CS 421 Lecture 12

- Compilation static languages, continued
  - Compiling in context (main for optimization)
    - **Assignment**
    - **Break statements**
    - **Short-circuit evaluation of boolean expressions**
  - Switch statements
  - Arrays
  - Code optimization
- Thursday's class: dynamic language execution via an example: the Sun HotSpot runtime system – tagged values; garbage collection

Notation

- [S] = compiled code for S
- [e] = compiled code for e
- Use subscripts on brackets for additional arguments, e.g. $[S]_L$ is compiled code for S, assuming S occurs within a switch statements labeled L.

Assignment statements

- Old scheme: [x=e] = let (I,t) = [e] in I; x=t.
- Can give poor results: [x=3] = ~~t=3; x=t~~ x=3

  [x=x+1] = t1=1; t2=x+t1; x=t2
- Compile expressions in context of target location:

  $[e]_x$ = code to calculate value of e *and* store it in x. $[e]_x$: instruction list
- $[x=e] = [e]_x$
- $[n]_x$ = "x=n"
- $[y]_x$ = "x=y", if y a different variable from x; ϵ, otherwise
- $[e1+e2]_x$ = let t = new location in $[e1]_t$; $[e2]_x$; x=t+x

Examples: $[x=x+1] = [x+1]_x = [x]_t ; [1]_x ; x=t+x$

$= t=x ; x=1 ; x=t+x$

$[x=1+x] = [1+x]_x = [1]_t ; [x]_x ; x=t+x$

$= t=1 ; x=1+x$

break statements

- break statement breaks from one level of switch or while. Cannot translate "break" without knowing context.
- $[S]_L$ = code for statement S, given that S occurs inside a switch or while statement, and L is the label just after that enclosing statement.

```
while ( ) {
    
    break;
    
}
```

```
switch ( ) {
    
    break;
    
}
```

$$[\![\text{while } (e)\ s]\!] = \begin{array}{l} \quad\ \text{JUMP } L2 \\ L1:\ [\![s]\!]_{L3} \\ L2:\ [\![e]\!]_t \\ \quad\ \text{CJUMP } t, L1, L3 \\ L3: \end{array}$$

$$[\![\text{break}]\!]_L = \text{JUMP } L$$

<u>Example</u>

```
[ while (true) {
    if (x==10)
        break;
    else x = 1+x;
  } ]
```

$$= \begin{array}{l} \quad\ \text{JUMP } L2 \\ L1:\ [\![\text{if} \cdots]\!]_{L3} \\ L2:\ [\![\text{true}]\!]t \\ \quad\ \text{CJUMP } t, L1, L3 \\ L3: \end{array}$$

(cont.)

$[\text{if } (x==10) \cdots]_{L3} =$

```
      t2 = x==10
      CJUMP t2, L4, L5
L4:   [break]_L3
      JUMP L6
L5:   t3 = 1
      x = t3 + x
L6:
```

$[\text{while} \cdots] =$

```
      JUMP L2
L1:   t2 = x==10
      CJUMP t2, L4, L5
L4:   JUMP L3
      JUMP L6
L5:   t3 = 1
      x = t3 + x
L6:
L2:   t = true
      CJUMP t, L1, L3
L3:
```

Notes:

- $[S]_L$ same as $[S]$ in most cases, but substatements are $[\ ]_L$.
- "continue" statement can be used in loops; terminates current iteration, but not whole loop. Also needs to be compiled "in context".
- More complicated versions of break are "break L" where L is the label of the while or switch, and "break n", which breaks out of n levels. Need more context to compile these.

Boolean expressions

- Current scheme: boolean expressions evaluated like any other, placing value in a temporary location:

$$[e1<e2] = \text{let } (l_1, t1) = [e1],\ (l_2, t2) = [e2],\ t = \text{newloc}() \text{ in } (l_1; l_2; t = t1<t2, t)$$

$$[e1\ \&\&\ e2] = \text{let } (l_1, t1) = [e1]\ (l_2, t2) = [e2] \text{ in } (l_1; l2; t = t1\ \&\&\ t2, t)$$

evaluates both operands - shouldn't do that.

$$[\text{if } e \text{ then } S1 \text{ else } S2] = \text{let } (l, t) = [e] \text{ in } (l; \text{CJUMP } t\ L1\ L2; \ldots)$$

E.g.

while (i<n && A[i] != x) {

don't evaluate if i<n is false

- *What's wrong?*

Boolean expressions w/ short-circuit evaluation

- Improved scheme:

$$
\begin{aligned}
[e1\ \&\&\ e2] = \textbf{let}\ \ & t = \text{newlocation}() \\
& l_1 = [e1]_t \\
& l_2 = [e2]_t \\
& L1,\ L2 = \text{newlabel}() \\
\textbf{in}\ \ (& l_1 \\
& \text{CJUMP}\ t,\ L1,\ L2 \\
L1{:}\ \ & l2 \\
L2{:}\ \ & \qquad\qquad ,\qquad t)
\end{aligned}
$$

- What's wrong now?

Unnecessary jumps when evaluating conjunction!

[if (x<y && y<z) S1 else S2]

= [x<y && y<z]$_t$
CJUMP t, L1, L2
L1: [S1]
JUMP L3
L2: [S2]
L3:

t = x<y
CJUMP t, L4, L5
L4: t = y<z
L5: CJUMP t, L1, L2

Jump to L5, but at L5 will definitely jump to L2. should jump to L2 directly.

Compiling boolean expressions in context

- Get better code if boolean expression can jump to correct label as soon as possible
- $[e]_{Lt,Lf}$ = code that calculates e and jumps to Lt if it is true, Lf if it is false. The code does not save the value anywhere.

- $[\text{true}]_{Lt,Lf}$ = JUMP L_t

  $[e1<e2]_{Lt,Lf}$ = $[e_1 < e_2]_t$
  CJUMP t, L_t, L_f
  = $[e_1]_{t1}$
  $[e2]_{t2}$
  $t = t1 < t2$
  CJUMP t, L_t, L_f

Compiling boolean expressions in context

- $[e1 \ \&\& \ e2]_{Lt,Lf}$ = $[e1]_{L1,Lf}$ (L1 a fresh label)
  $[e2]_{Lt,Lf}$

[while e do S] =
```
     JUMP L2
L1:  [S]
L2:  [e]_{L1,L3}
L3:
```

$[!e]_{Lt,Lf} = [e]_{Lf,Lt}$

$[\text{if } (e)\ S1 \text{ else } S2] =$ $[e]_{L1,L2}$

$L1:\ [S1]$

JUMP $L3$

$L2:\ [S2]$

$L3:$

Ex:

$[\text{if } (x<y \ \&\& \ y<z)$

$S1$

$\text{else } S2\ ]$

$= [x<y \ \&\& \ y<z]_{L1,L2}$

$L1:\ [S1]$

JUMP $L3$

$L2:\ [S2]$

$L3:$

$=$

$[x < y]_{L4, L2}$

$L4:$ $[y < z]_{L1, L2}$

$L1:$ $[s1]$

JUMP L3

$L2:$ $[s2]$

$L3:$

=

t1 = x < y

CJUMP t1, L4, L2

L4: t2 = y < z

CJUMP t2, L1, L2

L1: [s1]

JUMP L3

L2: [s2]

L3

Jump directly to L2 if x<y false

Compiling switch statement

- Use "jump table" and address calculation

$$\left[\begin{array}{l} \text{switch } (e) \ \{ \\ \quad \text{case } 0: \ S0 \\ \quad \text{case } 1: \ S1 \\ \quad \quad \vdots \\ \quad \text{case } n: \ Sn \\ \} \end{array}\right] = \begin{array}{l} [e]_t \\ \text{offset} = t * 4 \\ l = \text{table} + \text{offset} \\ \text{JUMPIND } l \\ L0: \ [S0]_L \\ L1: \ [S1]_L \\ \quad \vdots \\ Ln: \ [Sn]_L \\ L: \end{array}$$

"jump table"

In data section $\rightarrow$ table: $\underbrace{L0, L1, \ldots, Ln}_{\text{"jump table"}}$

Notes:

- Jump table more efficient than sequence of if stmts if n large
- If n small, if may be better; compilers will not always use jump table
- If cases are widely separated - case 0: -- , case 10000, ... - then jump table uses extra space - compiler probably will not use it
- For "default" - check if value of e is between lowest and highest case at start; fill in gaps in jump table to point to default stmt.

Compiling object references

- In expression e.t: at compile time
  - Type of e is known; call its class C
  - Location of field t within C is known; say its offset is o
  - [e] will produce (l, t), where t contains pointer to object
- [e.t] = let (l,t) = [e]
               t1 = newlocation()
          in (l; t1=t+o, t1)
- Method calls e.t(...) more complicated – will discuss in a couple of weeks

Compiling array references

- Simple rule: If A has elements of type T, and if elements of type T occupy n bytes, then address of A[i] is address of A + i*n.
- [A[e]] = let (l, t) = [e]
  in (l
  t1 = &A
  t2 = t*w (w size of A's elements)
  t3 = t1+t2
  t4 = LOADIND t3, t4)

Notes:
- Indexing from 1 would require extra decrement operation – that is why arrays are usually indexed from zero
- Haven't included bounds check. In C, size of array may not be known, so cannot do bounds check. Also, bounds checking adds several extra instructions.

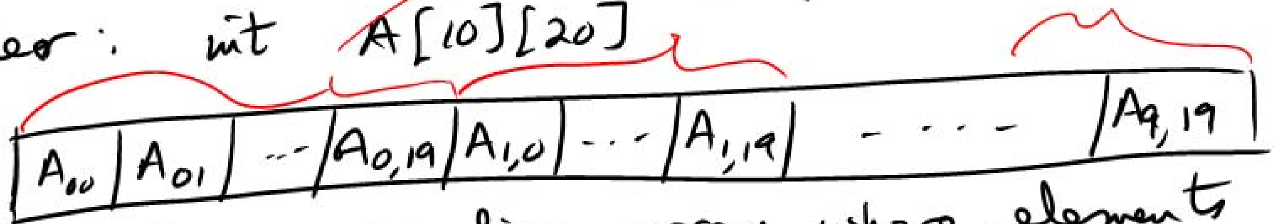
e.g. 1 for array of char's
4 for array of int's
8 for array of double

Compiling array references

elements of A

- Idea extends to multi-dimensional arrays.

Consider: int A[10][20]



Think of A as one-dim. array whose elements are of size (w) $4*20$; those elements are arrays whose elements are of size 4. Eg, A[i][j] is at address $A + i*80 + j*4$.

$A[e_1][e_2]$ is just A indexed by e_1 (previous slide), and the result indexed by e_2:

$A[e_1][e_2] =$

$$
\left.
\begin{array}{l}
[e_1]_{t1} \\
t2 = \&A \\
t3 = t1 * 80 \\
t4 = t2 + t3
\end{array}
\right] A[e_1]
$$

$$
\left.
\begin{array}{l}
[e_2]_{t5} \\
t6 = t5 * 4 \\
t7 = t4 + t6 \\
t8 = \text{LOADIND } t7
\end{array}
\right] A[e_1][e_2]
$$

Note: Java two-dim arrays are just arrays of pointers, so always have $w = 4$ and use LOADIND to get address of subarray

"l-values" vs. "r-values"

- In an assignment: e1 = e2, need to evaluate e1 and e2 differently: e1 is evaluated to an address, e2 to a value.
- When an expression produces a location, that is called its "l-value"; when it produces a value, it is the "r-value".
- Need l-value translation scheme, $[e]_{lv}$
- L-value scheme for array refs is same as r-value scheme, but omit final "LOADIND"
- Eg. A[i] = A[i]+1

Need address

Need value

$$[A[e]]_{lv} = \left(\begin{array}{l} [e]_{t1} \\ t2 = \&A \\ t3 = t1 * w \\ t4 = t2 + t3 \end{array}, t4 \right)$$

$$[e_1 = e_2] = \begin{array}{l} [e_1]_{lv,t1} \\ [e_2]_{t2} \\ \text{STOREIND } t2, t1 \end{array}$$

Machine-independent optimizations

- Machine-independent optimization = optimizations that can be done at the level of IR – i.e. does not depend upon features of target machine such as registers, pipeline, special instructions
- E.g. "loop-invariant code motion":

t1, t2 invariant across iterations of loop

Code motion

```
int A[100][100]
t
while (j<n) {
    x = x + A[i][j]
    j++;
}
```

```
t1 = &A
t2 = i*100
t3 = t2+j
t4 = t3*4
t5 = t1+t4
t6 = LOADIND t5
x = x+t6
j = j+1
```

Machine-dependent optimizations

- Machine-dependent optimization = optimizations that exploit features of target machine such as registers, pipeline, special instructions
  - Register allocation
  - Instruction selection
  - Instruction scheduling