The Poly and ML System: Abstract Machine and Realisation

David C.J. Matthews

July 5, 1995

1 Introduction

This document forms part of a set describing the Poly[3] and Standard ML[2] systems. It describes the abstract machine and its realisation on the Vax and the MC68020 and on a interpreted byte-code system.

2 Overview

The Poly code-generators generate segments of code from a tree representation. The tree is constructed by the Poly and ML front-ends but passes through machine- and language-independent global optimiser and pre-code passes before it gets to the code-generator. The code-generator for the interpreted system essentially flattens the tree into a byte-code so rather than describe the tree structure and its mapping into the various machine-codes, the byte-code system will be used as a model for the other two versions.

3 Basic Byte-Code System

The byte-code system is a simple stack machine in which each instruction is a single byte, possibly followed by a number of operand bytes. It has only a few primitive instructions, see Table 3, so there is plenty of scope for using the remainder of the 256 available opcodes for shorter versions of common cases. For example, in addition to having an instruction to load a full-length immediate constant it might be worth having an instruction to load a single byte constant to be used for values less than 256. Going one step further it is possible to combine frequently used operand combinations with the opcode, so that, for example an instruction to load the constant value zero would be a single byte. These optimisations will be described in due course.

The machine has only four registers, the stack pointer, the program counter, the handler register and a temporary register. The stack pointer, *sp,* points to the top entry on the stack which is used for instruction operands, function parameters and results, return addresses and for stacking exceptions. The program counter, *pc,* points to the current instruction in the code. The handler register, *hr,* points into the stack at the current exception entry. The temporary register, *t,* is only explicitly used during exception handling.

A single stack is used on which procedure parameters, local values, link information and exceptions are held. A procedure call involves first pushing the arguments onto the stack and then executing a call instruction which pushes two words of link information. Local declarations inside the procedure also result in values being pushed onto the stack, as does setting up an exception handler. Intermediate results are treated as local declarations, indeed the distinction between local declarations and intermediate results is arbitrary since the higher-level parts of the compiler may remove local declarations made by the user, for example where the value is a constant, and introduce extra declarations as a result, say, of expanding inline functions. Values are popped from the stack when they are no longer needed. There is no frame pointer register so values on the stack have to be addressed relative to the stack pointer. All the objects on the stack have fixed size so the code-generator knows the number of words on the stack at any time and can work out the offset. Not having a frame pointer means that procedure call and return is faster since there is no need to save an extra register. In the machine-code versions it also saves instructions which is perhaps more important.

The most obvious point is that the machine does not have any instructions for operating on data, such as adding integers, or reading from a file. These are all done using the standard closure call mechanism, but with special closure addresses. In the byte-code system these are trapped by the interpreter but in the code-generated versions they are the addresses of code in the run-time system.

This has several advantages. First of all, in a functional language, the operation to add two integers is a true function and can be passed around or stored in variables in any way that satisfies the type system. With this mechanism, addition behaves like any other function until it is applied.

It is still possible for the code-generator to implement an addition instruction by recognising the case of a function application when the function is a constant whose value is the special address for addition. This, however, is an optimisation, and the choice of whether to make it or not depends on the expected frequency of

jump	Unconditional jump.
jump_false	Pop the top of the stack and jump if it is false.
case	Indexed jump.
load_local	Load a value at a given offset from the top of the stack.
indirect	Indirect load using the address on the top of the stack and a given offset.
load_constant	Load an immediate value.
const_addr	Load a value from the constant vector.
call_closure	Call a procedure using the closure on the stack.
call_sl	Call a procedure using a static link.
return	Return from a procedure, popping a number of arguments.
get_store	Allocate a new object of a given size.
move_to_vec	Store a value in a new object.
lock	Lock a new object when all the values have been put in.
set_handler	Set up an exception handler.
del_handler	Delete the exception handler.
raise_ex	Raise an exception.
load_ex	Load the last exception name.
reset	Remove values from the stack.
reset_r	Remove values from the stack, sliding the top value down.
set_stack_val	Change a value on the stack (only used in tail-recursion).
pad	A no-operation.
enter_int	Enter interpreter (only used during bootstrapping).

Table 1: Primitive Opcodes

additions, and whether it justifies the extra complexity in the code-generator.

We will now look at the instructions in more detail and describe some of the derived instructions, which though not primitive, make the machine reasonably efficient. The choices were made on the basis of compiling a large program and counting the occurrences of various combinations of opcodes and operands. The most common choices were made into single byte-codes.

3.1 Control and Procedure Call

jump, jump_false, case, call_closure, call_sl, return, pad, enter_int
The conditional and unconditional branch instructions, *jump* and *jump_false,* are generated from high-level *if* and *while* expressions and from tail-recursion removal. The most common cases are short jumps over only a few instructions so branches would seem to be an ideal candidate for using short operands. However it is difficult for the code-generator, which is generating a stream of instructions, to recognise whether a shorter form of the branch can be used until the destination is reached. This would require the code-generator to generate a long branch initially and then to shorten the branch and shift up the code if a shorter branch could be used. Shifting the code is generally a bad idea since other addresses must be adjusted, so instead two forms of each instruction are used, a direct and an indirect jump, each with a single byte operand. The direct form is used for branches of less than 256 bytes, and the indirect form for longer jumps. The code-generator keeps a list of all branches whose destinations are not yet known and compiles in an indirection table for branches which are about to go out of range.

The *case* instruction does an indexed branch to one of the addresses which follow the instruction. This instruction could be replaced by a series of comparisons and conditional branches, but when the number of tests is large it provides a considerable speed-up.

There are two kinds of procedure call mechanisms used. The general case is where a procedure is a closure, in other words the code of the procedure together with the non-local values it uses. It must be used wherever a procedure is returned from a procedure or assigned to a variable.

The representation of a closure is a vector whose first word is the address of the code and whose other entries are the non-local values. The *call_closure* instruction pushes the return address on the stack, then the address of the closure, and then jumps to the code.

Constructing a closure involves allocating space on the heap and copying the non-local values into the closure. This is expensive so where possible an alternative procedure call mechanism is used. If a procedure is only ever called and is not used in any other way, such as being assigned to a variable, the non-local values it uses will always be on the stack when it is called and they can be found with a static-chain. These procedures are called with the *call_sl* instruction. The operands to the instruction are the procedure address, always a constant, and the static link entry to be used, either the address of the current frame or of another frame found from the static chain if the procedure is not local. Because there is no frame pointer register, in either case the base of the frame has to be found from the stack pointer. The instruction includes an operand which gives the current number of items on the stack so that the base of the frame can be found.

The *return* instruction is used to return from either kind of procedure. The value on the top of the stack is returned as the result of the call. It has an operand which is the number of arguments to be removed from the stack.

The *pad* instruction is no-operation and is used before a call instruction to ensure that the return address pushed onto the stack is not aligned on a word boundary. This is necessary so that the garbage-collector can distinguish return addresses, which can point into objects, from other addresses. While an interpreter could, for example, add a bit to a return address before pushing it on the stack, and mask it out when returning, this would be difficult with compiled machine-code where the return address would normally be a simple machine address. This way of distinguishing return addresses works equally with compiled machine-code.

The final instruction in this group is *enter_int*.
It is only used during the boot-strap phase when a machine-code version of the system must coexist with the interpreted code. Its purpose will be described in due course.

3.2 Data

load_local, indirect, load_constant, const_addr, set_stack_val, reset, reset_r, get_store, move_to_vec, lock

load_local copies a value from the current stack frame onto the top of the stack. It is used to load the values of local declarations or arguments. Because there is no frame pointer the operand gives the number of words from the stack pointer before the operation. Loading from the stack is a very common operation so single byte operations are used to load the first few words on the stack.

The *indirect* instruction replaces the top item on the stack by a value from the object it points to. An operand gives the offset from the start of the object of the value required. Small offsets are very common so single byte operations are used for them.

Often a value is loaded from the closure of a procedure or by following a static chain. The static chain or the closure address are both on the stack so they can be loaded by *load_local.* Loading a value from the closure or following the static chain can be done with one or more *indirect* instructions. Because of the frequency of these operations derived instructions are used for loading closure or static link values.

There are two kinds of value in the Poly system. *Addresses* are values greater than 65536 and are used to refer to objects. They will be changed by the garbage-collector if an object has to be moved to compact the store. *Numbers* have values of less than 65536 and are used for small integers. Either kind of constant can appear in a piece of code but they must be treated differently. Addresses are put in a section at the end of the code where the garbage-collector can find and modify them. They are loaded by the *const_addr* instruction which takes as its operand the offset from the current program counter of the value required. Numbers on the other hand always have the same value so they can be put in the code directly. They can be loaded by the *load_constant* which is followed by the lower 16-bits of the value. Single byte instructions are provided to load common constant values such as zero.

The Poly system is basically a functional machine with operations on variables done by special procedure calls. However tail-recursion removal means that values on the stack can be overwritten when a procedure tail-recurses. In the case of a normal procedure one or more of the arguments will be overwritten with a new value. The *set_stack_val* instruction takes a new value for an argument off the stack and sets the given location to that value. Because inline procedures as well as normal procedures can be tail-recursive it is possible for local values to be assigned new values with *set_stack_val.*

The *reset* instruction removes items from the stack by resetting the stack pointer. *reset_r* does a similar job but slides the value on the top of the stack down.

New objects are allocated on the heap by the *get_store* instruction. This returns the address of an object of specified size. Values are put into the object with the *move_to_vec* instruction which takes a word off the stack and puts it into a given offset in the new object. When each value has been filled in the *lock* instruction is used to indicate that the object is no longer changeable. It is needed because the garbage-collector treats objects that may be changed from objects that may not. This sequence is used for the general case where several objects may be constructed, such as closures for mutually recursive procedures. In most cases a single *tuple* instruction can be used which allocates an object, pops values from the stack into it, and locks it.

3.3 Exceptions

set_handler, del_handler, raise_ex, load_ex
An exception handler is set up by the *set_handler* instruction. This pushes two words onto the stack containing the address of the handler and the previous value of the handler register, *hr.* The register is then loaded from the stack pointer so that it points to these words. The address of the handler points into the code in the same way as a return address so it must not be aligned onto a word boundary. A *pad* instruction may be needed before it to ensure this.

An exception can be raised with a *raise_ex* instruction. The value on the top of the stack is the name of the exception and this is loaded into the temporary register *t.* The stack pointer is then reset to the value in the handler register. The two values on the stack are now the previous value of the handler register, which is reloaded, and the address of the handler code to which a jump is made. In this way the handler is entered with the stack pointer at a known value. The exception name can be recovered from *t* by the *load_ex* instruction.

At the end of the scope of the handler, if an exception has not been raised, the previous handler is reinstated with the *del_handler* instruction. This pops the two values off the stack, loading the handler register with the old value and discarding the handler address. The *del_handler* instruction has an operand which is the location to jump to. This is used because the code for the handler itself is compiled immediately afterwards so a jump must be made round it.

4 Machine-Code Versions

The machine-code code-generators for the Vax and the 68020 are basically very similar to the byte-code version. The original version for the Vax essentially macro-expanded the byte-code instructions, making use of the store-to-store instructions available on the Vax. In due course a more efficient version was written which made better use of the registers.

4.1 Pseudo-Stack

The present version of the code-generators attempts to use registers rather than the hardware stack for local declarations and intermediate results. In the first version of the code-generator a local declaration was generated by evaluating the expression and pushing the result onto the stack. Any references to the declared identifier involved loading the value from that stack location. It is often the case that a local declaration is made and the value used soon after, much in the way that a temporary result is used. The result of evaluating an expression usually ends up in a register and so if the identifier being declared is used soon after, the value can be taken directly from the register. If there are only a few uses of the identifier it may be possible to use the register for all the references in which case there is no point in pushing it on the stack at all.

Instead of using the real stack for these values a *pseudo-stack* is used in the code-generator to record the location of a value. Entries on the pseudo-stack are either literal constants, values in registers, or addresses relative to a register. Values on the real stack are represented by offsets from a notional frame pointer register and converted into offsets from the stack pointer when the instruction is generated. Making a local declaration involves evaluating the expression and pushing the location of the result onto the pseudo-stack. References to the values on the pseudo-stack use the appropriate location. Values in registers need not be transferred to store unless the register is required for something else, or if a branch or a procedure call is being made which could change the contents of the registers. In that case the registers are pushed onto the real stack and the pseudo-stack is updated to give the new locations.

For each entry there is a use-count giving the number of references to the value. This is calculated by the higher-level parts of the compiler and allows items to be removed from the pseudo-stack when they are no longer required. Registers holding these values can then be re-used. It is also useful to know when the last reference is being made, particularly if a value is in a register. Many operations involve changing the value in a register, for example the addition operation often involves adding a value to the contents of a register. Since this changes the register contents it can only be done if the value will no longer be required.

A highly optimising code-generator might attempt to merge multiple occurrences of the same piece of code. On the whole this should probably be done by the machine-independent optimiser, but the code-generator does attempt to remove repeated loads from the same location. The most obvious case is where a value on the stack is loaded into a register by two neighbouring instructions. When an entry on the pseudo-stack is loaded into a register the entry is marked with the register number. If it is to be loaded a second time the register can be used without having to be reloaded. Since values on the stack are simply held as addresses relative to a register this method can be used for any value which is relative to a register. For instance when following the static chain a register is loaded with the head of the chain and then the next frame is found by an indirection off it. This may be repeated until the required frame is found. The code-generator does not remove items from the pseudo-stack immediately if they have been loaded into a register. Instead it keeps them in case they are needed again. When a relative address is put on the pseudo-stack there is a search to see if it is there already and if it is, a reference is made to the existing entry rather than pushing a new entry. If the existing entry has been marked as having been loaded into a register so much the better and that register can be used if a further indirection is made. In this way following a chain of indirections for a second time will not generate any new code, provided that none of the registers pointing to entries in the chain have been reloaded with anything else. If the chain has been broken anywhere by using the register for something else the chain will have to be reloaded from the break onwards even if further entries in the chain are actually in registers.

4.2 Labels and Jumps

The code-tree represents flow of control by the structure of the tree but this has to be flattened using jumps when the machine code is generated. The canonical example is the if-expression but jumps are also produced from while-loops, exception handlers and tail-recursive calls. Conditional branches in the VAX instruction set only allow offsets of $+127$ or -128 bytes which is insufficient to branch round more than a few instructions. There is an unconditional branch instruction which can be used together with the conditional branch to make longer jumps. A long conditional jump can be made simply by using a conditional branch round an unconditional long branch and having the condition inverted, but this would be a considerable overhead if used for every conditional branch. It is better to try to use the short conditional branches where possible since many, perhaps most, of the branches will be within the range of a short branch.

As with many problems there is a trade-off between the improvement in the code gained by using short branches and the effort involved, but it is possible to go a long way without a great deal of effort. To solve the problem completely requires multiple passes because the size of one branch instruction may affect whether another can use the shorter form [4]. The solution adopted in the present version of the code-generator uses only the single pass but generates reasonably good code. It is based on a scheme described in [1] and assumes that long branches are only needed rarely. Short branches are used for all forward jumps, which works fine provided the code between the jump and its destination is not too large. We may, however, put in a short jump and start generating the code which it is jumping round, only to find that the short jump is not going to reach. The solution is to interrupt the code that is being generated and put in a long unconditional branch to carry on the jump. The destination of the original short jump is set to this new branch instruction and this long branch should now be enough to reach round any more code. Of course this new branch instruction cannot be put in anywhere we find a branch coming to the end of its range. We must ensure that the normal instruction stream is unaffected by having this long branch put down in the middle of it. This is done by only putting the long branch between two instructions and preceeding it by an instruction to jump round it.

```
        tstl r0 # test a value
        beql L1a# conditional branch, originally to L1
        .
        .   A lot of code
        .
        brb SKIP# skip the branch extension
L1a:    brw L1  # continue the conditional branch to L1
SKIP:   .       # and continue the previous code
        .
        .   Perhaps a lot more code
        .
L1:     ...     # The final destination
```

There is a fixed overhead in generating this code because of the need to branch round the extension. To reduce this there are two limits which are used to decide whether to extend a branch. The *hard limit* is reached when an instruction is about to be generated which will almost certainly cause a jump to go out of range. At this point the instruction stream must be interrupted and that jump extended. There may, however, be other jumps which, while they are not at the extreme of their range, are getting close to it. Having once interrupted the instruction stream and generated the skip round the extended jump, it is probably worth extending these other jumps as well. The *soft limit* is used to decide whether jumps are getting close enough to the end of their range to extend them if the stream has already been interrupted. The soft limit also comes into play if an unconditional branch is generated as part of the normal instruction stream since there is then no need to interrupt the stream.

5 Forward References

Recursive and particularly mutually recursive procedures pose a problem for the code-generator since it is not possible to get the address of the code of a procedure until it has been compiled. This means that some procedures may be compiled before all the procedures they refer to. Since there is no separate linking phase forward references have to be filled in by the code-generator.

References

[1] R. D. Evans, *Language Implementation in a Portable Operating System*, Ph. D. Dissertation, University of Cambridge. 1981.

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

[3] David C.J. Matthews, *The Poly Manual*, SIGPLAN Notices, Septemeber 1985.

[4] M. H. Williams, *Long/Short Address Optimization in Assemblers*,Software - Practice and Experience Vol 9,227-235 (1979)