- "Tail-call" elimination, ie. when the last thing in the code for a function is a call to another function this is done by rearranging the stack and doing a jump. This achieves tail-recursion elimination as a special case.

- When the value of an application is needed the graph corresponding to this is not built, instead the function is called the stack set up as if it would have looked if called with EVAL.

The code for the else part of the factorial function in the example above becomes:

```
PUSH 1; GET; PUSH 1; GET; PUSHBASIC 1; SUB;
MKINT; CALLFUN fac; MUL;
```

instead of the previous longer and slower code.

3.9 Machine code generation

Machine code generation from the G-code is rather straightforward since the G-code is well suited for execution on a von Neumann machine. A simple code generator would just do "macro-expansion" of the G-code; each G-instruction is replaced by a fixed piece of machine code. We use a more elaborate code generator which tries to avoid stack references and thereby memory references.

3.9.1 Node layout

In the current implementation each node has a tag field of one word followed by 0, 1, or 2 more words. The tag serves two purposes, first it indicates if the node has canonical form or not and second it tells the garbage collector how to treat the rest of the node. The tag part is not just a small number as is often the case in other implementations, but instead its an address into the machine code. By jumping to this address with different offsets different things can be done with the node, such as evaluation, garbage collection and printing. The evaluation of a node is accomplished by making a subroutine call to a certain offset from the tag, if the node is already on canonical form this address will contain a return instruction which then immediately returns, otherwise it will contain code to initiate the unwinding.

3.9.2 Memory allocation

Memory for the graph is allocated on a heap. Garbage collection is performed by copying the used part of the heap into another equally sized area.

During execution one machine register always points at the next free heap location, allocation of a cell is done by simply adding the cell size to the register. Heap overflow need not be tested before every allocation, instead it's tested once before a number of consecutive allocations. This brings down the allocation overhead. A normal (whatever that is) program allocates about 300 kbytes/sec.

The copying garbage collection used has some advantages:

- cells of varying sizes are easily handled.
- storage gets compacted at garbage collection, which is important when using virtual memory.
- the time spent in g.c. is proportional to the size of the used part of the heap, not the size of the heap as with mark-scan. This is important since the memory allocation is very large compared to the size of the used memory.

4. Current state of the compiler

The compiler as described here is not fully implemented at the moment. We have an older version of the compiler which does not do the case transformations described, but is otherwise very similar.

After completing the compiler we would like to test some other possible improvements:

- A global analysis to detect when call-by-value could be used instead of call-by-need has been proposed in [Mycr80]. This could bring down the amount of graph construction further.
- Vector nodes, ie. nodes with many parts of the same kind, could be used to store tuples in an efficient way.

5. Performance

All benchmarks below, and the code shown in the appendicies are produced with the older version of the compiler (see above).

It is difficult to do fair comparisons between different languages, but we have tried some benchmark programs with different compilers and interpreters. If not stated otherwise the same algorith has been used for all languages, this means that even the C and Pascal programs use lists if the functional program does. Of course this is not "fair" since one would not write the an imperative program this way, but if one starts using different versions for different languages this makes the comparison even more difficult. Nevertheless, in two examples, 8queen and kwic, the programs in Pascal and C have been written in an imperative style.

All execution times given i table 1 are in seconds of CPU time (garbage collection included).

Language processors:

LML The lazy ML compiler described in this paper. Lazy evaluation.

ML The ML/LCF system from Edinburgh, translates ML to LISP and interprets the LISP. Strict evaluation.

ML-C The compiling ML system by Luca Cardelli, translates to VAX-code. Strict evaluation.

SASL Turners SASL, translates SASL into SECD code and interprets it. Lazy evaluation.

LISP Interpreted Franz Lisp on UNIX. Strict evaluation.

Table 1.

	LML	ML	ML-C	SASL	LISP	Liszt	Pascal	C
8queen	3.2	170	9.0	170	48	5.2	1.0	0.4
fib20	0.83	46	0.5	31	21	1.1	0.92	0.46
primes	0.5	29	1.2	20	7.8	1.1	-	-
insort	0.37	15	1.0	12	6.4	0.8	-	-
tak	10	309	16	-	76	3.0	2.6	1.8
kwic	39	-	-	-	-	-	9.7	-

Table 2.

	allocated memory (K)	max used (K)	heap size (K)	GC time (%)
fib20	45	0.3	20	4
prime	88	1.7	10	5
insort	59	4	10	15
8queen	687	6.4	50	6

Liszt Compiled Franz Lisp on UNIX. Strict evaluation.

Pascal The VAX/UNIX Pascal compiler pc. Strict evaluation.

C The VAX/UNIX C compiler cc. Strict evaluation.

Programs:

8queen Counting the number of solutions to the 8 queen problem (actually 7 queens are used to limit the execution time). The Pascal and C versions of this program are coded differently (coded in a non-functional style).

fib20 Computation of the 20th Fibonacci number.

primes The first 300 primes using Erathostene's sieve.

insort Insertion sort of 100 elements in a list (repeated 10 times).

tak The tak function with arguments 18,12,6.

kwic Keyword in context, all significant rotations of a number of sentences sorted. The Pascal program was written in the usual Pascal style with arrays.

We have also performed some measurements of memory consumption (of the LML programs) which are presented in table 2.

allocated memory: is total amount of memory allocated by the program.

max used: is maximum amount of heap memory in use at any time, ie. the part of the heap not containg garbage.

heap size: is the size of the heap (twice this amount used because of the way the garbage collector works.)

GC time: is the time spent in garbage collection.

6. Acknowledgements

This work was supported by the Swedish Board for Technical Development (STU). The LML compiler has been developed in cooperation with Thomas Johnsson. I must also thank the whole of the Programming Methodology Group for many helpful comments, suggestions and ideas.

References

[Burs80] R. M. Burstall, D. B. McQueen, and D. T. Sannella, "Hope: An Experimental Applicative Language", pp. 136-143 in Proceedings of the 1980 LISP Conference, Stanford, CA (August 1980).

[Card84] L. Cardelli, "ML under UNIX", Polymorphism: The ML/LCF/Hope Newsletter, Vol. 1 no. 3 (January 1984).

[Fair82] J. Fairbairn, "Ponder and its Type System", Technical Report No. 31, University of Cambridge, Computer Laboratory (November 1982).

[Gord79] M. Gordon, R. Milner, and C. Wadsworth, "Edinburgh LCF", Lecture Notes in Computer Science, Vol. 78, Springer Verlag (1979).

[Huda84] P. Hudak and D. Kranz, "A Combinator-based Compiler for a Functional Language", pp. 122-132 in Proc. 11th ACM Symp. on Principles of Programming Languages (1984).

[Hugh82] J. Hughes, "Super Combinators: A New Implementation Method for Applicative Languages", pp. 1-10 in Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming, Pittsburgh (1982).

[John84] T. Johnsson, "Efficient Compilation of Lazy Evaluation" in Proceedings of the 1984 Symposium on Compiler Construction, Montreal (1984).

[Jone82] N.D. Jones and S.S. Muchnick, "A Fixed-Program Machine for Combinator Expression Evaluation", pp. 11-20 in Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming, Pittsburgh (1982).

[Miln78] R. Milner, "A Theory of Type Polymorphism in Programming", Journal of Computer and System Sciences, Vol. 17 no. 3 pp. 348-375 (1978).

[Miln84] R. Milner, "Standard ML Proposal", Polymorphism: The ML/LCF/Hope Newsletter, Vol. 1 no. 3 (January 1984).

[Mycr80] A. Mycroft, "The Theory and Practice of Transforming Call-by-Need into Call-by-Value", pp. 269-281 in Proc. 4th Int. Symp. on Programming, Lecture Notes in Computer Science, Vol. 83, Springer Verlag, Paris (April 1980).

[Turn79] D. A. Turner, "New Implementation Techniques for Applicative Languages", Software - Practice and Experience, Vol. 9 pp. 31-49 (1979).

Appendix A, Terminology

Program parts will (almost) always be enclosed in ` '. Certain names inside stand for special things:

i, i_k are variables (ie. identifiers that are not constructors).

C, C_k are constructors.

p, p_k are patterns (ie. "expressions" built up from constructors and variables).

I, I_k are **new** unique variables, ie. they are compiler generated variables distinct from all other identifiers in the program.

D, D_k are declarators, ie. the definition part of a let expression.

e, e_k are expressions.

Appendix B, Declarator syntax

Declarators can be built up in several ways:

`D_1 and D_2' the bindings of D_1 and D_2 are made simultaneously.

`rec D' makes the bindings in D recursive, ie. the variables bound in D may be used in the right hand sides of D.

`local D_1 in D_2' makes D_1 available in D_2, but not outside D_2.

`p = e' binds the variables in the pattern p to the corresponding parts of e.

$i p_{11} \ldots p_{1n} = e_1$
...
$|| \; i p_{m1} \ldots p_{mn} = e_m$
defines the function i by a number of equations.

Appendix C, Some benchmark programs

```
let nsoln nq =
        letrec ok [] = true
            || ok (x.l) =
                letrec  safe x d [] = true
                    ||  safe x d (q.l) = x ~= q & x ~= q+d &
                                         x ~= q-d & safe x (d+1) l
                in      safe x 1 l
        in
        letrec gen 0 = [[]]
            || gen n = concmap (\\b.(filter ok (map (\\q.q.b) (count 1 nq)))) (gen (n-1))
        in      length (gen nq)
in
        nsoln (stoi (hd argv))
```

Count the number of solutions of the n-queen problem (placing n queens on a n*n board).
Used functions:
map - apply a function to all elements of a list.
concmap - as map but concatenate the instead of cons.
length - gives the length of a list.
count - generate a list of consecutive numbers between two limits.
stoi - stringto integer.
argv - list of arguments to the program.

```
letrec fib n =
       if n < 2 then 1
       else fib (n - 1) + fib (n - 2)
in
       fib 20
```

The 20:th fibonacci number.

```
letrec  tak x y z =
              if ~(y < x) then z
              else    tak (tak (x-1) y z)
                          (tak (y-1) z x)
                          (tak (z-1) x y)
in
       tak 18 12 6
```

The tak function.

```
let mod x y = x - (x/y*y) in
let rec filter p l = if null l then []
                  else
                     if mod (hd l) p = 0 then
                            filter p (tl l)
                     else    hd l . filter p (tl l)
and     count a b = if a > b then []
                 else a . count (a+1) b
in
letrec sieve l = if null l then nil
              else hd l . sieve (filter (hd l) (tl l))
in
let primesto n = sieve (count 2 n)
in
primesto 300
```

Generating primes up to a limit.

Appendix D, G-code and VAX-code

This is the essential part of the run time machinery of all programs:

```
         jmp     APPLY_eval      this is were eval
         jmp     ....              of an APPLY jumps
         jmp     ....
APPLY:   movl    (%ep),r0        get pointer to apply
         movl    8(r0),-(%ep)    push argument part
         movl    4(r0),r0        get function part
         movl    r0,-(%ep)       and push it
         jmp     *(r0)           jump via UNWIND tag
APPLY_eval:
         movl    r0,-(%ep)       push node pointer
         movl    %ep,-(sp)       save current ep
         jbr     APPLY           enter unwind state
```

Function definition:

```
        filter p L =
                if null L then
                        []
                else if p (hd L) then
                        (hd L) . filter p (tl L)
                else
                        filter p (tl L)
```

G-machine code:

```
PUSH 3; EVAL; NULL; JFALSE 105;
        RET_NIL;
LABEL 105;
PUSH 3; HD; PUSH 2; MKAP; EVAL; GET; JFALSE 106;
        PUSH 3; TL; PUSH 2; PUSHFUN filter; MKAP; MKAP;
        PUSH 4; HD; CONS; UPDATE 5; RET 4;
LABEL 106;
        PUSH 3; TL; MOVE 4; JFUN filter.
```

Note: %ep is an alias for r10, used as stack pointer for the pointer stack.
%hp is an alias for r11, used as heap pointer.

Vax assebler code:

```
# Code for function _filter
        .globl  C_filter
C_filter:
CLt100: .long   FLt100,1f
1:      .asciz  "_filter"
2:      movl    $CLt100,r0
        rsb
        rsb;    .byte 0,0,0,0,0
        jmp     funprinterr
        jmp     2b
        .globl  F_filter
F_filter:                          Entry point for call via EVAL
FLt100:
        subl3   %ep,(sp),r0
        cmpl    r0,$16
        bgeq    1f
        jmp     return
1:
DLt100:
        .globl  D_filter
D_filter:                          Entry point for "direct" call

        movl    12(%ep),r0         PUSH 3
        movl    (r0),r1            EVAL
        jsb     eval(r1)
        cmpl    $CONS,(r0)         NULL; JFALSE L105
        jeql    L105

        movl    (sp)+,%ep          RET_NIL
        movl    (%ep)+,r0
        movl    $F_nil,(r0)
        rsb
L105:                              LABEL L105
        cmpl    %hp,ehp            Enough heap left?
        bleq    1f
        jsb     GARBMIN            no: collect garbage
1:
        movl    12(%ep),r2         PUSH 3
        movl    4(r0),r2           HD
        movl    $APPLY,(%hp)+      PUSH 2; MKAP
        movl    4(%ep),(%hp)+
        movl    r2,(%hp)+
        moval   -12(%hp),r0
        movl    (r0),r1            EVAL
        jsb     eval(r1)
        tstl    4(r0)              JFALSE L106
        jeql    L106

        cmpl    %hp,ehp
        bleq    1f
        jsb     GARBMIN
1:

        movl    12(%ep),r2         PUSH 3
        movl    4(r2),r2           TL
        movl    $APPLY,(%hp)+      PUSH 2; PUSHFUN filter; MKAP
        movl    $C_filter,(%hp)+
        movl    4(%ep),(hp)+
        movl    $APPLY,(%hp)+      MKAP
        moval   -16(%hp),(%hp)+
        movl    r2,(%hp)+
        movl    12(%ep),r2         PUSH
        movl    4(r2),r2           HD

        movl    (sp)+,r1           CONS; UPDATE 5; RET 4
        movl    (r1),r0
        movl    $CONS,(r0)
        movl    r2,4(r0)
        moval   -12(hp),8(r0)
        movl    r1,%ep
        rsb
L106:
        movl    12(%ep),r2         PUSH 3
        movl    8(r2),12(%ep)      TL; MOVE 4
        jmp     D filter           JFUN filter
```