Destination-Driven Code Generation

R. Kent Dybvig, Robert Hieb, Tom Butler
dyb@cs.indiana.edu

Indiana University Computer Science Department

Technical Report #302

February 1990

Destination-Driven Code Generation*

R. Kent Dybvig, Robert Hieb, Tom Butler
Computer Science Department
Indiana University
Bloomington, IN 47405

February 1990

Abstract

Destination-driven code generation is a simple top-down technique that allows the code generated for a program phrase to depend upon its context in an abstract syntax tree. The context is encapsulated in a data destination and a control destination. The data destination specifies where the value computed by an expression is to be stored, while the control destination specifies where program execution is to resume after computation of the value. Together, the destinations allow a code generator to specify data and control flow dependencies between subcomputations. As a result, these subcomputations can be "wired" together in an efficient manner. We illustrate the technique by presenting a code generator for a subset of the programming language C. This technique has been used in the implementation of an incremental compiler for the Scheme programming language that generates code for one of several computer architectures.

1 Introduction

In this article, we introduce a straightforward code generation strategy called *destination-driven code generation*, which is used to generate high-quality assembly code directly from abstract syntax trees. Code generation proceeds in one pass with no back-patching or code splicing. In most cases, destination-driven code generation results in code free of unnecessary loads and stores, unnecessary tests and branches, dead code, and some forms of useless code. Thus, the code generated is of sufficient quality that block reordering [12] and peephole optimization [4] are typically unnecessary.

A destination-driven code generator can be used as the single pass in the back-end of a simple compiler as one of the final passes in a complex optimizing compiler. Because high-quality code is generated in one pass, destination-driven code generation is especially useful in interactive computing environments where compilation speed is as important as the quality of the code that is generated.

The abstract syntax trees passed to the code generator may be produced by the front-end, or syntax analysis phase, of the compiler, or they may be the result of earlier optimization and analysis passes operating on the output of the front-end. The use of an abstract syntax tree representation for the intermediate language rather than a more assembly-code-like representation allows the intermediate code to more closely resemble the source program from which it was derived.

Destination-driven code generation is a simple top-down technique that allows the code generated for a program phrase to depend upon its context in an abstract syntax tree. This context

*This material is based on work supported by the National Science Foundation under grant number CCR-8803432.

is encapsulated in a *data destination* and a *control destination*. The data destination specifies whether the value computed by a subtree is needed to compute the value of its parent. It is often the case that this value is not needed; this is especially true when trees are derived from statements in a language that distinguishes statements from expressions. Moreover, many languages, notably Scheme [13] and C [11], allow expressions that perform side effects to be evaluated "for effect only." We treat syntax trees as representations of expressions and rely on the data destination to signal when an expression value is not needed. This simplifies the code generator and avoids generating store instructions for values that are not needed. When the value computed by a subtree is needed, the data destination specifies its storage location. Therefore, the code generator can specify intermediate locations in a way that is convenient for "higher level" computations. By contrast, the redundant loads and stores generated by techniques that require the value of a subcomputation to be placed in a predetermined location are avoided.

Just as the data destination eliminates unnecessary move instructions, the control destination eliminates unnecessary jumps. The control destination specifies where to go next. In many cases the control destination simply specifies that the computation is to continue with the next instruction; however, in order to accommodate constructs that alter the sequential flow of control, the control destination can also be a label, a pair of labels, or a special *return* destination. When the control destination is a pair of labels, the flow of control is to be altered according to the boolean value of the current expression. A single label indicates an unconditional jump over code for an expression that is conditionally evaluated, *e.g.*, the *else* part of an **if** statement. The return destination indicates when there is nothing left to do in the current procedure; a return sequence can then be generated immediately after the code that computes the value of an expression.

A code generator that handles structured control constructs gracefully may fail to handle nesting of these constructs and may therefore generate jumps to jumps or jumps to returns. For instance, when the last statement of a **while** loop is another **while** loop, any break from the inner **while** loop should jump directly to the top of the outer loop. However, the code generated for a break from the inner loop may in fact be a jump to the end of the loop, followed by a jump to the top of the outer loop. The redundant jumps can be avoided if the inner loop inherits the control destination specified for the body of the outer loop. In general, the generation of redundant jump sequences for nested control constructs can be avoided via a simple inheritance scheme on control destinations.

Traditional approaches to code generation divide code generation into several components. The code is broken into basic blocks and a control graph connecting the basic blocks is constructed to preserve the flow information of the original program. The basic blocks themselves are broken up into simpler, intermediate code in some form of "three-address code." This division allows the control graph and the basic blocks to be optimized independently. Most of the literature addresses a particular component of this compilation process. A variety of code generation strategies for converting from intermediate code into machine code have been proposed, generally with the intent to generate locally optimal and/or retargetable code [2, 1, 15, 6, 7, 10, 8, 9]. A general description of the traditional approach and a review of some of the major code generation techniques can be found in Aho, *et. al.* [3]. On the other hand, our approach treats code generation for control structures and expressions uniformly, generating object code in a single pass over the abstract syntax tree.

An interesting analogue to our approach can be found in *cps* (continuation-passing style) transformations [16]. The cps transformation simplifies code by breaking it into functionalized basic blocks that are invoked with an explicit continuation argument. The continuation embodies both the control and data destinations of our technique. However, this technique only results in good code generation if the resulting continuations are carefully analyzed and optimized.

$$
\begin{array}{lll}
n \in \mathit{Int} & & \\
x, f \in \mathit{Identifier} & & \\
 & & \\
P \in \mathit{Program} & ::= & F \ldots \\
 & & \\
F \in \mathit{Fundecl} & ::= & \textbf{fundecl}\ f\ \mathit{params}\ \mathit{locals}\ E \\
 & & \\
E \in \mathit{Expression} & ::= & \textbf{sequence}\ E\ E \\
 & | & \textbf{while}\ E\ E \\
 & | & \textbf{break} \\
 & | & \textbf{return}\ E \\
 & | & \textbf{if}\ E\ E\ E \\
 & | & \textbf{if}\ E\ E \\
 & | & \textbf{assign}\ x\ E \\
 & | & \textbf{binop}\ \mathit{op}\ E\ E \\
 & | & \textbf{and}\ E\ E \\
 & | & \textbf{or}\ E\ E \\
 & | & \textbf{not}\ E \\
 & | & \textbf{call}\ f\ E\ \ldots \\
 & | & \textbf{var}\ x \\
 & | & \textbf{int}\ n
\end{array}
$$

Figure 1: Abstract syntax for a subset of C.

The next three sections of this article are devoted to the development of a destination-driven code generator. The code generator produces assembly code for the DEC VAX [5] architecture from a representation of abstract syntax trees for a subset of the C programming language. Section 2 presents an abstract syntax for the chosen subset of C and briefly discusses syntactic and semantic issues relating to the source and target languages. Section 3 presents a destination-driven code generator that translates from the abstract syntax to the target language. The final section discusses the interaction between optimization and destination-driven code generation and summarizes the important aspects of the technique.

2 Source and Target Languages

2.1 Source Language

Figure 1 presents a grammer for a language of abstract syntax trees that corresponds to a subset of the C programming language. The grammer deviates from the concrete syntax of C in the omission of certain language constructs and in its treatment of blocks. Only **while** loops are treated; other loop forms can be treated in a similar manner or translated into equivalent **while** loops. Data is restricted to type *Int* and a representative subset of the standard operators are included. For simplicity we assume that block variables are renamed as necessary and that their declarations are lifted to the *locals* field of the enclosing procedure definition.

Many of the traditional distinctions between statements and expressions are not preserved in C. Variable assignments are expressions and expressions may be used as statements. Use of boolean

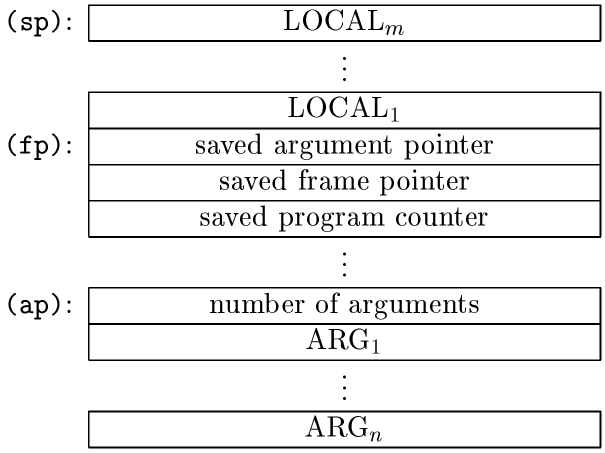

Figure 2: Format of a call frame.

operators is not limited to conditional and loop test expressions. Furthermore, expression analogs of the traditional statement forms **if** and **sequence** are provided. The end result is that whether a construct is classified as an expression or a statement depends largely upon its use, and consequently the abstract syntax in Figure 1 does not distinguish between statements and expressions.

2.2 Target Language

The VAX architecture provides an orthogonal set of instructions common to most multi-register machines. There are fifteen general purpose registers (excluding the program counter); three of these are reserved as indices into the control stack. Of the remaining registers we use only `r0`. Arithmetic is performed using the VAX complement of three, two, and one operand arithmetic instructions on long word (integer) data. For example, integer division may be performed with one of two instructions:

`divl3` *divisor, dividend, quotient*
`divl2` *divisor, dividend-quotient*

Single operand instructions are provided for incrementing and decrementing. Comparisons are performed using the standard two and one operand (comparison against zero) instructions.

The `calls` (call with stack arguments) and `ret` instructions are used to perform procedure call and return. The format of a call frame is shown in Figure 2. Before the `calls` instruction is executed, arguments to the called routine are pushed on the stack in reverse order. The number of arguments on the stack is the first parameter to the `calls` instruction; the second parameter is the address of the called routine. Upon execution of the `calls` instruction, the number of arguments is saved on the stack along with the contents of the program counter, frame pointer (`fp`), and argument pointer (`ap`) registers.[1] The `fp` and `ap` are then repositioned as shown. After control is transferred to the called routine, space is allocated for local variables. We adhere to the VAX calling convention, which requires that the return value be placed in register `r0`, and we do not generate code that uses any register that must be saved across a call. Execution of a `ret` instruction removes the call frame from the stack and restores the contents of the saved registers.

[1] Other information not relevant to our discussion is stored on the stack as a result of the `calls` instruction. Complete details can be found in [5, pp. 6-16–6-18, 9-13].

All data references are indirect, register direct, or immediate; arguments to a routine are referenced at an offset from the argument pointer, as shown in Figure 2. Stack-allocated (programmer-declared) locals are referenced at an offset from the frame pointer. Expression temporaries are generated "on the fly" and are therefore referenced relative to the stack pointer (`sp`). Assembler syntax and instruction mnemonics are typeset in typewriter font. Assembly code is generated for the UNIX "as" assembler [14]. This assembler provides generalized branch mnemonics that are automatically transformed into appropriate combinations of conditional and unconditional branches depending on the size of the branch.

3 Code Generation

This section presents a destination-driven VAX assembly code generator for the source language of Section 2. The code generator is assumed to be the single pass in the back end of a simple compiler. In particular, it is assumed that programmer-declared variables have not been preassigned locations and that evaluation of subexpressions may require the generation of expression temporaries. In the conclusion, relaxation of these assumptions is discussed.

Our presentation of the code generation function is divided into two parts. The first part discusses aspects of the assembly language interface, including instruction selection and allocation and deallocation of compiler-generated temporaries. The second part presents the code generation function.

3.1 The Assembly Language Interface

Every compiler must solve the problems of storage allocation and instruction selection. Since the destination-driven code generation technique is independent of the solutions for these problems, simple strategies for storage allocation and instruction selection have been chosen for the code generator presented in this article. These strategies are briefly discussed in this section.

Since local variables are declared in the *locals* field of the enclosing procedure declaration, their locations can be allocated in the call frame at the start of each procedure call. Parameters are also placed in the call frame, so all programmer-declared variables are preassigned to locations allocated within the control stack (see Figure 2). Thus, the map from programmer-declared variables to locations is constant for each **fundecl**. On the other hand, temporaries are created "on the fly" and do not have preassigned locations. In some cases, it is necessary to dynamically allocate and deallocate stack space for temporaries, although we also allow temporaries to reside in the accumulator (register `r0`). The set of locations we require thus includes the accumulator, stack temporaries, parameters, and locals:

$$A \in Location = ac \mid stack \mid param_i \mid local_i$$

The set of operands includes the locations plus integer constants:

$$O \in Operand = Location \mid integer$$

Operands are mapped to assembly language operands by the functions $\downarrow$ and $\uparrow$ shown in Figure 3. When a stack location is defined, the $\downarrow$ function allocates the necessary stack space using the pre-decrement (`sp` relative) addressing mode. When a stack location is referenced, the $\uparrow$ function deallocates its stack space using the post-increment addressing mode. Both functions map the

$$
\begin{array}{l}
\downarrow : \mathit{Location} \rightarrow \mathit{Code} \\
\downarrow \mathit{stack} \;\Rightarrow\; \langle\!\langle\, \texttt{-(sp)} \,\rangle\!\rangle \\
\downarrow \mathit{ac} \;\Rightarrow\; \langle\!\langle\, \texttt{r0} \,\rangle\!\rangle \\
\downarrow \mathit{param}_i \;\Rightarrow\; \langle\!\langle\, \texttt{4i(ap)} \,\rangle\!\rangle \\
\downarrow \mathit{local}_i \;\Rightarrow\; \langle\!\langle\, \texttt{-4i(fp)} \,\rangle\!\rangle
\end{array}
$$

$$
\begin{array}{l}
\uparrow : \mathit{Operand} \rightarrow \mathit{Code} \\
\uparrow \mathit{stack} \;\Rightarrow\; \langle\!\langle\, \texttt{(sp)+} \,\rangle\!\rangle \\
\uparrow \mathit{ac} \;\Rightarrow\; \langle\!\langle\, \texttt{r0} \,\rangle\!\rangle \\
\uparrow \mathit{param}_i \;\Rightarrow\; \langle\!\langle\, \texttt{4i(ap)} \,\rangle\!\rangle \\
\uparrow \mathit{local}_i \;\Rightarrow\; \langle\!\langle\, \texttt{-4i(fp)} \,\rangle\!\rangle \\
\uparrow n \;\Rightarrow\; \langle\!\langle\, \texttt{\$n} \,\rangle\!\rangle
\end{array}
$$

Figure 3: Operand conversion.

$$
\begin{array}{l}
\langle\!\langle\, \texttt{subl3 } A\texttt{, (sp)+, -(sp)} \,\rangle\!\rangle \;\Rightarrow\; \langle\!\langle\, \texttt{subl2 } A\texttt{, (sp)} \,\rangle\!\rangle \\
\langle\!\langle\, \texttt{subl3 } A_1\texttt{, } A_2\texttt{, } A_2 \,\rangle\!\rangle \;\Rightarrow\; \langle\!\langle\, \texttt{subl2 } A_1\texttt{, } A_2 \,\rangle\!\rangle \\
\langle\!\langle\, \texttt{subl3 } A_1\texttt{, } A_2\texttt{, } A_3 \,\rangle\!\rangle \;\Rightarrow\; \langle\!\langle\, \texttt{subl3 } A_1\texttt{, } A_2\texttt{, } A_3 \,\rangle\!\rangle \\
\langle\!\langle\, \texttt{subl2 \$1, } A \,\rangle\!\rangle \;\Rightarrow\; \langle\!\langle\, \texttt{decl } A \,\rangle\!\rangle \\
\langle\!\langle\, \texttt{subl2 \$0, } A \,\rangle\!\rangle \;\Rightarrow\; \langle\!\langle\, \,\rangle\!\rangle \\
\langle\!\langle\, \texttt{subl2 } A_1\texttt{, } A_2 \,\rangle\!\rangle \;\Rightarrow\; \langle\!\langle\, \texttt{subl2 } A_1\texttt{, } A_2 \,\rangle\!\rangle \\
\mathit{etc.}
\end{array}
$$

Figure 4: Instruction selection.

accumulator to `r0`. Since each compiler-generated temporary is defined and referenced exactly once, the $\downarrow$ and $\uparrow$ functions serve to allocate and deallocate locations for temporaries.

The environment parameter is a pair consisting of a control component and a data component:

$$\rho \in \mathit{Environment} = \mathit{Label} \times \mathit{Map}$$

The data component is a map from identifiers to locations:

$$m \in \mathit{Map} = \mathit{Identifier} \rightarrow \mathit{Location}$$

The control component is a label used to generate code for **break** expressions. It is discussed in detail in the next part of this section. Labels are assembly language statement labels or the special label *return*:

$$L \in \mathit{Label} = \mathit{return} \mid \text{assembly language label}$$

Instruction selection is generally dependent on temporary allocation. For instance, the judicious choice of a temporary location might allow the use of a two operand instruction when a three operand instruction would otherwise be necessary. However, to simplify the presentation, we separate instruction selection from temporary allocation and employ a simple instruction selection strategy as demonstrated in Figure 4.

3.2 The $\mathcal{CG}$ Function

The code generator $\mathcal{CG}$ takes an expression from the target language, an environment, a data destination, and a control destination, and returns assembly code in the object language:

$$\mathcal{CG} : \textit{Expression} \rightarrow \textit{Environment} \rightarrow \textit{Data Destination} \rightarrow \textit{Control Destination} \rightarrow \textit{Code}$$

The data destination is used to determine where the value of the expression parameter is to be placed. Data destinations are either actual locations as described in Section 3.1 or the special flag *effect*:

$$\delta \in \textit{Data Destination} = \textit{effect} \mid \textit{Location}$$

When the data destination is *effect*, the value of the expression need not be stored in a location. However, the value of the expression may be needed to control the flow of execution, as determined by the control destination.

The control destination indicates where control is to be transferred after execution of the expression parameter. The control destination is either a label or a pair of labels:

$$\gamma \in \textit{Control Destination} = \textit{Label} \mid \textit{Label} \times \textit{Label}$$

A single label indicates unconditional transfer of control to the control destination. A return from a procedure body is generated when the label is *return*. If the control destination is a pair of labels, the label chosen for transfer of control depends on the boolean value of the expression. A pair of labels indicates that the expression is being evaluated as a test expression, and control is to be transferred to one of the labels depending upon the boolean value of the expression.

The control and data destinations implicitly separate expressions into four categories:

1. expressions evaluated for side effects, distinguished by the *effect* data destination and single control destinations;

2. expressions evaluated for their value, distinguished by location data destinations and single control destinations;

3. expressions evaluated for their effect on control flow (and possibly for side effects), distinguished by the *effect* data destination and paired control destinations; and

4. expressions evaluated both for their value and their effect on control flow, distinguished by location data destinations and paired control destinations.

Expressions in the first category are usually referred to as *statements*, and we shall adhere to that convention. However, here the categorization of an expression as a statement is a result of the actual *use* of an expression rather than a result of the expression falling into a certain *syntactic* category. Although some expressions, such as while loops, are naturally classified as statements, other expressions such as sequences or conditionals may be used either as statements or as subexpressions in a context where the value of the expressions is important. The last two categories, characterized by paired control destinations, are collectively referred to as *test expressions.* A test expression is used to conditionally control a program's execution, and thus has two control destinations, one of which must be selected depending upon the value of the test expression. Test expressions in the third category are evaluated only for their effect on control flow and possibly for side effects, and thus have the *effect* data destination. However, since assignments in C return values, they may be used as boolean expressions in the test part of a conditional, resulting in test expressions whose values must be both stored in a location and used to select the appropriate control destination.

The $\mathcal{CG}$ function is called for each procedure body with a data destination of *effect*, a control destination of *return*, and an initial environment:

$$\mathcal{CG}\ [\![\, E \,]\!]\ \rho_{init}\ \mathit{effect}\ \mathit{return}$$

The map component of the initial environment ρ_{init} must have locations for the parameters and local variables. This map is constant during code generation for the body. We assume that stack space has already been allocated for local variables. Since the control component of the environment is only used for breaks from loops, its initial value is irrelevant. The initial data destination is *effect*, since in C values must be explicitly returned from procedure calls. On the other hand, the initial control destination is *return*, since C procedures return implicitly when control "falls off the end" of a procedure body without encountering a **return** expression. As the abstract syntax tree for the body of the **fundecl** is traversed, the *return* destination is inherited by any expression that occurs in *tail position*. As a result, a return instruction is generated whenever there is nothing left to do but return. The *return* destination also indicates when procedure calls are in tail position and can therefore be used to implement proper tail recursion, as required by Scheme [13].

The rest of this section discusses the code generated for each construct in the source language of Section 2. We start with the control expressions, shown in Figure 5. These expressions are used to determine the flow of control in a program. Since **while** and one-armed **if** constructs make sense only in effect contexts, and in fact are restricted to such contexts by the syntax of C, the code generation function form them shown in Figure 5 ignores the data destination and assumes a single label for a control destination. On the other hand, **sequence** and two-armed **if** expressions can occur in either effect or value contexts.

The choice of data and control destinations for the subexpressions of a **sequence** reflect the imperative, sequential nature of this construct. The first subexpression is executed only for effect, so it is given *effect* as a data destination and the label of the second subexpression as a control destination. The second subexpression of a **sequence** inherits the data and control destination of the entire **sequence**.

An **if** expression must be able to select appropriately between its *then* and *else* parts based upon the boolean value of its *test* part. This is accomplished by calling $\mathcal{CG}$ on the test part with a paired control destination, consisting of the labels attached to the *then* and *else* subexpressions. The value of the *test* subexpression is needed only for its effect on control flow so its data destination is *effect*. The *then* and *else* subexpressions inherit the data and control destinations of the entire expression. Since single-armed **if** statements lack *else* subexpressions, the *test* expression uses the control destination of the entire expression as a false destination.

Code generation for a **while** loop is initiated by labeling the loop test expression; this label is then used as the control destination of the loop body. The original control destination is used both as a break destination and as the false destination of the loop test expression. The break destination is stored in the control component of the environment used for the loop subexpressions. (A **break** statement cannot appear within the loop test expression in C, so the actual control component for the test expression is irrelevant.) The control environment can also be used to store control destinations for other exceptional control transfers such as C **continue** and **goto** statements.

Values are returned from C procedures by **return** expressions. A **return** is accomplished simply by making the control destination of the **return** subexpression *return* and its data destination the accumulator.

$$
\begin{aligned}
&\mathcal{CG}\,[\![\,\textbf{sequence}\ E_1\ E_2\,]\!]\ \rho\ \delta\ \gamma\ \Rightarrow\\
&\quad \mathcal{CG}\,[\![\,E_1\,]\!]\ \rho\ \mathit{effect}\ L\\
&L\text{:}\\
&\quad \mathcal{CG}\,[\![\,E_2\,]\!]\ \rho\ \delta\ \gamma
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{CG}\,[\![\,\textbf{if}\ E_{test}\ E_{then}\ E_{else}\,]\!]\ \rho\ \delta\ \gamma\ \Rightarrow\\
&\quad \mathcal{CG}\,[\![\,E_{test}\,]\!]\ \rho\ \mathit{effect}\ (L_{then}, L_{else})\\
&L_{then}\text{:}\\
&\quad \mathcal{CG}\,[\![\,E_{then}\,]\!]\ \rho\ \delta\ \gamma\\
&L_{else}\text{:}\\
&\quad \mathcal{CG}\,[\![\,E_{else}\,]\!]\ \rho\ \delta\ \gamma
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{CG}\,[\![\,\textbf{if}\ E_{test}\ E_{then}\,]\!]\ \rho\ \delta\ L\ \Rightarrow\\
&\quad \mathcal{CG}\,[\![\,E_{test}\,]\!]\ \rho\ \mathit{effect}\ (L_{then}, L)\\
&L_{then}\text{:}\\
&\quad \mathcal{CG}\,[\![\,E_{then}\,]\!]\ \rho\ \mathit{effect}\ L
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{CG}\,[\![\,\textbf{return}\ E\,]\!]\ \rho\ \delta\ \gamma\ \Rightarrow\\
&\quad \mathcal{CG}\,[\![\,E\,]\!]\ \rho\ \mathit{ac}\ \mathit{return}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{CG}\,[\![\,\textbf{while}\ E_{test}\ E_{body}\,]\!]\ (L_{break}, m)\ \delta\ L\ \Rightarrow\\
&L_{test}\text{:}\\
&\quad \mathcal{CG}\,[\![\,E_{test}\,]\!]\ (L, m)\ \mathit{effect}\ (L_{body}, L)\\
&L_{body}\text{:}\\
&\quad \mathcal{CG}\,[\![\,E_{body}\,]\!]\ (L, m)\ \mathit{effect}\ L_{test}
\end{aligned}
$$

Figure 5: Control expressions.

Since the control destination for a loop is always a label, the code generated for a **break** statement could be simply a jump:

$$
\begin{aligned}
&\mathcal{CG}\,[\![\,\textbf{break}\,]\!]\ (L_{break}, m)\ \mathit{effect}\ \gamma\ \Rightarrow\\
&\quad \mathcal{CG}_{jump}\ L_{break}
\end{aligned}
$$

However, this can result in redundant jumps when a **break** statement occurs as a branch of an **if** statement or as the last statement of a **sequence** statement. Such jumps can be avoided by watching for **break** in the $\mathcal{CG}$ functions for **if** statements and **sequence** statements, as shown in Figure 6. If **break** is the second statement of a **sequence** statement, the break destination becomes the control destination of the first statement. Similarly, if **break** is a branch of an **if** statement, the break destination becomes one of the control destinations supplied to the test expression.

This treatment of **break** allows a simplified treatment of loops, since all loop forms can be transformed into endless loops with explicit exits without generating unnecessary jumps:

$$\textbf{while}\ E_{test}\ E_{body}\ \Rightarrow\ \textbf{loop}\ (\textbf{if}\ E_{test}\ E_{body}\ \textbf{break})$$

$$
\begin{array}{l}
\mathcal{CG}\,[\![\,\textbf{sequence}\ E\ \textbf{break}\,]\!]\ (L_{break}, m)\ \mathit{effect}\ \gamma\ \Rightarrow \\
\quad \mathcal{CG}\,[\![\,E\,]\!]\ (L_{break}, m)\ \mathit{effect}\ L_{break}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{CG}\,[\![\,\textbf{if}\ E\ \textbf{break}\,]\!]\ (L_{break}, m)\ \delta\ L\ \Rightarrow \\
\quad \mathcal{CG}\,[\![\,E\,]\!]\ (L_{break}, m)\ \mathit{effect}\ (L_{break}, L)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{CG}\,[\![\,\textbf{if}\ E_{test}\ \textbf{break}\ E_{else}\,]\!]\ (L_{break}, m)\ \delta\ \gamma\ \Rightarrow \\
\quad \mathcal{CG}\,[\![\,E_{test}\,]\!]\ (L_{break}, m)\ \mathit{effect}\ (L_{break}, L_{else}) \\
L_{else}: \\
\quad \mathcal{CG}\,[\![\,E_{else}\,]\!]\ (L_{break}, m)\ \delta\ \gamma
\end{array}
$$

$$
\begin{array}{l}
\mathcal{CG}\,[\![\,\textbf{if}\ E_{test}\ E_{then}\ \textbf{break}\,]\!]\ (L_{break}, m)\ \delta\ \gamma\ \Rightarrow \\
\quad \mathcal{CG}\,[\![\,E_{test}\,]\!]\ (L_{break}, m)\ \mathit{effect}\ (L_{then}, L_{break}) \\
L_{then}: \\
\quad \mathcal{CG}\,[\![\,E_{then}\,]\!]\ (L_{break}, m)\ \delta\ \gamma
\end{array}
$$

Figure 6: Optimizing **break** statements.

$\mathcal{CG}$ for the generic loop construct is a simplified version of $\mathcal{CG}$ for **while** loops:

$$
\begin{array}{l}
\mathcal{CG}\,[\![\,\textbf{loop}\ E\,]\!]\ (L_{break}, m)\ \mathit{effect}\ \gamma\ \Rightarrow \\
L_{top}: \\
\quad \mathcal{CG}\,[\![\,E\,]\!]\ (\gamma, m)\ \mathit{effect}\ L_{top}
\end{array}
$$

The code generated from the above transformations is identical to the code produced by the original treatment of **while** loops.

Assignments in C can occur in a value or effect context. When the data destination is *effect*, assignments are handled by calling $\mathcal{CG}$ on the subexpression with the current control destination and with the location of the variable as the data destination:

$$
\begin{array}{l}
\mathcal{CG}\,[\![\,\textbf{assign}\ x\ E\,]\!]\ (L_{break}, m)\ \mathit{effect}\ \gamma\ \Rightarrow \\
\quad \mathcal{CG}\,[\![\,E\,]\!]\ (L_{break}, m)\ m(x)\ \gamma
\end{array}
$$

However, when the data destination of the assignment is not *effect* there are two data destinations. This can be handled by treating the assignment as a **sequence** consisting of the original assignment followed by a reference to the assigned variable:

$$
\textbf{assign}\ x\ E\ \Rightarrow\ \textbf{sequence}\ (\textbf{assign}\ x\ E)\ (\textbf{var}\ x)
$$

It may be preferable to use a different intermediate location, such as the accumulator, or to avoid assignments in value contexts altogether by a simple code transformation prior to code generation.

We turn next to code generation for boolean-valued expressions. These expressions include relational operators such as "<" and boolean connectives such as **or**, **and**, and the unary operator **not**. Generating code for these expressions is straightforward when they are being used for their effect on control flow, *i.e.*, when they have a data destination of *effect* and a pair of control destinations.

Figure 7 shows how $\mathcal{CG}$ generates code for **or**, **and**, and **not** when they are being used as test expressions. If the first subexpression of an **or** expression evaluates to true the whole expression is true and the second subexpression is not evaluated. On the other hand, if the first subexpression of

$$
\begin{array}{l}
\mathcal{CG}\;[\![\,\textbf{or}\;E_1\;E_2\,]\!]\;\rho\;\mathit{effect}\,(L_{true}, L_{false})\;\Rightarrow\\
\quad\mathcal{CG}\;[\![\,E_1\,]\!]\;\rho\;\mathit{effect}\,(L_{true}, L)\\
L\!:\\
\quad\mathcal{CG}\;[\![\,E_2\,]\!]\;\rho\;\mathit{effect}\,(L_{true}, L_{false})
\end{array}
$$

$$
\begin{array}{l}
\mathcal{CG}\;[\![\,\textbf{and}\;E_1\;E_2\,]\!]\;\rho\;\mathit{effect}\,(L_{true}, L_{false})\;\Rightarrow\\
\quad\mathcal{CG}\;[\![\,E_1\,]\!]\;\rho\;\mathit{effect}\,(L, L_{false})\\
L\!:\\
\quad\mathcal{CG}\;[\![\,E_2\,]\!]\;\rho\;\mathit{effect}\,(L_{true}, L_{false})
\end{array}
$$

$$
\begin{array}{l}
\mathcal{CG}\;[\![\,\textbf{not}\;E\,]\!]\;\rho\;\mathit{effect}\,(L_{true}, L_{false})\;\Rightarrow\\
\quad\mathcal{CG}\;[\![\,E\,]\!]\;\rho\;\mathit{effect}\,(L_{false}, L_{true})
\end{array}
$$

Figure 7: Boolean expressions in conditional contexts.

an **or** expression evaluates to false, the second subexpression is evaluated and its value is the value of the complete expression. Consequently, the first subexpression receives the true destination of the **or** expression as a true destination and the label of the second expression as a false destination. The second subexpression simply inherits the control destination of the entire **or** expression. **and** expressions are treated similarly. Nowhere is the utility of the control destination better evidenced than in the $\mathcal{CG}$ function for **not**—the semantics of **not** are captured simply by swapping the true and false labels of the input control destination.

The relational operators require their operands in suitably simple forms. Often the original operands are complex and must be simplified. This simplification is performed by $\mathcal{CG}$ for binary operators, as shown in Figure 8. $\mathcal{CG}$ examines the operands and if necessary assigns temporaries and invokes $\mathcal{CG}$ on the operands with the temporaries as data destinations. We use the stack and accumulator as temporaries. Since the accumulator is a register it is the preferred temporary, and the stack is only used if both operands are complex. In Figure 8, S indicates simple expressions. Simple operands consist of variable references and constants. The auxiliary function $\mathcal{CG}_{operand}$ is used to translate simple expressions into operands:

$$
\begin{array}{l}
\mathcal{CG}_{operand} : \mathit{Simple\ Expression} \rightarrow \mathit{Environment} \rightarrow \mathit{Operand}\\
\mathcal{CG}_{operand}[\![\,\textbf{var}\;x\,]\!]\rho = \rho[\![\,x\,]\!]\\
\mathcal{CG}_{operand}[\![\,\textbf{int}\;n\,]\!]\rho = n
\end{array}
$$

The operator and locations of the values of the subexpressions are passed to $\mathcal{CG}_{binop}$, which generates code for the binary operation:

$$
\mathcal{CG}_{binop} : \mathit{Expression} \rightarrow \mathit{Operand} \rightarrow \mathit{Operand} \rightarrow \mathit{Data\ Destination} \rightarrow \mathit{Control\ Destination} \rightarrow \mathit{Code}
$$

Generating code for the relational operators is straightforward when they are being used as boolean control expressions. For example, here is $\mathcal{CG}_{binop}$ for $<$ with an *effect* data destination and a paired control destination:

$$
\begin{array}{l}
\mathcal{CG}_{binop}\;[\![\,<\,]\!]\;O_1\;O_2\;\mathit{effect}\,(L_{true}, L_{false})\;\Rightarrow\\
\quad\langle\!\langle\,\texttt{cmpl}\;\uparrow\! O_1\texttt{,}\;\;\uparrow\! O_2\,\rangle\!\rangle\\
\quad\mathcal{CG}_{branch}\;(\langle\!\langle\,\texttt{jlss}\,\rangle\!\rangle, L_{true})\;(\langle\!\langle\,\texttt{jgeq}\,\rangle\!\rangle, L_{false})
\end{array}
$$

The branching code is produced by the auxiliary function $\mathcal{CG}_{branch}$, which is discussed below.

$$
\begin{array}{l}
\mathcal{CG}\ [\![\, \textbf{binop}\ op\ S_1\ S_2 \,]\!]\ \ \rho\ \delta\ \gamma\ \ \Rightarrow \\
\quad \mathcal{CG}_{binop}\ [\![\, op \,]\!]\ (\mathcal{CG}_{operand}[\![\, S_1 \,]\!]\rho)\ (\mathcal{CG}_{operand}[\![\, S_2 \,]\!]\rho)\ \delta\ \gamma
\end{array}
$$

$$
\begin{array}{l}
\mathcal{CG}\ [\![\, \textbf{binop}\ op\ S\ E \,]\!]\ \ \rho\ \delta\ \gamma\ \ \Rightarrow \\
\quad \mathcal{CG}\ [\![\, E \,]\!]\ \ \rho\ ac\ L \\
L: \\
\quad \mathcal{CG}_{binop}\ [\![\, op \,]\!]\ ac\ (\mathcal{CG}_{operand}[\![\, S \,]\!]\rho)\ \delta\ \gamma
\end{array}
$$

$$
\begin{array}{l}
\mathcal{CG}\ [\![\, \textbf{binop}\ op\ E\ S \,]\!]\ \ \rho\ \delta\ \gamma\ \ \Rightarrow \\
\quad \mathcal{CG}\ [\![\, E \,]\!]\ \ \rho\ ac\ L \\
L: \\
\quad \mathcal{CG}_{binop}\ [\![\, op \,]\!]\ (\mathcal{CG}_{operand}[\![\, S \,]\!]\rho)\ ac\ \delta\ \gamma
\end{array}
$$

$$
\begin{array}{l}
\mathcal{CG}\ [\![\, \textbf{binop}\ op\ E_1\ E_2 \,]\!]\ \ \rho\ \delta\ \gamma\ \ \Rightarrow \\
\quad \mathcal{CG}\ [\![\, E_1 \,]\!]\ \ \rho\ stack\ L_1 \\
L_1: \\
\quad \mathcal{CG}\ [\![\, E_2 \,]\!]\ \ \rho\ ac\ L_2 \\
L_2: \\
\quad \mathcal{CG}_{binop}\ [\![\, op \,]\!]\ stack\ ac\ \delta\ \gamma
\end{array}
$$

Figure 8: Evaluating binary operations.

So far we have dealt only with boolean expressions in contexts where they are used for controlling conditional execution. In C, boolean expressions may also be used in value contexts, in which case they evaluate to either zero (false) or one (true). The simplest way to generate code for a boolean expression whose data destination is not *effect* is to convert it to an equivalent **if** expression that explicitly returns one or zero:

$$E_{bool}\ \Rightarrow\ \textbf{if}\ E_{bool}\ (\textbf{int}\ 1)\ (\textbf{int}\ 0)$$

Since the boolean expression is now the test subexpression of an **if** statement, $\mathcal{CG}$ will assign it an *effect* data destination and a paired control destination, and it can be handled as described above.

It is also possible for a boolean expression to occur in a context in which the data destination is *effect*, but the control destination is not a pair. This situation is typically a result of using the results of the subexpressions of the boolean expression to control side effects. In this situation, **and** and **or** expressions can be treated as equivalent **if** expressions:

$$
\begin{array}{l}
\textbf{and}\ E_1\ E_2\ \Rightarrow\ \textbf{if}\ E_1\ E_2 \\
\textbf{or}\ E_1\ E_2\ \Rightarrow\ \textbf{if}\ (\textbf{not}\ E_1)\ E_2
\end{array}
$$

In the case of the relational operators and **not** expressions, since nothing will be done with the result of the operation it is only necessary that the subexpressions be evaluated if they can result in side effects:

$$
\begin{array}{l}
\textbf{binop}\ relop\ E_1\ E_2\ \Rightarrow\ \textbf{sequence}\ E_1\ E_2 \\
\textbf{not}\ E\ \Rightarrow\ E
\end{array}
$$

After code is generated for **relop** subexpressions and the required comparison is generated, the auxiliary function $\mathcal{CG}_{branch}$ is called to generate a branch in the flow of control:

$$\mathcal{CG}_{branch} : (Code \times Label) \times (Code \times Label) \rightarrow Code$$

$$\begin{array}{l} \mathcal{CG}_{branch}\ (jmp_1, L_1)\ (jmp_2, L_2)\ \Rightarrow \\ \quad \langle\!\langle\, jmp_1\ \ L_1 \,\rangle\!\rangle \\ \quad \mathcal{CG}_{jump}\ L_2 \\ \text{if } L_2 = return \text{ or } L_2 = L_{next} \text{ and } L_1 \neq return \\ \quad \langle\!\langle\, jmp_2\ \ L_2 \,\rangle\!\rangle \\ \quad \mathcal{CG}_{jump}\ L_1 \\ \text{otherwise} \end{array}$$

$\mathcal{CG}_{branch}$ receives two mnemonic-label pairs. One of the pairs is used to generate a conditional branch instruction; the label from the other pair is used by $\mathcal{CG}_{jump}$ to generate an unconditional branch or return instruction. Since *return* is not a valid target for a conditional branch, *return* labels are always passed to $\mathcal{CG}_{jump}$. Otherwise, if one of the labels refers to the next statement (indicated by L_{next} above), it is passed to $\mathcal{CG}_{jump}$, since no code is generated to fall through to the next statement.

Jumps and returns are generated by the $\mathcal{CG}_{jump}$ function, show below.

$$\begin{array}{l} \mathcal{CG}_{jump} : Label \rightarrow Code \\ \mathcal{CG}_{jump}\ return\ \Rightarrow\ \langle\!\langle\, \texttt{ret} \,\rangle\!\rangle \\ \mathcal{CG}_{jump}\ L_{next}\ \Rightarrow\ \langle\!\langle\ \rangle\!\rangle \\ \mathcal{CG}_{jump}\ L\ \Rightarrow\ \langle\!\langle\, \texttt{jbr}\ \ L \,\rangle\!\rangle, \quad \text{otherwise} \end{array}$$

If the label passed to $\mathcal{CG}_{jump}$ is *return*, a `ret` instruction is generated. If the label passed to $\mathcal{CG}_{jump}$ is the label of the next statement (L_{next}), no code is generated. Otherwise, a `jbr` instruction to the target label is generated.

Our discussion of $\mathcal{CG}_{branch}$ and $\mathcal{CG}_{jump}$ assume some mechanism for determining when a label is attached to the next statement. One approach is to provide a "next-label" argument to the code generation functions. Then, for instance, the $\mathcal{CG}$ function for **sequence** statements would be:

$$\begin{array}{l} \mathcal{CG}\ [\![\, \textbf{sequence}\ S_1\ S_2 \,]\!]\ \rho\ \delta\ \gamma\ L_{next}\ \Rightarrow \\ \quad \mathcal{CG}\ [\![\, S_1 \,]\!]\ \rho\ effect\ L\ L \\ L\text{:} \\ \quad \mathcal{CG}\ [\![\, S_2 \,]\!]\ \rho\ \delta\ \gamma\ L_{next} \end{array}$$

The constructs that remain—**arithop**, **call**, **var**, and **int**—are typically used to produce values. To simplify $\mathcal{CG}$ for **var** and **call** expressions we introduce a help function $\mathcal{CG}_{store}$:

$$\mathcal{CG}_{store} : Operand \rightarrow Data\ Destination \rightarrow Control\ Destination \rightarrow Code$$

It takes a source operand, a data destination and a control destination, stores the source in the data destination and transfers control as necessary to the control destination. In the simplest case, where the data destination is *effect* and the control destination is a label, $\mathcal{CG}_{store}$ ignores the source and uses $\mathcal{CG}_{jump}$ to generate a branch if necessary:

$$\begin{array}{l} \mathcal{CG}_{store}\ O\ effect\ L\ \Rightarrow \\ \quad \mathcal{CG}_{jump}\ L \end{array}$$

If the data destination is *effect* and the control destination is a pair of labels, the source is tested to set the condition codes and $\mathcal{CG}_{branch}$ is used to generate a conditional branch instruction:

$$\begin{aligned}
&\mathcal{CG}_{store}\ O\ \mathit{effect}\ (L_{true}, L_{false})\ \Rightarrow \\
&\quad \langle\!\langle\, \texttt{tstl}\ \uparrow O \,\rangle\!\rangle \\
&\quad \mathcal{CG}_{branch}\ (\langle\!\langle\, \texttt{jneq} \,\rangle\!\rangle, L_{true})\ (\langle\!\langle\, \texttt{jeql} \,\rangle\!\rangle, L_{false})
\end{aligned}$$

If the data destination is a location a store instruction is generated before the use of $\mathcal{CG}_{branch}$ or $\mathcal{CG}_{jump}$:

$$\begin{aligned}
&\mathcal{CG}_{store}\ O\ A\ L\ \Rightarrow \\
&\quad \langle\!\langle\, \texttt{movl}\ \uparrow O\texttt{,}\downarrow A \,\rangle\!\rangle \\
&\quad \mathcal{CG}_{jump}\ L
\end{aligned}$$

$$\begin{aligned}
&\mathcal{CG}_{store}\ O\ A\ (L_{true}, L_{false})\ \Rightarrow \\
&\quad \langle\!\langle\, \texttt{movl}\ \uparrow O\texttt{,}\downarrow A \,\rangle\!\rangle \\
&\quad \mathcal{CG}_{branch}\ (\langle\!\langle\, \texttt{jneq} \,\rangle\!\rangle, L_{true})\ (\langle\!\langle\, \texttt{jeql} \,\rangle\!\rangle, L_{false})
\end{aligned}$$

Variable or integer references are handled by calling $\mathcal{CG}_{store}$ directly:

$$\begin{aligned}
&\mathcal{CG}\ [\![\, \mathbf{var}\ x \,]\!]\ (L_{break}, m)\ \delta\ \gamma\ \Rightarrow \\
&\quad \mathcal{CG}_{store}\ (m[\![\, x \,]\!])\ \delta\ \gamma \\
&\mathcal{CG}\ [\![\, \mathbf{int}\ n \,]\!]\ \rho\ \delta\ \gamma\ \Rightarrow \\
&\quad \mathcal{CG}_{store}\ n\ \delta\ \gamma
\end{aligned}$$

The $\mathcal{CG}$ function for **call** adheres to the VAX calling conventions, which requires argument values to be pushed onto the stack in reverse order. After code for the call has been generated $\mathcal{CG}_{store}$ is called with the accumulator as a source location:

$$\begin{aligned}
&\mathcal{CG}\ [\![\, \mathbf{call}\ f\ E_1 \ldots E_n \,]\!]\ \rho\ \delta\ \gamma\ \Rightarrow \\
&\quad \mathcal{CG}\ [\![\, E_n \,]\!]\ \rho\ \mathit{stack}\ L_n \\
&L_n\!: \\
&\qquad\qquad \vdots \\
&\quad \mathcal{CG}\ [\![\, E_1 \,]\!]\ \rho\ \mathit{stack}\ L_1 \\
&L_1\!: \\
&\quad \langle\!\langle\, \texttt{calls \$}n\texttt{,}\ f \,\rangle\!\rangle \\
&\quad \mathcal{CG}_{store}\ \mathit{ac}\ \delta\ \gamma
\end{aligned}$$

The binary arithmetic operations are handled much like binary relational operations, except here a location data destination is the norm rather than the exception. If the data destination is *effect* rather than a location, it is simplest to use the accumulator as a temporary location:

$$\begin{aligned}
&\mathcal{CG}_{binop}\ [\![\, \mathit{arithop} \,]\!]\ A_1\ A_2\ \mathit{effect}\ \gamma\ \Rightarrow \\
&\quad \mathcal{CG}_{binop}\ [\![\, \mathit{arithop} \,]\!]\ A_1\ A_2\ \mathit{ac}\ \gamma
\end{aligned}$$

If the data destination is a location and the control destination is a single label, $\mathcal{CG}_{jump}$ is called to generate a jump if necessary, as in the following example for subtraction:

$$\begin{aligned}
&\mathcal{CG}_{binop}\ [\![\, - \,]\!]\ A_1\ A_2\ A_3\ L\ \Rightarrow \\
&\quad \langle\!\langle\, \texttt{subl3}\ \uparrow A_2\texttt{,}\uparrow A_1\texttt{,}\downarrow A_3 \,\rangle\!\rangle \\
&\quad \mathcal{CG}_{jump}\ L
\end{aligned}$$

Otherwise, if the control destination is a pair of labels, $\mathcal{CG}_{branch}$ is called to generate a branch:

$$\begin{array}{l} \mathcal{CG}_{binop}\ [\![-]\!]\ A_1\ A_2\ A_3\ (L_{true}, L_{false}) \ \Rightarrow \\ \quad \langle\!\langle\ \texttt{subl3}\ \uparrow\! A_2\texttt{,}\uparrow\! A_1\texttt{,}\downarrow\! A_3\ \rangle\!\rangle \\ \quad \mathcal{CG}_{branch}\ (\langle\!\langle\ \texttt{jneq}\ \rangle\!\rangle, L_{true})\ (\langle\!\langle\ \texttt{jeql}\ \rangle\!\rangle, L_{false}) \end{array}$$

4 Conclusions

Destination-driven code generation is a simple technique for generating efficient target code directly from abstract syntax trees. The abstract syntax trees received as input by a destination-driven code generator may be produced directly by the front end of a compiler, or they may be the result of earlier code improvement and register allocation passes that operate on intermediate forms of the abstract syntax trees produced by the front end. We envision a compiler back end in which each of the intermediate passes between the front end and the code generator are optional. Each pass takes as input a given language of abstract syntax trees, possibly with annotations resulting from data or control flow analysis, and produces (presumably) equivalent programs in the same language. The speed of the compiler and the quality of the code generated depends on which of the optional passes, if any, are selected.

It is useful to include a simplification prepass that reduces the size and complexity of the language. This prepass can perform many of the transformations described in this article, not only simplifying the code generator but also simplifying the intermediate, optional passes between the simplification prepass and the code generator.

The code generator described in Section 3 creates temporary stack locations for some complex arithmetic subexpressions occurring in value contexts. A better solution is to introduce temporaries during the simplification prepass. The resulting language of abstract syntax trees would contain no doubly-nested subexpressions, and all variables, whether present in the source or introduced by the compiler, would be recorded in the appropriate fields of the **fundecl** construct. By separating the introduction of temporaries from the allocation of locations it becomes possible to do live analysis and splitting of temporaries, followed by register allocation and assignment, before code generation.

It is often desirable to arrange the code for a **while** loops so that the test expression follows the loop body. This can result in one fewer instruction (a branch "not taken") in the loop. It is not difficult to do so within the framework we have described, but it does require the insertion of a jump instruction immediately before the **while** loop to the test expression. This jump instruction is redundant when the **while** loop is the target of a jump instruction. Also, the code for nested **if** expressions, *i.e.*, **if** expressions whose test expressions are themselves **if** expressions, sometimes contains redundant jumps, because our mechanism does not always result in the optimal ordering of the code for the various subexpressions involved in nested conditional expressions. It is possible to solve these problems by augmenting the code generator to return not only the code generated by an expression but also the entry point into the code, which is now always assumed to be the first instruction. Unfortunately, this cannot be made to work fully without introducing a second pass to perform "back-patching" for loop exits, and these problems occur rarely enough in practice that this additional effort is not clearly worth the cost.

While the use of a destination-driven code generator results in reasonable code without peephole optimization, some opportunities for optimization at the target code level still exist. For example, we have addressed instruction selection at the granularity of single instructions, and the choices made will not be optimal for all machines. A peephole optimizer that recognizes certain machine-dependent combinations of instructions may therefore be beneficial. Also, instruction scheduling employed to maintain good pipeline and cache behavior seems to make sense only at the target code level. However, optimizations requiring data and control flow analysis are less likely to be necessary when the target code is derived via destination-driven code generation, removing a source of complexity and potential unreliability from the target code optimizer.

Destination-driven code generation provides a simple and economical paradigm for the construction of syntax-directed code generators. The introduction of data and control destinations allows a destination-driven code generator to communicate decisions about the placement of code and data in a clear and concise manner. The destination-driven technique produces target code that rarely contains unnecessary move or jump instructions. Furthermore, because the technique is based upon a recursive, top-down traversal of an abstract syntax tree, destination-driven code generation is highly intuitive and easily codified.

We have used destination-driven code generation techniques in the implementation of an incremental optimizing compiler for the Scheme programming language that generates assembly code "on the fly" for one of several architectures, including the VAX, MC68000, MC88000, and SPARC. The code generator is parameterized by a table of machine-specific code generators for various low-level operations.

References

[1] Aho, A. V., Ganapathi, M., and Tjiang, S. W. K. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems 11*, 4 (1989), 491–516.

[2] Aho, A. V., and Johnson, S. C. Optimal code generation for expression trees. *Journal of the ACM 23*, 3 (1976), 488–501.

[3] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.

[4] Davidson, J. W., and Fraser, C. W. The design and application of a retargetable peephole optimizer. *ACM Transactions on Programming Languages and Systems 2*, 2 (1980), 191–202.

[5] Digital Equipment Corporation. *VAX Architecture Handbook*, 1986.

[6] Ganapathi, M., and Fischer, C. N. Affix grammar driven code generation. *ACM Transactions on Programming Languages and Systems 11*, 4 (1985), 560–599.

[7] Ganapathi, M., Fischer, C. N., and Hennessy, J. L. Retargetable compiler code generation. *ACM Computing Surveys 14*, 4 (1982), 573–592.

[8] Glanville, R. S. *A machine independent algorithm for code generation and its use in retargetable compilers.* PhD thesis, University of California, Berkeley, 1977.

[9] Glanville, R. S., and Graham, S. L. A new method for compiler code generation. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages* (1978), pp. 231–240.

[10] Graham, S. L. Table-driven code generation. *IEEE Computer 13*, 8 (1980), 25–34.

[11] KERNIGHAN, B. W., AND RITCHIE, D. M. *The C Programming Language.* Prentice Hall, 1978.

[12] RAMANATH, M. V. S., AND SOLOMON, M. Jump minimization in linear time. *ACM Transactions on Programming Languages and Systems 6*, 4 (1984), 527–545.

[13] REES, J., AND CLINGER, W. E. Revised^3 report on the algorithmic language Scheme. *ACM SIGPLAN Notices 21*, 12 (1986).

[14] REISER, J. F., AND HENRY, R. R. *Berkeley VAX/UNIX assembler reference manual*, 1983.

[15] SETHI, R., AND ULLMAN, J. The generation of optimal code for arithmetic expressions. *Journal of the ACM 17*, 4 (1970), 715–728.

[16] STEELE, G. L. J. Rabbit: A compiler for Scheme (a study in compiler optimization). Master's thesis, Massachusetts Institute of Technology, 1977.