**Compilation as Tree Transformation**

Motivation:

* elegant and efficient compilation for conditionals
* compilation for more complex control structures

To describe this compilation we introduce an imaginary, big, instruction

branch(c,nThen,nElse)

Here

* c is a potentially complex Java boolean expression
* nThen is label to jump to when c evaluates to true
* nFalse is label to jump to when c evaluates to false

Next, we show how to expand branch(c,nThen,nElse) into actual instructions. This is a recursive process.

**Using 'branch' in Compilation**

[[ if (c) sThen else sElse ]] =

branch(c,nThen,nElse)

nThen: [[ sThen ]]

goto nAfter

nElse: [[ sElse ]]

nAfter:

[[ while (c) s ]] =

lBegin: branch(c,start,lExit)

start: [[ s ]]

goto lBegin

lExit:

**Decomposing Condition in 'branch'**

**Negation**

branch(!c,nThen,nElse) =

branch(c,nElse,nThen)

**And**

branch(c1 && c2,nThen,nElse) =

branch(c1,nNext,nElse)

nNext: branch(c2,nThen,nElse)

Here, nNext is a fresh label.

**Or**

branch(c1 || c2,nThen,nElse) =

branch(c1,nThen,nNext)

nNext: branch(c2,nThen,nElse)

**Boolean Constant**

branch(true,nThen,nElse) =

goto nThen

branch(false,nThen,nElse) =

goto nElse

**Boolean Variable**

Option one:

branch(xN,nThen,nElse) =

iload\_N

ifeq nElse

goto nThen

Option two:

branch(xN,nThen,nElse) =

iload\_N

ifne nThen

goto nElse

**Relation**

Option one:

branch(e1 R e2,nThen,nElse) =

[[ e1 ]]

[[ e2 ]]

if\_cmpR nThen

goto nElse

Option two:

branch(e1 R e2,nThen,nElse) =

[[ e1 ]]

[[ e2 ]]

if\_cmpNegR nElse

goto nThen

**Storing Result into Boolean Variable**

What if we need to compute

x = c

where x,c are boolean?

* What are nThen,nElse labels?

Producing boolean expression on stack:

[[ c ]] =

branch(c,nThen,nElse)

nThen: iconst\_1

goto nAfter

nElse: iconst\_0

nAfter:

Then we can store the value as usual

**Simple Peephole Optimizations**

Note also that we can eliminate the pattern

goto L

L:

if it is generated in the process above

We can pick the option that eliminates the jump

We can detect this kind of optimization by looking only at neighboring instructions

* example of ‘peephole optimization’

Generated instructions:

-----------------------------------

| | | | | | | | |

-----------------------------------

| |

----------------

|

optimizer looks at a 'window' of instructions

Other examples:

* recognize pattern for ‘x=x+c’, replace with iinc
* recognize copying of boolean variables (b1=b2) - no need for ‘branch’
* how to compile this assignment: b1= b2 && b3; (no better than simple scheme, but for larger expressions and simple comparisons we have an improvement)

Further advanced reading

* [Compiling with Continuations](http://books.google.ch/books?id=3RjLXL2DTEoC&lpg=PP1&dq=Compiling%20with%20continuations&hl=en&pg=PP1#v=onepage&q=&f=false)
* [The essence of compiling with continuations](http://doi.acm.org/10.1145/155090.155113)