Intermediate Representation

Leonidas Fegaras
Intermediate Representation (IR)

• The semantic phase of a compiler
  1) translates parse trees into an intermediate representation (IR), which is independent of the underlying computer architecture
  2) generates machine code from the IRs

• This makes the task of retargeting the compiler to another computer architecture easier to handle

• The IR data model includes
  – raw memory (a vector of words/bytes), infinite size
  – registers (unlimited number)
  – data addresses

• The IR programs are trees that represent instructions in a universal machine architecture
Some IR specs are actually machine-dependent:
  - 32bit, instead of 64bit addresses
  - some registers have a special meaning (sp, fp, gp, ra)

Most IR specs are left unspecified and must be designed:
  - frame layout
  - variable allocation
    - in the static section, in a frame, as a register, etc
  - data layout
    - eg, strings can be designed to be null-terminated (as in C) or with an extra length (as in Java)
IR Example

• Represents the IR:
  MOVE(MEM(+(TEMP(fp),CONST(-16))),
   +
   MEM
   +
   TEMP
   CONST
   -16
   fp
   -20
   TEMP
   CONST
   10

• which evaluates the program:
Expression IRs

- **CONST(i):** the integer constant i
- **MEM(e):** if e is an expression that calculates a memory address, then this is the contents of the memory at address e (one word)
- **NAME(n):** the address that corresponds to the label n
  - eg. MEM(NAME(x)) returns the value stored at the location X
- **TEMP(t):** if t is a temporary register, return the value of the register,
  - eg. MEM(BINOP(PLUS,TEMP(fp),CONST(24))) fetches a word from the stack located 24 bytes above the frame pointer
- **BINOP(op,e1,e2):** evaluate e1, evaluate e2, and perform the binary operation op over the results of the evaluations of e1 and e2
  - op can be PLUS, AND, etc
  - we abbreviate BINOP(PLUS,e1,e2) by +(e1,e2)
- **CALL(f,[e1,e2,...,en]):** evaluate the expressions e1, e2, etc (in that order), and at the end call the function f over these n parameters
  - eg. CALL(NAME(g),ExpList(MEM(NAME(a)),ExpList(CONST(1),NULL))) represents the function call g(a,1)
- **ESEQ(s,e):** execute statement s and then evaluate and return the value of the expression e
Statement IRs

- MOVE(TEMP(t),e): store the value of the expression e into the register t
- MOVE(MEM(e1),e2): evaluate e1 to get an address, then evaluate e2, and then store the value of e2 in the address calculated from e1
  - eg, MOVE(MEM(+(NAME(x),CONST(16))),CONST(1))
- EXP(e): evaluate e and discard the result
- JUMP(L): Jump to the address L
  - L must be defined in the program by some LABEL(L)
- CJUMP(o,e1,e2,t,f): evaluate e1 & e2. If the values of e1 and e2 are related by o, then jump to the address calculated by t, else jump the one for f
  - the binary relational operator o must be EQ, NE, LT etc
- SEQ(s1,s2,...,sn): perform statement s1, s2, ... sn is sequence
- LABEL(n): define the name n to be the address of this statement
  - you can retrieve this address using NAME(n)
Local Variables

- Local variables located in the stack are retrieved using an expression represented by the IR:
  \[
  \text{MEM}(+(\text{TEMP}(fp), \text{CONST}(\text{offset})))
  \]

- If a variable is located in an outer static scope \( k \) levels higher than the current scope, we follow the static chain \( k \) times, and then we retrieve the variable using the offset of the variable:
  - eg, if \( k=3 \):
    \[
    \text{MEM}(+(\text{MEM}(+(\text{MEM}(+(\text{MEM}(+(\text{TEMP}(fp), \text{CONST}(\text{static})), \text{CONST}(\text{static})), \text{CONST}(\text{static})), \text{CONST}(\text{static})), \text{CONST}(\text{offset})))
    \]

where static is the offset of the static link
L-values

• An l-value is the result of an expression that can occur on the left of an assignment statement
  – eg, \( x[f(a,6)].y \) is an l-value
• It denotes a location where we can store a value
• It is basically constructed by deriving the IR of the value and then dropping the outermost MEM call
• For example, if the value is
  \[
  \text{MEM}(+(\text{TEMP}(fp),\text{CONST}(offset)))
  \]
  then the l-value is:
  \[
  +(\text{TEMP}(fp),\text{CONST}(offset))
  \]
Data Layout: Vectors

- Usually stored in the heap
- Fixed-size vectors are usually mapped to n consecutive elements
- Otherwise, the vector length is also stored before the elements
- In Tiger, vectors start from index 0 and each vector element is 4 bytes long (one word), which may represent an integer or a pointer to some value
- To retrieve the $i$th element of an array $a$, we use
  \[
  \text{MEM}(+(A,*(I,\text{CONST}(4))))
  \]
  where $A$ is the address of $a$ and $I$ is the value of $i$
- But this is not sufficient. The IR should check whether $I<\text{size}(a)$:
  \[
  \text{ESEQ}(\text{SEQ}(\text{CJUMP}(\text{gt},I,\text{CONST}(<\text{size}_A)), \text{NAME}(\text{next}),\text{NAME}(\text{error}_\text{label})),
  \text{LABEL}(\text{next}),
  \text{MEM}(+(A,*(I,\text{CONST}(4))))))
  \]
Records

- For records, we need to know the byte offset of each field (record attribute) in the base record
- Since every value is 4 bytes long, the $i$th field of a structure $a$ can be retrieved using $\text{MEM}(+(A,\text{CONST}(i*4)))$, where $A$ is the address of $a$
  - here $i$ is always a constant since we know the field name
• For example, suppose that i is located in the local frame with offset 24 and a is located in the immediate outer scope and has offset 40. Then the statement

\[ a[i+1].\text{first} := a[i].\text{second} + 2 \]

is translated into the IR:

\[
\text{MOVE(MEM(MEM(+A,*(+I,CONST(1)),CONST(4)))))),} \\
\quad (+\text{MEM}(+(\text{MEM}(+(A,*(I,CONST(4)))),CONST(4))))}, \\
\quad \text{CONST(2)})}
\]

where \( I = \text{MEM}(+(\text{TEMP(fp)},\text{CONST(24)}) \)

and \( A = \text{MEM}(+(\text{TEMP(fp)},\text{CONST(40)}) \)

since the offset of first is 0 and the offset of second is 4
Strings

- In Tiger, strings of size n are allocated in the heap in n+4 consecutive bytes, where the first 4 bytes contain the size of the string.
- The string is simply a pointer to the first byte.
- String literals are statically allocated.
- Other languages, such as C, store a string of size n into the heap in n+1 consecutive bytes.
  - the last byte has a null value to indicate the end of string.
- Then, you can allocate a string with address A of size n in the heap by adding n+1 to the global pointer (gp):
  
  ```
  MOVE(A, ESEQ(MOVE(TEMP(gp),
                 +(TEMP(gp), CONST(n+1))),
            TEMP(gp)))
  ```
Control Statements

• The while loop
  while c do body;

  is evaluated in the following way:
  loop: if c goto cont else goto done
  cont: body
    goto loop
  done:

  which corresponds to the following IR:
  SEQ(LABEL(loop),
    CJUMP(EQ,c,1,NAME(done),NAME(cont)),
    LABEL(cont),
    s,
    JUMP(NAME(loop)),
    LABEL(done))
For-Loops

- The for statement
  
  for i:=lo to hi do body

  is evaluated in the following way:

  i := lo
  j := hi
  if i>j goto done

  loop: body
  i := i+1
  if i<=j goto loop

  done:
Other Control Statements

- The break statement is translated into a JUMP
  - The compiler keeps track which label to JUMP to on a “break” statement by maintaining a stack of labels that holds the “done:” labels of the for- or while-loop
  - When it compiles a loop, it pushes the label in the stack, and when it exits a loop, it pops the stack
  - The break statement is thus translated into a JUMP to the label at the top of the stack.

- A function call $f(a_1,...,a_n)$ is translated into the IR
  \[ \text{CALL}(	ext{NAME}(L),[sl,e_1,...,en]) \]
  where $L$ is the label of the first statement of the $f$ code, $sl$ is the static link, and $ei$ is the IR for $ai$

- For example, if the difference between the static levels of the caller and callee is one, then $sl$ is
  \[ \text{MEM}(+(\text{TEMP}(fp),\text{CONST}(	ext{static\_link\_offset}))) \]
Example

• Suppose that records and vectors are implemented as pointers (i.e. memory addresses) to dynamically allocated data in the heap. Consider the following declarations:
  
  ```
  struct { X: int, Y: int, Z: int } S; /* a record */
  int i;
  int V[10][10];                     /* a vector of vectors */
  ```

• where the variables S, i, and V are stored in the current frame with offsets -16, -20, and -24 respectively

• We will the following abbreviations:
  
  ```
  S = MEM(+ (TEMP(fp), CONST(-16)))
  I = MEM(+ (TEMP(fp), CONST(-20)))
  V = MEM(+ (TEMP(fp), CONST(-24)))
  ```
Example (cont.)

- S.Z+S.X
  
  \[+(\text{MEM}(+(S,\text{CONST}(8))),\text{MEM}(S))\]

- if \((i<10)\) then S.Y := i else i := i-1

  \[
  \text{SEQ}(\text{CJUMP}(\text{LT},I,\text{CONST}(10),\text{trueL},\text{falseL}), \text{LABEL}(\text{trueL}), \text{MOVE}(\text{MEM}(+(S,\text{CONST}(4))),I), \text{JUMP}(\text{exit}), \text{LABEL}(\text{falseL}), \text{MOVE}(I,-(I,\text{CONST}(1))), \text{JUMP}(\text{exit}), \text{LABEL}(\text{exit}))
  \]
Example (cont.)

- \( V[i][i+1] := V[i][i]+1 \)
  
  ```
  MOVE(MEM(+ (MEM(+ (V, *(I, CONST(4)))),
  *(+(I, CONST(1)), CONST(4)))),
  MEM(+ (MEM(+ (V, *(I, CONST(4)))), *(I, CONST(4))))))
  ```

- for \( i:=0 \) to \( 9 \) do \( V[0][i] := i \)
  
  ```
  SEQ(MOVE(I, CONST(0)),
  MOVE(TMP(t1), CONST(9)),
  CJUMP(GT, I, TMP(t1), done, loop),
  LABEL(loop),
  MOVE(MEM(+ (MEM(V), *(I, CONST(4))))), I),
  MOVE(I, +(I, CONST(1))),
  CJUMP(LEQ, I, TMP(t1), loop, done),
  LABEL(done))
  ```