

Chapter 3.1 — Expressions

An expression, typically, stands for a value; the formula is evaluated, not executed. It must be translated into a target machine instruction sequence that will be executed, for its effect. We have to decide what effect is appropriate. Once a decision has been taken, it's easy to see how *side-effects* are incorporated into the chosen scheme, and why language designers are sometimes reluctant to specify exactly what should be the resulting of evaluating an expression with side-effects.

The simplest idea is that execution of the translated code should change the state of the machine so that the stack grows by one value. *No other changes to the state of the machine should be made.* Values are calculated in order to be consumed (perhaps by a statement, when the value is printed or assigned to a destination location, or by further calculation yielding new values). So the code sequences for expressions will always be embedded in a longer sequence that removes the formula's value from the stack. In the case of an assignment statement with a simple variable as the destination, the expression valued will simply be popped into the appropriate location.

Many expressions include function calls, and many Pascal 'functions' have bodies that change certain parts of the state. Toy has no procedures or functions, so won't be thinking about such cases in this chapter, and we don't have to devise a specification of expression translation that allows for such side-effects.

Many languages, like Toy, are defined so that their *integer* type has exactly the properties of 2s-complement 'integer' arithmetic on common hardware. The idea, of course, is that each Pascal *integer* value, or C *int* value, (or Modula-2, or..) can be stored in a small, easily addressed sequence of bits. On most machines at present, 32 bits are used, although the language definition may permit as few as 16 bits. Some languages don't specify an upper limit, but it's unusual for an implementor to differ from the 'natural' size for the target machine. We will use one byte for character and boolean values, and four bytes for integer values.

The following rules although specific to Toy are suitable for translating a large subset of C or Pascal expressions. They include the translation of operators similar to C's address-forming and indirection operators & and *, in certain simple situations.

Integer Constants

Rule 3.1. $TE[N]$ = `push1 %N`

The *syntactic brackets* or *semi-quotes* [] are used so that it is easy to see what is source language and what is implementation language. This convention allows me to write the source language construct under consideration in the usual way – rather as italics are used in a sentence such as

Where in English one says *adjective noun* as in "pretty compiler", in French one says *noun adjective* as in "something else".

We don't use ordinary quotes in the translation rules because we want to be able to talk about whole classes of source-language phrases in a single rule – rule 3.1 doesn't mean literally "N", it means the whole class of things which have the form N (numerals).

To implement the translation rule **TE** in ML, we have to be explicit about the tree-like abstract syntax:

```
datatype expr = Ident of string | Numb of int
              | Plus of expr*expr | .....
```

There is one alternative for each kind of expression; I have shown only the first three as an illustration.

Underlining is used on the right hand side of the rule to indicate target language constructs. The source language syntax class *N* shown on the left in syntactic brackets, appears on the right as part of a target language instruction. The rule doesn't include any conversions that may be necessary – for example, the source language may use decimal constants while the assembler insists on octal or hexadecimal ones. I've also omitted the necessary checks on the size of the constant. The assembler instruction chosen is a *long* variant, so I'm allowing 32 bit integers: if the source language permits integers to be restricted to 32 bits there must still, in practice, be a compile-time check that the constant isn't too big. If the source language doesn't permit the restriction, a more complicated translation must be used. For example, a multi-word integer library package may have to be written, and the arithmetic expressions compiled into function calls.

We must decide exactly what we want to be the result of executing an implementation of the **TE** rule. Obviously we don't want to calculate the value of the expression, we want to generate a sequence of instructions. But what representation should we choose for an instruction? And how should we represent the sequence? An attractive idea that will be enough for our present purposes is to represent instructions as strings, and use a string list to represent complete target language programs. We define two functions

```
fun gens = [s]
infix ++
fun i++j = i@j
```

Now the function **TE** can be given type *expr->string list*. The ML version of the first rule in **TE** can be written

```
fun TE(Numb n) = gens("pushl %" ^ makestring n)
```

It is clear that the *N* in the abstract version of the rule is acting as a pattern-variable in the usual functional programming sense. The ML rule has to include a constructor (*Numb*) to restrict the translation to work only on numbers. In

the abstract version, we rely on our knowledge of the syntax classes of Toy to determine that N is a variable that matches only one kind of expression, rather than anything at all.

The ML implementations of subsequent rules make use of ++ to concatenate the instruction lists specified by the separate parts of the right-hand-sides.

Arithmetic operations

Rule 3.2 $TE[E1+E2]$ = $TE[E1]; TE[E2]; \text{add1}(sp)+, (sp)$

Rule 3.3 $TE[E1-E2]$ = $TE[E1]; TE[E2]; \text{sub1}(sp)+, (sp)$

The target SF4 instructions in these rules perform stack based arithmetic. They use auto-increment mode, so that the first operand is popped from the top of the stack. The second operand, which is also the destination, is the value underneath the first. The net effect is of a zero-address **add** (or **sub**) instruction. The other operations which correspond closely to the primitive instructions of the hardware can be implemented similarly.

Example

The ML implementation of the translation scheme can be simply extended with addition rules:

```
fun TE(Numb n) =
    gen("pushl %" ^ makestring n)
|   TE(Plus(E1,E2)) =
    TE E1 ++ TE E2 ++ gen"addl (sp)+, (sp)"
```

Multiplications are, as for many real machines, a little tricky. Given four byte operands, the multiply instruction produces an eight byte result. If you try to use stack addressing as above, the most significant and least significant parts aren't put in particularly appropriate places. With register addressing the behaviour is easier to understand, so I would translate multiplications as follows:

Rule 3.4 $TE[E1 * E2]$ = $TE[E1]; TE[E2]; \text{copy1}(sp)+, r1; \text{mull}(sp), r1; \text{copy1}(r1), (sp)$

this changes r2, as well as r1. ¹

Given a translation scheme it is very easy to calculate, on paper, the instruction sequence which will be generated for any particular source program. You have to mentally parse the source language phrases in order to discover which operator is at the root of the tree – otherwise you will pick the wrong translation rule.

Example

¹Acknowledgements to Bruce Chapman for pointing out the need for special treatment of multiplication on this machine.

The second + sign in the expression (1+2*3+4) is at the root of the parse tree. Draw the derivation tree using a grammar of expressions, terms and numbers in order to verify this.

```

TE[[ 1+2*3+4]]
=   TE[[ 1+2*3]] ; TE[[ 4]] ; add1 (sp)+, (sp)
=   TE[[ 1]] ; TE[[ 2*3]] ; add1 (sp)+, (sp);
    TE[[ 4]] ; add1 (sp)+, (sp)
=   TE[[ 1]] ; TE[[ 2]] ; TE[[ 3]] ;
    copy1 (sp)+,r1; mull (sp),r1; copyl r1,(sp); add1 (sp)+, (sp);
    TE[[ 4]] ; add1 (sp)+, (sp)
=   TE[[ 1]] ; TE[[ 2]] ; TE[[ 3]] ;
    copy1 (sp)+,r1; mull (sp),r1; copyl r1,(sp); add1 (sp)+, (sp);
    TE[[ 4]] ; add1 (sp)+, (sp)
=   pushl %1; pushl %2; pushl %3;
    copy1 (sp)+,r1; mull (sp),r1; copyl r1,(sp); add1 (sp)+, (sp);
    pushl %4; add1 (sp)+, (sp)

```

Convince yourself that the net effect of this code is the same as **pushl %11**.

Boolean Constants

In Toy, boolean formulas can be evaluated in the same manner as integer formulas, using a standard representation such as a zero-filled byte for *false*, and a byte with at least one bit set for *true*.

Rule 3.5. $TE[[\text{true}]] = \text{pushB } \%1$

Rule 3.6. $TE[[\text{false}]] = \text{pushB } \%0$

Boolean Operations

The boolean connectives are implemented as stack operations, just like arithmetic operators. A consequence is that both operands of *or* (*and*) are evaluated, even when the value of the left operand seems to make it unnecessary. We won't consider function or procedure calls just yet, but it's clear that given a 'function' such as

```
fun p(n) = (print n; n=0);
```

if the source language specifies that *or* evaluates both its arguments, we must make sure that the evaluation of the expression

```
p(3) or p(0)
```

prints out both numbers as 'side effects', not just the first.

Rule 3.7. $TE[[P \text{ or } Q]] = TE[[P]] ; TE[[Q]] ; \text{orb (sp)+, (sp)}$

Rule 3.8. $TE[[P \text{ and } Q]] = TE[[P]] ; TE[[Q]] ; \text{andb (sp)+, (sp)}$

Some source languages specify a 'conditional' or *left-strict* interpretation of

boolean formulas, in which the right operand isn't evaluated if the left operand is true (for *or*) or if it is false (for *and*). We'll look at various ways of doing this later. The above rules are correct for Pascal, which doesn't have conditional boolean connectives. They are almost right for C's infix *&* (*and*) and *|* (*or*) operators. The latter are required to perform the boolean operation bitwise on their arguments (which my rules do), but C doesn't have a boolean type, so for that language the operators would be working on integer values and should specify 32bit instructions.

The convention used by translation schemes which involve labels is that whenever the right-hand-side of a rule includes *upper case* labels, fresh values of these labels are used for each application of the rule. An implementation would have to choose new names for each boolean formula it translates, so that the labels don't clash. One way this can be done is shown below, and study of this implementation should make the principle clear.

Rule 3.9. $TE[E1 = E2]$
 $= TE[E1]; TE[E2]; comp1 (sp)+, (sp)+; jeq L0;$
 $pushB \%0; jump L1; L0: pushB \%1; L1:$

The comparison of the values of the arithmetic expressions E1 and E2 is done in two stages: first, there is code for the evaluation of E1 and E2, then there is code to compare the resulting values. In this section, to keep the rules as simple as possible, we use the same scheme for arithmetic expressions no matter in what context they appear. The code for each expression causes a long to be pushed; only arithmetic values can be compared in Toy, not characters or booleans.

To be consistent with the other kinds of boolean valued formula, the net effect of the comparison must be to push a byte to represent the boolean result. So two longs get pushed, both are popped again, then either *%1* (for true) or *%0* (for false) get pushed. It's a very complicated (and *long!*) sequence of instructions, and you can surely do better much of the time — particularly when the comparison is being used to control a choice or iteration formula.

Example

$TE[4 = 0]$
 $= TE[4]; TE[0]; comp1 (sp)+, (sp)+; jeq L0;$
 $pushB \%0; jump L1; L0: pushB \%1; L1:$
 $= pushl \%4; pushl \%0; comp1 (sp)+, (sp)+; jeq L0;$
 $pushB \%0; jump L1; L0: pushB \%1; L1:$

Generating Labels

The ML translation of this part of the **TE** scheme shows how the new labels can be generated. The 'function' *newLabel* returns a different number each time it is called. We make this slight compromise of our principles in order to avoid (for the moment) complicating **TE** with additional parameters.

```

val labelCounter = ref(0);
fun newLabel():string = (currentLabel:=!currentLabel+1;
                        "L" ^ makestring(!currentLabel));

fun TE(EQ(E1,E2)) =
  let val aNewLabel = newLabel();
      val anotherNewLabel = newLabel();
  in
    TE(E) ++ TE(F) ++
    gen("comp1 (sp)+, (sp)+\n") ++
    gen("jeq " ^ aNewLabel) ++
    gen("pushB %0") ++
    gen("jump " ^ anotherNewLabel) ++
    gen(aNewLabel ^ ":") ++
    gen("pushB %1\n") ++
    gen(anotherNewLabel ^ ":")
  end

```

Some assemblers have a notion called **local labels** which saves you from making up fresh names each time.

Character Constants

Most assemblers will generate an ASCII bit pattern for a character constant, relieving the compiler writer from building the mapping explicitly, but we have to be careful. The assembler and source language may accept slightly different character sets, and the quoting conventions may be different. The back-slash convention for newlines and tabs is common (too common!) so that a C compiler implemented in C may have to read an input containing '\n' (exactly one slash) and generate an instruction that pushes exactly one character: but the *printf* statement that outputs the instruction will interpret \n as meaning 'start a new line', so we must write \\n in the *printf* call. This is much easier to get right than it sounds!

Rule 3.10. $TE[\text{'x'}] = \text{pushB \% 'x'}$.

Variables

We keep to our rule that whatever the value of an expression, **TE** generates code to push it onto the top of the stack. Since variables stand for locations, we in fact have a choice: should we push the contents of the location, or some value that represents the location itself? The obvious answer is that we want the contents of the location.

Rule 3.11. $TE[V] = \text{push1 } V$

when V is an integer variable. On the SF4 machine, a label used in the above fashion specifies direct addressing; the instruction accesses the contents of the location as required. If it was intended that variables in expressions stand for their location, and not their contents, we'd want some other addressing mode that would enable us to push the address. In fact, this is what we do for the addressing-forming operator; the translation rule is given later.

Rule 3.12. $TE[V] = \text{pushB } V$

when V is a character or boolean value.

For each variable we translate, we also generate a labelled assembler pseudo-instruction to obtain the location were the value of the variable can be stored. The sequence of labelled locations can be generated after translation is complete by printing out an assembly language pseudo-instruction for each variable in the lexical analyser's symbol table, if any. A particularly simple translator may not use a lexical analyser. For example, if variables are restricted to single letters (and there are no keywords) lexical analysis is unnecessary. In this case, the compiler can emit 26 pseudo-instructions regardless of whether or not they are all needed.

Example

```
TE[1+x*2]
= TE[1] ; TE[x*2] ; addl (sp)+,(sp)
= TE[1] ; TE[x] ; TE[2];
  copy1 (sp)+,r1 ; mull (sp),r1 ; copyl r1,(sp) ; addl (sp)+,(sp)
= pushl %1 ; pushl x ; pushl %2;
  copy1 (sp)+,r1 ; mull (sp),r1 ; copyl r1,(sp) ; addl (sp)+,(sp)
```

This code can be easily tested if your translator emits the instruction sequence

```
call __printnumber
stop
x: long 0
...the library procedure given at the start of the section...
```

after the code for the expression.

Since variables stand for store locations, they can be used on the left hand side of assignment statements (so-called **L-expressions**), in which case we aren't interested in the old contents of the container, only in putting something new into it: in fact, we want its address. See the next chapter under **assignment statements** for details.

The above discussion hasn't said what to do if an array variable appears in an expression. The most common case, of a subscripted array variable, is dealt with below, but we can get interesting and occasionally useful behaviour if allow unsubscripted array names as well. The rule is that such expressions stand

for the *byte* address of the first (0th) array element. This address is a 32bit 2s-complement value, and so the Toy addition operator can be used to calculate the address of any particular element within the array. Having found the address, the Toy indirection operator can be used to find the contents at that address. Indirection gives a 32 bit value in Toy

$$\text{TE}[\text{V}] = \text{pushL } \%V$$

Given an array of integers, it is possible to do *address arithmetic* in order to access a particular array element. Toy addresses are always byte addresses, regardless of the kind of array involved, so some care is needed. For example, *I is the 32bit value at the start of the array, which can also be accessed as I[0]; *(I+4) is the 32bit value four bytes from the start, which can also be accessed as I[1]; and *(I+8) is the 32 bit value 8 bytes on from the start of array I, that is, the contents of I[2].

It isn't in keeping with the spirit of Toy to make the translation of the operators such as + and prefix * depend on the types of their arguments. There isn't, therefore, any way of accessing a single byte value from a character array by using address arithmetic. Array subscripting must be used instead.

Array subscripting

Explicit subscripting is provided in Toy so that integer arrays can be accessed without the Toy programmer having to scale the subscripts explicitly, and so that character and boolean arrays can be accessed appropriately. In an expression, C[0] means the first character in C, and C[2] means the third; I[0] means the first integer in I and I[2] means the third. It's easy to get this right since the syntax of array subscripting in Toy makes sure that the array name, and therefore its type, is explicit in every subscript-expression.

one-dimensional arrays.

Space for the arrays can be statically allocated in the same way as for (lower case) scalar variables. We generate a labelled pseudo-instruction for each array, allocating 10 bytes or longs as appropriate. An array-indexing expression (on the right hand side of an assignment) such as J[n] refers to the contents of the n^{th} location in the array J; we can find the contents by calculating the address of the n^{th} location and indirecting through it.

$$\begin{aligned} \text{TE}[\text{V}[\text{E}]] &= \text{TE}[\text{E}]; \text{popL } r0; \text{addL } \%V,r0; \text{pushB } (r0) \\ &\quad \text{,if V is a 1-byte array var} \\ &= \text{TE}[\text{E}]; \text{popL } r0; \text{mull } \%4,r0; \text{addL } \%V,r0; \text{pushL } (r0) \\ &\quad \text{,if V is a 4-byte array var} \end{aligned}$$

Addresses of Store Locations

Known as *references* in ML, these values can be obtained by using the @

operator in Toy, which is like `&` in C. The most obvious imitation is to use the address of the container. The only possibility which we'll consider here is the address operator is applied to a variable.

We can use immediate addressing. Each global container is given a different store location. We must push the address (not the contents at that address!). Since we're translating variables as above, the variable name is also the label of its runtime location. On the SF4 machine, labels are treated exactly like (integer or unsigned) constants; the assembler simply works out the address of the labelled machine location. So to push the address, rather than the contents, immediate addressing is used

Rule 3.13 $TE[[@V]] = \text{pushL } \%V$

Some assemblers treat labels differently from constants: read the manual carefully!

Formulas which obtain the contents of a variable via its address (`*` in C, `^` in Pascal, `..`) will be translated to a code sequence that indirects through the address of the store location used for the variable. Indirect addressing must be via a register on the SF4 machine, so the address is popped into a register:

Rule 3.14. $TE[[[]E]] = TE[[E]]; \text{popL } r0; \text{pushL } (r0)$

This is correct even when the address expression E is a function call, as we'll see, but can easily be optimised for the more common case that E is a pointer variable holding the address. My rule generates code to push a 32 bit value, and therefore can't be used to handle one-byte characters: a quick fix would be to extend Toy to use a different operator symbol for each size of object. Later we'll see how the compiler can infer from the context which length to use.

N.B. the translation pushes the contents of the container described by E , so it isn't suitable for situations where you want the container itself — see the next chapter for how to translate down-arrow formulas on the left hand side of assignments — that is, in L-expressions.

Example

$TE[[[]@x]]$

Evaluation of this formula should give the same overall effect as evaluation of x when it appears in the same context. Let's assume we're seeking a value rather than a container. Using rule 3.14, we get

$TE[[@x]]; \text{popl } r0; \text{pushL } (r0)$

And then by rule 3.13 we get

$\text{pushL } \%x; \text{popL } r0; \text{pushL } (r0)$

The first instruction pushes the address of x . Next, the stack is popped into $r0$. This restores the stack to its previous state and leaves the address of x in $r0$. Finally, we push the contents of the location whose address is in $r0$; which is therefore the contents of x , as required.

Using Registers

The above translations always use the stack for storing any intermediate values that may be calculated. The stack can grow arbitrarily large, so there is no practical need to consider the possibility of stack overflow. Most machines have a relatively small number of registers, but if a source program is such that this small number is enough, it is sensible to use them. Even in a big program, short sequences of instructions that use the stack can be replaced by similar sequences that use registers. Chapter 3.3 describes a straightforward method of generating code that uses registers for intermediate results during expression evaluation.

Project

Build an expression translator for a language of integer expressions. Implement integer constants, addition, multiplication, subtraction and division. Provide 26 'variables' named a, b, c, ..., z. Each variable should be implemented as a labelled global storage location as described above. Devise a method for initialising the contents of the locations to various constant values (the next chapter explains how assignment statements can be translated – don't bother with that yet). Implement C's & and *, so that $\&x=4+\&y$, etc.. Suppose that x is initialised to 42, and y is initialised to 77. What should be the value of $*(4+\&y)$? Check the code produced by your translator by assembling and running it, with a suitable test program. C compilers use type information to scale similar address calculations according to the type of the pointer. We won't add this refinement until we've looked at symbol tables.

First do it without a lexical analyser, and then by making small changes to the parser and no changes to the translator, add a scanner module.

Chapter 3.2 – Translating Statements

The following sections describe the translation of formulas which are used for their effect rather than their value. The idea behind these translations is to produce sequences of SF4 instructions that, taken together, leave the size of the stack unchanged. They may make other changes – that's what statements are for! But printing instructions will only change the screen and assignment instructions will only change the store location that they are intended to change. Other temporary changes may be made to the stack, but if so, the change will be undone again. Some instructions change the registers. When these rules refer to registers, it is always to load a value into a register and then immediately use it. It doesn't matter what happens to the register after that.

Output statements

Pascal's *write* and *writeln* statements are relatively hard to compile, since the effect of the statement depends on the types of the arguments. The following translations don't depend on type information – they rely on the user choosing the appropriate printing method. I will first describe Toy's simple printing statements, and how to compose them, so that you can try some complete programs right away.

Rule 3.15. $TS[\text{printchar}(E)] = TE[E]; \text{call printchar}; \text{discardb } \%1$

The ML implementation uses procedure *gen* to build the list of instructions or to print them to an output file. Of course, in ML we must also extend the type used to implement the abstract syntax. The primitive printing instructions could be represented by constructors as in the following

datatype statement = | **PCharInstr** of expr
| **PNumblInstr** of expr |

The type *expr* is the one described in the previous chapter.

| **TS(PCharInstr E)** = (TE E ++
gen"call printchar" ++
gen"discardb %1")

Of course, if *E* turns out not to be a character-valued expression, the discard instruction will not do the right the thing: this Toy language has to be used with care.

Rule 3.16.

$TS[\text{printnumb}(E)] = TE[E]; \text{call printnumb}; \text{discard1 } \%1$

Rules 3.15 and 3.16 depend on the scheme for expressions in Chapter 7: the $TE[E]$ are the translations of the character valued formula *E* in rule 3.14 and the integer-valued formula *E* in rule 3.16. The rules can be used with any appropriate expression, no matter how simple or complicated, since the TE rules describe quite generally how to invent sequences of instructions that push exactly the required value.

Sequential Composition

Rule 3.17. $TS[[I; J]] = TS[[I]]; TS[[J]]$

And again, the ML datatype describing the abstract syntax would be extended to include an alternative

| **Seq** of statement*statement

Example

Using the rules above, programs can be translated step-by-step by a simple process of substitution. When the whole formula is underlined, the translation is complete :

$TS[[\text{printchar}(\backslash'n'); \text{printchar}('*')]]$
= $TS[[\text{printchar}(\backslash'n')]]; TS[[\text{printchar}('*')]]$ by
3.17
= $\text{pushB } \%'\backslash'n'; \text{call printchar}; \text{discardb } \%1$; $TS[[\text{printchar}('*')]]$
by
3.15
= $\text{pushB } \%'\backslash'n'; \text{call printchar}; \text{discardb } \%1$;
 $\text{pushB } \% '*'; \text{call printchar}; \text{discardb } \%1$ by
3.15

Assignment statements

These are perhaps the simplest statements to translate: the translation is a code sequence to evaluate the right hand side, followed by a pop to store the result in the correct place. We begin with the simplifying assumption that the destination is indicated with a simple variable name – later we will consider the possibility of ‘address arithmetic’ and assignment to array elements.

Rule 3.18. $TS[[V:=E]] = TE[[E]]; \text{pop1 } V$

The rule as given assumes the variable V is a four byte value. Otherwise, the rule is a general one. We can use the same scheme as in the previous chapter for encoding the type (length) of the variable in its name.

Example

In ML, rule 3.18 might be encoded as below. The function carefully stores single byte values into variables whose names require it; of course, if the expression size and variable size don't match, the function will not meet its specification – the part that requires it to generate code sequences whose execution does not change the size of the stack. Making sure that this doesn't happen is one of the jobs of the type-checking phase. Below I sketch a crude method of determining which variant of the pop instruction to use. Later, we will consider how to process declarations, which in most languages are the mechanism by which types are associated with variable names.

```

fun vartype(v:string) =
  if v>="i" andalso v<="n" then arith
  else if v="p" orelse v="q" then truth
  else silly;

fun TS(Assign(v,e) ) =
  (TE e;
   case vartype v of
     truth => gen("popb " ^ v)
   | arith => gen("popl " ^ v)
   | silly => error()
  )

```

When E is a particularly simple formula - a numerical constant, perhaps - a copy instruction may be used for the whole assignment. This is easy to add as a special case, using a more specific pattern (*numb N*) where the rule given uses E .

If the left hand side of the assignment is not a simple variable (perhaps it is an indirection), the translation is a little more complicated, as the left hand side must then be evaluated to find the store location which is to be updated. Formula $E1$ will yield an address, which is popped into $r0$ so that the final pop can indirect through it.

Rule 3.19. $TS[\llbracket E1:=E2 \rrbracket] = TE[E2]; TE[E1]; \text{pop1 } r0; \text{pop1 } (r0)$

Each call of TE pushes one long value and there are two pops to restore the stack.

Exercise

An array-indexing expression on the left hand side of an assignment can be treated in the same way, omitting only the final push instruction, so that a register is loaded with address of the destination location and the rhs value popped into it. Devise a suitable rhs for a rule for such assignments:

$TS[\llbracket V[E1] := E2 \rrbracket] = \dots$

Bounded Iteration

The Toy *times-do* statement consists of an expression and a body – another statement. It is executed by first evaluating the expression to obtain a number (four bytes), and then executing the body that number of times. The only problem is to devise a translation that works no matter how deeply nested the loops are. This scheme uses the stack to store the count of the number of iterations, rather than a register which would be faster but less general.

Rule 3.20. $TS[\text{times } E \text{ do } S]$
 $= TE[E]; L1: \text{compL } \%0,(\text{sp}); \text{jumpL } L2;$
 $TS[S]; \text{subL } \%1,(\text{sp}); \text{jump } L1;$
 $L2: \text{discardL } \%1$

I've used upper case labels L1 and L2 which must be uniquely generated each time this rule is used. This is done using the method described in the previous

chapter on translating expressions. The shape of this code is as follows:

```

..code for E, which pushes one long value..;
..code which does not change the stack size..;
..a discard instruction which removes one long value..

```

The execution of the code sequence $TE[E]$ will push one integer value (a long). The (execution of the) code sequence $TS[S]$ will leave the size of the stack unchanged. It is the translation of a statement, and only the translations of expressions have a net effect on the size of the stack. So the value of the loop counter will still be in the topmost stack location after each iteration, until it is discarded on exit from the loop. Note that this is an essence a proof by induction: we can see by inspection that the simplest statements are translated into code sequences that do not affect the stack size; we make the inductive assumption that the recursive call of TS also respects the stack size, and on the basis of this assumption observe that the code for a times-statement also respects the stack size.

The next rule is for Toy *for-do* loops, which are modelled on those of Pascal.

```

Rule 3.21  $TS[\text{for } i:= E1 \text{ to } E2 \text{ do } S]$ 
=  $TE[E1]; TE[E2];$ 
   $L1: \text{compL } (sp)4, (sp); \text{ jumpgt } L2;$ 
   $\text{copyL } (sp)4, i;$ 
   $TS[S]; \text{ addL } \%1, (sp)4; \text{ jump } L1;$ 
   $L2: \text{discardL } \%2$ 

```

The code sequence $TE[E1]; TE[E2]$ pushes two long values. The value of $E2$ will be left on top, so this is accessed using (sp) . The value of $E1$ is underneath, so this is accessed using $(sp)4$. The (copy of the) value of $E1$ is incremented until it 'overtakes' the value of $E2$. Both the pushed values are discarded again at the end of the loop, so this translation keeps to the rule: no net effect on the size of the stack from execution of a statement. Thus you can translate nested *for*-loops. Programming languages vary greatly in their bounded iteration constructs. In Algol68, loops similar to this introduces a new scope (which we'll therefore think about after we've looked at how local blocks and procedure calls are implemented). In Pascal, the loop variable is declared outside the loop, and gets updated by it – the same as in my translation. But, if the loop body is executed zero times (which happens when $E2$ is smaller than $E1$), the variable does not get updated.

It's often the case that the upper bound of the sequence, given by $E2$ is a constant. The compare instruction can refer to a constant directly, using immediate addressing. Then there is then no need to push $E2$, so the addressing mode for $E1$ is different, and the discard instruction is different too. It's instructive to consider what happens when $E1$ and $E2$ are not constant formulas – their values depend on the contents of one or more locations. What should be the meaning of a loop such as the following?

```
for i:=1 to n do n:=n+1
```

A naive answer is that it doesn't matter, as no-one would write such a silly statement. But in fact most programming languages specify a particular outcome. Toy is similar to Pascal: the loop bounds are evaluated once before the loop is entered, so that the number of iterations does not depend on the loop body.

An alternative language design is for the bounds to be re-evaluated each time around the loop. In fact, it would be incorrect to call such a loop behaviour *bounded* iteration, since they may fail to terminate.

The translation of statement S is indicated as usual by $TS[S]$. All occurrences of i which are free in S (in other words, all the ones which correspond to this particular loop counter) are translated in the usual way for a variable access. Until we look at procedures and local blocks, this is just as described in the previous chapter.

Exercise

For-loops in C have an unusual syntax and semantics. The syntax is
"for" "(" E1 ";" E2 ";" E3 ")" S

In C, assignment is treated as an operator with a side effect, so any one of these expression could directly update a counter. It isn't hard to extend TE to deal with assignment – the rule is similar to the TS rule for Pascal-style assignment, except that the value of the right hand side expression is copied rather than popped into the destination location.

The informal semantics of the loop construct is that E1 is evaluated (possibly initialising a counter), then E2 is evaluated; if E2 evaluates to zero, the loop terminates, otherwise the statement S is executed. After each execution of S, E3 is evaluated (possibly incrementing a counter) and then E2 is re-evaluated, causing additional executions of S until either the effects of S or the side-effects of E2 and E3 make E2 evaluate to zero.

Devise a translation scheme for C for-loops.

Unbounded Iteration

The only unbounded iteration construct we'll look at is the *while*-loop. The condition is a boolean expression. Using my rules for the translation of expressions, the code for this will push a one byte value onto the stack. So the following code checks to see if *false* was pushed, popping the byte at the same time.

Rule 3.22. $TS[\text{while } E \text{ do } S]$
= $\underline{L1}$: $TE[E]$; $\text{compB } \%0,(\text{sp})+$; $\text{jeq } \underline{L2}$;
 $TS[S]$; $\text{jump } \underline{L1}; \underline{L2}$:

Exercise

Devise an ML function for *translating* (don't worry about parsing!)

Dijkstra's multiple-guarded unbounded iteration construct, which is written as

```
do E1: S1  E2: S2  E3: S3  .... od
```

The semantics is such that all of the expressions E_n are evaluated. Suppose some subset of the expressions $\{E_i, E_j, \dots\}$ evaluates to true. Then exactly one of the statements $\{S_i, S_j, \dots\}$ is executed and another iteration is performed – the expressions are re-evaluated, etc. When all of the expressions evaluate to false, the loop terminates. Since there can be any number of guards E_n , it is convenient to use a list of guards and their controlled statements in the representation of a Dijkstra *do*-statement:

```
datatype statement
  = .... | iterate of (expr*statement) list | .....
```

Choice Formulas

In this section we'll concentrate on *if*-statements and *case*-statements. The rules for *if* will in fact work for choices between values (conditional expressions) too. The rules for *case* can be easily modified to work for expressions, although at present this is required only for functional languages.

If statements

If-then-else statements can be implemented with a single test, followed by a conditional jump. An unconditional jump after the code for the *then*-part makes sure the *else*-part doesn't get executed too:

Rule 3.23. $TS \llbracket \text{if } E_0 \text{ then } E_1 \text{ else } E_2 \rrbracket$
 $= \quad TE \llbracket E_0 \rrbracket ; \text{compB } \%1,(\text{sp})+; \text{jne } L1;$
 $\quad TS \llbracket E_1 \rrbracket ; \text{jump } L2;$
 $\quad \underline{L1}: TS \llbracket E_2 \rrbracket ;$
 $\quad \underline{L2}:$

The condition part of an *if*-statement is probably the most frequent context in which boolean valued formulas are used. Some source languages provide special operators or connectives for use in this situation, that short-circuit evaluation of the condition. For example, in the ML expression

```
if p orelse q then i else j
```

it isn't necessary (and in fact it's wrong) to evaluate q if p is true; instead, the code for evaluation of q should be skipped over and the code for i executed immediately. Consider

```
if x=0 orelse y/x = 1 then i else j
```

A correct implementation of *orelse* must not cause a 'divide by zero' exception or interrupt for this statement.

It's quite tricky to generate optimal *skipping code* – code which does the

minimum amount of jumping, so we leave that to another chapter.

Suppose we use TE, with the same specification as before, to translate the arguments of *andalso* and *orelse*. Then the translation of *if*-statements provides a clue to the translation of the connectives, for

```
p andalso q  <=>  if p then q else false
p orelse q   <=>  if p then true  else q
```

You may like to confirm this by writing down the truth tables. Using the scheme above, we can calculate

```
TS[[if p then q else false]] = TE[[p]]; compB %1,(sp)+;
                               jne E;
                               TS[[q]]; jump F;
                               E:TS[[false]];
                               F:
```

A similar calculation may be done for *orelse*. We can therefore translate the conditional connectives

```
TE[[E1 andalso E2]] = TE[[E1]]; compB %1,(sp)+;
                     jne E;
                     TE[[E2]]; jump F;
                     E:pushB %0;
                     F:
```

```
TE[[E1 orelse E2]] = TE[[E1]]; compB %1,(sp)+;
                   jne E;
                   pushB %1; jump F;
                   E:TE[[E2]];
                   F:
```

Case formulas

Pascal insists that the guards in *case*-statements are constants and that the alternatives are statements. The simplest implementation is to treat it like a nest of if-then-elses, an idea that works even when the guards are not constants.

```
TS[[case E0 of E1 : S1; E2 : S2; ...; En : Sn end]]
= TE[[E0]];
  N1: TE[[E1]]; compL (sp)+,(sp); jne N2; TS[[S1]]; jump Nn+2;
  N2: TE[[E2]]; compL (sp)+,(sp); jne N3; TS[[S2]]; jump Nn+2;
  ...
  Nn: TE[[En]]; compL (sp)+,(sp); jne Nn+1; TS[[Sn]]; jump Nn+2;
  Nn+1: call blowup
  Nn+2: discardL %1
```

First the value of the controlling expression is pushed, then it is compared, in turn, with each of the constants. It is finally discarded only when execution of the whole *case*-statement is complete. This translation doesn't work if the *case* is an expression, because then the S_i would be expressions and their translations would cause values to be pushed on top of the value of E . The discard instruction at the end of the translation would discard the **result**, not the value of E . You can adapt the idea easily enough, though: simply replace `discard1 %1` by `pop1 (sp)4`.

A possible ML datatype for the abstract syntax of *case* -statements is a *(int*statement) list*; with one list element for each arm. This is particularly suitable for implementing the above translation scheme. Using list recursion, the base case is an empty list of arms (generate `call blowup`); the induction step is to generate code to test and conditionally execute for the first case in the list, branching to code for the remaining arms if the test is unsuccessful.

Exercise

Suppose it is decided to extend Pascal to include *case*-expressions as well as *case*-statements. Use the adapted version of the rule to find the result of

TS[[x:=case 1 of 0: 5; 1: 7 end]]

Trace the execution of the resulting code (on paper if necessary). Does x receive the correct value? Is there any net effect on the size of the stack?

—

There are, of course, better and cleverer ways to translate *case*- statements, particularly when the E_i are number constants. The jump tables mentioned in 'Notes on machine level programming' are one such way. Another is binary search.

binary search

When a case statement is very large, a linear sequence of tests is very slow. For a case-statement with N arms, binary search can find the correct arm in $\log_2(N)$ tests. In the abstract syntax, the list of pairs *int*statement* used to represent the case-statement is very reminiscent of a dictionary data structure: it can be thought of as a map from the values in the guards (each is of course different from all the others) to the statements. Suppose then that the case is represented by a balanced binary tree, perhaps an AVL tree. [for example, see Sedgewick, 'Algorithms'].

datatype statement = | **casestat** of expr*bintree |
and bintree = **empty** | **node** of bintree*int*statement*bintree

```

fun trancase(chooser,body) =
  let fun trantree empty =
        jump blowup
      | trantree (node(empty,g,s,empty)) =
        compL (sp),%g;
        jne blowup;
        TS s;
        jump D
      | trantree (node(L,g,s,R)) =
        (* S,G are new labels in every recursive call *)
        compL (sp),%g;
        jlt S;
        jgt G;
        TS s;
        jump D;
        S: trantree L;
        G: trantree R
    in
      val D=newLabel()
    in
      tranexpr chooser; trantree body;
      D: discardL %1
    end
end

```

The integer at the root of the tree is the median value of all of the guards; the statement is the one that it guards. A suitable translation function is given above in pseudo-ML, omitting the detail of the code generation functions. The label blowup is the entry point of an assembler routine to produce a runtime error message: it is used when the expression controlling the case-statement evaluates to an integer which is not one of the guards.

Of course it isn't necessary to use an AVL tree to obtain the same results. For example, if a list of pairs is used, this can be sorted using *quicksort* in the same time as it takes to construct an AVL tree. The median element of the list is the first that should be tested for, after which all the list elements before the median, then all the list elements after the median, can be translated recursively.

Exercise

For very short case statements, the binary search code is longer and perhaps slower than the linear, if-then-else style of code. Produce a hybrid of the two schemes, that given a small tree generates linear code, and given a large tree generates binary search code. Hint: modify *trantree* so that when the tree is small, it calls the 'linear' procedure. Think of it as an extra base case.

Dijkstra's guarded commands: select formulas

These are just like *iterate*-formulas, except that after execution of a chosen arm they jump to the end of the code sequence not the start; and if no guards are true, they call a procedure to print an error message. Each of the guards is a

boolean formula, so the translation scheme will generate code to push one byte, hence the **compB** instructions.

```

select
P1: F1          N1:  T[[P1]]; compB %1,(sp)+; jne N2; T[[F1]];
                    jump Nn+2;
P2 F2          N2:  T[[P2]]; compB %1,(sp)+; jne N3; T[[F2]];
                    jump Nn+2;
...
Pn: Fn          ...
                    Nn:  T[[Pn]]; compB %1,(sp)+; jne Nn+1; T[[Fn]];
                    jump Nn+2;
                    Nn+1: call blowup
endselect          Nn+2:

```

This is a particularly simple translation; used without thought, it will frequently cause values to be pushed and then immediately popped again by the *comp* instructions. If the conditions P_i are comparisons, you can do much better by encoding them directly; see especially the description below on the translation of conditional expressions.

Jumping statements - goto, break, and continue.

It's easy to translate *goto*-statements for a language like Toy, of statements and expressions in which all variables are global. The statements themselves are modelled closely on the unconditional jump instructions of the von Neumann machine, and so are translated directly into JUMPs. Similarly, labelled statements are translated into labelled instruction sequences.

There is a little more to it, however, if the source language permits the local declaration of variables. For then a *goto* may transfer control from a statement in which there is a variable x , say, to another statement where there isn't. If the storage for variables is allocated dynamically on block entry and deallocated on block exit, we must make sure that the code for a *goto*-statement performs deallocation for all the variables in all of the blocks that are exited by the jump. If *goto*-statements are permitted to transfer control into the middle of a block – so that there are more variables in scope after the jump than before – allocation and perhaps initialisation may also need to be done. In practice most language descriptions forbid jumps into blocks, so this is rarely a problem. It is even more complicated when *gotos* are permitted to transfer control from one procedure to another, for then the de-allocation that is necessary depends on the particular run-time behaviour of the program: consider a recursive procedure that jumps to a label in the procedure that called it! There is some discussion of these problems in Section 4.

In the spirit of our toy language, we'll assume that all source language labels start with the letter L, and that these can be output as target language labels without confusion, just as the global variable names are used to label store locations in the output program.

$$T[\text{goto } L] = \text{JUMP } L$$
$$T[L: S] = \underline{L}; T[S]$$

Project

Extend your expression parser/ translator (the project exercise in Chapter 3.1) to handle a selection of the statement constructs of Toy.

Include a simple *print*-statement for integer expressions. Translation of Pascal's write-statement requires information about the types of the expressions, which we don't have yet, so there is no need to attempt that.

If you don't have a lexical analyser ready, try using single upper-case letters for 'keywords': in this little language each keyword starts with a different letter. In this case, use "=" for ":", "|" for "or" and "&" for "and".

Optional extra: implement the one-dimensional arrays described in this chapter [3.1 or 3.2?].

Chapter 3.3 – Simple Optimisation

The translations we've studied so far have emphasised correctness rather than efficiency. The implementations of procedure calling in Section 4 are quite good – much more analysis of the source program is needed to do significantly better – but the implementation of expressions is particularly poor. It is also quite easy to do a lot better.

We'll look first at two elegant translation ideas. One is a short set of rules for use when an expression only contains strict operators (such as arithmetic, comparison, bitwise and strict boolean operators). These operators produce undefined results if any of their operands are undefined, so that it is appropriate to fully evaluate the operands before executing any code for the operation. Most expressions used in assignment statements and as argument formulas in procedure calls involve only strict operators. The other method is for translating the left-strict boolean operators (ML's *andalso*, *orelse*, C's *&&*, *||*) and conditional expressions (ML's *if-then-else*, C's *p?:e:f*). These are most often seen as the condition in *if*-statements and *while*-statements. Because these expressions may be defined even when one of their components is undefined, it is appropriate to interleave the evaluation of their operands with the code for the operation. To get the best results in this latter case, it is necessary to reconsider the translations of *if*-statements and *while*-statements.

Taking advantage of Associativity

The procedure *readE* is based on the EBNF rule

$$E = T ("+" T)^*$$

Instead of building a binary tree of “+” nodes, *readexpression* may build a list of trees, one for each term, as a single node.

```
datatype formula
  = name of char
  | numb of integer
  | bigadd of formula list
  | etc
```

Now, the data structure for $x+1+2$ would be **bigadd**([**name**(‘x’), **numb**(1), **numb**(2)]). A function can be written to calculate the sum of all the **numb** nodes in such a list. A procedure can be written to print out all the remaining nodes in postfix form.

```
printaddoperands(nil) = donothing
printaddoperands(cons(f, fs)) = postorder(f); aux(fs)
```

If the list has more than two trees in it, we wish to print each tree, except for the first, immediately followed by a “+” symbol, as in

```
ab+c+d+e+
```

since given this form, stack evaluation can be completed with a stack of maximum size 2.

```
aux(nil)=donothing
aux(cons(f,fs)) = postorder(f); printstring("+"); aux(fs)
```

Given a list which contains a mixture of **numb** nodes and others, we can print the shortest equivalent postfix expression

```
printshort(L) = printaddoperands(M); printnumber(sum); printstring("+")
    where M = ..a list of those elements of L which are not numb nodes..
           sum = ..the sum of those elements of L which are numb nodes..
```

Exercise

Complete the definition of *printshort*. Calculate the output printed by

```
printshort([numb(1), name('x'), numb(2), name('y')]).
```

Evaluating expressions using registers

The Sethi-Ullman expression translation algorithm is in widespread use. It is simple to implement, provides optimal code within a certain class of applications, and facilitates 'targeting' of expression evaluation, so that it is usually not necessary to copy the result from the register into which it is computed. The important weakness of the algorithm is that it assumes that the variables in the expression are implemented as store locations, and not as registers. Translation schemes that cache some variables in registers are usually much more complicated – the optimal choice of which variables to cache depends on the choice of instructions for evaluation of the expressions in which they are used, which in turn affects the registers which are available for caching. Although making an optimal choice is in fact impractical, and making a merely very good choice is hard, a reasonable choice is to reserve some registers for use as temporaries during expression evaluation – using the S-U algorithm given below – with others dedicated for holding the most frequently used variables.

The Sethi-Ullman algorithm

The algorithm is developed from an analysis of the number of registers needed for the evaluation, without stores to temporary non-register locations, of expressions of the available forms. Expressions consisting of a single variable (shown in the abstract syntax as v) can be evaluated by loading one register. Binary expressions where the right operand is a variable can be evaluated by first evaluating the left operand (perhaps a complicated expression), but then simply modifying the value in the result register using a memory to register operation. Binary operations involving two complicated operands can be evaluated in the same number of registers as the more complicated of the operands – the more complex is evaluated first and saved in one register, then the other is evaluated using the remaining registers. If both operands require the same number of registers, the binary expression will need an additional one

to hold the value of one subexpression while the other is evaluated.

$$\begin{aligned}
 \mathbf{N}[\![v]\!] &= 1 \\
 \mathbf{N}[\![E+v]\!] &= \mathbf{N}[\![E]\!] \\
 \mathbf{N}[\![E+F]\!] &= f \quad ,\text{if } e < f \\
 &= e+1 \quad ,\text{if } e = f \\
 &= e \quad ,\text{if } e > f
 \end{aligned}$$

where $(e,f) = (\mathbf{N}[\![E]\!], \mathbf{N}[\![F]\!])$

Where two cases 'overlap', the first rule is used in preference.

The translation function $E:expression^*register\ list \rightarrow code$ uses contextual information, a list of available registers, to translate expressions using the minimum number of registers and the minimum number of **copy** instructions; that is, register to memory moves.

The specification of E requires it to generate code that, when executed, stores the value of E into the register at the head of the list. The remaining registers in the list can have their contents altered, but no other registers can be changed. An unlimited supply of store locations, possibly a stack, is available for intermediate results if the number of free registers is insufficient.

There are several cases to consider, since we have to face the possibility that the number of free registers is too few to avoid stores altogether. A non-commuting operator is used to illustrate the algorithm, rather than the usual addition operator, to emphasize that non-commutativity poses no special difficulty. The instruction $SUB\ T1\ r$ can be understood, and pronounced, 'subtract $T1$ from r '.

$$\begin{aligned}
 \mathbf{E}[\![v]\!] (r::rs) &= \text{copy } v\ r \\
 \mathbf{E}[\![E-v]\!] (r::rs) &= (\mathbf{E}[\![E]\!] (r::rs)); \text{sub } v\ r \\
 \mathbf{E}[\![E-F]\!] (r::rs) &= (\mathbf{E}[\![F]\!] (r::rs)); \text{copy } r\ T1; (\mathbf{E}[\![E]\!] (r::rs)); \text{sub } T1\ r \\
 &\quad \text{if } \mathbf{N}[\![E]\!] \geq \text{len}(r::rs) \text{ and } \mathbf{N}[\![F]\!] \geq \text{len}(r::rs) \\
 \mathbf{E}[\![E-F]\!] (r0::r1::rs) &= (\mathbf{E}[\![E]\!] (r0::r1::rs)); (\mathbf{E}[\![F]\!] (r1::rs)); \text{sub } r1\ r0 \\
 &\quad \text{if } \mathbf{N}[\![E]\!] > \mathbf{N}[\![F]\!] \\
 \mathbf{E}[\![E-F]\!] (r0::r1::rs) &= (\mathbf{E}[\![F]\!] (r1::r0::rs)); (\mathbf{E}[\![E]\!] (r0::rs)); \text{sub } r1\ r0 \\
 &\quad \text{if } \mathbf{N}[\![E]\!] \leq \mathbf{N}[\![F]\!]
 \end{aligned}$$

The target language identifier $T1$ is used to indicate a new temporary variable; we do not show the mechanism by which recursive applications of this translation rule generate new variables. One possibility is to use push r instead of copy $r\ T1$, and correspondingly sub $(sp)+,r$ instead of sub $T1\ r$. It isn't necessary to use the stack for the temporaries; globals could be used if direct addressing is faster than indexed addressing on the target machine.

Correctness is proved by induction on the structure of the source expression. Of course, the translation function must be called with a non-empty list. Validity of the base case is obvious, and the others are almost as simple. One point that

may require explanation is exhaustiveness of the final three rules: only one of the three can be used for lists of length one, and it is guarded by an additional test. The remaining rules require longer register lists. How can we be certain that one of the rules is applicable?

$N[E] \geq 1$ for any formula E. Therefore, if rule (3) is not chosen because either $N[E]$ or $N[F]$ is less than the length of the list, the list must be at least two registers long and rule (4) or (5) can be used.

Example

The above rule generates code with no moves for many simple expressions:

```

E[(a-b)+(c*d+e*f)] [r0,r1]
  □ (E[c+d+e*f] [r1,r0]); (E[a-b] [r0]); ADD r1[] r0
  □ (E[e*f] [r0,r1]); (E[c*d] [r1]); ADD r0[] r1; (E[a-b] [r0]); ADD r1[] r0
  □ (E[e*f] [r0,r1]); (E[c*d] [r1]); ADD r0[] r1;
    (E[a] [r0]);SUB b[] r0; ADD r1[] r0
  □ (E[e*f] [r0,r1]); (E[c*d] [r1]); ADD r0[] r1;
    LOAD a[] r0; SUB b[] r0; ADD r1[] r0
  □ (E[e*f] [r0,r1]); (E[c] [r1]); MUL d[] r1; ADD r0[] r1;
    LOAD a[] r0; SUB b[] r0; ADD r1[] r0
  □ (E[e*f] [r0,r1]); LOAD c[] r1; MUL d[] r1; ADD r0[] r1;
    LOAD a[] r0; SUB b[] r0; ADD r1[] r0
  □ (E[e] [r0,r1]); MUL f[] r0; LOAD c[] r1; MUL d[] r1; ADD r0[] r1;
    LOAD a[] r0; SUB b[] r0; ADD r1[] r0
  □ LOAD e[] r0; MUL f[] r0; LOAD c[] r1; MUL d[] r1; ADD r0[] r1;
    LOAD a[] r0; SUB b[] r0; ADD r1[] r0

```

When translating large expressions, a naive implementation of this algorithm will lead to a very large number of calls of N. This can be avoided by labelling each node of the expression tree with the appropriate number using a single walk of the expression tree.

Optimising Control Flow

Our earlier translation of statements such as

```
if x<>0 andalso y div x = 1 then S else T
```

produces very poor code, in which many conditional jumps are made to control the pushing of truth values, which are then popped only to be used to control more conditional jumps. The code that one would write by hand is, by comparison, fast and obvious

```

compL x,%0
jeq elselab
copyL y,r0

```

```

divL x,r0
compL r0,1
jne elselab
...S...
jump finishlab
elselab:
...T...
finishlab:

```

Boolean Expressions

Martyn Richards, the designer of BCPL, devised a procedure which translated a boolean expression relative to a label and a boolean value. The generated code jumps to the label if the expression evaluates to the same sense as the boolean value, and falls through if not.

Richard Bornat has described a modification of Richards' algorithm, which produces the same target code as Richards' in most cases, using two labels rather than one. Bornat's algorithm works better than Richards' for conditional formulas such as

```
if (if p then x=0 else y=0) then S else T
```

or the equivalent C

```
if(p?x==0:y==0) S; else T;
```

These translation schema recursively translate the left operands of **andalso** and **orelse** differently, which precludes the translation of the left operand before the connective has been parsed. They are convincing evidence of the usefulness of translation from a parse tree intermediate form!

Richards' scheme **R** has the simpler specification so this is given first.

The R Scheme

This scheme can be used by the statement translator for translating conditional statements. Here the environment and other parameters to TS are omitted for clarity.

$$\begin{aligned}
 \mathbf{TS}[\text{if } p \text{ then } a \text{ else } b] \\
 &= \mathbf{R}[p] \text{ L false;} \\
 &\quad \mathbf{TS}[a]; \\
 &\quad \mathbf{JUMP } F; \\
 &\quad \mathbf{L: TS}[b] \\
 &\quad \mathbf{E:}
 \end{aligned}$$

where L and F are new labels

If p evaluates to true, we want control to fall through to the code for the *then*-part; if p evaluates to false, we want control to transfer to the label L. The **TS** rule for *if*-statements generates two 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 formal parameters. It's trivial to get the right effect when the conditional expression is a single boolean variable:

$$\mathbf{R}[\![v]\!] \ l \ \text{false} = \underline{\text{test } v}; \underline{\text{jumpfalse } l}$$

Clearly control transfers to the label l exactly when the (runtime) value of the variable v is the same as the (compile time) value of the final parameter of \mathbf{R} .

Although *if*- and *while*- statements always call \mathbf{R} with a *false* boolean argument, we'll see that we sometimes need to use a *true* argument, so that the code generated by \mathbf{R} falls through on false, and branches on true. In particular, this makes it unnecessary to generate any code for boolean negation – we just invert the argument to \mathbf{R} :

$$\mathbf{R}[\![v]\!] \ l \ \text{true} = \underline{\text{TEST } v}; \underline{\text{JUMPTRUE } l}$$

$$\mathbf{R}[\![\text{not } p]\!] \ l \ b = \mathbf{R}[\![p]\!] \ l \ (\neg b)$$

In the usual way when designing recursive procedures, the correctness of the recursive call can be assumed, so the resulting code will transfer to l exactly when the value of p is the same as the value of $\neg b$, that is, when *not* p is the same as b , as required. Similar induction assumptions can be made for the following rules:

$$\mathbf{R}[\![p \ \text{andalso } q]\!] \ l \ \text{true} = \mathbf{R}[\![p]\!] \ \underline{M} \ \text{false}; \mathbf{R}[\![q]\!] \ l \ \text{true}; \underline{M}:$$

$$\mathbf{R}[\![p \ \text{andalso } q]\!] \ l \ \text{false} = \mathbf{R}[\![p]\!] \ l \ \text{false}; \mathbf{R}[\![q]\!] \ l \ \text{false}$$

$$\mathbf{R}[\![p \ \text{orelse } q]\!] \ l \ \text{true} = \mathbf{R}[\![p]\!] \ l \ \text{true}; \mathbf{R}[\![q]\!] \ l \ \text{true}$$

$$\mathbf{R}[\![p \ \text{orelse } q]\!] \ l \ \text{false} = \mathbf{R}[\![p]\!] \ \underline{M} \ \text{true}; \mathbf{R}[\![q]\!] \ l \ \text{false}; \underline{M}:$$

The \mathbf{R} rules generate no code for **andalso**, **orelse** and **not**, and at most one test and conditional jump for each of the variables. There is therefore no shorter translation available.

Conditional formulas have more complicated translation schema:

$$\mathbf{R}[\![\text{if } p \ \text{then } q \ \text{else } r]\!] \ l \ \text{true} =$$

$$\mathbf{R}[\![p]\!] \ \underline{E} \ \text{false};$$

$$\mathbf{R}[\![q]\!] \ l \ \text{true};$$

$$\underline{\text{JUMP } F};$$

$$\underline{E}: \mathbf{R}[\![r]\!] \ l \ \text{true};$$

$$\underline{F}:$$

$$\mathbf{R}[\![\text{if } p \ \text{then } q \ \text{else } r]\!] \ l \ \text{false} =$$

$$\mathbf{R}[\![p]\!] \ \underline{E} \ \text{false};$$

$$\mathbf{R}[\![q]\!] \ l \ \text{false};$$

$$\underline{\text{JUMP } F};$$

$$\underline{E}: \mathbf{R}[\![r]\!] \ l \ \text{false};$$

$$\underline{F}:$$

Richards' scheme can generate a jump that lands directly on another; Richards'

compiler did not optimise the translation of the BCPL equivalent of (boolean valued) **if-then-else** expressions in conditional contexts. The problem is that the instruction JUMP F may be labelled: for example, this happens whenever the expression q has the form $x \text{ andalso } y$. (See the rule for *andalso*, above).

Exercise

Implement the **R** scheme in ML. Hint: look at the ML implementation of **TE** for translating expressions (Chapter 3.1; **R** has the ML type `expression*label*bool` unit. Labels can be represented in the implementation language as integers (also Chapter 3.1).

The B Scheme

Bornat's solution to this problem was to devise a set of translation rules **B** generate JUMP instructions only where it can be guaranteed that there will be no preceding label.

A boolean formula F is translated by a procedure **B** relative to two labels and a boolean constant. The generated code may jump to the first label, but only if F evaluates to True; it may jump to the second label, but only if F evaluates to False; or neither label may be jumped to (that is, control can fall through) if the source formula evaluates to the same sense as the third parameter.

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

$$\mathbf{B}[\![v]\!] \text{ t f true} = \text{TEST } v; \text{ JUMPFALSE } f$$

$$\mathbf{B}[\![v]\!] \text{ t f false} = \text{TEST } v; \text{ JUMPTTRUE } t$$

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

$$\mathbf{B}[\![a=b]\!] \text{ t f true} = \text{COMPARE } a,b; \text{ JUMPNOTEQUAL } f$$

$$\mathbf{B}[\![a=b]\!] \text{ t f false} = \text{COMPARE } a,b; \text{ JUMPEQUAL } t$$

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.

Exercise

Modify the **R** and **B** schemes to allow for comparison of arbitrary expressions. Hint: use the Sethi-Ullman translator for the subexpressions.

Exercise

Modify the Sethi-Ullman translation scheme **E** to allow left-strict operators embedded within an otherwise strict expression. For example, consider this C expression, which should load a register with the value of the bracketed subexpression then modify it using **and p,reg**:

$$(x/=0 \ \&\& \ y/x == 1) \ \& \ p$$

—

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

$TS[\text{if } p \text{ then } a \text{ else } b]$
 $= B[p] \text{ T F true};$
 $\underline{T}: TS[a];$
 $\text{jump } \underline{N};$
 $\underline{E}: TS[b]$
 $\underline{N}:$

where T, F and N are new labels

The boolean connectives can be recursively translated with no additional target code:

$B[p \text{ andalso } q] \text{ t f n} = B[p] \text{ N f true}; \underline{N}: B[q] \text{ t f n}$
 $B[p \text{ orelse } q] \text{ t f n} = B[p] \text{ t N false}; \underline{N}: B[q] \text{ t f n}$
 $B[\text{not } p] \text{ t f n} = B[p] \text{ f t } (-n)$
 $B[\text{if } p \text{ then } q \text{ else } r] \text{ t f n} =$
 $B[p] \text{ T E true};$
 $\underline{T}: B[q] \text{ t f false};$
 $\underline{JUMP} \text{ f};$
 $\underline{E}: B[r] \text{ t f n}$

The **B** scheme never generates 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 $JUMP \text{ F}; \text{F};$ or $JUMP \text{ FALSE } \text{F}; \text{F};$).

Top-down translators cannot elegantly generate optimal code for source phrases that include constant subphrases. It is of course possible to explicitly test, in each rule, for the presence of a constant subexpression and give such cases special treatment. Such a constant subphrase can be arbitrarily large so no finite pattern could match all cases. This approach would therefore involve explicit simplification as part of the process of rule selection:

$B[p \text{ orelse } q] \text{ t f n} = B[q] \text{ t f n} \quad ,if \text{ Simplify}[p] = [false]$
 $B[p \text{ orelse } q] \text{ t f n} = B[p] \text{ t N false}; \underline{N}: (B[q] \text{ t f n}) \quad ,otherwise$

This is not only inelegant, it is inefficient. It's far better to include a tree-improving pass before translation, eliminating constant subexpressions as far as possible. This would include replacing formulas containing constants (such as **if p then true else q**) by equivalent expressions (**p orelse q**) without constants.

Simple constant conditions can be translated well; good code is generated for, eg, **while true do S**. No code need be generated for **true** in such a statement, while in others an unconditional jump can be used.

$B[True] \text{ t f true} = \text{NO-OP}$
 $B[True] \text{ t f false} = \underline{JUMP} \text{ t}$
 $B[False] \text{ t f true} = \underline{JUMP} \text{ f}$
 $B[False] \text{ t f false} = \text{NO-OP}$

$$\begin{aligned} \text{TS}[\text{while } P \text{ do } S] \\ &= \underline{N}: \mathbf{B}[P] \text{ T F true;} \\ &\quad \underline{T}: \text{TS}[S]; \\ &\quad \text{jump } \underline{N}; \\ &\quad \underline{F}: \\ &\quad \text{where T, F and N are new labels} \end{aligned}$$

Exercise

These rules mean that in a statement (which may appear in a procedure body) such as

```
while true do if x>0 then x:=x div 2 else return;
```

the jump from the end of the if-statement back to the start of the body of the loop takes control directly to the code for the test of x.

Calculate the target code for the sample statement.

—

The **B** scheme makes it easy to arrange for very efficient flow of control through complicated boolean formulas with left-strict connectives. The translation schemes for statements can be extended in a similar way so that inefficient sequences of jumps and return instructions can be avoided for complicated nested conditional statements.

Case expressions and enumerated types

The **B** scheme generalises to *case*-expressions, used in conditional contexts, on any enumeration type. Case expressions with constant right hand sides correspond to explicit tabulation of finite functions, like truth tables. When the value of the case expression is used simply to control another case expression, explicit construction of the intermediate values is unnecessary, for example:

```
case (case e of 0=>1 | 1=>2 | 2=>0) of 0=>X | 1=>Y | 2=>Z
```

The inner case expression can be translated relative to a list of labels and a set of values acceptable if the code 'falls through'. As in the translation of **not**, code for the inner expression can be completely eliminated.

Conditional Statements

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.

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

$S[\text{if } p \text{ then } t \text{ else } e] \text{ Normal} = (B[p] \text{ L M true}); L: (S[t] \text{ (Goto N)});$
 $M: (S[e] \text{ Normal}); N:$

$S[\text{if } p \text{ then } t \text{ else } e] \text{ (Goto n)} = (B[p] \text{ L M true}); L: (S[t] \text{ (Goto n)});$
 $M: (S[e] \text{ (Goto n)})$

$S[s; t] \text{ c} = (S[s] \text{ Normal}); (S[t] \text{ c})$

The term (**Goto n**) on the left hand side of the second rule is a *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 instruction if the continuation parameter specifies one.

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

$S[\text{while } p \text{ do } t] \text{ Normal} = R: (B[p] \text{ L M true}); L: (S[t] \text{ (Goto R)}); M:$
 $S[\text{while } p \text{ do } t] \text{ (Goto m)} = R: (B[p] \text{ L m true}); L: (S[t] \text{ (Goto R)})$

Example

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

$S[\text{while } p \text{ do if } q \text{ then } r \text{ else } s] \text{ Normal}$

- $R: (B[p] \text{ L M true}); L: (S[\text{if } q \text{ then } r \text{ else } s] \text{ (Goto R)}); M:$
- $R: (B[p] \text{ L M true}); L: (B[q] \text{ L' M' true});$
 $L': (S[r] \text{ (Goto R)}); M': (S[s] \text{ (Goto R)}); M:$
- $R: \text{TEST } p; \text{JUMPFALSE } M; \text{TEST } q; \text{JUMPFALSE } M';$
 $\dots I \dots;$
 $\text{JUMP } R; M': \dots s \dots; \text{JUMP } R; M:$

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 could 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 (see below).

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 Ullman¹¹, 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.

Labelled statements and source language gotos

Consider this program fragment:

```
while P do begin
    if Q then goto L;
    S;
L:
end;
```

Our translations so far would emit a target-language label for *L*, and a JUMP *L* instruction for *goto L*. This is unfortunate, as the next target instruction is another JUMP, back to the top of the loop.

We can do much better than this by binding statement labels to values of our continuation type, using an environment, symbol table or similar data structure.

The description is simpler for restricted forms of **goto** such as Tennent's **redo** and **leave**¹². These are similar to BCPL's and *C*'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.

An environment that behaves exactly like e except that it maps a to b is written as $e[a \mapsto b]$.

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

$$\begin{aligned} \mathcal{S}[\text{I: C}] \text{ Normal } e = N: (\mathcal{S}[\text{C}] \text{ Normal } (e[\text{I} \mapsto (\text{Goto N, Goto N'})])); N': \\ \mathcal{S}[\text{I: C}] \text{ c } e = N: (\mathcal{S}[\text{C}] \text{ c } (e[\text{I} \mapsto (\text{Goto N, c})])) \\ \mathcal{S}[\text{redo I}] \text{ c } e = \text{C first}(e[\text{I}]) \\ \mathcal{S}[\text{leave I}] \text{ c } e = \text{C second}(e[\text{I}]) \\ \mathcal{S}[\text{I; redo J}] \text{ c } e = \mathcal{S}[\text{I}] (\text{first}(e[\text{J}])) e \\ \mathcal{S}[\text{I; leave J}] \text{ c } e = \mathcal{S}[\text{I}] (\text{second}(e[\text{J}])) e \end{aligned}$$

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.

Although this solves the immediate problem it isn't good enough since it is still capable of generating a labelled JUMP or RETURN instruction. The difficulty is when *leave*-statements appear in *if*-statements, for the *then*-part and *else*-parts are both labelled.

Example

$$\begin{aligned} \mathcal{S}[x: (\text{if } p \text{ or } q \text{ then leave } x \text{ else print "hello"})] (\text{Goto } c) e \\ \square \quad N: (\mathcal{S}[\text{if } p \text{ or } q \text{ then leave } x \text{ else print "hello"}] (\text{Goto } c) e' \\ \qquad \qquad \qquad \text{where } e' \text{ is } (e[x \mapsto (\text{Goto N, Goto c})]) \\ \square \quad N: (\mathcal{B}[p \text{ or } q] \text{ T F true}); \\ \quad \quad T: (\mathcal{S}[\text{leave } x] (\text{Goto } c) e'); \\ \quad \quad F: (\mathcal{S}[\text{print "hello"}] (\text{Goto } c) e') \\ \square \quad N: \text{TEST } p; \text{JUMPTRUE } T; \text{TEST } Q; \text{JUMPFALSE } F; \\ \quad \quad T: (\mathcal{S}[\text{leave } x] (\text{Goto } c) e'); \\ \quad \quad F: (\mathcal{S}[\text{print "hello"}] (\text{Goto } c) e') \\ \square \quad N: \text{TEST } p; \text{JUMPTRUE } T; \text{TEST } Q; \text{JUMPFALSE } F; \\ \quad \quad T: \text{JUMP } c; \\ \quad \quad F: \text{PRINT "hello"; JUMP } c \end{aligned}$$

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. A later section 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.

The B scheme with continuation parameters

We next modify the **B** rule to accept continuation parameters instead of simple target language labels. The modification makes it easier to optimise **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.

$$\begin{aligned} \mathbf{B}[\mathbf{v}] \text{ t Proc true} &= \text{TEST } v; \text{ JUMPTRUE } L; \text{ RETURN}; L: \\ \mathbf{B}[\mathbf{v}] \text{ t (Goto L) true} &= \text{TEST } v; \text{ JUMPFALSE } L \\ \mathbf{B}[\mathbf{v}] \text{ Proc f false} &= \text{TEST } v; \text{ JUMPFALSE } L; \text{ RETURN}; L: \\ \mathbf{B}[\mathbf{v}] \text{ (Goto L) f false} &= \text{TEST } v; \text{ JUMPTRUE } L \\ \mathbf{B}[\text{True}] \text{ t f true} &= \text{NO-OP} \\ \mathbf{B}[\text{True}] \text{ t f false} &= \mathbf{C} \text{ t} \\ \mathbf{B}[\text{False}] \text{ t f true} &= \mathbf{C} \text{ f} \\ \mathbf{B}[\text{False}] \text{ t f false} &= \text{NO-OP} \end{aligned}$$

Armed with this more powerful scheme, we can make the code for the condition do the job of the *leave*-statement:

$$\begin{aligned} \mathbf{S}[\text{if } p \text{ then leave } x \text{ else } y] \text{ c e} \\ &= (\mathbf{B}[p] \text{ second}(e[x]) \text{ (Goto N) false}); \text{N: } (\mathbf{S}[y] \text{ c e}) \\ \mathbf{S}[\text{if } p \text{ then } s \text{ else } s'] \text{ Normal e} \\ &= \mathbf{B}[p] \text{ (Goto T) (Goto E) true;} \\ &\quad \text{T: } (\mathbf{S}[s] \text{ (Goto N) e}); \text{E: } (\mathbf{S}[s'] \text{ Normal e}); \text{N:} \\ \mathbf{S}[\text{if } p \text{ then } s \text{ else } s'] \text{ c e} \\ &= (\mathbf{B}[p] \text{ (Goto T) (Goto E) true}); \text{T: } (\mathbf{S}[s] \text{ c e}); \text{E: } (\mathbf{S}[s'] \text{ c e}) \end{aligned}$$

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**.

Example

This rule generates code for the condition **p** that chooses the continuation denoted by **x** as soon as possible:

$$\begin{aligned} \mathbf{S}[x: (\text{if } p \text{ or } q \text{ then leave } x \text{ else skip; print "hello"})] \text{ (Goto L) e} \\ \square \text{ N: } (\mathbf{S}[\text{if } p \text{ or } q \text{ then leave } x \text{ else skip; print "hello"}] \text{ (Goto L) e'}) \\ \quad \text{where } e' \text{ is } (e[x] \text{ (Goto N, Goto L)}) \\ \square \text{ N: } (\mathbf{S}[\text{if } p \text{ or } q \text{ then leave } x \text{ else skip}] \text{ Normal e}); (\mathbf{S}[\text{print "hello"}] \text{ (Goto L) e'}) \\ \square \text{ N: } (\mathbf{B}[p \text{ or } q] \text{ (Goto L) (Goto N') false}); \end{aligned}$$

N': (S[skip] Normal e'); (S[print "hello"] (Goto L) e')

□ N: (B[p] (Goto L) (Goto N') false); N': (B[q] (Goto L) (Goto N') false)

N': NO-OP; (S[print "hello"] (Goto L) e')

□ N: TEST P; JUMPTRUE L; N': TEST Q; JUMPTRUE L;

N': NO-OP; (S[print "hello"] (Goto L) e')

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 the appendix to this section).

Assignment Statements

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

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 R0.

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

□ (A[if p then y else z] Normal); MOVE R0[] x; RETURN

□ (B[p] (Goto T) (Goto F) true); T: (A[y] (Goto L));

F: (A[z] Normal);

L: MOVE R0[] x; RETURN

□ TEST p; JUMPFALSE F; T: MOVE y[] R0; JUMP L;

F: MOVE z[] R0;

L: MOVE R0[] 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

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.

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 Fraser¹³, 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.

Appendix

The final version of the rules for a target stack machine is given here. All the optimisations discussed in this section 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 | Normal s | Proc

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

$\text{isContinuation}[\text{skip}] (\text{Normal } s) e s' = (s=s')$
 $\text{isContinuation}[\text{skip}] \text{Proc } e s = \text{true}$
 $\text{isContinuation}[\text{skip}] (\text{Goto } n s) e s' = (s=s')$
 $\text{isContinuation}[\text{leave } x] c e s = \text{true if } (\text{second}(e[x]) = \text{Proc})$
 $= (s=s') \text{ where } (\text{Goto } n s') = \text{second}(e[x]) \text{ otherwise}$
 $\text{isContinuation}[\text{redo } x] c e s = \text{true if } (\text{first}(e[x]) = \text{Proc})$
 $= (s=s') \text{ where } (\text{Goto } n s') = \text{first}(e[x]) \text{ otherwise}$
 $\text{isContinuation}[\text{statement}] c e s = \text{false}$

$\text{continuationOf}[\text{skip}] (\text{Normal } s) e c = c$
 $\text{continuationOf}[\text{skip}] c e c' = c$
 $\text{continuationOf}[\text{leave } x] c e c' = \text{second}(e[x])$
 $\text{continuationOf}[\text{redo } x] c e c' = \text{first}(e[x])$

$S[\text{int } v=F; B] c e s = (E[F] e); (S[B] c e' (s+1))$

where $e' = e[v \square s]$

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

if $(\text{isContinuation}[x] c e s) \square (\text{isContinuation}[y] c e s)$

where $c x = \text{continuationOf}[x] c e (\text{Goto } E s)$

and $c y = \text{continuationOf}[y] c e (\text{Goto } E s)$

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

where $c x = \text{continuationOf}[x] c e (\text{Goto } T s)$

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

where $c y = \text{continuationOf}[y] c e (\text{Goto } F s)$

$S[\text{if } p \text{ then } x \text{ else } y] (\text{Normal } s) e s' = (B[p] (\text{Goto } L s') (\text{Goto } M s') \text{ true } s');$

$L: (S[x] (\text{Goto } N s) e s'); M: (S[y] (\text{Normal } s) e s'); N:$

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

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

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

$T: (S[x:=y] (\text{Goto } N s') e s);$

$F: (S[x:=z] (\text{Normal } s') e s); N:$

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

$T: (S[x:=y] c e s); F: (S[x:=z] c e s)$

$S[x:=y] \text{ c e s} = \text{MOVE}(e[y], e[x]); (\text{C c s})$ if y is a variable
 $= E[y] e; \text{POP}(e[x]); (\text{C c s})$ otherwise
 $S[I: C] (\text{Normal s}) e s' = \text{N}: (S[C] (\text{Normal s}) (e[I] (\text{Goto N s}', \text{Goto N' s}))); \text{N}':$
 $S[I: C] \text{ c e s} = \text{N}: (S[C] \text{ c} (e[I] (\text{Goto N s}, \text{c})))$
 $S[\text{redo I}] \text{ c e s} = \text{C first}(e[I]) s$
 $S[\text{leave I}] \text{ c e s} = \text{C second}(e[I]) s$
 $S[\text{call p}] \text{ Proc e s} = \text{DISCARD s}; \text{JUMP}(e p)$
 $S[\text{call p}] \text{ c e s} = \text{CALL}(e p); (\text{C c s})$
 $S[i;j] \text{ c e s} = (S[i] \text{ c}' e s)$ if i is continuation of j c e c where $\text{c}' = \text{continuationOf}[j] \text{ c e c}$
 $S[i;j] \text{ c e s} = (S[i] (\text{Normal s}) e s); (S[j] \text{ c e s})$
 $S[\text{while p do t}] (\text{Normal s}) e s' = \text{R}: (\mathbf{B}[p] (\text{Goto L s}') (\text{Goto M s}') \text{true s}');$
 $\text{L}: (S[t] (\text{Goto R s}') e s'); \text{M}: \text{DISCARD}(s'-s)$
 $S[\text{while p do t}] \text{ Proc e s} = \text{R}: (\mathbf{B}[p] (\text{Goto L s}) \text{Proc true s}); \text{L}: (S[t] (\text{Goto R s}) e s)$
 $S[\text{while p do t}] (\text{Goto c s}') e s = \text{R}: (\mathbf{B}[p] (\text{Goto L s}) (\text{Goto c s}') \text{true s}); \text{L}: (S[t] (\text{Goto R s}'$
 $\text{if } s=s'$
 $= \text{R}: (\mathbf{B}[p] (\text{Goto L s}') (\text{Goto M s}) \text{true s}');$
 $\text{L}: (S[t] (\text{Goto R s}') e s'); \text{M}: \text{DISCARD}(s'-s); \text{JUMP c}$
 otherwise
 $S[\text{unconditional statement u}] \text{ c e s} = \dots \text{code for u} \dots; (\text{C c s})$

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

$\mathbf{B}[v] \text{ t} (\text{Goto j s}) \text{true s}' = \text{TEST } v; \text{JUMPFALSE } j$
 $\mathbf{B}[v] (\text{Goto j s}) \text{f false s}' = \text{TEST } v; \text{JUMPTRUE } j$
 $\mathbf{B}[v] \text{ t Proc true s} = \text{TEST } v; \text{JUMPTRUE } L; \text{RETURN}; \text{L}:$
 $\mathbf{B}[v] \text{ Proc f false s} = \text{TEST } v; \text{JUMPFALSE } L; \text{RETURN}; \text{L}:$
 $\mathbf{B}[\text{True}] \text{ t f true s} = \text{NO-OP}$
 $\mathbf{B}[\text{True}] \text{ t f false s} = \text{C t s}$
 $\mathbf{B}[\text{False}] \text{ t f true s} = \text{C f s}$
 $\mathbf{B}[\text{False}] \text{ t f false s} = \text{NO-OP}$
 $\mathbf{B}[p \text{ and } q] \text{ t f n s} = (\mathbf{B}[p] (\text{Goto N s}) \text{f true s}); \text{N}: (\mathbf{B}[q] \text{ t f n s})$
 $\mathbf{B}[p \text{ or } q] \text{ t f n s} = (\mathbf{B}[p] \text{ t} (\text{Goto N s}) \text{false s}); \text{N}: (\mathbf{B}[q] \text{ t f n s})$
 $\mathbf{B}[\text{not } p] \text{ t f n s} = \mathbf{B}[p] \text{ f t } (\neg n) s$

$E[n] e = \text{PUSH } \%n$ if n is a number
 $E[v] e = \text{PUSH}(e[v])$ if v is a local variable
 $E[x+y] e = (E[x] e); (E[y] e); \text{ADD}$

