

One pass Code Generation Using Continuations

Keith Clarke
Department of Computer Science
Queen Mary College
University of London

Abstract

Translation schema are described, using the functional notation of Peyton Jones, which make possible the generation of high quality code for conditional formulas and boolean expressions. These use contextual information describing the continuation of evaluation after that of the formula being translated. The technique can be used for the incremental improvement of a code generator, leading eventually to one that never generates jumps to jumps and making it possible to optimise target machine instructions that have been described in the literature as 'logically adjacent'. The method is efficient and can be applied to conventional programming languages such as C and Modula 2.

Keywords

Continuations, code generation, boolean expressions, jumping code, logically adjacent code.

This Technical Report, #468, has been scanned in from a paper original. There may be errors of character substitution as a result.

In many places a right-arrow character -> has been replaced by a ~ symbol. Please read this as meaning "rewrites to".

1. Introduction

The ORBIT optimising compiler for the Scheme dialect of Lisp (Kranz et al, 1987) uses the idea of continuations, familiar from denotational semantics, to produce target code that is competitive with that from C and Modula-2 implementations. The compiler is a development of Steele's experimental compiler Rabbit.

When a formula is translated with respect to a continuation, the continuation describes some of the contextual information (namely, what computational step follows evaluation of the formula) that is normally ignored during translation. Thus ORBIT avoids generating code that merely leads from one continuation to another - for example, jumps to RETURN instructions. Even better, many procedure calls can be replaced by jumps, since the compiler will know that immediately following the call, the calling procedure will itself return. Optimisation of *tail-calls* can be very valuable, since it makes it easier to improve register use over the call; any locals of the calling procedure that are held in registers can be overwritten by the argument values, avoiding use of an argument stack.

Conventional optimising compilers often achieve similar results by painstaking analysis of a relatively low-level intermediate code, performing elaborate data-flow analysis in order to discover the same possibilities of optimisation.

The ORBIT compiler uses nine phases, and must be considered a large and elaborate program; conventional optimising compilers are also complex pieces of software. ORBIT's *continuation-passing style* (CPS) is intimately related to the functional nature of Scheme, using function-closures to represent continuations. Good translation of closures is essential in Scheme, and therefore it is natural to use the same techniques to generate good code for continuations. This paper shows how most, if not quite all, of the benefits of a CPS compiler can be obtained in a more conventional setting, where closure analysis would be a largely unnecessary burden.

Section 2 describes earlier two-pass approaches using contextual information. Section 3 shows how Bornat's 'skipping code' translator for boolean expressions appears in the functional style. Sections 4 to 6 extend the notion of a label to include statement and procedure continuations. Sections 7 and 8 show how source language labels can be compiled using pairs of continuations. Section 9 treats logical adjacency using conditional assignments as an example. Similar ideas are used in Section 10 to show how block exit stack manipulations can be improved when compiling to a stack-based machine.

2. Two-pass compilation

Compilers for 'simple' imperative languages such as C and Pascal are often organised as a first pass, generating an intermediate code which is improved and finally translated into target machine instructions by a second pass. Optimisation is usually performed on the intermediate code or the target code, or both, since a one-pass 'front-end' is so tightly constrained that the direct generation of high-quality code is virtually impossible. This organisation has the major benefit that the same complicated and expensive back-ends can be used with several different source languages.

However, an alternative arrangement is to construct a *parse-tree* as intermediate form, and to translate from this. In extreme cases, the parse tree for the entire source program is constructed before any translation is done; in others, parse trees for expressions or perhaps procedure bodies are constructed and translated individually, saving on storage space.

Using a parse tree has the advantage that the *order* in which source language phrases are translated need not be the same as their original textual ordering. For example, expression

translation can take advantage of the commutativity of some operators to generate code for the right operand before the left, as in the well-known Sethi-Ullman algorithm (Sethi and Ullman, 1970). All the information in the original source program is available when optimisations are being considered; programming idioms can be easily recognised and translated into corresponding target language idioms.

The Sethi-Ullman algorithm also illustrates how context - in this case the number of registers that are required for evaluation of the whole expression - can be used to guide translation of a sub-expression. In general, and for other syntactic forms, the more context considered, the higher the quality of the resulting code. The next section shows how a simple consideration of context makes it possible to generate excellent quality 'skipping' or 'jumping' code for boolean expressions.

3. Translation of Boolean Expressions

Bornat (1979) has described a beautifully simple translation scheme (still not widely known) for the conditional interpretation of boolean connectives, so-called 'jumping code'. Each boolean formula F is translated by the procedure B relative to three pieces of context: a label, transfer to which implies that F evaluated to False; a second label, transfer to which implies that F evaluated to True; if the source formula evaluates to the same sense as the third parameter, neither label need be jumped to, that is, control can 'fall through'. This idea is presented here in a modern referentially transparent programming notation, as used for example in Peyton Jones (1987). The procedure is defined in the form of a four-argument semantic function or *translation scheme*, with the first parameter being the syntactic value being translated.

Boolean variables are translated by B into simple test-and-branch code:

$$\begin{aligned} B[[v]] \ t \ f \ \text{true} &= \text{TEST } v; \text{ JUMPFALSE } f \\ B[[v]] \ t \ f \ \text{false} &= \text{TEST } v; \text{ JUMPTRUE } t \end{aligned}$$

The source language variable v , shown on the left in 'syntactic' brackets, appears on the right as part of a target language instruction. In practice, the variable would be mapped to an addressing mode suitable for accessing its run-time representation.

Similarly, the implementation language variables t and f reappear as target language labels. These conversions have been omitted for clarity. In what follows, all implementation language variables are shown in lower case, and all target language constructs in upper case.

No code need be generated for boolean constants in some common situations, while in others an unconditional jump can be used:

B[[True]] t f true = NO-OP
BI[True]] t f false =JUMP t
BI[False]] t f true =JUMP f
B[[False]] t f false = NO-OP

It is simple to confirm that these code sequences only transfer control to the label required by the specification, or permit control to fall through when the source expression has the specified sense.

Rules for arithmetic comparisons can easily be included. Assuming conventional COMPARE and conditional jump instructions, we can take equality tests as typical:

B[[a=b]] t f true = COMPARE a,b; JUMPNOTEQUAL f
B[[a=b]] t f false = COMPARE a,b; JUMPEQUAL t

The B scheme can be simply incorporated into a translation scheme for conditional statements:

S[[if p then a else b]] = (B[[p]] T F true); T:(S[[a]]; JUMP N; F:(S[[b]]); N:
where T, F and N are new labels

The rule for if-statements generates three new labels, which are shown in upper case. In subsequent rules, upper case will be used for labels in the rule where they are generated and lower case where they appear as 'formals'.

Finally the boolean connectives, assuming a conditional interpretation,¹ can be translated without the use of unconditional jump instructions:

B[[p and q]] t f n = (B[[p]] N f true); N: (B[[q]] t f n)
B[[p or q]] t f n =(B[[p]] t N false); N: (B[[q]]t f n)
B[[not p]] t f n =B[[p]] f t (~n)

Correctness can be proved by structural induction on the source expression. Simple constant conditions are translated well; we will see below that good code can be generated for, eg, **while true do S**. Compound conditions such as (**p or false**) should be optimised by an initial 'tree improving' pass, to eliminate the constants. In the rest of this paper, it is assumed that algebraic simplification is performed before translation.

The **B** rules generate no code for the connectives, and at most one test and conditional jump for each of the variables. There is therefore no shorter translation available. In more detail, the **B**

scheme does not generate labelled jump instructions and there can therefore be no jumps to jumps. As there are no phrases that compile to just a jump, the scheme does not generate jumps over jumps either. Finally, there are no trivial jumps (**of the form JUMP F; F: or JUMPFALSE F; F:**).

The rules for connectives preclude the translation of the left operand before the connective has been parsed, as the left operands of **and** and **or** are treated differently.² Thus a parse tree or equivalent intermediate representation is essential.

Implementation of translation rules matched against a tree, such as these, can be very efficient. Augustsson (1985) and Wadler (1987) give methods for compiling the rule selection (or pattern matching). Symbolic assembly code can be generated by replacing the target machine labels and instructions on the right hand side of the rules by suitable printing statements: there is no need to construct lists of target instructions.

The elegance of this translation scheme led the author to investigate how other translation schemes (such as that for statements) could be modified to use similar contextual information.

4. Statement continuations

1 as in **cand** and **cor** (Gries, 1981).

2 unless prefix notation is used for **cand** and **cor**.

4-2/11/88-7:22 am

The translation scheme S, given above, though obviously correct, generates a jump instruction following the then-part of an if-statement regardless of the context of the whole statement. When it is itself a then-part, this jump will lead directly to another jump. In the ORBIT compiler, this unnecessary jump is avoided by giving explicitly the continuation of *all* of the formulas in the source program, but we have seen that in some situations it is convenient to allow the generated code to 'fall through' to the next instruction; this corresponds to taking the 'normal' continuation.

We therefore define a union type to represent two possibilities; first that the continuation is an unconditional jump, and second that it is the next instruction in line:

type continuation ::= Goto x | Normal

**S[[if p then t else e]] Normal = (B[[p]] L M true); L: (S[[t]] (Goto N));
M: (S[[e]] Normal);N:**

**S[[if p then t else e]] (Goto n) = (B[[p]] L M true); L: (S[[t]] (Goto n));
M:(S[[e]] (Goto n))**

S[[s;t]] c = (S[[s]] Normal) ; (S[[t]] c)

The term (**Goto n**) on the left hand side of the second rule is *pattern*, which if applicable will cause the variable **n** to be bound to some target language label generated by another rule. The two rules for translating **if-then-else** statements are exhaustive because alternative patterns are given for both kinds of continuation. The rule for statement sequences is sufficient because the variable pattern **c** matches both kinds of continuation.

Unconditional statements are translated in the usual way, followed by a jump instruction¹ if the continuation parameter specifies one.

The scheme is easily extended to include **while**-statements:

S[[while p do t]] Normal = R: (B[[p]] L M true); L: (S[[t]] (Goto R)); M:

S[[while p do t]] (Goto m) = R: (B[[p]] L m true); L: (S[[t]] (Goto R))

Example

The above rule generates good code when the final statement in a while-body is a conditional:

S[[while p do if q then r else s]] Normal

~ R: (B[[p]] L M true); L: (S[[if q then r else s]] (Goto R)); M:

~ R: (B [[p]] L M true); L: (B[[q]] L' M' true);

L': (S[[r]] (Goto R)); M':(S[[s]] (Goto R)); M:

~ R: TEST p; JUMPFALSE M; TEST q; JUMPFALSE M';

...r...; JUMP R; M':...s...; JUMP R; M:

¹ Translation schema are inevitably complicated by the need to refer at once to source language, implementation language and target language. We have tried to avoid confusion by referring always to source language *statements*, target language *instructions* and implementation language *rules* or *schema*.

The S scheme generates labels in four positions: before the code for a boolean expression, which will therefore label a **TEST** or **COMPARE** instruction; before the code for a statement (a then-part, else-part, or while-body); and after a complete if-statement or while-statement. To avoid jumps to jumps, we must show that none of these labels are immediately followed by a jump instruction. It is sufficient to show (a) that the translation of a statement by the S scheme never generates an empty sequence of instructions or code starting with a jump and (b) that, if a call of S can generate a trailing label, the rule in which the call occurs does not generate a jump immediately following. Case (b) is satisfied since trailing labels are only produced when a statement is translated relative to a Normal continuation; inspection shows that Normal continuations only arise in the translation of sequential composition, when the following instruction is generated by a

recursive call of S. In the rest of the paper, it is shown how (a) can also be satisfied

The translation of the **donothing** or **skip** statement (indicated in C, for example, by an isolated semi-colon) generates no target instructions. This can lead to a labelled jump instruction being generated, and hence in a jump to a jump. It is not possible to avoid this by changing the translation of this statement, instead the rules that call for its translation relative to a Goto continuation must be eliminated. Similarly, source language goto statements require special attention. These problems are the topics of sections 7 and 8.

The above rules use only the left context (and the continuation parameter) to determine the appropriate continuation for each constituent statement of an if or a **while**. As they stand, they are therefore suitable for use in a one pass compiler, during parsing. However, in the next section we examine a further optimisation that requires, for most source languages, the construction of the parse tree for a whole procedure body.

A similar syntax directed translation scheme is given in Aho, Sethi and Ullman (1986), using an inherited 'next label' attribute for all statements. The present method can be seen as an improvement of their scheme that does not generate labels unnecessarily. This greatly simplifies the task of proving optimality, and improves the readability of the generated code, simplifying debugging of the code generator. In subsequent sections we will develop the idea further.

5. Procedure continuations

It is common for the body of a procedure to be a conditional statement. This will be translated into a sequence of instructions followed by an *epilogue* that restores the environment of the calling procedure and then resets the program counter. On modern machines, this is often accomplished with a single **RETURN** instruction.

It is extravagant to compile a procedure body into code that jumps to a single return instruction, and easy to do better. We extend the notion of a continuation:

type continuation : :- Goto L I Normal I Proc

And similarly extend the translation scheme:

S[[unconditional-statement]] c = ...standard code sequence...; (C c)

The translation function C simply generates appropriate code for a continuation:

$C(\text{Goto } x) = \text{JUMP } x$
 $C \text{ Normal} = \text{NO-OP}$
 $C \text{ Proc} = \text{RETURN}$

The translation scheme, P , for a procedure body, uses S :

$P[[I]] = S[[I]] \text{ Proc}$

Since **RETURN** instructions are never labelled (like jumps, they always follow the target instructions for a source language unconditional statement), this translation scheme does not generate jumps to **RETURN**s either.

Source languages that introduce sequences of statements explicitly (like occam, for example), give the translator sufficient left context to generate similar code during parsing, but in general construction of the parse tree is needed.

6. Tail calls

If the compiler can verify that non-local accesses to a procedure's local variables are not possible, then calls of other procedures can be translated into jumps, when translated relative to a procedure continuation. For a procedure with no parameters, the extra rule needed is simply:

$S[[\text{call } p]] \text{ Normal} = \text{CALL } p$
 $S[[\text{call } p]] (\text{Goto } x) = \text{CALL } p; \text{JUMP } x$
 $S[[\text{call } p]] \text{ Proc} = \text{JUMP } p$

This translation includes the common 'tail recursion' conversion into a loop. If the called procedure has parameters, the jump is preceded by a simultaneous assignment of the actual argument formulas to the formal parameter locations. Since none of the locals of the calling procedure are live at this point, the translation can make effective use of the registers. Often tail recursion involves passing a parameter on unchanged and in the original parameter position. The resulting assignment will be of the form $x:=x$ and is easily recognised as another special case.

On many **RISC** machines, the calling sequence explicitly saves the various registers, including the program counter. It is therefore simple to modify the second of the above rules so that the address of the continuation is stacked instead of the current program counter, thus saving another jump. Again, on a heavily pipe-lined machine, this can be very valuable.

When the called procedure has no parameters, the rule once again introduces the chance of generating a labelled jump instruction. This can be avoided as before by adding more special cases for dealing with conditional statements.

7. Labelled statements and source language gotos

In the denotational description, a statement label is said to directly denote a continuation. The denotation of a sequence of statements, the second of which is a **goto**, is that of the first statement relative to the continuation specified by the **goto**. We can achieve a similar effect

by binding statement labels to values of our continuation type, using an environment, symbol table or similar data structure.

As with denotational semantics, the description is simpler for restricted forms of **goto** such as Tennent's (1981) **redo** and **leave**. These are similar to **BCPL's continue** and **break** statements, with named continuations. The former statement is used to repeat execution of the named statement (in other words it is a backwards goto) and the latter to terminate execution (in other words it is like a goto to the following statement). A **redo** statement requires that the label denote the continuation that begins with execution of the labelled statement, and a **leave** statement that it denotes the continuation of the labelled statement. For this reason, the environment maps labels to *a pair* of continuations.

Following Tennent, an environment that behaves exactly like e except that it maps a to b is written as $e[a \sim b]$.

A labelled statement is therefore translated relative to a continuation and an environment:

$$S[[I: C]] \text{ Normal } e = N: (S[[C \sim \text{Normal } (e[I \sim (\text{Goto } N, \text{Goto } N')]])]; N':$$

$$S[[I: C]] c e = N: (S[[C]] c (e[I \sim (\text{Goto } N, c)]))$$

$$S[[\text{redo } I]] c e = C \text{ first } (e[[I]])$$

$$S[[\text{leave } I]] c e = C \text{ second } (e[[I]])$$

$$S[[I; \text{redo } J]] c e = S[[I]] (\text{first}(e[[J])) e$$

$$S[[I; \text{leave } J]] c e = S [[I]] (\text{second}(e[[J]))e$$

The operator **first** selects the first element of a pair, the operator **second** selects the second element. The first of these rules ensures that a label is never bound to a **Normal** continuation, as

the meaning of these is context dependent.

Our objective, of never generating a labelled **JUMP** or **RETURN** instruction, cannot be met if source language **gotos** receive this simple translation. It is clear that **leave** applied to a statement forming the complete body of a procedure would be translated directly into a **RETURN** instruction, as required, but **leave** compiled relative to a **Goto** continuation may in some circumstances compile poorly.

Example

```
S[[x: (if p or q then leave x else skip; print "hello")]] (Goto c) e
~ N: (S[[if p or q then leave x else skip; print "hello"]]] (Goto c) e'
      where e' is (e[x~(Goto N,Goto c)])
~ N: (B[[p or q]] T F true); T: (S[[leave x]] (Goto c) e'); F: (S [[print "hello"]]] (Goto
c) e')
~ N: TEST p; JUMPTRUE T; TEST Q; JUMPFALSE F;
      T: (S[[leave x]] (Goto c) e'); F: (S[[print "hello"]]] (Goto c) e')
~ N: TEST p; JUMPTRUE T; TEST Q; JUMPFALSE F;
      T: JUMP c; F: PRINT "hello"; JUMP c
```

Since **leave x** compiles to a **JUMP**, and **p or q** compiles to code including a branch to T, this translation is sub-optimal.

The following section shows how this situation can be alleviated.

The translation of a source language **goto** must arrange that execution continues not merely with the correct instruction, but in the correct environment. Section 10 gives some idea of what that implies in terms of discarding local variables from a stack, but the general problem of **gotos** that cross procedure boundaries, and so require the discarding of entire stack frames, is not dealt with here.

8. The B scheme with continuation parameters

It is useful to modify the **B** rule to accept continuation parameters instead of simple target language labels. The modification facilitates the optimisation of **leave** and **redo** statements, as well as providing other opportunities.

The modified scheme differs significantly only in the rules for constants and variables. The rules for boolean connectives pass the continuation parameters on in the recursive calls in the same manner as before. There are no rules for **Normal** continuations, as **B** is never used with these.

These rules, by merging the continuation-taking with the condition, also avoid the problem of generating a conditional jump over an unconditional one, which can easily happen with a naive translation of **leave** or **redo**.

It is now possible to generate RETURN instructions within the code for boolean expressions. The Acorn RISC Machine (ARM) has general conditional instructions, and in this case conditional returns are possible and should be used. Moving procedure continuations inside the code for conditional expressions makes this a simple task.

The problem of source language skip statements can also be solved by adding special cases to the **S** scheme, now that the **B** scheme is parameterised on general continuations:

S[[if p then s else skip]] c = (B[[p]] (Goto T) c true); T: (S[[s]] c)

Although all these special cases make the translator rather large, each of them independently improves the quality of the resulting code. They can therefore be added and tested incrementally, with little danger of introducing difficult to diagnose bugs. In fact, there is some commonality that can be used to abbreviate the rules (see Appendix).

9. Continuations and Assignment Source language statements of the form

x:=if p then y else z

can be transformed (given that evaluation of p cannot affect the L-value of x) to **if p then x:=y else**

x:=z

Very often it will be possible to evaluate the formulas y and z directly into the location used for x , and in many more cases the result can be moved to the correct location with a single instruction. Whereas the original assignment will be seen by the translator as an unconditional statement, the transformed version is a straightforward conditional so that a jump can be saved in this case too and two instructions on either side of the extra jump are replaced by one.

Example

This example uses the translation scheme **A**, which is similar to **S**, but takes a value-expression as syntactic parameter and generates code to evaluate the expression into register **RO**.

S[[x:=if p then y else z]] Proc

```

~ (A[[if p then y else z]] Normal); MOVE RO~x; RETURN
~ (B[[p]] (Goto T) (Goto F) true); T: (A[[y]] (Goto L));
                                     F: (A[[z]] Normal);
                                     L: MOVE RO~x; RETURN
~ TEST p; JUMPFALSE F;                T: MOVE y~Ro; JUMP L;
                                     F: MOVE z~RO;
                                     L: MOVE RO~x; RETURN

```

S[[if p then x:=y else x:=z]] Proc

```

~ (B[[p]] (Goto T) (Goto F) true); T: (S[[x:=y ]] Proc); F:(S[[x:=z]] Proc)
~ TEST p; JUMPFALSE F;
  T: MOVE y~x; RETURN; F:MOVE z~x; RETURN

```

It is not necessary to explicitly transform the original program before translation; all that is needed is an extra rule using the original source on the left hand side and the improved translation on the right hand side.

Similar opportunities exist whenever a conditional expression is embedded in a 'small' expression followed by an abnormal continuation. The technique appears to offer similar possibilities to that of Davidson and Fraser (1982), who studied optimisation of *logically adjacent instructions*. Two instructions are logically adjacent if they are linked by flow of control. By considering such links, Davidson and Fraser were able to eliminate jumps to jumps and other redundant computations. The next section considers the optimisation of logically adjacent 'stack tidying' code.

10. Stack machines.

Many implementations use a stack to hold the values of local variables, as well as performing arithmetic on the stack top. Each block is translated into a *prologue* that allocates and initialises stack space for the local variables of the block, code for the body of the block, and an *epilogue* that resets the stack. No matter how large the prologue, the epilogue is always small, and so fits the criterion for optimisation described in the previous section. Unfortunately we cannot in general transform the program as above:

$$\text{Int } x=F; \text{ if } p \text{ then } s \text{ else } s' \quad ? \sim \text{ if } p \text{ then } (\text{int } x=F; s) \text{ else } (\text{int } x=F; s')$$

Assuming, then, that 'distribution' of the declarations over a conditional body is either incorrect or merely undesirable, we are forced to introduce a new complication to the notion of continuations. First we rehearse the idea behind stack implementations. We can characterise the idea of variable access with a simple translation scheme for expressions, which is parameterised on an environment binding names to stack offsets:

$$\begin{aligned} E[[n]] e &= \text{PUSH}\%n \\ E[[v]] e &= \text{PUSH}(ev) \\ E[[x+y]] e &= (E[[x]] e); (E[[y]] e); \text{ADD} \end{aligned}$$

The first rule generates an immediate mode instruction for a numerical constant; the second generates a suitable addressing mode to access a variable; the third generates a stack arithmetic instruction after first arranging for the values of the operands to be pushed. Suitable variable bindings can be created on translation of a block by using the E scheme to translate the initialisation formula, then translating the body of the block relative to the size of the new stack and the new environment:

$$\mathbf{S}[[\text{int } v=F; \mathbf{B}]] \mathbf{c} \mathbf{e} \mathbf{s} = (\mathbf{E}[[F]] e); (\mathbf{S}[[\mathbf{B}]] (\mathbf{Goto} \mathbf{N}) e' (s+1)); \mathbf{N}; \mathbf{DISCARD} \mathbf{1}; (\mathbf{C} \mathbf{c}) \text{ where } e' e[v \sim s]$$

The extra parameter, 5, to the S scheme enables variables to be addressed via a 'name base' or 'frame pointer' for the current procedure. The essential idea is to do allocation and deallocation on block entry and exit, combined with so-called 'procedure level addressing'. The DISCARD instruction removes the local variable from the stack on block exit. It is easy to see that continuations, as they stand, cannot easily be distributed inside the body of a block. Thus chains of jumps can be generated for nested blocks, each leading to a simple block epilogue that discards a few locals before jumping to another block epilogue.

Example

The code for the inner block is followed by a jump to the epilogue of the outer block:

```
S[[int x=3; if p then mt y=x+x; print y else print x]] Normal e s ~
  PUSH %3; TEST p; JUMPFALSE L;
  PUSH s+1; PUSH s+1; ADD; PRINTLOCAL s+2; DISCARD 1; JUMP L';
  L: PRINLOCAL s+1;
  L': DISCARD 1
```

Clearly these successive jumps should be merged into one, and the several DISCARD instructions similarly combined. This is achieved by annotating **Goto** continuations with the stack size expected at the destination label and similarly extending **Normal** continuations. Now that the **S** scheme is parameterised on the stack size, this information is readily available. It is the responsibility of the code at the point of the jump, not the destination, to set the stack correctly.

type continuation ::= Goto j k | Normal s | Proc

S[[int v=F; B]] c e s (E[[F]] e); (S[[B]] c e' (s+l))

where e' = e[v~s]

S[[if p then x else y]] (Normal t) e s = (B[[p]] (Goto L s) (Goto M s) true s);

L: (S[[x]] (Goto N t) e s); M: (S[[y]] (Normal t) e s); N:

S[[if p then x else y]] c e s = (B[[p]] (Goto L s) (Goto M s) true s);

L: (S[[x]] c e s); M: (S[[y]] ces)

S[[unconditional statement u]] c e s = ...code for u...; (C C s)

etc

C (Goto j s) s' = JUMP j	if s=s'
= DISCARD (s'-s); JUMP j	otherwise
C (Normal s) s' = NO-OP	if s=s'
= DISCARD (s'-s)	otherwise
C Proc s	= RETURN

Though the rules given are for *statement-blocks*, they can easily be extended for the case when blocks may have values (as in ML and similar languages). This is not attempted here, although it would be interesting to consider the design of translation rules for blocks in a conditional context, that is, in extending the **B** scheme to produce high quality code for boolean valued blocks.

An unfortunate consequence of these extensions is an apparent need to modify the **B** scheme to allow for stack manipulations; the continuations now specify a stack size. In general, the stack pointer may need to be adjusted when a conditional jump is taken, but left unmodified when the jump is not taken. There appears to be no completely satisfactory solution to this problem on stock hardware, but it should be noted that the above rules always supply identical stack sizes to the **B** scheme for all the alternative continuations: 'falling through', jumping to the then-part or jumping to the else-part. There is therefore no need to adjust the stack before taking one of the continuations. Problems would arise with the special case rules for skip and leave statements, which may well specify different stack offsets. In an experimental implementation, these special case rules are only used when the stack offsets match. They are, after all, optimisations.

Example

The situation below could arise if the source fragment were the final part of a larger block declaring seven other variables.

```
S[[int x=3; if p then mt y=x+x; print y else print x]] (Normal 0) e' 7
  ~(E[[3]] e); (S[[if p then mt y--x+x; print y else print x]] (Normal 0) e' 8)
                                                    where e' = e[x~8]

~(E[[3]] e); (B[[p]] (Goto L 8) (Goto M 8) true 8);
  L: (S[[int y=x+x; print y ]] (Goto N 0) e' 8);
  M: (S[[print x]] (Normal 0) e' 8); N:

~(E[[3]] e); TEST p; JUMPFALSE M;
  L: (S[[int y=x+x; print y ]] (Goto N 0) e' 8);
  M: (S[[print x]] (Normal 0) e' 8); N:

~(E[[3]] e); TEST p; JUMPFALSE M;
  L: (S[[int y=x+x; print y]] (Goto N 0) e' 8);
  M: PRINT 8; DISCARD 8; N:

~(EI[3]] e); TEST p; JUMPFALSE M;
  L: (E[[x+x]] e'); (S[[print y ]] (Goto N 0) e~' 9);
  M: PRINT 8; DISCARD 8; N:
                                                    where e'' = e[x~8,y~9]

~PUSH %3; TEST p; JUMPFALSE M;
  L: PUSH 8; PUSH 8; ADD; PRINT 9; DISCARD 9; JUMP N;
  PRINT 8; DISCARD 8; N:
```

11. Conclusion

The translation schema described above have been implemented in an experimental compiler for the programming language described in Bornat (1987). This implementation is intended as a vehicle for experiments in type checking and translation, with a bias to practical usefulness. The stack machine chosen as the principle target was originally designed for ease of interpretation combined with simple compilation techniques. In this it was quite successful, but it must be admitted that ultimately, the instruction set mitigates against efficient generated code.

A similar compilation strategy directed at a more conventional register machine would be less complicated and produce better results.

Pattern matching on the abstract syntax offers at least the same benefits as peephole optimisation for the general problem of instruction selection, at less cost, since the same information is available in a more convenient form. We have shown how a single pass over the abstract syntax of a program is also sufficient for the generation of high quality code for the control flow aspects of the translation. There remains the problem of register allocation. Further work will investigate how syntax directed data flow analysis can be combined with continuation-based translation, approaching more closely the excellence of the code produced by ORBIT.

Acknowledgements

I would like to thank Richard Bornat for much help and advice, and William Roberts for a very valuable careful reading of earlier drafts of this report.

REFERENCES

Aho A.V., Sethi R. and Ullman J.D. (1986), "Compilers: Principles, Techniques and Tools", pp 490-495. Addison-Wesley.

Augustsson, L. (1985), "Compiling Pattern Matching", in Functional Programming Languages and Computer Architecture, LNCS 201, pp 368-381.

Bornat, R. (1979). "Understanding and writing compilers", publ Macmillan, London.

Bornat, R. (1987), "Programming from first principles", publ Prentice-Hall. Davidson J.W.

and Fraser C.W., (1982). "Eliminating redundant object code",

Ninth Annual Symposium on Principles of Programming Languages, pp 128-32. Gries,

D., (1981) "The Science of Programming", publ Springer-Verlag. p68

Kranz, Kelsey, Rees, Hudak, Philbin and Adams (1986)
"ORBIT: an optimising compiler for Scheme", CACM p219

Peyton Jones, S. (1987). "The Implementation of functional programming languages", publ Prentice-Hall.

Sethi, R. and Ullman, 3. (1970) "The generation of optimal code for arithmetic expressions", 3. ACM 17:4, 715-728.

Tennent R.D. (1981). "Principles of Programming Languages", publ Prentice-Hall. Wadler,

P (1987), "Efficient Compilation of Pattern Matching", in Peyton Jones (1987).

Appendix

The final version of the rules for a target stack machine is given here. All the optimisations discussed in the main text have been included. A vertical bar at the left of a rule indicates that the rule is a special case optimisation. Rather than try for the maximum generality in all the rules, some specific cases that have no obvious good translation (in particular, the B scheme could be extended) are avoided by checks in the rules that might generate those cases.

type continuation ::= Goto j k I Normal s I Proc

C (Goto j s) s'	=JUMP j	if s=s'
	=DISCARD (s'-s); JUMP j	otherwise

C (Normal s) s'	= NO-OP =DISCARD (s'-s)	if s=s' otherwise
CProcs	=RETURN	
isContinuation[[skip]] (Normal s) e s	= (s=s')	
isContinuation[[skip]] Proc e s	= true	
isContinuation[[skip]] (Goto n s) e s'	= (s=s')	
isContinuation[[leave x]] c e s	= true if(second(e[[x]]) = Proc) = (s=s') where (Goto n s') = second(e[[x]])	
otherwise		
isContinuation[[redo x]] c e s	= true if(first(e[[x]]) = Proc) = (s=s') where (Goto n s') = first(e[[x]]) otherwise	
isContinuation[[statement]] c e s	= false	
continuation of [[skip]] (Normal s) e c	= c	
continuation of [[skip]] c e c	= c	
continuation of [[leave x]] c e c	= second(e[[x]])	
continuationof[[redo x]] c e c'	= first(e[[x]])	
S[[int v=F; B]] c e s = (E[[F]] e); (S[[B]] c e' (s+l))		where e' = e[v~s]
S[[if p then x else y]] c e s = (B[[p]] cx cy true s); (C cx s); E:		if(isContinuation[[x]] c e s) A (isContinuation[[y]] c e s)
		where cx = continuationOf![[x]]1 c e (Goto E s)
		and cy = continuationOf[[y]] c e (Goto E s)
S[[if p then x else y]] c e s = (B[[p]] cx (Goto F s) false s); F:(S[[y]] c e s); T: if		isContinuation[[x]] c e s where cx =
		continuationof[[x]] c e (Goto T s)
S[[if p then x else y]] c e s = (B[[p]] (Goto Ts) cy true s); T: (S[[x]] c e s); F: if		isContinuation[[y]] c e] where cy =
		continuationof[[y]] c e (Goto F s)
S[[if p then x else y]] (Normal s) e s' = (B[[p]] (Goto L s') (Goto M s') true s');		L: (S[[x]] (Goto N s) e s'); M: (S[[y]] (Normal s) e s'); N:
S[[if p then x else y]] c e s = (B[[p]] (Goto L s) (Goto M s) true s);		L: (S[[x]] c e s); M:(S[[y]] c e s)

$S[[x:= \text{if } p \text{ then } y \text{ else } z]] (\text{Normal } s') e s = (B[[p]] (\text{Goto } T \text{ s}) (\text{Goto } F \text{ s}) \text{true s});$
 $T: (S[[x:=y]] (\text{Goto } N \text{ s}') e s); F: (S[[x:=z]]$
 $(\text{Normal } s') e s); N:$

$S[[x:= \text{if } p \text{ then } y \text{ else } z]] c e s = (B[[p]] (\text{Goto } T \text{ s}) (\text{Goto } F \text{ s}) \text{true s});$
 $T: (S[[x:=y]] c e s); F: (S[[x:=z]] c e s)$

$S[[x:=y]] c e S = \text{MOVE } (e[[y]])\sim(e[[x]]): (C c s) \quad \text{if } y \text{ is a variable}$
 $= E[[y]] e ; \text{POP } (e[[x]]); (C c s) \quad \text{otherwise}$

$S[[I: C]] (\text{Normal } s) e s' = N: (S[[C]] (\text{Normal } s) (e[I\sim(\text{Goto } N \text{ s}', \text{Goto } N' \text{ s}'))); N':$

$S[[I: C]] C e s = N: (S[[C]] c (e[I\sim(\text{Goto } N \text{ s}, c)]))$

$S[[\text{redo } I]] c e s = C \text{first}(e[[I]]) s$

$S[[\text{leave } I]] c e s = C \text{second}(e[[I]]) s$

$S[[\text{call } p]] \text{Proc } e s = \text{DISCARD } s; \text{JUMP } (e p)$

$S[[\text{call } p]] c e s = \text{CALL } (e p); (C c s)$

$S[[i;j]] c e s = (S[[i]] c' e s) \text{ if } \text{isContinuation}[[j]] c e S \text{ where } c' = \text{continuationof}[[j]] c e c S[[i;j]] c$
 $e S = (S[[i]] (\text{Normal } s) e s) ; (S[[j]] C e s)$

$S[[\text{while } p \text{ do } t]] (\text{Normal } s) e s' = R: (B[[p]] (\text{Goto } L \text{ s}') (\text{Goto } M \text{ s}') \text{true s}');$

$L: (S[[t]] (\text{Goto } R \text{ s}') e s'); M: \text{DISCARD } (s'-s)$

$S[[\text{while } p \text{ do } t]] \text{Proc } e S = R: (B[[p]] (\text{Goto } L \text{ s}) \text{Proc true s}); L: (S[[t]] (\text{Goto } R \text{ s}) e s) S[[\text{while } p \text{ do } t]]$
 $(\text{Goto } c \text{ s}') e S = R: (B[[p]] (\text{Goto } L \text{ s}) (\text{Goto } c \text{ s}') \text{true s}); L: (S[[t]] (\text{Goto } R \text{ s}) e s)$

If s=s

= R: (B[[p]] (Goto L s') (Goto M s) true s');

L:(S[[t]] (Goto R s') e s'); M: DISCARD (s'-s); JUMP c

otherwise

$S[[\text{unconditional statement } u]] C e S = \dots \text{code for } u \dots; (C c s)$

(these rules for B require S and s', when both are present, to be the same)

$B[[v]] t (\text{Goto } j \text{ s}) \text{true s}' = \text{TEST } v; \text{JUMPFALSE } j$

$B[[v]] (\text{Goto } j \text{ s}) \text{f false s}' = \text{TEST } v; \text{JUMPTRUE } j$

$B[[v]] t \text{Proc true } S = \text{TEST } v; \text{JUMPTRUE } L; \text{RETURN}; L:$

$B[[v]] \text{Proc f false } S = \text{TEST } v; \text{JUMPFALSE } L; \text{RETURN}; L:$

$B[[\text{True}]] t \text{f true } s = \text{NO-OP}$

$B[[\text{True}]] t \text{f false } s = C t s$

$B[[\text{False}]] t \text{f true } s = C f s$

$B[[\text{False}]] t \text{f false } S = \text{NO-OP}$

$B[[p \text{ and } q]] t \text{f n s} = (B[[p]] (\text{Goto } N \text{ s}) \text{f true s}); N: (B[[q]] t \text{f n s})$

$B[[p \text{ or } q]] t \text{f n s} = (B[[p]] t (\text{Goto } N \text{ s}) \text{false s}); N: (B[[q]] t \text{f n s})$

$B[[\text{not } p]] t \text{f n s} = B[[p]] \text{f t } (\sim n) s$

$E[[n]] e = \text{PUSH } \%n \quad \text{If } n \text{ is a number}$

$E[[v]] e = \text{PUSH } (e[[v]]) \quad \text{if } v \text{ is a local variable}$

$E[[x+y]] e = (E[[x]] e); (E[[y]] e); \text{ADD}$

