Lecture 32
Control Structures: Efficient Translation
Instructor: Hossein Hojjat
April 19, 2017
Recap: Code Generation for while

\[
\text{[while (cond) stmt]} = \\
\text{nStart: } \quad [\text{cond}] \\
\text{ifeq(nExit) } \quad [\text{stmt}] \\
\text{goto(nStart)} \\
\text{nExit:}
\]
$[e_1 < e_2] =$

$[e_1]$

$[e_2]$

if_icmplt(nTrue)

iconst_0

goto(nExit)

nTrue:   icontst_1

nExit:
while (counter < to) {
    counter = counter + step;
}

Translation 1:

nBegin: iload #counter
    iload #to
    if_icmplt nTrue
    istore #counter
    goto nAfter
nTrue:  iconst_1
nAfter: ifeq nExit
    iload #counter
    iload #step
    iadd
    istore #counter
    goto nBegin
nExit:

Translation 2:

nBegin: iload #counter
    iload #to
    if_icmplt nBody
    goto nExit
nBody:  iload #counter
        iload #step
        iadd
        istore #counter
        goto nBegin
nExit:
Compare Two Translations

```java
while (counter < to) {
    counter = counter + step;
}
```

Translation 1:
```
nBegin: iload #counter
    iload #to
    if_icmplt nTrue
    iconst_0
    goto nAfter
nTrue:  iconst_1
nAfter: ifeq nExit
    iload #counter
    iload #step
    iadd
    istore #counter
    goto nBegin
```

Translation 2:
```
nBegin: iload #counter
    iload #to
    if_icmplt nBody
    goto nExit
nBody:  iload #counter
    iload #step
    iadd
    istore #counter
    goto nBegin
nExit:  
```

Translation 2 immediately jumps to body, no intermediate result for while condition.
Macro \texttt{branch} Instruction

- Introduce an imaginary big instruction
  \begin{center}
  \texttt{branch}(c, nTrue, nFalse)
  \end{center}
- \texttt{c} is a potentially complex Java boolean expression
  - Main reason why branch is not a real instruction
- \texttt{nTrue} is label to jump to when \texttt{c} evaluates to true
- \texttt{nFalse} is label to jump to when \texttt{c} evaluates to false
- No “fall through” - always jumps (symmetrical)

We show how to:

- Use branch to compile \texttt{if}, \texttt{while}, etc.
- Expand branch recursively into concrete bytecodes
Using `branch` in Compilation

\[
\text{[if (c) } t \text{ else e]} = \\
\text{branch}(c, nTrue, nFalse)
\]

- \text{nTrue: } [ t ]
- \text{goto (nAfter)}

\[
\text{nFalse: } [ e ]
\]

\[
\text{nAfter:}
\]

\[
\text{[while (c) } s \text{]} = \\
\text{nTest: } \text{branch}(c, nBody, nExit)
\]

- \text{nBody: } [ s ]
- \text{goto (nTest)}

\[
\text{nExit:}
\]
Decomposing `branch`

\[
\text{branch}(!c, \text{nThen}, \text{nElse}) = \text{branch}(c, \text{nElse}, \text{nThen})
\]

\[
\text{branch}(c_1 \&\& c_2, \text{nThen}, \text{nElse}) = \\
\quad \text{branch}(c_1, \text{nNext}, \text{nElse}) \\
\text{nNext:} \quad \text{branch}(c_2, \text{nThen}, \text{nElse})
\]

\[
\text{branch}(c_1 \mid\| c_2, \text{nThen}, \text{nElse}) = \\
\quad \text{branch}(c_1, \text{nThen}, \text{nNext}) \\
\text{nNext:} \quad \text{branch}(c_2, \text{nThen}, \text{nElse})
\]

\[
\text{branch}(\text{true}, \text{nThen}, \text{nElse}) = \text{goto} \ \text{nThen}
\]

\[
\text{branch}(\text{false}, \text{nThen}, \text{nElse}) = \text{goto} \ \text{nElse}
\]

boolean variable \(b\) with slot \(N\)

\[
\text{branch}(b, \text{nThen}, \text{nElse}) = \\
\quad \text{iload}_N \\
\quad \text{ifeq} \ \text{nElse} \\
\quad \text{goto} \ \text{nThen}
\]
branch($e_1 \mathbf{R} e_2, \text{nThen}, \text{nElse}$) =

\[
\begin{align*}
&[e_1] \\
&[e_2] \\
&\text{if\_icmp}\mathbf{R}(\text{nThen}) \\
&\text{goto}(\text{nElse})
\end{align*}
\]

\textbf{R} can be $<$, $>$, $==$, $!=$, $\leq$, $\geq$, ...
• Consider storing $x = c$ where $x, c$ are boolean and $c$ has &&, ||
• How to put result of branch on stack to allow istore?

```java
[x = c] =
  branch(c, nThen, nElse)
  nThen:  iconst_1
           goto(nAfter)
  nElse:  iconst_0
  nAfter:  istore #x
```
If \((x < y) \land \neg((y < z) \land \text{cond})\) then:

- return

Else:

- \(y = y + 1\)

Branch cases:

- \(x < y\):
  - \(n1\):
    - \(y < z\):
      - \(n2\):
        - \(\text{cond}\):
          - \(\text{else}\):
            - \(\text{return}\)
          - \(\text{then}\):
            - \(\text{return}\)
    - \(\text{goto after}\)
  - \(\text{else}\):
    - \(\text{iload y}\)
    - \(\text{iconst}_1\)
    - \(\text{iadd}\)
    - \(\text{istore y}\)

After:

Fewer push/pop of boolean constants compared to previous translation
Implementing `branch`

- Option 1: emit code using `branch`, then rewrite
- Option 2: `branch` is a just a function in the compiler that expands into instructions

```
branch(c, nTrue, nFalse)
```

```
public List<Bytecode> compileBranch(Expression c, Label nTrue, Label nFalse) {
    ...
}
```

- The function takes two destination labels
• A common way to exit from a loop is to use a `break` statement

```java
while (true) {
    code1
    if (cond) break;
    cond2
}
```

• Consider a language that has expressions, assignments, blocks {...}, `if`, `while`, and a `break` statement

• `break` statement exits the innermost loop and can appear inside arbitrarily complex blocks and `if` conditions

• How would translation scheme for such construct look like?

• We need a generalization of compilation functions […]

**break Statement**
• Pass a **label** to compilation functions \([\cdots]\) indicating to which instructions to jump after they finish
  - No fall-through

\[
[x = e] \after = \quad // \text{new parameter 'after'}
\]

\[
[e]
\]

\[
\text{istore } \#x
\]

\[
goto(\after) \quad // \text{at the end jump to it}
\]

\[
[s_1; s_2] \after =
\]

\[
[s_1] \text{freshL}
\]

\[
freshL: \quad [s_2] \after \quad \{ \quad \text{we could have any junk in here}
\]

\[
\quad \because ([s_1] \text{freshL}) \text{ends in a jump}
\]
Translation of if, while, return

\[
\begin{align*}
\text{[if (c) t else e] after} & = \\
& \text{branch (c, nTrue, nFalse)} \\
& \quad \text{nTrue: [t] after} \\
& \quad \text{nFalse: [e] after} \\
\end{align*}
\]

\[
\begin{align*}
\text{[while (c) s] after} & = \\
& \text{nTest: branch (c, nBody, after)} \\
& \quad \text{nBody: [s] nTest} \\
\end{align*}
\]

\[
\begin{align*}
\text{[return e] after} & = \\
& \quad [e] \\
& \quad \text{ireturn}
\end{align*}
\]
[if (x < y) return; else y = 2;] after =

  iload  #x
  iload  #y
  if_icmplt nTrue
  goto nFalse

nTrue:    return
nFalse:   iconst_2
           istore  #y
           goto after

Note: no goto after return because

  • translation of if does not generate goto as it did before, since it passes it to the translation of the body
  • translation of return knows it can ignore the after parameter
Two Destination Parameters

\([s_1; s_2] \text{ after brk} = \]
\([-s_1] \text{ freshL brk} \]

\(\text{freshL: } [s_2] \text{ after brk} \]

\([x = e] \text{ after brk} = \]
\([-e] \]

\(\text{istore } \#x \]
\(\text{goto after} \]

\([\text{return } e] \text{ after brk} = \]
\([-e] \]

\(\text{ireturn} \]

\([\text{break} ] \text{ after brk} = \]
\(\text{goto brk} \]

\([\text{while (c) s}] \text{ after brk} = \]
\(\text{test: branch(c, body, after)} \]
\(\text{body: } [s] \text{ test } \text{ after} \]

this is where the second parameter gets bound to the exit of the loop
if with two parameters

\[
\text{[if } (c) t \text{ else } e \text{]} \text{ after brk } = \\
\text{branch}\ (c, nTrue, nFalse) \\
\text{nTrue: } [t] \text{ after brk} \\
\text{nFalse: } [e] \text{ after brk}
\]
break and continue

[break] after brk cont =

goto brk

[continue] after brk cont =

goto cont

[while (c) s] after brk cont =

nTest: branch (c, nBody, after)

nBody: [s] test after nTest