IR trees: Expressions

CONST i	Integer constant i
NAME n	Symbolic constant n [a code label]
TEMP t	Temporary t [one of any number of "registers"]
BINOP e_1 e_2	Application of binary operator: ADD, SUB, MUL, DIV [arithmetic] AND, OR, XOR [bitwise logical] SLL, SRL [logical shifts] SRA [arithmetic right-shift] to integer operands e_1 (evaluated first) and e_2 (evaluated second)
MEM e	Contents of a word of memory starting at address e
CALL f $[e_1,\ldots,e_n]$	Procedure call; expression f is evaluated before arguments $e_1,\ldots,e_n$
ESEQ s e	Expression sequence; evaluate s for side-effects, then e for result

IR trees: Statements

MOVE TEMP t e	Evaluate e into temporary t
MOVE MEM e_1 e_2	Evaluate e_1 yielding address a, e_2 into word at a
EXP e	Evaluate e and discard result
JUMP e $[l_1,\ldots,l_n]$	Transfer control to address e; $l_1,\ldots,l_n$ are all possible values for e
CJUMP e_1 e_2 t f	Evaluate e_1 then e_2, yielding a and b, respectively; compare a with b using relational operators: BEQ, BNE [signed and unsigned integers] BLT, BGT, BLE, BGE [signed] jump to t if true, f if false
SEQ s_1 s_2	Statement s_1 followed by s_2
LABEL n	Define constant value of name n as current code address; NAME(n) can be used as target of jumps, calls, etc.

Kinds of expressions

Expression kinds indicate "how expression might be used"

Ex(exp) expressions that compute a value

Nx(stm) statements: expressions that compute no value

Cx conditionals (jump to true and false destinations)

RelCx.*op*(left, right) eq, ne, gt, lt, ge, le

IfThenElseExp expression or statement, depending on use

Conversion operators allow use of one form in context of another:

unEx convert to tree expression that computes value of inner tree

unNx convert to tree statement that computes inner tree but returns no value

unCx(t, f) convert to statement that evaluates inner tree and branches to true destination if non-zero, false destination otherwise

Translating MiniJava

Local variables: Allocate as a temporary t

TEMP t Ex(TEMP t)

Array elements: Array expression is reference to array in heap.

For exressions e and i, translate $e[i]$ as:

Ex(MEM(ADD(e.unEx(), $\times$(i.unEx(), CONST(w)))))

where w is the target machine's word size: all values are word-sized (scalar) in MiniJava

Array bounds check: array index $i <$ `e.size`; runtime will put size in word preceding array base

Object fields: Object expression is reference to object in heap.

For expression e and field f, translate e.`f` as:

Ex(MEM(ADD(e.unEx(), CONST(o))))

where o is the byte offset of the field `f` in the object

Null pointer check: object expression must be non-null (i.e., non-zero)

Translating MiniJava

String literals: Allocate statically:

```
        .word 11
label:  .ascii "hello world"
```

Translate as reference to label:

Ex(NAME(`label`))

Object creation: Allocate object in heap.

For class T, translate `new` $T()$ as:

Ex(CALL(NAME("new"), CONST(fields), NAME(label for T's vtable)))

Array creation: Allocate array in heap.

For type T, array expression e, translate `new`$T[e]$ as:

Ex(ESEQ(MOVE(TEMP(s), e.unEx()),
CALL(NAME("new"), MUL(TEMP(s), CONST(w)), TEMP(s))))

where s is a fresh temporary, and w is the target machine's word size.

Control structures

Basic blocks:

- a sequence of straight-line code
- if one instruction executes then they all execute
- a maximal sequence of instructions without branches
- a label starts a new basic block

Overview of control structure translation:

- control flow links up the basic blocks
- ideas are simple
- implementation requires bookkeeping
- some care is needed for good code

while loops

while (c) s:

1. evaluate c
2. if false jump to next statement after loop
3. evaluate loop body s
4. evaluate c
5. if true jump back to loop body

e.g.,

if not(c) jump *done*
body:
s
if c jump *body*
done:

Nx(SEQ(SEQ(c.unCx(b, x), SEQ(LABEL(b), s.unNx())),
SEQ(c.unCx(b, x), LABEL(x))))

for loops

for (i, c, u) s

1. evaluate initialization statement i
2. evaluate c
3. if false jump to next statement after loop
4. evaluate loop body s
5. evaluate update statement u
6. evaluate c
7. if true jump to loop body

Nx(SEQ(i.unNx(),
SEQ(SEQ(c.unCx(b, x), SEQ(LABEL(b), SEQ(s.unNx(), u.unNx()))),
SEQ(c.unCx(b, x), LABEL(x)))))

For **break** statements:

- when translating a loop push the *done* label on some stack
- **break** simply jumps to label on top of stack
- when done translating loop and its body, pop the label

Method calls

$e_0.m(e_1,\ldots,e_n)$:

Ex(CALL(MEM(MEM(e_0.unEx(), $-w$), m.index $\times$ w), e_1.unEx(),
$\ldots e_n$.unEx()))

Null pointer check: expression e_0 must be non-null (i.e., non-zero)

Comparisons

Translate a *op* b as:

RelCx.*op*(a.unEx(), b.unEx())

When used as a conditional unCx(t,f) yields:

CJUMP(a.unEx(), b.unEx(), t, f)

where t and f are labels.

When used as a value unEx() yields:

```
ESEQ(SEQ(MOVE(TEMP(r), CONST(1)),
         SEQ(unCx(t, f),
             SEQ(LABEL(f),
                 SEQ(MOVE(TEMP(r), CONST(0)), LABEL(t))))),
     TEMP(r))
```


Conditionals

Translate short-circuiting Boolean operators (`&&`, `||`, `!`) as if they were conditionals

e.g., $x<5$ `&&` $a>b$ is treated as $(x<5)$? $(a>b):0$

We translate e_1 ? $e_2:e_3$ into IfThenElseExp(e_1, e_2, e_3)

When used as a value IfThenElseExp.unEx() yields:

```
ESEQ(SEQ(SEQ(e1.unCx(t, f),
             SEQ(SEQ(LABEL(t),
                     SEQ(MOVE(TEMP(r), e2.unEx()),
                         JUMP(j))),
                 SEQ(LABEL(f),
                     SEQ(MOVE(TEMP(r), e3.unEx()),
                         JUMP(j))))),
         LABEL(j)),
     TEMP(r))
```

As a conditional IfThenElseExp.unCx(t,f) yields:

```
SEQ(e1.unCx(tt, ff), SEQ(SEQ(LABEL(tt), e2.unCx(t, f)),
                         SEQ(LABEL(ff), e3.unCx(t, f))))
```


Conditionals: Example

Applying unCx(t,f) to $(x<5)$? $(a>b):0$:

```
SEQ(BLT(x.unEx(), CONST(5), tt, ff),
    SEQ(SEQ(LABEL(tt, BGT(a.unEx(), b.unEx(), t, f)),
        SEQ(LABEL(ff, JUMP(f))))
```

or more optimally:

```
SEQ(BLT(x.unEx(), CONST(5), tt, f),
    SEQ(LABEL(tt, BGT(a.unEx(), b.uneX(), t, f)))
```

One-dimensional fixed arrays: Pascal/Modula/C/C++

var a : **array** $[2..5]$ **of** $integer$;
...
$a[e]$

translates to:

MEM(ADD(TEMP(FP), ADD(CONST $k-2w$, $\times$(CONST w, e.unEx))))

where k is offset of static array from the frame pointer FP, w is word size

In Pascal, multidimensional arrays are treated as arrays of arrays, so `A[i,j]` is equivalent to A[i][j], so this translation works for subarrays. Not so in Fortran.

Multidimensional arrays

Array allocation:

constant bounds

- allocate in static area, stack, or heap
- no run-time descriptor is needed

dynamic arrays: bounds fixed at run-time

- allocate in stack or heap
- descriptor is needed

dynamic arrays: bounds can change at run-time

- allocate in heap
- descriptor is needed

Multidimensional arrays

Array layout:

Contiguous:

1. *Row major*
   Rightmost subscript varies most quickly:
   `A[1,1], A[1,2], ...`
   `A[2,1], A[2,2], ...`
   Used in PL/1, Algol, Pascal, C, Ada, Modula, Modula-2, Modula-3
2. *Column major*
   Leftmost subscript varies most quickly:
   `A[1,1], A[2,1], ...`
   `A[1,2], A[2,2], ...`
   Used in FORTRAN

By vectors

Contiguous vector of *pointers* to (non-contiguous) subarrays

Multi-dimensional arrays: row-major layout

```
array [1..N,1..M] of T
≡ array [1..N] of array [1..M] of T
```

no. of elt's in dimension j: $D_j = U_j - L_j + 1$

position of `A[`i_1`, ..., `i_n`]`:

$$\begin{aligned}&(i_n - L_n)\\ &+(i_{n-1} - L_{n-1})D_n\\ &+(i_{n-2} - L_{n-2})D_nD_{n-1}\\ &+\cdots\\ &+(i_1 - L_1)D_n\cdots D_2\end{aligned}$$

which can be rewritten as

$$\overbrace{i_1D_2\cdots D_n + i_2D_3\cdots D_n + \cdots + i_{n-1}D_n + i_n}^{\text{variable part}}$$
$$\underbrace{-(L_1D_2\cdots D_n + L_2D_3\cdots D_n + \cdots + L_{n-1}D_n + L_n)}_{\text{constant part}}$$

Address of `A[`i_1`, ..., `i_n`]`:

address(`A`) + ((variable part $-$ constant part) $\times$ element size)

case (switch) statements

case E **of** V_1: $S_1 \ldots V_n$: S_n **end**

1. evaluate the expression
2. find value in case list equal to value of expression
3. execute statement associated with value found
4. jump to next statement after case

Key issue: finding the right case

- sequence of conditional jumps (small case set)
  $\mathbf{O}(|\text{ cases }|)$
- binary search of an ordered jump table (sparse case set)
  $\mathbf{O}(\log_2 |\text{ cases }|)$
- hash table (dense case set)
  $\mathbf{O}(1)$

case (switch) statements

case E **of** V_1: $S_1 \ldots V_n$: S_n **end**

One translation approach:

```
        t := expr
        jump test
L1:     code for S1
        jump next
L2:     code for S2
        jump next
        ...
Ln:     code for Sn
        jump next
test:   if t = V1 jump L1
        if t = V2 jump L2
        ...
        if t = Vn jump Ln
        code to raise run-time exception
next:
```


Labels and gotos

A little complicated!

Resolving references to labels multiply-defined in different scopes:

```
begin
  L: begin
     goto L;
     ... { possible definition of L }
   end
end
```

- Scope labels like variables
- On use, label definition is either resolved or unresolved
- On definition, backpatch previous unresolved label uses

Jumping out of blocks or procedures:

1. Pop run-time stack
2. Fix display (if used); static chain needs no fixing
3. Restore registers if jumping out of a procedure

After translation

To simplify translation there are mismatches between tree code and actual machine instructions:

1. CJUMP to two labels; machine conditionals fall through on false
2. ESEQ and CALL order evaluation of subtrees for side-effects – constrains optimization
3. CALL as argument to another CALL causes interference between register arguments

Canonical trees

Rewrite into an equivalent canonical form:

- SEQ can only be subtree of another SEQ
- SEQs clustered at top of tree
- might as well turn into simple linear list of statements

Basic blocks and traces

3-stage transformation:

1. to linear list of *canonical trees* without SEQ/ESEQ
2. to *basic blocks* with no internal jumps or labels
3. to *traces* with every CJUMP immediately followed by false target

Linear trees

1. No SEQ or ESEQ nodes
2. A CALL can only be a subtree of an EXP(...) or a MOVE(TEMP t,...)

Transformations:

- lift ESEQs up tree until they can become SEQs
- turn SEQs into linear list

Linearizing trees

ESEQ(s_1, ESEQ(s_2, e))	= ESEQ(SEQ(s_1,s_2), e)
BINOP(*op*, ESEQ(s, e_1), e_2)	= ESEQ(s, BINOP(*op*, e_1, e_2))
MEM(ESEQ(s, e_1))	= ESEQ(s, MEM(e_1))
JUMP(ESEQ(s, e_1))	= SEQ(s, JUMP(e_1))
CJUMP(*op*, ESEQ(s, e_1), e_2, l_1, l_2)	= SEQ(s, CJUMP(*op*, e_1, e_2, l_1, l_2))
BINOP(*op*, e_1, ESEQ(s, e_2))	= ESEQ(MOVE(TEMP t, e_1), ESEQ(s, BINOP(*op*, TEMP t, e_2)))
CJUMP(*op*, e_1, ESEQ(s, e_2), l_1, l_2)	= SEQ(MOVE(TEMP t, e_1), SEQ(s, CJUMP(*op*, TEMP t, e_2, l_1, l_2)))
MOVE(ESEQ(s, $e1$), e_2)	= SEQ(s, MOVE(e_1, e_2))
CALL(f, a)	= ESEQ(MOVE(TEMP t, CALL(f, a)), TEMP(t))

Taming conditional branches

1. Form *basic blocks*: sequence of statements always entered at the beginning and exited at the end:
   - first statement is a LABEL
   - last statement is a JUMP or CJUMP
   - contains no other LABELs, JUMPS or CJUMPs
2. Order blocks into *trace*:
   - every CJUMP followed by false target
   - JUMPs followed by target, if possible, to eliminate JUMP

Basic blocks

Control flow analysis discovers basic blocks and control flow between them:

1. scan from beginning to end:
   - LABEL l starts a new block and previous block ends (append JUMP l if necessary)
   - JUMP or CJUMP ends a block and starts next block (prepend new LABEL if necessary)
2. prepend new LABELs to blocks with non-LABEL at beginning
3. append JUMP(NAME done) to last block

Traces

1. Pick an untraced block, the start of some trace
2. Follow a *possible* execution path, choosing false targets first
3. Repeat until all blocks are traced

Cleaning up:

- CJUMP followed by true target: switch targets, negate condition
- CJUMP(o, a, b, l_t, l_f) followed by neither l_t nor l_f:
  1. create new l'_f
  2. rewrite as CJUMP(o, a, b, l_t, l'_f), LABEL l'_f, JUMP l_f
- JUMP l, LABEL l $\rightarrow$ LABEL l