Implementing Control Flow Constructs

Comp 412

Copyright 2017, Keith D. Cooper & Linda Torczon, all rights reserved.
Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.
Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Chapter 7 in EaC2e
Boolean & Relational Expressions

First, we need to add Boolean & relational expressions to the grammar

\[
\begin{align*}
\text{Boolean} & \quad \to \quad \text{Boolean} \lor \text{AndTerm} & \quad \text{Expr} & \quad \to \quad \text{Expr} + \text{Term} \\
& \quad \mid \quad \text{AndTerm} & & \quad \mid \quad \text{Expr} - \text{Term} \\
\text{AndTerm} & \quad \to \quad \text{AndTerm} \land \text{RelExpr} & & \quad \mid \quad \text{Term} \\
& \quad \mid \quad \text{RelExpr} & & \quad \mid \quad \text{Term} \times \text{Value} \\
\text{RelExpr} & \quad \to \quad \text{RelExpr} < \text{Expr} & & \quad \mid \quad \text{Term} \div \text{Value} \\
& \quad \mid \quad \text{RelExpr} \leq \text{Expr} & & \quad \mid \quad \text{Value} \\
& \quad \mid \quad \text{RelExpr} = \text{Expr} & & \quad \mid \quad ! \text{Factor} \\
& \quad \mid \quad \text{RelExpr} \neq \text{Expr} & & \quad \mid \quad \text{Factor} \\
& \quad \mid \quad \text{RelExpr} \geq \text{Expr} & & \quad \mid \quad ( \text{Expr} ) \\
& \quad \mid \quad \text{RelExpr} > \text{Expr} & & \quad \mid \quad \text{number} \\
& \quad \mid \quad \text{Expr} & & \quad \mid \quad \text{Reference}
\end{align*}
\]

This grammar allows $w < x < y < z$

... where Reference derives a name, a subscripted name, a structure reference, a string reference, a function call, ...
Boolean Values

The compiler needs to represent the values TRUE and FALSE

Numerical representation

• Assign values to TRUE and FALSE (design time)
  – Common choices are 1 / 0, -1 / 0, or non-zero / 0
  – Not a lot of difference between the choices
• Use hardware AND, OR, and NOT operations

⇒ Make a choice and use it uniformly

Implementation of Boolean values and relational comparisons depends heavily on features of the ISA.
Relational Values

Different instruction sets (ISAs) provide different kinds of support

<table>
<thead>
<tr>
<th>Comparison Produces</th>
<th>ISA Impact</th>
<th>Impact on Code Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean condition code</td>
<td>multiple compare ops</td>
<td>⇒ complexity into expressions</td>
</tr>
<tr>
<td></td>
<td>multiple branch ops</td>
<td>⇒ complexity into branches</td>
</tr>
</tbody>
</table>

Branch consumes what compare produces
• Fundamental design issue: operations intended to work together
• Most ISAs settle on a single model

Implementation of Boolean values and relational comparisons depends heavily on features of the ISA.
Boolean & Relational Values

Boolean-valued comparisons
• Code can use the result of a relational comparison directly
• Presume that Boolean ops (e.g., AND, OR, NOT) work on result
• Leads to straightforward translation

Examples

\[
\begin{align*}
x < y & \quad \text{becomes} \quad \text{cmp}_\text{LT} \quad r_x, r_y \implies r_1 \\
\text{if } (x < y) & \quad \text{then stmt}_1 \quad \text{becomes} \quad \text{cmp}_\text{LT} \quad r_x, r_y \implies r_1 \\
\text{else stmt}_2 & \quad \text{cbr} \quad r_1 \quad \rightarrow \text{L_stmt}_1, \text{L_stmt}_2
\end{align*}
\]

r_1 contains a Boolean value
Boolean & Relational Values

When compare produces a condition code
• Must use a branch to interpret result of compare
• Necessitates branches in the evaluation

Example

If \textbf{cmp} sets a condition-code register, rather than returning a Boolean value, the implementation of \texttt{x < y} becomes much more complex.
Boolean & Relational Values

When compare produces a condition code
• Must use a branch to interpret result of compare
• Necessitates branches in the evaluation

Example

if (x < y) then \textit{stmt}_1 becomes
\begin{align*}
\text{cmp} & \quad r_x, r_y \Rightarrow CC_1 \\
\text{cbr}_{\text{LT}} & \quad CC_1 \rightarrow L_T, L_F \\
\text{L}_T: & \quad \text{stmt}_1 \\
\text{br} & \quad \rightarrow L_E \\
\text{L}_F: & \quad \text{stmt}_2 \\
\text{br} & \quad \rightarrow L_E \\
\text{L}_E: & \quad \ldots \text{other statements} \ldots
\end{align*}

If the sole use of the relational expression’s value is to control an if-then-else construct, then we can fold the statements into the evaluation.
Boolean & Relational Values

When compare produces a condition code:
- Must use a branch to interpret result of compare
- Necessitates branches in the evaluation

Example

if (x < y) then \( stmt_1 \) becomes
else \( stmt_2 \)

This code represents the result of the comparison implicitly, in the PC

This idea works on most ISAs and in many circumstances

If the sole use of the relational expression’s value is to control an if-then-else construct, then we can fold the statements into the evaluation.
Other Ops That Interpret A Comparison

ISAs sometimes include other ops that can interpret a comparison

**Conditional Move Operation**
- Chooses its argument based on result of the comparison
- Can envision either a Boolean or a CC-based version
  Avoids a disruptive branch

**Predicated Instructions**
- Allows a Boolean value to control execution of a single operation
- Typically, controls whether result is assigned or discarded
  Avoids a disruptive branch

Branch on a compare may be hard to predict
Implementing **IF-THEN-ELSE**

**Example**

if \((x < y)\)
then \(a \leftarrow c + d\)
else \(a \leftarrow e + f\)

Both avoid branches and lead to shorter code. Note, however, that the path lengths through the code are about the same.

**Bottom line:** your mileage may vary. The best implementation depends on specific details of the ISA.

<table>
<thead>
<tr>
<th>Conditional Move</th>
<th>Predicated Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{comp})</td>
<td>(\text{cmp}_\text{LT})</td>
</tr>
<tr>
<td>(r_x,r_y)</td>
<td>(r_x,r_y)</td>
</tr>
<tr>
<td>(\Rightarrow \text{CC}_1)</td>
<td>(\Rightarrow r_1)</td>
</tr>
<tr>
<td>(\text{add})</td>
<td>((r_1))</td>
</tr>
<tr>
<td>(r_c,r_d)</td>
<td>(r_c,r_d)</td>
</tr>
<tr>
<td>(\Rightarrow r_1)</td>
<td>(\Rightarrow r_a)</td>
</tr>
<tr>
<td>(\text{add})</td>
<td>((!r_1))</td>
</tr>
<tr>
<td>(r_e,r_f)</td>
<td>(r_e,r_f)</td>
</tr>
<tr>
<td>(\Rightarrow r_2)</td>
<td>(\Rightarrow r_a)</td>
</tr>
<tr>
<td>(\text{i2i}_\text{LT})</td>
<td>(\text{CC}_1,r_1,r_2)</td>
</tr>
<tr>
<td>(\Rightarrow r_a)</td>
<td>(\Rightarrow r_a)</td>
</tr>
</tbody>
</table>

**Encodes** \(x < y\) **in** \(\text{CC}_1\), **but never creates a Boolean value**

**Creates an Boolean value for** \(x < y\)
Generating a branching **IF-THEN-ELSE** with SDT

To generate branching code with SDT, need to create & track the labels

**Stmt** \(\rightarrow\) **if** **Expr** \(\text{then WithElse}\) \(\text{else WithElse}\)

1. Create labels for “then part”, “else part”, and “exit”
   - Generate three labels and push them onto the stack
2. Emit branch to appropriate part (then or else) after **Expr**
   - Emit the branch followed by the label for **then part** (labelled **nop**)
3. At end of WithElse, emit branch to the exit
   - Emit a jump to the exit label followed by the label for the **else part** (labelled **nop**)
4. At end of second WithElse, emit branch to the exit
   - Emit a jump to the exit label followed by the label for the **exit** statement
Generating a predicated **IF-THEN-ELSE** with **SDT**

**The predicated code is easier to generate with SDT**

1. Evaluate the controlling *Expr* into a register
   - *Processor may use one or more dedicated predicate registers*
   - *May require both the predicate and its logical complement*

2. Generate a dense stream of ops for the **then** clause
   - *Predicate each instruction with the *Expr*’s value*

3. Generate a dense stream of ops for the **else** clause
   - *Predicate each instruction with the logical complement of the *Expr*’s value*

4. Count on the instruction scheduler to interleave them.
   - *Requires the scheduler to understand the anti-dependences created by the use of a predicate and its logical complement*
   - *Adds complication to the scheduler, but it comes from predication, not from this particular code-generation scheme*

*Your performance may vary.*
Loops

- Compiler should follow a uniform template
- Evaluate condition before loop \((\textit{if needed})\)
- Evaluate condition after loop
- Branch back to the top \((\textit{if needed})\)

Merges test with last block of loop body

\textbf{while, for, do, & until} all fit this basic model
Generating a **FOR** Loop

A typical for loop might be specified as follows:

\[
Stmt \rightarrow \text{FOR NAME} = \text{Expr}_1 \text{ TO } \text{Expr}_2 \text{ BY } \text{Expr}_3 \{ \text{Stmts} \}
\]

With the usual meaning

⇒ **NAME** is an index variable that starts with **Expr}_1 and runs to **Expr}_2 by increments of **Expr}_3

- The rules for the individual statements in **Stmts** will emit code for the loop body
- The rule for the loop must emit code for the control structure

**while, until, repeatuntil, & repeatwhile** all fit the same basic template
Generating a **FOR** Loop

**How should the code look? Two obvious choices ...**

- **Single copy of test**
  ```
  i ← 1
  loop:
  if (i > 100) goto end
  ... loop body ...
  i ← i + 1
  goto loop
  end: ... next statement ...
  ```

- **Two copies of test**
  ```
  i ← 1
  if (i > 100) goto end
  ... loop body ...
  i ← i + 1
  goto loop
  end:
  ... next statement ...
  ```

- One copy version may be easier to generate
  - Can generate initialization & test before body, increment and jump after

- Two copy version may optimize better
  - Single block loop body will incorporate increment and test in one block

See Dave Whalley’s paper in 1991 **PLDI**

COMP 412, Fall 2017
Generating a **FOR** Loop with **SDT**

Generating the single test version of the loop

\[
\text{Stmt} \rightarrow \text{FOR NAME} = \text{Expr}_1 \ TO \ \text{Expr}_2 \ \text{BY} \ \text{Expr}_3 \ \{ \text{Stmts} \}
\]

1. **Use bison**’s mid-production action to insert code after \(\text{Expr}_2\)
   - Initialize **NAME** to \(\text{Expr}_1\)
     - Let **SDT** generate code to evaluate \(\text{Expr}_1\) into a register
   - Test **NAME** against \(\text{Expr}_2\)
     - Let **SDT** generate code to evaluate \(\text{Expr}_2\) into a register
     - Compare against **NAME** and branch accordingly
     - Emit a label before the loop body

2. **On the reduction**, emit the end-of-loop code
   - Increment **NAME** by \(\text{Expr}_3\)
     - Let **SDT** generate code to evaluate \(\text{Expr}_2\) into a register
   - Jump to **loop**

** initialization  
** test  
** loop body  
** increment  

** COMP 412, Fall 2017 **
Generating a **FOR** Loop with **SDT**

### Generating the two test version of the loop

**Stmt** → **FOR** **NAME** = **Expr**$_1$ **TO** **Expr**$_2$ **BY** **Expr**$_3$ { **Stmts** }

1. **Use bison**’s mid-production action to insert code after **Expr**$_2$
   - Initialize **NAME** to **Expr**$_1$
     - Let **SDT** generate code to evaluate **Expr**$_1$ into a register
   - Test **NAME** against **Expr**$_2$
     - Let **SDT** generate code to evaluate **Expr**$_2$ into a register
     - Compare against **NAME** and branch accordingly
     - Emit a label before the loop body

2. On the reduction, emit the end-of-loop code
   - Increment **NAME** by **Expr**$_3$
     - Let **SDT** generate code to evaluate **Expr**$_3$ into a register
   - Generate the other form of the test and branch
     - This process involves evaluating some of the **Expr**’s multiple times
Generating a **FOR Loop with SDT**

**Generating the two test version of the loop**

\[
Stmt \rightarrow \text{FOR NAME} = \text{Expr}_1 \text{ TO } \text{Expr}_2 \text{ BY } \text{Expr}_3 \{ \text{Stmts} \}
\]

(1) Use **bison**’s mid-production action to insert code after \(\text{Expr}_2\)

- Initialize **NAME** to \(\text{Expr}_1\)
  - Let **SDT** generate code to evaluate \(\text{Expr}_1\) into a register
- Test **NAME** against \(\text{Expr}_2\)
  - Let **SDT** generate code to evaluate \(\text{Expr}_2\) into a register
  - Compare against **NAME** and branch accordingly
  - Emit a label before the loop body

(2) On the reduction, emit the end-of-loop code

- Increment **NAME** by \(\text{Expr}_3\)
  - Let **SDT** generate code to evaluate \(\text{Expr}_3\) into a register
- Generate the other form of the test and branch
  - This process involves evaluating some of the \(\text{Expr’s}\) multiple times

In general, the compiler needs a data structure to hold the information for the loop. One way to handle the second test is to include a buffer of code in that “loop structure”. The compiler generates the test into the buffer, then copies it into each place where it is needed.
Code For a Two-Test **FOR** Loop

for (i = 1; i < 100; i++)
   
   { *loop body* }

*next statement*

---

The two-test template produces a code shape for loops that works well for optimization. ([271] in EaC2e)

---

### Initialization

- `loadl 1 ⇒ r_1`
- `loadl 1 ⇒ r_2`
- `loadl 100 ⇒ r_3`
- `cmp_GE r_1,r_3 ⇒ r_4`
- `cbr r_4 → L_2,L_1`

### Pre-test

- `L_1:` *loop body*
- `add r_1,r_2 ⇒ r_1`
- `cmp_LT r_1,r_3 ⇒ r_5`
- `cbr r_5 → L_1,L_2`

### Post-test

- `L_2:` *next statement*

---

Text shows examples of other kinds of loops translated into the basic template.
Break statements

Many programming languages include a break statement

Exits from the innermost control-flow statement

• Out of the innermost loop
• Out of a case statement

Break translates into a jump

• Targets statement outside the control-flow construct
• Creates a multiple-exit construct
• Skip in a loop goes to next iteration

**break** only makes sense if loop has > 1 block
**skip** or **continue** becomes branch to post-test block
Control Flow

Case Statements
1. Evaluate the controlling expression
2. Branch to the selected case
3. Execute the code for that case
4. Branch to the statement after the case

Parts 1, 3, & 4 are well understood, part 2 is the key
Control Flow

Case Statements
1. Evaluate the controlling expression
2. Branch to the selected case
3. Execute the code for that case
4. Branch to the statement after the case

Parts 1, 3, & 4 are well understood, part 2 is the key

Strategies
• Linear search (nested if-then-else constructs)
• Build a table of case expressions & binary search it
• Directly compute an address (requires dense case set)

Case statements are a place where attention to code shape pays off.
Case Statement Implementation

Using Linear Search

```c
switch( expr ) {
    case 0:    block_0;
                break;
    case 17:   block_17;
                break;
    case 23:   block_23;
                break;
    default:   block_d;
                break;
}
```

Case Statement

Implementation

- Cost of execution depends on frequency of values for `expr`
- Clear benefit to reordering cases
Case Statement Implementation

Using Binary Search

switch(expr) {
    case 0: block_0; break;
    case 17: block_{17}; break;
    ... ...
    case 99: block_{99}; break;
    default: block_d; break;
}

Case Statement

- Cost of execution depends on log_2 of number of cases
- Separate label for default case

Implementation

<table>
<thead>
<tr>
<th>Value</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LB_0</td>
</tr>
<tr>
<td>17</td>
<td>LB_{17}</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>99</td>
<td>LB_{99}</td>
</tr>
</tbody>
</table>

Search Table

temp ←expr;
down ←0
up ←n
while (down + 1 < up) {
    middle ←(up + down) / 2;
    if (Table.value[middle] ≤ temp)
        then down ←middle;
    else up ←middle;
}

if (Table.value[down] == temp)
    then jump to Table.label[down];
else jump to label for default;
Case Statement Implementation

Using a Jump Table

\[ \text{switch(} \text{expr}) \{ \]
\begin{align*}
\text{case } 0 &: \text{ block}_0; \\
&\quad \text{break;}
\text{case } 1 &: \text{ block}_1; \\
&\quad \text{break;}
\text{case } 2 &: \text{ block}_2; \\
&\quad \text{break;}
\vdots \\
\text{case } 9 &: \text{ block}_9; \\
&\quad \text{break;}
\text{default} &: \text{ block}_d; \\
&\quad \text{break;}
\}
\]

\text{temp} \leftarrow \text{expr};

\begin{align*}
\text{If } (0 \leq \text{temp} \leq 9) \text{ then } \{ \\
&\quad \text{addr} \leftarrow \text{@LT} + \text{temp} \times 4 \\
&\quad \text{target} \leftarrow \text{Mem[addr]} \\
&\quad \text{jump to target}
\}
\end{align*}

\text{else jump to LB}_d

\begin{align*}
\text{Case Statement}
\end{align*}

\begin{itemize}
  \item \text{Cost is } O(1)
  \item \text{Can optimize code further}
\end{itemize}

\text{Implementation}

\begin{align*}
\text{If “expr” is one byte, use a full table and avoid the if-then-else}
\end{align*}

COMP 412, Fall 2017
Case Statement Implementation

Using a Jump Table

temp ← expr;

If (0 ≤ temp ≤ 9) then {
    addr ← @(LT + temp * 4
    target ← Mem[addr]
    jump to target
}
else jump to LB

Original Code

Mapping into low-level operations (e.g., ILOC)

• We need the label, @LT, in a register
  – If case statement is in a loop, load @LT before the loop (LICM)
  – Could put @LT in the AR, but that requires initializations

• Code maps cleanly into a few operations (say 3?)

COMP 412, Fall 2017
Generating a **CASE** Statement With **SDT**

The grammar for a case statement might resemble:

\[
\begin{align*}
  \text{CaseStmt} & \rightarrow \text{SWITCH} (\text{Expr}) \text{ INTO} \{ \text{CaseList} \} ; \\
  \text{CaseList} & \rightarrow \text{Caselist \ OneCase} \\
    & \mid \text{OneCase} \\
  \text{OneCase} & \rightarrow \text{CASE \ LABEL} : \text{Stmt} ; \text{BREAK} ;
\end{align*}
\]

The difficulty is deciding how to implement the “switch”

- Need to see the cases before making the decision
- The reductions occur in the “wrong” order
Generating a **CASE** Statement With **SDT**

1. At (1), create and preserve an **EVAL** label and an **EXIT** label, then emit a jump to the **EVAL** label.

2. In each **OneCase**, create a code label, preserve it in a list of `<case label, code label>` pairs, emit the code label, and generate the **Stmt**.

3. After **Stmt**, emit a jump to the **EXIT** label.

4. At (2), analyze the list of labels, choose an evaluation strategy, emit the **EVAL** label, then emit the code to evaluate the **Expr** and jump to the selected case label.

5. Emit the **EXIT** label.

---

**CaseStmt** → \( \text{SWITCH}\ \{\text{Expr}\}\ \text{INTO}\ \{\text{CaseList}\}\ )

**CaseList** → \( \text{Caselist\ OneCase}_1\ \text{OneCase}_2\ |

**OneCase** → \( \text{CASE}\ \text{LABEL}\ :\ \text{Stmt} ;\ \text{BREAK} ;\ )
Generating a **CASE** Statement With **SDT**

The resulting code has the following shape:

<table>
<thead>
<tr>
<th>Code Block</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CL₁:</td>
<td>code for first case</td>
</tr>
<tr>
<td></td>
<td>jump EXIT</td>
</tr>
<tr>
<td>CL₂:</td>
<td>code for second case</td>
</tr>
<tr>
<td></td>
<td>jump EXIT</td>
</tr>
<tr>
<td>CLₙ:</td>
<td>code for second case</td>
</tr>
<tr>
<td></td>
<td>jump EXIT</td>
</tr>
<tr>
<td>EVAL:</td>
<td>code to evaluate Expr to a label CLᵢ and jump to CLᵢ</td>
</tr>
<tr>
<td>EXIT:</td>
<td>nop</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Step (1)**
- Jump `EVAL`

**Step (2)**
- Jump `EXIT`

**Step (3)**
- Jump `EXIT`

**Step (4)**
- Jump `EXIT` and jump to `CLᵢ`

**Step (5)**
- NOP

While building the case list

While walking the case list
Implementing the Switch (Step 4)

When Step (4) executes in SDT, all the information is available

• Compiler has a list of the case labels and their code labels
• Compiler has access to the result of evaluating $Expr$
  – Step (4) happens on reduction by $CaseStmt$ production (at vertical line 2)
• Compiler can implement any of the three schemes for switch

Choosing the method

• Compiler can walk the list and do some analysis
  – Look at number of labels & density of labels
• Choose among if-then-else, binary search, or a jump table
Implementing the Switch (Step 4)

When Step (4) executes in SDT, all the information is available

- Compiler has a list of the case labels and their code labels
- Compiler has access to the result of evaluating $Expr$
  - Step (4) happens on reduction by $CaseStmt$ production (at vertical line 2)
- Compiler can implement any of the three schemes for switch

Using an if-then-else structure

- Compiler walks the list of <case label, code label> pairs
  - For each pair, generate a test on $Expr = $case label, with branch to code label
  - Use the taken case for the code label & the fall through for the next case
    - Reduces average penalty
    - Handles default as the last `else` clause
- Creates the expected code
- An extra transfer of control per case, beyond other two methods
Implementing the Switch (Step 4)

When Step (4) executes in SDT, all the information is available

- Compiler has a list of the case labels and their code labels
- Compiler has access to the result of evaluating $Expr$
  - Step (4) happens on reduction by $CaseStmt$ production (at vertical line 2)
- Compiler can implement any of the three schemes for switch

Using a binary search

- Compiler walks the list of <case label, code label> pairs
  - Generates a sorted table of the pairs, which it statically initializes
    → Assembler directive
- Inserts either “canned” inline code or a call to library routine
  - Followed by a jump to the matching code label
- One extra transfer of control per case as price of SDT
Implementing the Switch (Step 4)

When Step (4) executes in SDT, all the information is available

• Compiler has a list of the case labels and their code labels
• Compiler has access to the result of evaluating $Expr$
  – Step (4) happens on reduction by $CaseStmt$ production (at vertical line 2)
• Compiler can implement any of the three schemes for switch

Using a jump table

• Compiler walks the list of <$case\ label, code\ label>$ pairs
  – Generates an ordered table of code labels
• Inserts index on $Expr$ into the table, followed by jump
• One extra transfer of control per case as price of SDT
Generating a **CASE** Statement With **SDT**

**The Bottom Line**
- With some careful planning, you can generate good code for a case statement using **SDT**
- The resulting code will have extraneous **jump** operations
  - Optimization can eliminate them (branch straightening)
  - The code for the if-then-else implementation is particularly ugly

**The alternative**
- Compiler could use **SDT** to build a small **AST**
- Compiler could walk the **AST** multiple times to analyze, and then generate the right code for the case statement