An Implementation of Standard ML in Poly

David C.J. Matthews

1 Introduction

This document describes an implementation of Standard ML [1] written in Poly [2]. This is a complete implementation of the language including the modules system.

The original aim of the implementation was largely as an experiment in using Poly for implementing reasonably sized programs, and only secondarily to produce a working ML system. In fact, though, the Poly/ML compiler has been very successful and is in use for a number of serious projects.

This description is mostly aimed at those interested in the details of the implementation or planning to implement their own compiler for ML or a similar language. It includes only the parts of the system specific to ML. The low-level system is described in a separate report.

2 Overview

The compiler can be divided fairly neatly into two main parts, the front-end which translates the ML source into a machine- and language-independent code-tree, and the back-end which optimises this code and translates it into machine-code. The back-end is shared between the Poly and ML compilers.

The ML front-end takes three passes to get to the intermediate code. The first pass parses the source text and generates the parse tree. It deals with infix precedences and type variables, but does no other identifier matching, in particular it does not distinguish between variables and constructors. The second pass matches up identifiers to their declarations and does the type checking. The third pass generates the intermediate code. It uses type information provided by the previous pass to resolve overloading and to do the completeness checking for patterns.

The back-end consists of an optimiser which takes the code tree from the front end and generates an optimised copy for the code-generator. It recognises tail-recursion and procedures for which a full closure need not be made. The other optimisations, such as inline code expansion and handling type-vectors, are intended for code from the Poly compiler. The machine-code generator compiles the code into vectors in store. Currently there are code-generators for the Vax, for the Sun2 and Sun3 (68010 and 68020) and a version which generates an interpretable code. Other code-generators may be written in the future. This part of the system is described in a separate document.

The compiler is divided into several modules which implement different data structures and operations on them. When the compiler modules are composed the result is a function with a Poly type equivalent to:

```
(unit->string) * (string->unit) * environ -> unit -> unit
```

that is a function of three arguments which returns a function. The first argument returns the next character from the input stream and the second is used to write out error messages. `environ` is a tuple of functions (in Poly a *type*) which enter and look up the different classes of values (types, values, infix status, exceptions, structures and signatures). It is used both to look up names in the outer scope during compilation and to enter the names when the code is executed. The result of compilation is a function which, when called, executes the code and "side-effects" the environment. In normal use this function is executed immediately the code is compiled.

The compiler compiles only one top-level phrase and returns a function which will execute it. If it finds any errors in the phrase, either in the syntax or perhaps a type mismatch, it raises an exception.

The basic compiler module is not used directly on files or from the terminal. Instead a "shell" procedure is wrapped round it which handles input from and output to the terminal. A separate procedure is used for compiling files. If there is an exception either from compilation errors or through the compiled code raising an exception the procedure for compiling files exits immediately and does not try to compile any more top-level phrases, whereas the interactive procedure prints out the name of the exception and then continues.

Other top-level procedures are possible, for example the *make* function uses a special environment to compile a program consisting of several modules by only compiling the modules that have changed since the last compilation. It works by having a special look-up function which checks to see whether a module is up to date and recursively calls the compiler.

3 The Parser

The parser reads ML source text and generates the parse tree. It is structured as a lexical analyser module *lexan* and parser modules *parse_dec,* the main expression and declaration parser and *parse_type* for types.

3.1 The Lexical Analyser

The lexical analyser module is fairly conventional. The input stream to the compiler is used by the lexical analyser to return the next lexical symbol. The main point to note is that names such as `A.b.+` which are structure selections, are returned as single entities and are separated out later on. Reserved words are looked up at this point. Symbols such as `=, * and ->` which can be treated as identifiers in some circumstances, are passed back as keywords.

The lexical analyser also contains the error-message routine. Error messages are produced using an Oppen-style [4] pretty printer. This is particularly useful because the parse-tree can be unparsed to give some context information. Generating an error message also sets a flag to abort the compilation at the end of the current pass.

3.2 Parsing Expressions and Declarations

The parser is a fairly conventional recursive-descent parser. This was chosen in preference to using a parser-generator such as YACC partly because of the problems of handling declarations of operator precedence but also because a parser-generator for Poly was not available.

The parser converts infix declarations into their equivalent prefix form and to do this it has access to the environment of infix precedences. Operator precedences in ML are under the control of the user and can be changed by local declarations. For example

```
let infix 8 + in 1+2*3 end
```

returns the value 9 because the `+` has been given a higher precedence than `*` for that expression. The parser must have access to global precedence declarations and know about the scope of declarations. This is quite easy with a recursive-descent parser since the parsing structure is represented in the recursion so all that is required is to pass the current scope table as an extra parameter to the parsing procedures.

Parsing an expression involving infixes is done by a recursive method. A simple way of doing this would be to have a procedure which recurses once for each precedence level and handles all the operators at that level. This is very inefficient since all expressions will require at least 10 procedure calls in order to reach the atomic expressions such as identifiers and constants. A much more efficient method is to recurse only when a more binding operator follows a less binding one. For example, the expression `a*b+c=d` can be parsed without recursing because each operator is followed by one with a lower precedence. The reverse-Polish representation is simply `a b * c + d =`. A slightly more complex expression `a*b+c=d+e` requires one recursion to handle the right-hand operand of the equality but no more. Right-associative operators can be handled by this scheme simply by recursing if the operator is right-associative and followed by an operator of the same precedence[1].

[1]This is slightly more difficult than it might be because in ML two different operators of the same precedence always associate to the right, even if they are both declared as right-associative.

Since the parser has to handle infix declarations in order to parse an expression it is sensible if it is also responsible for passing top-level infix declarations back to the global environment. It is possible for top-level declarations involving infixes to fail if several declarations are combined in a "local-declaration" so top-level infix declarations are first put into a temporary environment until the compilation and execution are complete when they can be put into the real global environment.

The parser also matches up explicit type variables since these have fairly simple scope rules, but it does not attempt to match up other identifiers with their declarations because of mutually recursive declarations.

The classes of identifiers accepted depends on the context. In particular `=` can be used in an expression but cannot be declared. The reserved type constructors `*` and `->` can be used as ordinary identifiers.

The expression and declaration parser calls procedures in the "parsetree" module to make the parse tree. The parse tree is returned as the result.

Parsing a pattern is complicated by the syntax of a layered pattern.

> pat ::=
> apat
> pat : ty
> <<**op**>>**var**<<**:ty**>> **as pat**

This is ambiguous because a variable can also appear in an atomic pattern (apat). The solution to this is to change the syntax to

> pat ::=
> apat
> pat : ty
> pat **as pat**

and parse it that way. The check that the first pattern is just a variable is then done on the parse tree.

Constructors and variables are not distinguished in the parser and this makes parsing clausal functions, in particular, rather more difficult. The declarations

> **infix** %**;**
> **fun** $(a :: b)\ \%\ c =$ **...**
> **fun** $(a\%b)\ c\ d =$ **...**

both declare % as a function. To find the name being declared it is necessary to unravel the infix application. The simple solution is to parse each clause as though it were a single pattern and the syntax were

> **fun pat = exp** — __ — **pat = exp**

Since the parser makes no distinction between constructors and variables this actually parses the same syntax as the full definition. The business of extracting the name of the function is then left to the second pass.

3.3 Parsing Types

Type expressions are parsed a separate module since the syntax is completely separate from that of expressions and declarations. The main concern here is that type constructor names must not include `=`, `*` or `->`. The result of parsing a type expression is a type tree generated by the *typetree* module.

4 The Parse Tree

The parse tree, or more accurately the abstract syntax tree, is at the heart of the compilation. The parsing routines call construction functions to make the tree and the result of parsing a top-level declaration is this tree.

4.1 Internal Structure

Entries in the parse-tree can be divided into classes: types, declarations, patterns and expressions, for example. The syntax of ML allows them to be combined in limited ways, a declaration appears in an expression only as part of a **let** declaration, so it would be possible to represent the parse-tree of a declaration with a different parse-tree type than an expression. In practice it is convenient to distinguish only between the parse-tree for types, *typetree* the parse-tree for structures and signatures *structs* and that for everything else, *parsetree*. Splitting this way can be made without circular dependencies since a type cannot contain an expression or a structure and an expression cannot contain a structure. All the type-checking code forms part of *typetree* while most of the intermediate code generating procedures are in *parsetree*.

4.2 Structure

The tree is close enough to the original source text that it can be unparsed into a recognisable form to generate error messages. Each parse-tree type is a union of records with a separate record type for each abstract syntax form.

5 The Second Pass

Each of the parse-tree modules contains procedures used in the second pass. The second pass is responsible for matching identifiers to declarations and doing the type checking. The core of the second pass is a procedure in *parsetree* which recurses down the tree and returns its type. The environment is updated by declarations so a side-effect of the second pass is the declarations. Identifiers are represented as variables which are initialised to an "undefined" value when they are created by the parser and then assigned a pointer to the actual declared variable.

5.1 Unification

Type-checking in ML is done by unifying type expressions. A type is represented as an expression involving terms which are either constants, such as `int`, operators such as `->` or `list` and type-variables such as α, written in ML as `'a`. To match two types the type expressions have to be combined so that a common type can be made from them. As a side-effect some type-variables may acquire values. For example, the *append* function has a type `'a list * 'a list -> 'a list`. This means that it can be applied to two lists and produces a list but that the elements of the list are the same in each case. Suppose this is to be applied to a the value `(nil, [1])` , a pair of values with the first being the nil value and the second a list containing a single integer. This pair has type `'b list * int list` where `'b` is a different type variable to `'a`. When *append* is applied to the pair the type expressions `'a list * 'a list` and `'b list * int list` are unified. The operators $*$ and `list` must match exactly but the type variables will match anything. The first occurrence of `'a` matches `'b` so `'a` and `'b` become the same type variable. The second occurrence of `'a` matches `int` so this combined type variable now takes on the value `int`. This is a constant which will only match other occurrences of `int`. The result of this is that `'a`, `'b` have both acquired the type `int` so the result of the application of *append* has type `int list`.

When types are represented as trees with the operators as nodes and the constants as leaves the type-variables can be represented using updatable references (variables). Unification of two expressions is then simply a matter of checking that constants and operators are the same in both expressions and where type-variables occur in one expression assigning the corresponding tree in the other expression into it. When two type-variables occur at the same point in both expressions either type-variable may be assigned to the other.

Unfortunately there are a number of complications which have to be dealt with before this scheme is satisfactory. Any number of type-variables can be unified together to make a single type-variable. In the example above `'a` and `'b` were unified and we have said that this results in one being assigned to the other. Since there is no difference between them suppose this resulted in `'a` pointing to `'b`. A little while later `'a` is unified with `int` which would result in `'a` now pointing to `int`. However `'b` must also point to `int` so rather than overwrite the pointer in `'a` we chain down the type-variables and point only the last one at `int`. This is shown in figure 5.1. The result of this is that the occurrence of a type-variable in a tree does not immediately mean that it will match anything. It is only after searching to the end of the chain of assignments and finding that the final entry is unset that we can conclude that it really is a type-variable. If it assigned to a constant or an expression then that becomes the value to be unified. As an optimisation all the type-variables in a chain can be pointed directly at the final entry to save following the chain repeatedly.

The next complication to be overcome is what is known as the “occurs” check. Suppose we are type-checking the following expression.

Initially.
'a ⟶ <unset>
'b ⟶ <unset>
After first unification.
'a ⟶ 'b ⟶ <unset>
After second unification.
'a ⟶ 'b ⟶ int

Table 1: Unification of 'a, 'b and int

```
fn x => x ::  x
```

This is a lamba-expression with bound variable `x` and whose body is the list constructed from using `x` as both the head and the tail. This is of course a type-error because `x` would have to have both a list type, for the tail, and the type of the element of that list, for the head. Detecting the error is quite difficult. Bound variables are given a type which is a new type variable, call it `'x`. If the rules described above are followed then this would result in `'x` pointing at `'x list`, in other words a circular structure. To avoid this we must check when a type variable is assigned to an expression that the expression does not contain the type variable anywhere.

A further complication has to do with what are called *generic* and *non-generic* types. Lambda-bound variables, such as `x` in the previous example, have the same type everywhere they occur, that is `'x` was used for the type of both occurrences of `x`. On the other hand a function such as *append* can have a different type at different occurrences. It may be applied to lists of integers in one place and lists of strings in another.

This is done by copying the type of the function each time it is used and making new type-variables for each of the old ones, being careful to share the type-variables correctly. For example each instance of the type of *append* must look like `'a list * 'a list -> 'a list` and not `'a list * 'b list -> 'c list` which would result if a new type-variable were made every time a type-variable was encountered. The *copy_type* procedure which does the copying keeps a list of each type-variable in the original structure and the corresponding new type-variable and only makes a new one if it finds one which is not in the list.

A lambda-bound variable has the same type everywhere but a function such as *append* can have different types in different places. How do we know whether to copy a type expression or not? A simple mechanism would be for type-variables belonging to lambda-bound variables to be marked as "not-copyable" so that *copy_type* would put them in the result rather than copying them. Otherwise they would behave like normal type-variables. Whenever an identifier is found its type is submitted to *copy_type* to get its effective type and in this way lambda-bound variables would have the same type at each occurrence, but generic variables would have their type variables copied. When the function has been compiled the type of the result would have to be processed to convert all the "not-copyable" type-variables into "copyable" ones so that function is generic.

This works only provided lambda-bound variables are not unified with other type-variables. Unification of a "not-copyable" type-variable with a "copyable" one can be arranged so that the copyable one points to the non-copyable one, the result being a non-copyable type-variable, but suppose we have to unify two non-copyable type-variables. For example

```
fn a => fn x => a
fn a => let val f = fn x => a in f end
```

These two lambda-expressions are both the same, they take a value and return a function which returns that value. The type should be `'a -> 'b -> 'a`. The only difference is that in the second case the inner lambda-expression is bound to an identifier before being returned. Suppose the type-variables corresponding to `a` and `x` are `'a` and `'x` then the type of `fn x => a` would be `'x -> 'a`. However what is the type of `f`? Our simple-minded scheme would say that at the end of compiling `fn x => a` the type-variables are made copyable so that `f` would have a type `'p -> 'q` where `'p` and `'q` are new copyable type-variables. This is wrong because while `'x` should be made copyable `'a` which is still in the scope of the outer lambda, should not. The correct type for `f` is `'p -> 'a`. To get this correct type-variables are given a level number and the procedure to make the type-variable copyable only makes them copyable if they are above a certain level. When type-variables with different levels are unified the higher number is set to point to the lower. In this way at the end of compiling `fn x => a` the type will be `'x -> 'a` and `'x` with level 2, say, will be made copyable, but `'a` with level 1 will not. It will only become copyable at the end of the outer lambda-expression when level 1 type-variables are made copyable.

5.2 Constructors

Type constructors are processed in the second pass and value constructors and equality functions are made for datatype declarations. Datatypes with only one constructor do not require a union to be made so the value constructor is a null operation. If there is more than one constructor then a tagged union is made. Tagging can be avoided if the values in the union are disjoint. Luca Cardelli [3] describes a scheme where values which are addresses can be distinguished from those which are not, allowing an efficient implementation of datatypes like lists and trees. At the moment the only optimisation is that datatypes consisting only of constants, essentially an enumerated type, are treated specially. The built-in constructor "list" is a special case.

6 The Third Pass

The third pass generates the intermediate code tree from the parse tree. It does some checking which cannot be done in the second pass. The result of the third pass is code which produces a vector containing the values and exceptions declared in the top-level declaration.

6.1 Completeness and Irredundancy Checking

Checking for the completeness and irredundancy in a set of patterns is done during the third pass largely because knowing that a match is complete allows somewhat better code to be generated. The method used is due to Alan Mycroft and was used in the Edinburgh ML compiler.

The algorithm can be thought of as a process of covering an area, the *floor* with *tiles* of different shapes and sizes. Each pattern in a match corresponds to a different tile and together the tiles must cover the floor if the match is to be exhaustive. The irredundancy check is represented by saying that each new tile layed must cover some of the floor which was previously exposed.

The basic kinds of patterns to be handled are wild cards, constructors and pairs. Variables are treated as wild cards since they match the whole of a tile. Integer and string constants are a special kind of constructor which unlike other constructors cannot make a complete match by enumerating all the possible values. Tuples with more than two elements are represented by pairs of a value and a pair. Labelled records are treated like tuples.

The completeness checking procedure takes an area of floor and a pattern (a tile) and returns the area of floor left after the tile has been layed. Initially the floor is *all_there,* if the match is complete the result should be *all_gone.* At intermediate stages some of the floor area may have to be blown up so that part of it can be removed. Suppose we had the following set of patterns in a match.

a :: nil
nil
x :: y

Initially the floor is *all_there* which will be represented by "...". The first pattern to be matched is `op :: (a, nil)` in prefix notation, that is the constructor `::` applied to a pair. This does not match the whole of "..." so it must be expanded into the possible constructors. The match becomes

:: (...)
nil (...)

In other words the two possible constructors, each applied to *all_there.* For convenience all constructors are treated as having one argument, constants such as `nil` in patterns are treated as though they were `nil (_)`. We can now attempt to remove the `::` entry from the list by matching its argument. The argument in the pattern is a tuple so the argument in the floor must be similarly expanded.

:: (..., ...)
nil (...)

This applies only to the particular sub-tile being matched, so the argument to `nil` is not affected.

Pairs are matched using the identity

$$(x \times y) \setminus (a \times b) = (x \setminus a) \times y \cup x \times (y \setminus b).$$

The two special cases are detected. Where $x \setminus a$ (or $y \setminus b$) results in a complete match the $(x \setminus a, y)$ (or $(x, y \setminus b)$) term is removed. The other special case is when $x \setminus a = x$ ($y \setminus b = y$) when the result is the original tile (x, y). This can result in an explosion of areas of floor since two terms are produced for each pattern, and in particular the sub-term $(x \setminus a) \times (y \setminus b)$ is present in both the resulting terms, but in practice most patterns fit into one or other of the special cases so the tiles do not grow too complex. This identity is also quite easy to program since the $x \setminus a$ expressions are just recursive calls. An alternative, such as

$$(x \times y) \setminus (a \times b) = (x \setminus a) \times b \cup x \times (y \setminus b),$$

which would avoid the common sub-term, is actually more difficult to program. This is because it requires b, which represents a tile, to instead represent an area of floor.

In this example the two terms produced by matching `(a,nil)` are `(∅, ...)`, since the identifier `a` matches everything, and `(..., ::(...))` after the second `...` has been expanded into `nil(...)` and `::(...)` . The first of these reduces to $\emptyset$ and is removed, leaving the resulting tile as

:: (..., :: (...))
nil (...)

The next pattern, `nil`, removes the second entry, and the final pattern removes the first. The final result is an empty set and the match has been shown to be exhaustive.

6.2 Matches

Matches are code-generated by processing each pattern twice, first to generate a test to see if the value matches the pattern, and then to generate code to extract the values of variables. The reason for not doing these both together is that the code tree being generated can return only a single result. It would be possible to combine the passes and raise an exception if a match did not work, but on the whole this does not seem worthwhile. A better solution is to generate more efficient testing code by making a tree of patterns so that each constructor is tested only once and the leaves would then be the expressions corresponding to each pattern.

6.3 Exceptions

Exceptions in ML have a scope just like other values, but by their nature, they may be declared, be raised but then propagate out of the scope of their declaration. They cannot then be caught except by the wild-card handler. It is possible to construct pathological examples involving recursive declarations and functions declared in the scope of an exception which then pass out of its scope which essentially mean that an exception must be treated as a run-time value.

An exception declaration generates-code to create a unique value by creating a one-word value on the heap. A handler for an exception then has to do a simple pointer equality test to see if an exception matches. This works for ML but needs a slight modification so that it is compatible with code from the Poly compiler. Exceptions in Poly are represented by a string and do not have parameters, so raising an ML exception involves saving the exception identifier and the parameter away in global variables, raising a Poly exception with the exception name as a string, and then taking out the identifier and parameter when the exception is handled. The string is useful for identifying uncaught exceptions at the outer level.

6.4 Equality Testing

The `=` function, along with `print` and `makestring`, is special in that it is infinitely overloaded. Each new datatype that is declared can give it a new meaning. `print` and `makestring` are defined on all types, but the result of printing a function is just “fun”. `=` on the other hand is not defined on functions or on type variables.

A simple way of handling equality would be to have a procedure which compared any two structures by using the contents of structures alone. This assumes that there is enough information in the values to distinguish addresses from data which must be there for the garbage-collector. It must also be able to distinguish references from other data because references are only regarded as equal if they are actually the same reference. This procedure could be used to compare values of any type without the type information.

It is however necessary to check at compile-time whether equality testing is allowed on values of the particular type. For example,

```
fun f x = (x = nil)
fun f p = (p = op +)
```

are both illegal, in this first case because `nil` has type `'a list` and so contains a free type variable, and the second because it involves comparison with a function. The first case would be legal if a type constraint, such as `(x: int list)` , had been used.

Equality testing is allowed for `X list` provided that it is allowed for `X`, because the definition of `list` as

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

contains no free type variables or functions. It would be possible to check this on every occurrence of `list` in a comparison, but it is more sensible to check this once when the datatype is declared. Comparing values of a type construction such as `int list` is allowed if it is allowed for each type constructor in it. Actually seeing if comparison is allowed for a type constructor is not trivial if there are mutually recursive type declarations where comparison may be allowed for one type only if it is allowed for another.

The result of this checking is a flag in each type constructor showing whether values of the type can be compared. It is not difficult to extend this slightly so that the flag is actually the code for a function which does the comparison with a special value which represents "comparison-not-allowed". For nullary type-constructors like `int` this is quite easy, but it is slightly more difficult for constructors like `int` which are polymorphic. How values of type `X list` are compared depends on the comparison function for `X`. The code actually generated for the type constructor `list` is roughly equivalent to

```
fun eq_list eq_X=
               fn (a::x, b::y) = eq_X(a,b) cand eq_list(x,y) |
                  (nil, nil)   = true |
                  _            = false
```

7 Modules

The definition of modules in ML is still under development so this can only be a description of the design as it is at the moment. The current design is based on the *structure,* a group of declarations packaged together; a *signature,* which is the "type" of a structure; and the *functor* which is a structure that can be parameterised by other structures. None of these can appear inside expressions or declarations so the module mechanism can be largely added on top of the core language system. The main exception is that names can be qualified by the name of the structure which contains the declaration and qualified names can appear anywhere. There is also an "open" declaration that can be used to make the names in a structure available without having to qualify each of them with the structure name. This can be used anywhere a declaration can appear.

8 Abstract Machine

The code generated by the third pass is a code tree which contains instructions and data but the flow of control is given by the tree itself. Using a tree makes global optimisation and inline code insertion somewhat easier than with segments of code, though the representation does take rather more store.

The code does not contain many of those instructions for manipulating data, such as adding together two integers, which are normally considered as part of an instruction set. Instead these operations are treated as calls to functions in a standard vector. A simple code-generator would generate these as ordinary function calls but the code-generators for the Vax and 68020 both recognise special entries in the vector and generate the appropriate instruction in-line.

9 Machine Code Generation

The final phase of compilation is to generate the machine code. The only code-generator available at the moment is for the VAX but other code-generators may be written in the future. They are all likely to be fairly similar. On the whole the code-generator is similar to that for a language like Pascal, but there are a few differences. The main difference is that variables (references) are not held on the stack. This means that a value on the stack cannot change and so can be cached in a register without too much difficulty.

The VAX machine code is based on bytes and an instruction can start at any byte in the word and occupy almost any number of bytes depending on the arguments. The code is generated into a vector of bytes which can expand as necessary and is then copied into a code-vector of the correct size at the end of code-generating the function.

9.1 Linkage Conventions and Register Usage

The VAX provides instructions for procedure call and return which set up and remove stack frames and save and restore registers. Apart from the program counter and stack pointer registers which have special uses, two registers are designated as frame pointer and argument pointers and are used by the standard call and return instructions. The standard linkage conventions are fairly powerful but do not provide any support for a static link to get access to free variables. They are also fairly slow which is a serious problem for a functional language where function call must be fast. For these reasons a very light-weight linkage convention is used which uses the subroutine call and return instructions.

The conventions differ slightly according to whether the function is being called with a static link or as a full closure. In the case of static link call register 1 contains the static link and the subroutine jump is to the start of the code. For a closure call the register contains the address of the closure and an indirect jump is made using the address in the first word of the closure as the destination. The arguments to the function are passed on the stack so after the call the stack contains the return address, pushed by the subroutine call, and the arguments. No other values are explicitly put on the stack. Register 1, containing either the static link or the closure address, is marked on the pseudo-stack and will be pushed onto the real stack if the register is required, in particular for another function call, in same way as any other register. Small functions which do not do any function calls themselves, never need to push register 1 and any references to variables in the closure can be done by indirecting through the register.

The code-generator keeps a note of the number of items on the stack and addresses all the values on it using the stack pointer register. This avoids the need for a frame pointer register but makes reading the assembly code listing somewhat difficult because the offsets of local values and arguments change according to the number of items currently on the stack.

The only use made of the frame pointer register in the code is in dealing with exceptions. In the standard VAX convention the word at the base of the frame is the address of a procedure for handling exceptions and it is initialised to zero by the standard procedure call instruction. The Poly/ML convention treats this word, the exception location, somewhat differently. When a handler is set up two words are put onto the stack, the old value of the exception location and the address of the new handler. The exception location is then set to the value of the stack pointer, in other words to the address of these two words. If an exception happens the stack pointer is reset to the value in the handler and these two words are popped from the stack, one back into the exception location to reinstate the previous handler, and the other into the program counter to execute the handler. If control reaches the end of the scope of an exception handler without the handler itself being executed the previous handler is reinstated and the handler address is removed from the stack.

References

[1] Robin Milner, *The Standard ML Core Language*, Edinburgh University.

[2] David C.J. Matthews, *The Poly Manual*, SIGPLAN Notices, September 1985.

[3] Luca Cardelli, *Compiling a Functional Language*, 1984 ACM Symposium on Lisp and Functional Programming.

[4] D. C. Oppen, *Prettyprinting*, ACM ToPLAS Vol. 2 No. 4 Oct 1980.