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 $n \times m$ 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.

![Diagram of compilers for five languages and four target machines: (left) without an IR, (right) with an IR.](image)

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 Expression Trees

**TREE continued:**

```
and stmt = MOVE of exp * exp
         | EXP of exp
         | JUMP of exp * Temp.label list
         | CJUMP of relop * exp * exp *
             Temp.label * Temp.label
         | SEQ of stmt * stmt
         | 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
```

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
               | BSEQ of stmt * exp
```

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, e1, e2)** e1, op, e2, e1 evaluated before e2
  - 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(TMP(t), e) evaluate e and move result into temporary t.
MOVE(MEM(e1), e2) evaluate e1, yielding address a; evaluate e2, store result in
wordSize bytes of memory starting 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, e1, e2, t, f) evaluate e1, then e2; 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

Translation of Abstract Syntax

SEQ(s1, s2) statement s1 followed by s2

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 (Conditionals)

Conditional:

\[ x > y: \]
\[ Cx(fn (t, f) \Rightarrow CJUMP(GT, x, y, t, f)) \]

\[ a > b \vee c < d: \]
\[ Cx(fn (t, f) \Rightarrow SEQ(CJUMP(GT, a, b, t, z), \]
\[ SEQ(LABEL z, CJUMP(LT, c, d, t, f)))) \]

May need to convert conditional to value:

\[ a := x > y: \]
\[ Cx \] corresponding to “\(x > y\)” must be converted into Tree.exp e.
\[ MOVE(\text{TEMP}(a), e) \]

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

Implementation of function UnEx:

structure T = Tree

fun unEx(Ex(e)) = e
    | unEx(Nx(s)) = T.ESEQ(s, T.CONST(0))
    | unEx(Cx(genstmt)) =
        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)),
                    genstmt(t, f),
                    T.LABEL(f),
                    T.MOVE(T.TEMP(r), T.CONST(0)),
                    T.LABEL(t)],
                   T.TEMP(r))
        end

Translation of Abstract Syntax (Conditionals)

The three conversion functions:

\[ \text{val unEx: exp} \Rightarrow \text{Tree.exp} \]
\[ \text{val unNx: exp} \Rightarrow \text{Tree.stm} \]
\[ \text{val unCx: exp} \Rightarrow (\text{Temp.label} \times \text{Temp.label} \Rightarrow \text{Tree.stm}) \]

\[ a := x > y: \]
\[ \text{MOVE(TEMP}(a), \text{unEx}(Cx(t,f) \Rightarrow ...)) \]

unEx makes a Tree.exp even though \(e\) was Cx.

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
  
  \[ \text{InFrame}(k): \]
  \[ \text{MEM}(\text{BINOP}(\text{PLUS}, \text{TEMP}(FP), \text{CONST}(k))) \]
  
  $k$: offset in own frame

  FP is declared in FRAME module.

- **Case 2:** variable $v$ declared in temporary register
  
  \[ \text{InReg}(t_{103}): \]
  
  \[ \text{TEMP}(t_{103}) \]

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
- \text{MEM} nodes added to tree with static link offsets ($k_1, \ldots, k_{n-1}$)
- When $l_g$ reached, offset $k_n$ used.

Array Access

Given array variable $a$,

\[ \& (a[0]) = a \]
\[ \& (a[1]) = a + w, \text{ where } w \text{ is the word-size of machine} \]
\[ \& (a[2]) = a + (2 * w) \]

... Let $e$ be the IR tree for $a$:

\[ a[i]: \]
\[ \text{MEM}(\text{BINOP}(\text{PLUS}, e, \text{BINOP}(\text{MUL}, i, \text{CONST}(w)))) \]

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

```plaintext
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

```plaintext
if e1 then e2 else e3
  - Treat $e_3$ 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(TMP(r), unEx(e2)),
          SEQ(JUMP(NAME(join)),
            SEQ(LABEL(f),
              SEQ(MOVE(TMP(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 $l$.

String Representation:

- **Pascal** fixed-length character arrays, padded with blanks.
- **C** variable-length character sequences, terminated by ‘/00’
- **Tiger** any 8-bit code allowed, including ‘/00’

```
<table>
<thead>
<tr>
<th>label:</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>f</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
```

Strings
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

```plaintext
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

```plaintext
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.

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

Becomes:

```plaintext
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.

Declarations

Consider type checking of “let” expression:

```plaintext
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 Calls

```plaintext
f(a1, a2, ..., an) =>
  CALL(NAME(1_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:

```plaintext
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:
  \[ \text{MOVE(TEMP(RV), unEx(body))} \]

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

<table>
<thead>
<tr>
<th>Source</th>
<th>Lexer</th>
<th>Identifier Tokens</th>
<th>Parser</th>
<th>Abstract Syntax Tree</th>
<th>Semantic Analysis</th>
<th>IR Trees</th>
<th>Canonicalizer</th>
<th>IR Trees</th>
<th>Back End</th>
<th>Tripe</th>
</tr>
</thead>
</table>

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.