

Translation to Intermediate Code Trees

Copyright ©2000 by Antony L. Hosking. *Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or fee. Request permission to publish from hosking@cs.purdue.edu.*

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

o e_1 e_2

Application of binary operator o :

PLUS, MINUS, MUL, DIV

AND, OR, XOR

[bitwise logical]

LSHIFT, RSHIFT

[logical shifts]

ARSHIFT

[arithmetic right-shift]

to integer operands e_1 (evaluated first) and e_2 (evaluated second)

IR trees: Expressions (cont.)

MEM

e

Contents of a word of memory starting at address e

CALL

$f [e_1, \dots, e_n]$

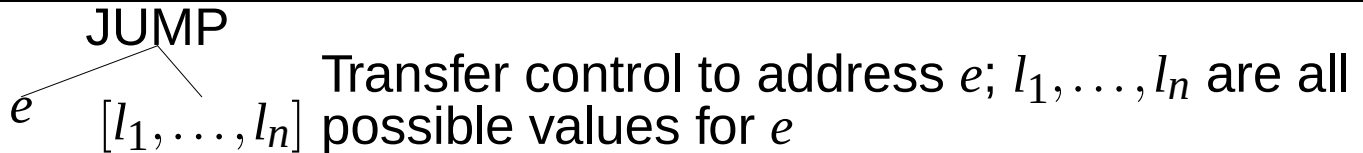
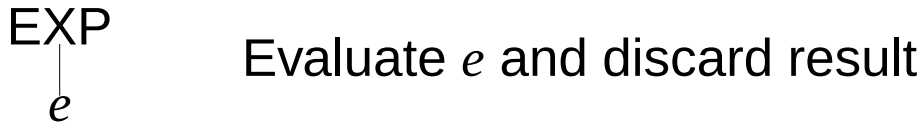
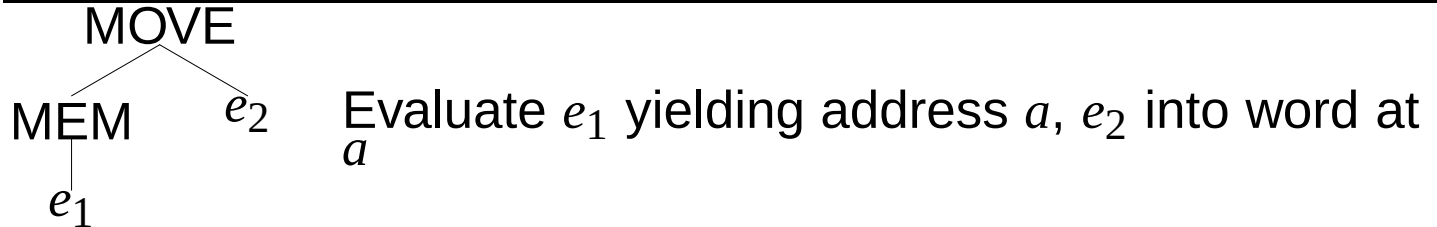
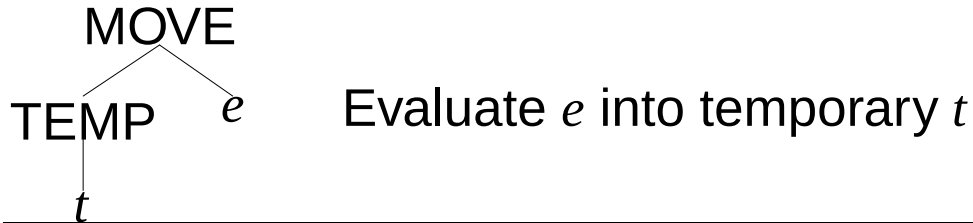
Procedure call; expression f is evaluated before arguments $e_1, \dots, e_n$

ESEQ

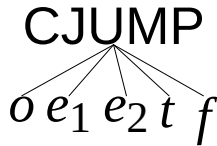
s e

Expression sequence; evaluate s for side-effects, then e for result

IR trees: Statements



IR trees: Statements (cont.)



Evaluate e_1 then e_2 , yielding a and b , respectively; compare a with b using relational operator o :

EQ, NE [signed and unsigned integers]

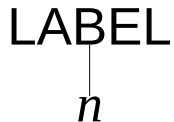
LT, GT, LE, GE [signed]

ULT, ULE, UGT, UGE [unsigned]

jump to t if true, f if false



Statement s_1 followed by s_2



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)

IfThenElseExp expression/statement depending on use

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

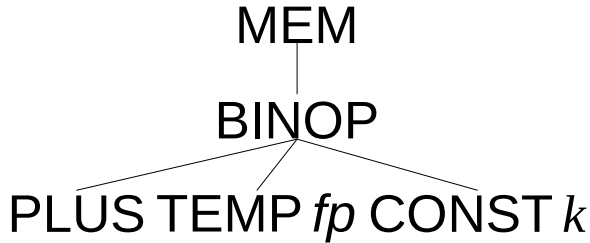
unEx converts to tree expression that computes value of inner tree

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

unCx(t, f) converts to statement that evaluates inner tree and branches to false target if zero, true target otherwise

Translating

Simple variables: fetch with a MEM:



Ex(MEM(+ (TEMP *fp*, CONST *k*)))

where *fp* is home frame of variable, found by following static links; *k* is offset of variable in that level

Translating

Array variables: An array variable holds a pointer to the array base, so fetch with a MEM like any other variable:

$$\text{Ex}(\text{MEM}(+(\text{TEMP } fp, \text{CONST } k)))$$

Thus, for $e[i]$:

$$\text{Ex}(\text{MEM}(+(e.\text{unEx}, \times(i.\text{unEx}, \text{CONST } w))))$$

i is index expression and w is word size.

Note: must first check array index $i < \text{size}(e)$; runtime will put size in word preceding array base

Translating

Record variables: Again, a record variable holds a pointers to the record base, so fetch like other variables.

Thus, for $e.f$:

$$\text{Ex}(\text{MEM}(+(e.\text{unEx}, \text{CONST } o)))$$

where o is the byte offset of the field f in the record

Note: must check record pointer is non-nil (i.e., non-zero)

Translating

String literals: Statically allocated, so just use the string's label:

Ex(NAME(*label*))

where the literal will be emitted as:

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

Translating

Record creation: $\tau\{f_1 = e_1, f_2 = e_2, \dots, f_n = e_n\}$ in the (preferably GC'd) heap, first allocate the space then initialize:

```
Ex( ESEQ(SEQ(MOVE(TEMP r,  
                externalCall("allocRecord",  
                             [CONST n])),  
        SEQ(MOVE(MEM(TEMP r, e1.unEx)),  
              SEQ(...,  
                  MOVE(MEM(+ (TEMP r,  
                             CONST (n - 1)w)),  
                          en.unEx))),  
    TEMP r))  
)
```

where w is the word size

Translating

Array creation: $t[e_1]$ of e_2 :

$\text{Ex}(\text{externalCall}(\text{"initArray"}, [e_1.\text{unEx}, e_2.\text{unEx}])))$

Block instructions: while loops

while c **do** s :

1. evaluate c
2. if false jump to next statement after loop
3. if true fall into loop body
4. branch to top of loop

test:

if not(c) jump *done*

s

jump *test*

done:

$Nx(\text{SEQ}(\text{SEQ}(\text{SEQ}(\text{LABEL } test, c.unCx(body, done)),$
 $\text{SEQ}(\text{SEQ}(\text{LABEL } body, s.unNx),$
 $\text{JUMP}(\text{NAME } test))))$, LABEL *done*))

for loops

for $i := e_1$ **to** e_2 **do** s

1. evaluate lower bound into index variable
2. evaluate upper bound into limit variable
3. if $\text{index} > \text{limit}$ jump to next statement after loop
4. fall through to loop body
5. increment index
6. if $\text{index} \leq \text{limit}$ jump to top of loop body

$t_1 \leftarrow e_1$

$t_2 \leftarrow e_2$

if $t_1 > t_2$ jump *done*

body: s

$t_1 \leftarrow t_1 + 1$

if $t_1 \leq t_2$ jump *body*

done:

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

Function calls

$f(e_1, \dots, e_n)$:

$\text{Ex}(\text{CALL}(\text{NAME } \textit{label}_f, [s/, e_1, \dots e_n]))$

where $s/$ is the static link for the callee f , found by following k static links from the caller, k being the difference between the levels of the caller and the callee

Comparisons

Translate $a \text{ op } b$ as:

$\text{RelCx}(op, a.\text{unEx}, b.\text{unEx})$

When used as a conditional $\text{unCx}(t, f)$ yields:

$\text{CJUMP}(op, a.\text{unEx}, b.\text{unEx}, t, f)$

where t and f are labels.

When used as a value unEx yields:

$\text{ESEQ}(\text{SEQ}(\text{MOVE}(\text{TEMP } r, \text{CONST } 1),$
 $\text{SEQ}(\text{unCx}(t, f),$
 $\text{SEQ}(\text{LABEL } f,$
 $\text{SEQ}(\text{MOVE}(\text{TEMP } r, \text{CONST } 0),$
 $\text{LABEL } t))))),$
 $\text{TEMP } r)$

Conditionals

The short-circuiting Boolean operators have already been transformed into if-expressions in abstract syntax.

e.g., $x < 5 \ \& \ a > b$ turns into

if $x < 5$ **then** $a > b$ **else** 0

Translate **if** e_1 **then** e_2 **else** e_3 into: `IfThenElseExp( $e_1$ ,  $e_2$ ,  $e_3$ )`

Conditionals

When used as a *value* unEx yields:

```
ESEQ(SEQ(SEQ( $e_1$ .unCx(t, f),
              SEQ(SEQ(LABEL t,
                      SEQ(MOVE(TEMP r,  $e_2$ .unEx),
                          JUMP join))),
              SEQ(LABEL f,
                  SEQ(MOVE(TEMP r,  $e_3$ .unEx),
                      JUMP join))))),
    LABEL join),
    TEMP r)
```

As a *conditional* unCx(t, f) yields:

```
SEQ( $e_1$ .unCx(tt,ff), SEQ(SEQ(LABEL tt,  $e_2$ .unCx( $t, f$ )),
                          SEQ(LABEL ff,  $e_3$ .unCx( $t, f$ ))))
```

Conditionals: Example

Applying $\text{unCx}(t, f)$ to **if** $x < 5$ **then** $a > b$ **else** 0:

```
SEQ(CJUMP(LT, x.unEx, CONST 5, tt, ff),  
    SEQ(SEQ(LABEL tt, CJUMP(GT, a.unEx, b.unEx, t, f)),  
        SEQ(LABEL ff, JUMP f)))
```

or more optimally:

```
SEQ(CJUMP(LT, x.unEx, CONST 5, tt, f),  
    SEQ(LABEL tt, CJUMP(GT, a.unEx, b.uneX, t, f)))
```

One-dimensional fixed arrays

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

...

a[*e*]

translates to:

MEM(+ (TEMP fp, + (CONST $k - 2w$, × (CONST w ,
e.unEx))))

where k is the offset of the static array from 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 can translate as above.

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:

$A[1,1], A[1,2], \dots$

$A[2,1], A[2,2], \dots$

Used in PL/1, Algol, Pascal, C, Ada, Modula-3

2. *Column major*

Leftmost subscript varies most quickly:

$A[1,1], A[2,1], \dots$

$A[1,2], A[2,2], \dots$

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 $T \equiv$ 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 $A[i_1, \dots, i_n]$: $(i_n - L_n)$
 $+ (i_{n-1} - L_{n-1})D_n$
 $+ (i_{n-2} - L_{n-2})D_n D_{n-1}$
 $+ \dots$
 $+ (i_1 - L_1)D_n \cdots D_2$

which can be rewritten as

$$\underbrace{i_1 D_2 \cdots D_n + i_2 D_3 \cdots D_n + \cdots + i_{n-1} D_n + i_n}_{\text{variable part}} - \underbrace{(L_1 D_2 \cdots D_n + L_2 D_3 \cdots D_n + \cdots + L_{n-1} D_n + L_n)}_{\text{constant part}}$$

Address of $A[i_1, \dots, i_n]$:

address(A) + ((variable part – constant part) $\times$ elt. size)

case statements

case E **of** $V_1: S_1 \dots V_n: S_n$ **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)
 $O(|\text{cases}|)$
- binary search of an ordered jump table (sparse case set)
 $O(\log_2 |\text{cases}|)$
- hash table (dense case set)
 $O(1)$

case statements

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

One translation approach:

```
     $t := \text{expr}$ 
    jump test
 $L_1$ :   code for  $S_1$ 
        jump next
 $L_2$ :   code for  $S_2$ 
        jump next
 $L_n$ :   code for  $S_n$ 
        jump next
test:   if  $t = V_1$  jump  $L_1$ 
        if  $t = V_2$  jump  $L_2$ 
        if  $t = V_n$  jump  $L_n$ 
        code to raise run-time exception
next:
```

Labels and gotos

A little tricky!

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

Parameter passing

Place information in formal parameter location for callee to access actual parameter:

- value
- address
- dope vector

Parameter passing modes:

value (copy-in), result (copy-out), value-result

Copy actual into formal on call, formal into actual on return

*reference (**var**), read-only*

Copy address of actual into formal

name, formal procedures, label parameters

Name parameters are re-evaluated on every reference

Compile-time representations

Distinguish:

- values (constants)
- locations (ordinary variables)
- addresses of locations containing values (indirect references, **var** parameters)

Value, result, value-result parameters

Value:

- treat formal as a local variable initialized with actual
- actual can be any expression of correct type

Result:

- treat formal as uninitialized local variable
- on return formal is copied into actual
- actual must be an *l-value*

Value-result:

- treat formal as local variable initialized with actual
- on return formal is copied to actual
- actual must be an *l-value*

Value, result, value-result parameters

Basic idea is to copy to/from the callee

- caller passes address of actual
- callee copies from/to actual's address
- result/value-result $\Rightarrow$ two local locations:
address of actual and local copy of actual's value
- value $\Rightarrow$ one local locations:
overwrite passed address with copied value

Value, result, value-result parameters

Scalars:

- result/value-result $\Rightarrow$ pass address of actual, copy value to/from local copy
- value $\Rightarrow$ simply pass value directly

Arrays:

- pass dope vector
- static arrays $\Rightarrow$ pass pointer to base of array
- result/value-result $\Rightarrow$ two local dope vectors
- value $\Rightarrow$ one local dope vector

Records:

- handle as scalar (since fixed in size)
- best to pass address, let callee copy (more compact calling sequences)

Reference and read-only parameters

Usually pass address

Scalars:

- reference $\Rightarrow$ pass address of actual
- read-only $\Rightarrow$ pass value (rather than address):
copy actual into read-only local

Arrays: pass dope vector (simple pointer if static)

Records: pass base address