COMPILERS
Intermediate Code

hussein suleman
uct csc3003s 2007

IR Trees

- An Intermediate Representation is a machine-independent representation of the instructions that must be generated.
- We translate ASTs into IR trees using a set of rules for each of the nodes.
- Why use IR?
  - IR is easier to apply optimisations to.
  - IR is simpler than real machine code.
  - Separation of front-end and back-end.

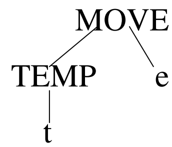
IR Trees – Expressions 1/2

CONST | i	Integer constant i
NAME | n	Symbolic constant n
TEMP | t	Temporary t - a register
MEM | m	Contents of a word of memory starting at m

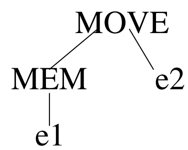
IR Trees – Expressions 2/2

BINOP op e1 e2	e1 op e2 - Binary operator Evaluate e1, then e2, then apply op to e1 and e2
CALL f (e1….en)	Procedure call:evaluate f then the arguments in order, then call f
ESEQ s e	Evaluate s for side effects then e for the result

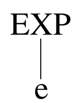
IR Trees – Statements 1/2



Evaluate e then move the result to temporary t

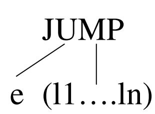


Evaluate e1 giving address a, then evaluate e2 and move the move the result to address a

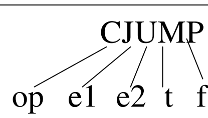


Evaluate e then discard the result

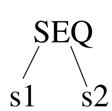
IR Trees – Statements 2/2



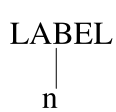
Transfer control to address e; optional labels l1..ln are possible values for e



Evaluate e1 then e2; compare the results using relational operator op; jump to t if true, f if false



The statement S1 followed by statement s2



Define constant value of name *n* as current code address; NAME(*n*) can be used as target of jumps, calls, etc.

Expression Classes 1/2

- Expression classes are an abstraction to support conversion of expression types (expressions, statements, etc.)
- Expressions are indicated in terms of their natural form and then "cast" to the form needed where they are used.
- Expression classes are not necessary in a compiler but make expression type conversion easier when generating code.

Expression Classes 2/2

- Ex(exp) expressions that compute a value
- Nx(stm) statements that compute no value, but may have side-effects
- RelCx (op, l, r) conditionals that encode conditional expressions (jump to true and false destinations)

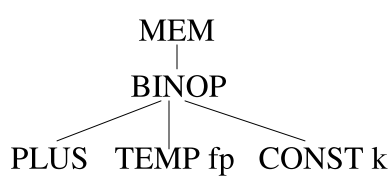
Casting Expressions

- 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.

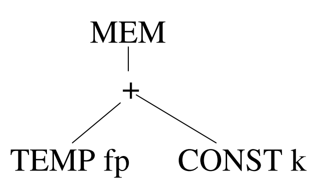
- Trivially, unEx (Exp (e)) = e
- Trivially, unNx (Stm (s)) = s
- But, unNx (Exp (e)) = MOVE[TEMP t, e]

Translation

- Simple Variables
  - simple variable v in the current procedure's stack frame



  - could be abbreviated to:



Expression Example

- Consider the statement:
  - A = (B + 23) * 4;
- This would get translated into:
  - ```
    Nx (
      MOVE (
        MEM (
          +(TEMP fp, CONST k_A)
        ),
        * (
          + (
            MEM (
              +(TEMP fp, CONST k_B)
            ),
            CONST 23
          ),
          CONST 4
        )
      )
    ```

Simple Array Variables

- C-like arrays are pointers to array base, so fetch with a MEM like any other variable:
  - Ex(MEM(+(TEMP fp,CONST k)))
- Thus, for e[I]:
  - Ex(MEM(+(e.unEx,x(i.unEx, CONST w))))
  - *i* is index expression and *w* is word size – all values are word-sized (scalar)
- Note: must first check array index i<size(e); runtime can put size in word preceding array base
    ```

Array creation

- t[e1] of e2:
  - Ex(externalCall(”initArray”, [e1.unEx, e2.unEx]))

General 1-Dimensional Arrays

- var a : ARRAY [2..5] of integer;

- a[e] translates to:
  - MEM(+(TEMP fp, +(CONST k-2w,x(CONST w, e.unEx))))
    - where k is offset of 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 1/3

- Array layout:
  - Contiguous:
    - 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-3
    - 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

Multidimensional Arrays 2/3

- array [1..N,1..M] of T
  - Equivalent to :
    - array [1..N] of array [1..M] of T
- no. of elt's in dimension j:
  - $Dj = U_j - L_j + 1$
- Memory address of $A[i_1, ..., i_n]$:
  - Memory addr. of $A[L_1, …, L_n]$ + sizeof(T) *
    [
    $+ (i_n - L_n)$
    $+ (i_{n-1} - L_{n-1}) * D_n$
    $+ (i_{n-2} - L_{n-2}) * D_n * D_{n-1}$
    $+ …$
    $+ (i_1 - L_1) * D_n * D_{n-1} * … * D_2$
    ]

Multidimensional Arrays 3/3

- which can be rewritten as

Variable part

- $i_1 * D_2 * \ldots * D_n + i_2 * D_3 * \ldots * D_n + \ldots + i_{n-1} * D_n + i_n$
- $- \; (L_1 * D_2 * \ldots * D_n + L_2 * D_3 * \ldots * D_n + \ldots + L_{n-1} * D_n + L_n)$

Constant part

- address of $A[i_1, \ldots, i_n]$:
  - address(A) + ((variable part - constant part) * element size)

Record Variables

- Records are pointers to record base, so fetch like other variables. For e.f
  - Ex(MEM(+(e.unEx, CONST o)))
    - where o is the byte offset of the field in the record
- Note: must check record pointer is non-nil (i.e., non-zero)

Record Creation

- t{f1=e1;f2=e2;….;fn=en} in the (preferably GC'd) heap, first allocate the space then initialize it:
  - 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

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"

Comparisons

- Translate a op b as:
  - RelCx(op, a.unEx, b.unEx)
- When used as a conditional unCx(t,f) yields:
  - CJUMP(op, 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)

If Expressions 1/3 [not for exams]

- If statements used as expressions are best considered as a special expression class to avoid spaghetti JUMPs.
- Translate if e1 then e2 else e3 into:
  - IfThenElseExp(e1,e2,e3)
- When used as a value unEx yields:
  - ESEQ(SEQ(SEQ(e1 .unCx(t, f),SEQ(SEQ(LABEL t,SEQ(MOVE(TEMP r, e2.unEx),JUMP join)),SEQ(LABEL f,SEQ(MOVE(TEMP r, e3.unEx),JUMP join)))),LABEL join),TEMP r)

If Expressions 2/3

- 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))))

If Expressions 3/3

- Applying 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)))

While Loops 1/2

- while c do s:
  - evaluate c
  - if false jump to next statement after loop
  - if true fall into loop body
  - branch to top of loop
  - e.g.,
    - test:
    - if not(c)jump done
    - s
    - jump test
    - done:

While Loops 2/2

- The tree produced is:
- Nx(SEQ(SEQ(SEQ(LABEL test, c.unCx(body,done)),
  SEQ(SEQ(LABEL body, s.unNx),
  JUMP(NAME test))),
  LABEL done))
- repeat e1 until e2 is the same with the evaluate/compare/branch at bottom of loop

For Loops 1/2

- for i:= e 1 to e 2 do s
  - evaluate lower bound into index variable
  - evaluate upper bound into limit variable
  - if index > limit jump to next statement after loop
  - fall through to loop body
  - increment index
  - if index < limit jump to top of loop body

For Loops 2/2

```
          t1  <-  e1
          t2  <-  e2
         if t1 > t2 jump done
body: s
          t1  <-  t1 +1
         if t1 < 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

Case Statement 1/3

- case E of V 1 : S 1 ... Vn: Sn end
  - evaluate the expression
  - find value in list equal to value of expression
  - execute statement associated with value found
  - jump to next statement after case
- Key issue: finding the right case
  - sequence of conditional jumps (small case set)
    - O(|cases|)
  - binary search of an ordered jump table (sparse case set)
    - O(log 2 |cases|)
  - hash table (dense case set)
    - O(1)

Case Statement 2/3

- case E of V 1 : S 1 ... Vn: Sn end
- One translation approach:

```
t :=expr
jump test
L 1 :    code for S1;   jump next
L 2 :    code for S 2;   jump next
...
Ln:       code for Sn    jump next
test:     if t = V1 jump L 1
if t = V2 jump L 2
...
if t = Vn jump Ln
code to raise run-time exception
next:
```

Case Statement 3/3

- Another translation approach:

```
t :=expr
check t in bounds of 0…n-1 if not code to raise run-
  time exception
jump jtable + t
L 1 :    code for S1;   jump next
L 2 :    code for S 2;   jump next
...
Ln:       code for Sn    jump next
Jtable:   jump L 1
jump L 2
...
jump Ln
next:
```

Function Calls

- f(e1;….. ;en):
  - Ex(CALL(NAME label f , [sl,e1 ,... en]))
- where sl is the static link for the callee f
  - Non-local references can be found by following m static links from the caller, m being the difference between the levels of the caller and the callee.
- In OO languages, you can also explicitly pass "this".