



CSCI 742 - Compiler Construction

Lecture 30

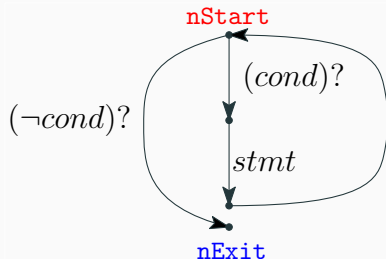
Control Structures: Efficient Translation

Instructor: Hossein Hojjat

April 9, 2018

Recap: Code Generation for `while`

```
[[while (cond) stmt]] =  
  nStart: [[cond]]  
          ifeq(nExit)  
            [[stmt]]  
          goto(nStart)  
  nExit:
```



Code Generation for Relational Expressions

```
[[e1 < e2]] =  
    [[e1]]  
    [[e2]]  
    if_icmplt (nTrue)  
    iconst_0  
    goto (nExit)  
nTrue:  iconst_1  
nExit:
```

Compare Two Translations

```
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  
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

```
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  
nExit:
```

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 `branch` Instruction

- Introduce an imaginary big instruction

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

- `c` is a potentially complex Java boolean expression
 - Main reason why `branch` is not a real instruction
- `nTrue` is label to jump to when `c` evaluates to true
- `nFalse` is label to jump to when `c` evaluates to false
- No “fall through” - always jumps (symmetrical)

We show how to:

- Use `branch` to compile `if`, `while`, etc.
- Expand `branch` recursively into concrete bytecodes

Using branch in Compilation

```
[[if (c) t else e]] =  
    branch(c, nTrue, nFalse)
```

```
nTrue:  [[ t ]]
```

```
        goto(nAfter)
```

```
nFalse: [[ e ]]
```

```
nAfter:
```

```
[[while (c) s]] =
```

```
nTest:  branch(c, nBody, nExit)
```

```
nBody:  [[ s ]]
```

```
        goto(nTest)
```

```
nExit:
```

Decomposing branch

```
branch (!c, nThen, nElse) = branch (c, nElse, nThen)
```

```
-----  
branch (c1 && c2, nThen, nElse) =  
    branch (c1, nNext, nElse)  
    nNext: branch (c2, nThen, nElse)
```

```
-----  
branch (c1 || c2, nThen, nElse) =  
    branch (c1, nThen, nNext)  
    nNext: branch (c2, nThen, nElse)
```

```
-----  
branch (true, nThen, nElse) = goto nThen
```

```
-----  
branch (false, nThen, nElse) = goto nElse  
-----
```

boolean variable b with slot N

```
branch (b, nThen, nElse) =  
    iload_N  
    ifeq nElse  
    goto nThen
```



```
branch( $e_1$  R  $e_2$ , nThen, nElse) =  
    [[ $e_1$ ]]  
    [[ $e_2$ ]]  
    if_icmpR(nThen)  
    goto(nElse)
```

R can be < , > , == , != , <= , >=, ...

Put Boolean Variable on Stack

- Consider storing $x = c$ where x, c are boolean and c has `&&`, `||`
- How to put result of branch on stack to allow `istore`?

```
[[x = c]] =  
    branch(c, nThen, nElse)  
nThen:   iconst_1  
         goto(nAfter)  
nElse:   iconst_0  
nAfter:  istore #x
```

Complex Boolean Expression: Example

Fewer push/pop of boolean constants compared to previous translation

```
if ((x < y) && !((y < z) && cond))
    return
else
    y = y + 1

                                branch(x<y, n1, else)
n1:   branch(y<z, n2, then)
n2:   branch(cond, else, then)
then: return
      goto after
else: iload #y
      iconst_1
      iadd
      istore #y
after:
```

Implementing `branch`

- Option 1: emit code using `branch`, then rewrite
- Option 2: `branch` is 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

break Statement

- A common way to exit from a loop is to use a `break` statement

```
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 $[[\dots]]$

Destination Parameters in Compilation

- Pass a **label** to compilation functions $\llbracket \dots \rrbracket$ indicating to which instructions to jump after they finish
 - No fall-through

```
 $\llbracket x = e \rrbracket$  after = // new parameter 'after'  
   $\llbracket e \rrbracket$   
  istore #x  
  goto(after) // at the end jump to it
```

```
 $\llbracket s_1; s_2 \rrbracket$  after =  
   $\llbracket s_1 \rrbracket$  freshL { we could have any junk in here  
freshL:  $\llbracket s_2 \rrbracket$  after   { because ( $\llbracket s_1 \rrbracket$  freshL) ends in a jump
```

Translation of `if`, `while`, `return`

$\llbracket \text{if } (c) \ t \ \text{else } e \rrbracket \text{ after} =$

```
                                branch(c, nTrue, nFalse)
      nTrue:   $\llbracket t \rrbracket \text{ after}$ 
      nFalse:  $\llbracket e \rrbracket \text{ after}$ 
```

$\llbracket \text{while } (c) \ s \rrbracket \text{ after} =$

```
      nTest:  branch(c, nBody, after)
      nBody:   $\llbracket s \rrbracket \text{ nTest}$ 
```

$\llbracket \text{return } e \rrbracket \text{ after} =$

```
                                 $\llbracket e \rrbracket$ 
                                ireturn
```

Generated Code for Example

```
[[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

`[[s1; s2]] after brk =`

`[[s1]] freshL brk`

`freshL: [[s2]] after brk`

`[[break]] after brk =`

`goto brk`

`[[x = e]] after brk =`

`[[e]]`

`istore #x`

`goto after`

`[[while (c) s] after brk =`

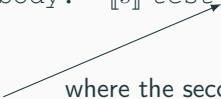
`test: branch(c, body, after)`

`body: [[s] test after`

`[[return e]] after brk =`

`[[e]]`

`ireturn`

this is  where the second parameter gets bound to the exit of the loop

if with two parameters

`[[if (c) t else e]]` after brk =

`branch(c, nTrue, nFalse)`

`nTrue:` `[[t]]` after brk

`nFalse:` `[[e]]` 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
```