Implementing Control-Flow Constructs
Comp 412

A lecture in which we will push SDT in directions you had never contemplated.
What is Left in Translation?

We have talked about the mechanism of translation. Next, we need to focus on the product, of translation.

Expressions

• Type-related issues in expressions  
  – Mixed-type operations and value conversions.

• Assignment

• More operators in expressions  
  – Boolean expressions, relational expressions, string operations

Control Flow

• Code shape and translation strategies  

Procedure and Function Calls

• Functionality: what, precisely, does a procedure call do?  
• Implementation: how does the compiler emit code to do that?
More operators, more levels of precedence

| Boolean          | →  Boolean ∨ AndTerm                                                                 |
|                 | | AndTerm                                                                                     |
| AndTerm         | →  AndTerm ∧ RelExpr                                                                 |
|                 | | RelExpr                                                                                      |
| RelExpr         | →  RelExpr < Expr                                                                 |
|                 | | RelExpr ≤ Expr                                                                              |
|                 | | RelExpr = Expr                                                                              |
|                 | | RelExpr ≠ Expr                                                                              |
|                 | | RelExpr ≥ Expr                                                                              |
|                 | | RelExpr > Expr                                                                              |
|                 | | Expr                                                                                         |

| Expr            | →  Expr + Term                                                                              |
|                 | | Expr - Term                                                                                 |
|                 | | Term                                                                                         |
| Term            | →  Term × Value                                                                              |
|                 | | Term ÷ Value                                                                                 |
| Value           | →  ! Factor                                                                                   |
| Factor          | →  ( Expr )                                                                                  |
| number          | | Reference                                                                                   |

... where Reference derives a name, a subscripted name, a structure reference, a string reference, a function call, ...
Translation can be pretty simple and formulaic

- Assign values to `true` and `false`
- Implement a translation scheme
  - *Binary operators, similar to arithmetic*

Numerical representation

- Assign values to `TRUE` and `FALSE` at design time
- Use hardware `AND`, `OR`, and `NOT` operations

Implicit representation

- Represent the Boolean value as an implicit function of the PC
- Works well if the only use of the Boolean value is to control branches

**Best choice depends on context, hardware, and available optimization**

\[ (x = 0) \]

\[ \text{true} \quad \text{false} \]

\[ \text{x is zero} \quad \text{x is nonzero} \]

\[ \text{...} \]
Different instruction sets (ISAs) provide different kinds of support

<table>
<thead>
<tr>
<th>Comparison Produces</th>
<th>Relational Values</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

Processor architect makes these choices
Compiler writer is expected to make the architect look smart.
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

<table>
<thead>
<tr>
<th>x &lt; y</th>
<th>becomes</th>
<th>cmp_LT r_x, r_y (\Rightarrow r_1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (x &lt; y) then stmt_1 else stmt_2</td>
<td>becomes</td>
<td>L1: (stmt_1) jump (\rightarrow L3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>L2: (stmt_2) jump (\rightarrow L3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>L3: nop</td>
</tr>
</tbody>
</table>

\(r_1\) contains a Boolean value
When comparison produces a condition code

- Must use a branch to interpret result of compare
- Necessitates branches in the evaluation

### Example

| $x < y$ | becomes | $\text{cmp } r_x, r_y \Rightarrow \text{CC}_1$
|---------|---------|----------------------------------|
|         | $\text{cbr}_{LT}$ | $\text{CC}_1 \Rightarrow \text{L}_T, \text{L}_F$
| $\text{L}_T$: | $\text{loadI}$ | $\text{TRUE} \Rightarrow r_2$
|         | jump | $\Rightarrow \text{L}_E$
| $\text{L}_F$: | $\text{loadI}$ | $\text{FALSE} \Rightarrow r_2$
|         | jump | $\Rightarrow \text{L}_E$
| $\text{L}_E$: | ... other statements ... |

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

When comparison produces a condition code:

- Must use a branch to interpret result of comparison
- Necessitates branches in the evaluation

Example

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

becomes

\[
\begin{align*}
\text{cmp} & \quad r_x, r_y \Rightarrow CC_1 \\
\text{cbr}_{LT} & \quad CC_1 \rightarrow L_T, L_F \\
L_T: & \quad stmt_1 \\
\text{jump} & \quad \rightarrow L_E \\
L_F: & \quad stmt_2 \\
\text{jump} & \quad \rightarrow L_E \\
L_E: & \quad \ldots other statements \ldots
\end{align*}
\]

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.
Translating an **IF-THEN-ELSE** with SDT

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

\[
Stmt \rightarrow \text{if } Expr \text{ then } WithElse \text{ else } WithElse
\]

1. Create labels for “then part”, “else part”, and “exit”
   - Generate three labels and push them onto the stack
     Emit branch to appropriate part (then or else) after Expr
   - Emit the branch followed by the label for **then part** (labelled \texttt{nop})

1. At end of \texttt{WithElse}, emit branch to the exit
   - Emit a jump to the exit label followed by the label for the **else part** (labelled \texttt{nop})

2. At end of second \texttt{WithElse}, emit branch to the exit
   - Emit a jump to the exit label followed by the exit label (labelled \texttt{nop})

Need a structure to hold 3 labels due to nested if-then-else’s
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.*
Generating a **predicated** IF-THEN-ELSE

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

**NOTE:** Implementing the if-then-else with predicated operations turns multiple blocks with control flow into a single longer block.

This approach builds longer blocks with more opportunities for the scheduler. That may, or may not, lead to better code.

Scheduling is still **NP-Complete**.

Your performance may vary.
Control Flow

Loops

• Compiler should follow a uniform template
• Evaluate condition before loop (*if needed*)
• Evaluate condition after loop
• Branch back to the top (*if needed*)

Merges test with last block of loop body

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

A typical for loop might be specified as follows:

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

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 translation 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 ...**

```plaintext
for i = 1 to 100 by 1
{ ...
```

**Single copy of test**

```
i ⇔ 1
loop: if (i > 100) branch to end
    ... loop body ...
i ⇔ i + 1
jump to loop
end: ... next statement ...
```

**Two copies of test**

```
i ⇔ 1
loop: if (i > 100) branch to end
    ... loop body ...
i ⇔ i + 1
if (i < 101) jump to 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**
Generating a **FOR Loop with SDT**

**Generating the single test version of the loop**

\[
Stmt \rightarrow \text{FOR NAME } = Expr_1 \ TO \ Expr_2 \ BY \ Expr_3 \ \{ \ Stmts \} \]

(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 \(Expr_2\) into a register

- Jump to **loop**
Generating a **FOR** Loop with **SDT**

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

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

(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
- Generate the second the test and branch
  - This process involves evaluating some of the \( \text{Expr} \)'s multiple times
Generating a **FOR** Loop with **SDT**

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

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

(1) Use **bison**'s mid-production action to insert code

- 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
- 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 second test. (Loops can and do nest.)

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.

With care, the test & branch sequences can be identical.
Code For a Two-Test FOR Loop

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

loadl 1 \Rightarrow r_1
loadl 1 \Rightarrow r_2
loadl 100 \Rightarrow r_3
cmp_GE r_1,r_3 \Rightarrow r_4
cbr r_4 \rightarrow L_2,L_1
L_1: loop body
add r_1,r_2 \Rightarrow r_1
cmp_GE r_1,r_3 \Rightarrow r_4
cbr r_4 \rightarrow L_2,L_1
L_2: next statement

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

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

(use break)

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;
}
```

Cost of execution depends on frequency of values for `expr`

Clear benefit to reordering cases
Case Statement Implementation

Using Binary Search

```c
switch( expr ) {  
case 0: block0;
    break;
case 17: block17;
    break;
...
case 99: block99;
    break;
default: block_d;
    break;
}
```

<table>
<thead>
<tr>
<th>Value</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LB0</td>
</tr>
<tr>
<td>17</td>
<td>LB17</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>99</td>
<td>LB99</td>
</tr>
</tbody>
</table>

Search Table

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

While the search code is complex, it is easy for the compiler to emit. The compiler writer gets it correct one time and it works from then on out.
Case Statement Implementation

Using a Jump Table

```c
switch( expr ) {
    case 0:  block0;  break;
    case 1:  block1;  break;
    case 2:  block2;  break;
    ...   ...   ...
    case 9:  block9;  break;
    default: blockd;  break;
}
```

Temp ← expr;

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

Case Statement

- Cost is $O(1)$
- Can optimize code further

If “expr” is one byte, use a full table and avoid the if-then-else
Case Statement Implementation

Using a Jump Table

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

If \((0 \leq \text{temp} \leq 9)\) then {
\[
\text{addr} \leftarrow \text{@LT} + \text{temp} \times 4
\]
\[
\text{target} \leftarrow \text{Mem}[\text{addr}]
\]
\[
\text{jump to target}
\]
} else jump to \(\text{LB}_d\)

Original Code

\[
\text{temp} \leftarrow \text{expr} \times 4
\]

If \((0 \leq \text{temp} \leq 36)\) then {
\[
\text{offset} \leftarrow \text{@LT} + \text{temp}
\]
\[
\text{addr} \leftarrow \text{Mem}[\text{ARP} + \text{offset}]
\]
\[
\text{jump to addr}
\]
} else jump to \(\text{LB}_d\)

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

- We need the label, \(\text{@LT}\), in a register
  - If case statement is in a loop, load \(\text{@LT}\) before the loop \(\text{(LICM)}\)
  - Could put \(\text{@LT}\) in the AR, but that requires initializations

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

This stuff follows from the array address optimization material.
Generating a **CASE** Statement With **SDT**

The grammar for a case statement might resemble:

```plaintext
CaseStmt ⇾  SWITCH ( Expr ) INTO { CaseList } ;
CaseList ⇾  Caselist OneCase
        |  OneCase
OneCase ⇾  CASE LABEL : Stmt ; BREAK ;
```

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**

\[
\text{CaseStmt} \rightarrow \text{SWITCH (Expr) INTO} \{ \text{CaseList} \} ; \\
\text{CaseList} \rightarrow \text{Caselist OneCase} \quad \text{1} \quad \text{2} \\
| \quad \text{OneCase} \\
\text{OneCase} \rightarrow \text{CASE LABEL : Stmt ; BREAK ;}
\]

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.
Generating a **CASE** Statement With **SDT**

The resulting code has the following shape:

<table>
<thead>
<tr>
<th>CL₁: code for first case</th>
<th>Step (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>jump EVAL</td>
<td></td>
</tr>
<tr>
<td>CL₂: code for second case</td>
<td>Step (2)</td>
</tr>
<tr>
<td>jump EXIT</td>
<td></td>
</tr>
<tr>
<td>CLₙ: code for second case</td>
<td>Step (3)</td>
</tr>
<tr>
<td>jump EXIT</td>
<td></td>
</tr>
<tr>
<td>EVAL: code to evaluate (Expr) to a label (CLᵢ) and jump to (CLᵢ)</td>
<td>Step (4)</td>
</tr>
<tr>
<td>EXIT: nop</td>
<td>Step (5)</td>
</tr>
</tbody>
</table>

While building the case list:
- CL₁
- CL₂
- CLₙ

While walking the case list:
- EVAL
- EXIT
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 $Expr = \text{case label}$, 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 $\text{else}$ 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

See “Clean,” § 10.2.2 in EaC2e