Machine Independent Optimisation in Poly and Poly/ML

David C.J. Matthews

1 Introduction

The Poly and Poly/ML code-generators generate a machine-independent code which is processed by an optimiser before being passed to a machine-dependent code-generator. The purpose of this note is to describe the machine-independent code and the optimisations performed on it.

The machine-independent code is in the form of a tree structure, the code-tree, which is easier to manipulate than the more conventional sequence of instructions, and with a persistent storage system does not have to be flattened to be preserved. Each node of the tree contains an instruction together with some other data and/or pointers to other nodes. For example a node containing a function call, the **eval** instruction, has a pointer to code returning the function, a list of arguments and the number of arguments.

The code-tree could be directly interpreted by a recursive procedure which walked over it evaluating the expressions and returning the results. At the most basic level the code is purely functional though there are some special instructions for creating and manipulating a "store". Declarations are handled by the "decl" instruction which saves the result of an expression on the "declaration stack". The value can then loaded by the "extract" instruction which gives the location in terms of an offset in one of currently accessible stack frames. As described below, references to non-local declarations must be converted into addresses in the closure of the procedure if higher-order procedures are to work. There are no references to global declarations in the code since they are always replaced by their values when they are first looked up.

2 The Optimiser

The function of the optimiser is to perform transformations on the code-tree which preserve the original meaning, but allow the machine-dependent code-generator to produce better code. A simple example is constant propagation. A declaration may consist of a giving a name to a manifest constant. This will generate code to put the value onto the stack. References to this declaration will be made through the identifier which will generate code to load the value off the stack. If the two are matched up by the optimiser then the instruction to load the value can be replaced by the value itself, and the instruction to push the value onto the stack can be deleted. A similar process is used where a declaration involves simply giving a name to a value which has been declared previously e.g. let x == ...; let y == x. Declarations of inline procedures are treated in the same way. Simplifying these declarations is particularly important when expanding inline procedures.

A large part of the work of the optimiser is in processing inline procedures. Many instructions in Poly are implemented as inline procedures, for example the instructions which extract fields from records or even the instruction to add two integers together. In addition there are polymorphic functions which select the type from which an operation is to be applied, for example the + function which applies **integer$+** if the arguments have type **integer**. This means that adding two values together involves two inline procedure expansions.

In essence an inline procedure is expanded by replacing the call by a block consisting of declarations of the formal parameters with their actual values, and then the body of the procedure. The process described above for removing redundant declarations is used to simplify the code. A few problems are found with recursion. If the inline procedure recurses then it cannot of course be expanded indefinitely. Instead the recursive call is generated as a normal procedure call. Tail-recursion is treated specially and is passed on to the code-generator to generate a jump back to the start of the body of the inline procedure. This adds a few complications. The start of the body of the procedure, as opposed to the code which declares the actual parameter values, must be marked. The second complication is that any parameters which may change as a result of the recursion must be forced into variables and not optimised out even if their initial values are constants. Optimising tail-recursive inline procedures is probably worthwhile because it allows **for** and **first** functions which sequence over abstract sets to be implemented as efficiently as a built-in **for** statement.

The optimiser detects tail-recursion of ordinary procedures and marks tail-recursive calls so that they can be processed specially by the machine-dependent code-generator.

One function the optimiser performs is not optimisation at all since it must be done to allow higher-order functions, and that is converting references to identifiers outside a procedure into references via the closure. A procedure in Poly is represented as the code of the procedure together with the non-local values referred to by it. It is implemented as a vector whose first word is the code and subsequent words are copies of the non-local values. Since variables (references) are not held directly on the stack all values put in the closure can be copied without affecting the semantics. When a procedure is declared the closure is constructed and that is the value which represents the procedure. It can be passed around or assigned to variables and it will always work. This implementation has the advantage that a stack can be used for all the local values and stack frames do not have to be processed by the general storage allocator.

Creating a closure is still an expensive operation since the vector must be allocated and the values copied. It is frequently the case that a full closure is not needed, as is shown by the number of languages which manage without procedures as first-class values at all. A common case is the procedure with no non-local references at all. This case is handled specially by combining the closure and the code. Another case is where the procedure has non-local references but is used in a way that does not require a closure. If a procedure is only ever called and is never returned as a result from a procedure, passed as a parameter to a procedure (this includes assignment to a variable) or referred to from the closure of another procedure then it need not be made into a closure. The optimiser keeps track of references to a procedure and marks it as not needing a closure if the conditions are satisfied. The code-generator can then use a static link or a display for these cases.

The optimiser processes the declarations of a block by recursing down the block optimising each declaration or statement in turn. As the recursion unwinds it looks at the procedures which have been declared and marks the procedures which do not need full closures. Those that do need full closures have all the non-locals they refer to marked as "loaded" and so they will be made into closures if they are procedures themselves. This works because the scope rules (excluding mutual recursion) ensure that a procedure only refers to declarations made before it, i.e. earlier on the list.

A situation that occurs frequently in ML is generating tuples which are immediately taken apart by other code. The optimiser detects such cases and avoids generating redundant tuples.