Topic 7: Intermediate Representations

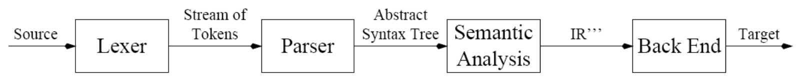
COS 320

Compiling Techniques

Princeton University
Spring 2015

Prof. David August

Intermediate Representations



Intermediate Representation (IR):

- An abstract machine language
- Expresses operations of target machine
- Not specific to any particular machine
- Independent of source language

IR code generation not necessary:

- Semantic analysis phase can generate real assembly code directly.
- Hinders portability and modularity.

Intermediate Representations

Suppose we wish to build compilers for n source languages and m target machines.

Case 1: no IR

- Need separate compiler for each source language/target machine combination.
- A total of $n * m$ compilers necessary.
- Front-end becomes cluttered with machine specific details, back-end becomes cluttered with source language specific details.

Case 2: IR present

- Need just n front-ends, m back ends.

Intermediate Representations

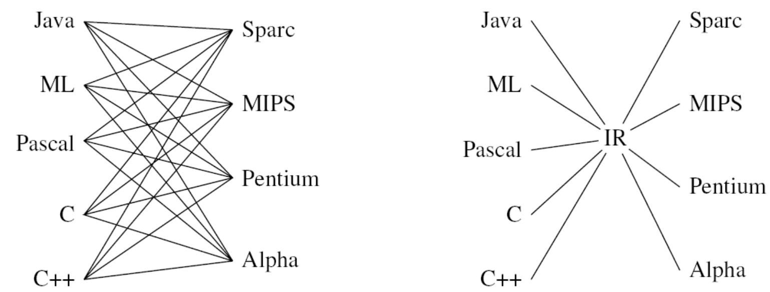


FIGURE 7.1. Compilers for five languages and four target machines: (left) without an IR, (right) with an IR.
From *Modern Compiler Implementation in ML,* Cambridge University Press, ©1998 Andrew W. Appel

Properties of a Good IR

- Must be convenient for semantic analysis phase to produce.
- Must be convenient to translate into real assembly code for all desired target machines.
  - RISC processors execute operations that are rather simple.
    - Examples: load, store, add, shift, branch
    - IR should represent abstract load, abstract store, abstract add, etc.
  - CISC processors execute more complex operations.
    - Examples: multiply-add, add to/from memory
    - Simple operations in IR may be "clumped" together during instruction selection to form complex operations.

IR Representations

The IR may be represented in many forms:

Expression trees:

- `exp`: constructs that compute some value, possibly with side effects.
- `stm`: constructs that perform side effects and control flow.

```
signature TREE = sig
datatype exp    = CONST of int
                | NAME of Temp.label
                | TEMP of Temp.temp
                | BINOP of binop * exp * exp
                | MEM of exp
                | CALL of exp * exp list
                | ESEQ of stm * exp
```

IR Expression Trees

`TREE` **continued:**

```
     and stm    = MOVE of exp * exp
                | EXP of exp
                | JUMP of exp * Temp.label list
                | CJUMP of relop * exp * exp *
                           Temp.label * Temp.label
                | SEQ of stm * stm
                | LABEL of Temp.label
     and binop  = PLUS|MINUS|MUL|DIV|AND|OR|
                  LSHIFT|RSHIFT|ARSHIFT|XOR
     and relop  = EQ|NE|LT|GT|LE|GE|ULT|ULE|UGT|UGE
end
```

Expressions

Expressions compute some value, possibly with side effects.

`CONST`(i) integer constant i

`NAME`(n) symbolic constant n corresponding to assembly language label (abstract name for memory address)

`TEMP`(t) temporary t, or abstract/virtual register t

`BINOP`$(op,\ e_1,\ e_2)$ $e_1\ op\ e_2$, e_1 evaluated before e_2

- integer arithmetic operators: PLUS, MINUS, MUL, DIV
- integer bit-wise operators: AND, OR, XOR
- integer logical shift operators: LSHIFT, RSHIFT
- integer arithmetic shift operator: ARSHIFT

Expressions

MEM(e) contents of `wordSize` bytes of memory starting at address e

- `wordSize` is defined in `Frame` module.
- if MEM is used as left operand of MOVE statement ⇒ store
- if MEM is used as right operand of MOVE statement ⇒ load

CALL(f, l) application of function f to argument list l

- subexpression f is evaluated first
- arguments in list l are evaluated left to right

ESEQ(s, e) the statement s evaluated for side-effects, e evaluated next for result

Statements

Statements have side effects and perform control flow.

MOVE(TEMP(t), e) evaluate e and move result into temporary t.

MOVE(MEM(e_1), e_2) evaluate e_1, yielding address a; evaluate e_2, store result in `wordSize` bytes of memory stating at address a

EXP(e) evaluate expression e, discard result.

JUMP(e, $labs$) jump to address e

- e may be literal label (NAME(l)), or address calculated by expression
- $labs$ specifies all locations that e can evaluate to (used for dataflow analysis)
- jump to literal label l: JUMP(NAME(l), [l])

CJUMP(op, e_1, e_2, t, f) evaluate e_1, then e_2; compare results using op; if true, jump to t, else jump to f

- EQ, NE: signed/unsigned integer equality and non-equality
- LT, GT, LE, GE: signed integer inequality
- ULT, UGT, ULE, UGE: unsigned integer inequality

Statements

SEQ(s_1, s_2) statement s_1 followed by s_2

LABEL(l) label definition - constant value of l defined to be current machine code address

- similar to label definition in assembly language
- use NAME(l) to specify jump target, calls, etc.

- The statements and expressions in TREE can specify function bodies.
- Function entry and exit sequences are machine specific and will be added later.

Translation of Abstract Syntax

- if `Absyn.exp` computes value ⇒ `Tree.exp`
- if `Absyn.exp` does not compute value ⇒ `Tree.stm`
- if `Absyn.exp` has boolean value ⇒ `Tree.stm` and `Temp.labels`

```
datatype exp = Ex of Tree.exp
             | Nx of Tree.stm
             | Cx of Temp.label * Temp.label -> Tree.stm
```

- `Ex` "expression" represented as a `Tree.exp`
- `Nx` "no result" represented as a `Tree.stm`
- `Cx` "conditional" represented as a function. Given a false-destination label and a true-destination label, it will produce a `Tree.stm` which evaluates some conditionals and jumps to one of the destinations.

Translation of Abstract Syntax (Conditionals)

Conditional:

```
x > y:
  Cx(fn (t, f) => CJUMP(GT, x, y, t, f))


a > b | c < d:
  Cx(fn (t, f) => SEQ(CJUMP(GT, a, b, t, z),
                      SEQ(LABEL z, CJUMP(LT, c, d, t, f))))
```

May need to convert conditional to value:

```
a := x > y:
```

`Cx` corresponding to "x $>$ y" must be converted into `Tree.exp` e.

```
  MOVE(TEMP(a), e)
```

Need three conversion functions:

```
  val unEx: exp -> Tree.exp
  val unNx: exp -> Tree.stm
  val unCx: exp -> (Temp.label * Temp.label -> Tree.stm)
```

Translation of Abstract Syntax (Conditionals)

The three conversion functions:

```
  val unEx: exp -> Tree.exp
  val unNx: exp -> Tree.stm
  val unCx: exp -> (Temp.label * Temp.label -> Tree.stm)

a := x > y:
  MOVE(TEMP(a), unEx(Cx(t,f) => ...)
```

`unEx` makes a `Tree.exp` even though e was `Cx`.

Translation of Abstract Syntax

Implementation of function `UnEx`**:**

```
structure T = Tree

fun unEx(Ex(e)) = e
  | unEx(Nx(s)) = T.ESEQ(s, T.CONST(0))
  | unEx(Cx(genstm)) =
      let val r = Temp.newtemp()
          val t = Temp.newlabel()
          val f = Temp.newlabel()
      in T.ESEQ(seq[T.MOVE(T.TEMP(r), T.CONST(1)),
                    genstm(t, f),
                    T.LABEL(f),
                    T.MOVE(T.TEMP(r), T.CONST(0)),
                    T.LABEL(t)],
                T.TEMP(r))
      end
```

Translation of Abstract Syntax

- Recall type and value environments `tenv`, `venv`.
- The function `transVar` return a record {exp, ty} of `Translate.exp` and `Types.ty`.
- exp is no longer a place-holder

Simple Variables

- **Case 1:** variable v declared in current procedure's frame

```
InFrame(k):
  MEM(BINOP(PLUS, TEMP(FP), CONST(k)))
```

```
k: offest in own frame
```

FP is declared in FRAME module.

- **Case 2:** variable v declared in temporary register

```
InReg(t_103):
  TEMP(t_103)
```

Simple Variables

- **Case 3:** variable v not declared in current procedure's frame, need to generate IR code to follow static links

```
InFrame(k_n):
  MEM(BINOP(PLUS, CONST(k_n),
      MEM(BINOP(PLUS, CONST(k_n-1),
             ...
             MEM(BINOP(PLUS, CONST(k_2),
                 MEM(BINOP(PLUS, CONST(k_1), TEMP(FP))))))))
```

```
k_1, k_2,..., k_n-1: static link offsets
k_n: offset of v in own frame
```

Simple Variables

To construct simple variable IR tree, need:

- l_f: level of function f in which v used
- l_g: level of function g in which v declared
- MEM nodes added to tree with static link offsets (k_1,..,k_n-1)
- When l_g reached, offset k_n used.

Array Access

Given array variable a,

```
&(a[0]) = a
&(a[1]) = a + w, where w is the word-size of machine
&(a[2]) = a + (2 * w)
...
```

Let e be the IR tree for a:

```
a[i]:
  MEM(BINOP(PLUS, e, BINOP(MUL, i, CONST(w))))
```

Compiler must emit code to check whether i is out of bounds.

Record Access

```
type rectype = {f1:int, f2:int, f3:int}
                    |        |        |
        offset:     0        1        2
```

```
var a:rectype := rectype{f1=4, f2=5, f3=6}
```

Let `e` be IR tree for `a`:

```
a.f3:
  MEM(BINOP(PLUS, e, BINOP(MUL, CONST(3), CONST(w))))
```

Compiler must emit code to check whether `a` is `nil`.

Conditional Statements

`if` e_1 `then` e_2 `else` e_3

- Treat e_1 as `Cx` expression $\Rightarrow$ apply `unCx`.
- Treat e_2, e_3 as `Ex` expressions $\Rightarrow$ apply `unEx`.

```
Ex(ESEQ(SEQ(unCx(e1)(t, f),
         SEQ(LABEL(t),
          SEQ(MOVE(TEMP(r), unEx(e2)),
           SEQ(JUMP(NAME(join)),
            SEQ(LABEL(f),
             SEQ(MOVE(TEMP(r), unEx(e3)),
              LABEL(join)))))))
        TEMP(r)))
```

Strings

- All string operations performed by run-time system functions.
- In Tiger, C, string literal is constant address of memory segment initialized to characters in string.
  - In assembly, label used to refer to this constant address.
  - Label definition includes directives that reserve and initialize memory.

``foo'':

1. Translate module creates new label l.
2. `Tree.NAME(`l`)` returned: used to refer to string.
3. String *fragment* "foo" created with label l. Fragment is handed to code emitter, which emits directives to initialize memory with the characters of "foo" at address l.

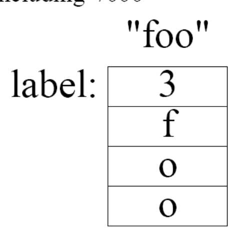
Strings

String Representation:

Pascal fixed-length character arrays, padded with blanks.

C variable-length character sequences, terminated by '/000'

Tiger any 8-bit code allowed, including '/000'



Strings

- Need to invoke run-time system functions
  - string operations
  - string memory allocation
- `Frame.externalCall: string * Tree.exp -> Tree.exp`

```
Frame.externalCall("stringEqual", [s1, s2])
```

  - Implementation takes into account calling conventions of external functions.
  - Easiest implementation:

```
fun externalCall(s, args) =
    T.CALL(T.NAME(Temp.namedlabel(s)), args)
```

Array Creation

```
type intarray = array of int
var a:intarray := intarray[10] of 7
```

Call run-time system function `initArray` to malloc and initialize array.

```
Frame.externalCall("initArray", [CONST(10), CONST(7)])
```

Record Creation

```
type rectype = { f1:int, f2:int, f3:int }
var a:rectype := rectype{f1 = 4, f2 = 5, f3 = 6}

ESEQ(SEQ( MOVE(TEMP(result),
            Frame.externalCall("allocRecord",
                                [CONST(12)])),
     SEQ( MOVE(BINOP(PLUS, TEMP(result), CONST(0*w)),
               CONST(4)),
     SEQ( MOVE(BINOP(PLUS, TEMP(result), CONST(1*w)),
               CONST(5)),
     SEQ( MOVE(BINOP(PLUS, TEMP(result), CONST(2*w)),
               CONST(6)))))),
     TEMP(result))
```

- `allocRecord` is an external function which allocates space and returns address.
- `result` is address returned by `allocRecord`.

While Loops

One layout of a **while loop**:

```
while CONDITION do BODY

test:
    if not(CONDITION) goto done
    BODY
    goto test
done:
```

A **break** statement within body is a `JUMP` to label `done`.
`transExp and transDec` need formal parameter "break":

- passed done label of nearest enclosing loop
- needed to translate breaks into appropriate jumps
- when translating while loop, `transExp` recursively called with loop done label in order to correctly translate body.

For Loops

Basic idea: Rewrite AST into let/while AST; call transExp on result.

```
for i := lo to hi do
  body
```

Becomes:

```
let
  var i := lo
  var limit := hi
in
  while (i <= limit) do
    (body;
     i := i + 1)
end
```

Complication:
If `limit == maxint`, then increment will overflow in translated version.

Function Calls

```
f(a1, a2, ..., an) =>
   CALL(NAME(l_f), sl::[e1, e2, ..., en])
```

- `sl` static link of `f` (computable at compile-time)
- To compute static link, need:
  - `l_f` : level of f
  - `l_g` : level of g, the calling function
- Computation similar to simple variable access.

Declarations

Consider type checking of "let" expression:

```
fun transExp(venv, tenv) =
  ...
  | trexp(A.LetExp{decs, body, pos}) =
      let
        val {venv = venv', tenv = tenv'} =
          transDecs(venv, tenv, decs)
      in
        transExp(venv', tenv') body
      end
```

- Need `level`, `break`.
- What about variable initializations?

Declarations

Consider type checking of "let" expression:

```
fun transExp(venv, tenv) =
  ...
  | trexp(A.LetExp{decs, body, pos}) =
      let
        val {venv = venv', tenv = tenv'} =
          transDecs(venv, tenv, decs)
      in
        transExp(venv', tenv') body
      end
```

- Need `level`, `break`.
- What about variable initializations?

Function Declarations

- Cannot specify function headers with IR tree, only function bodies.
- Special "glue" code used to complete the function.
- Function is translated into assembly language segment with three components:
  - prologue
  - body
  - epilogue

Function Prolog

Prologue precedes body in assembly version of function:

1. Assembly directives that announce beginning of function.
2. Label definition for function name.
3. Instruction to adjust stack pointer (SP) - allocate new frame.
4. Instructions to save escaping arguments into stack frame, instructions to move non-escaping arguments into fresh temporary registers.
5. Instructions to store into stack frame any *callee-save* registers used within function.

Function Epilog

Epilogue follows body in assembly version of function:

6. Instruction to move function result (return value) into return value register.
7. Instructions to restore any *callee-save* registers used within function.
8. Instruction to adjust stack pointer (SP) - deallocate frame.
9. Return instructions (jump to return address).
10. Assembly directives that announce end of function.

- Steps 1, 3, 8, 10 depend on exact size of stack frame.
- These are generated late (after register allocation).
- Step 6:

```
MOVE(TEMP(RV), unEx(body))
```

Fragments

```
signature FRAME = sig
  ...
  datatype frag = STRING of Temp.label * string
                | PROC of {body:Tree.stm, frame:frame}
end
```

- Each function declaration translated into fragment.
- Fragment translated into assembly.
- `body` field is instruction sequence: 4, 5, 6, 7
- `frame` contains machine specific information about local variables and parameters.

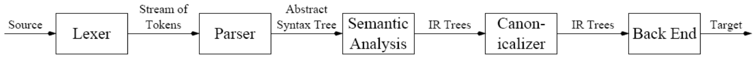
Problem with IR Trees

Problem with IR trees generated by the `Translate` module:

- Certain constructs don't correspond exactly with real machine instructions.
- Certain constructs interfere with optimization analysis.
- `CJUMP` jumps to either of two labels, but conditional branch instructions in real machine only jump to *one* label. On false condition, fall-through to next instruction.
- `ESEQ, CALL` nodes within expressions force compiler to evaluate subexpression in a particular order. Optimization can be done most efficiently if subexpressions can proceed in any order.
- `CALL` nodes within argument list of `CALL` nodes cause problems if arguments passed in specialized registers.

Solution: Canonicalizer

Canonicalizer



Canonicalizer takes `Tree.stm` for each function body, applies following transforms:

1. `Tree.stm` becomes `Tree.stm list`, list of canonical trees. For each tree:
   - No `SEQ, ESEQ` nodes.
   - Parent of each `CALL` node is `EXP(...)` or `MOVE(TEMP(t), ...)`
2. `Tree.stm list` becomes `Tree.stm list list`, statements grouped into *basic blocks*
   - A *basic block* is a sequence of assembly instructions that has one entry and one exit point.
   - First statement of basic block is `LABEL`.
   - Last statement of basic block is `JUMP, CJUMP`.
   - No `LABEL, JUMP, CJUMP` statements in between.
3. `Tree.stm list list` becomes `Tree.stm list`
   - Basic blocks reordered so every `CJUMP` immediately followed by false label.
   - Basic blocks flattened into individual statements.