



复旦大学

编译原理/Compiler Translation to Intermediate Code

周雅倩
复旦大学计算机科学技术学院
zhouyaqian@fudan.edu.cn
2018/11/23



References

- <http://www.stanford.edu/class/cs143/>
– Partial contents are copied from the slides of Alex Aiken

编译原理

2




Outline

- IR Trees
- Translation into Trees

编译原理

2018/11/23

3




Why Intermediate Representations (IR)?

- Individual pieces of abstract syntax can be complicated things
– such as array subscripts, procedure calls, and so on
- Individual “real machine” instructions can also have a complicated effect.
- Therefore, the IR should have individual components that describe only extremely simple things:
– a single fetch, store, add, move or jump.

编译原理

2018/11/23

4




Intermediate Representations (IR)

- What makes a good IR ?
– easy to convert from the AST;
– easy to convert into the machine code;
– must be clear and simple;
– must support various machine-independent optimizing transformations;

编译原理

2018/11/23

5



VS Intermediate language

- A popular format for intermediate languages is **three address code**.
- **C** is used as an intermediate language by many programming languages.
- Sun's Java bytecode and Microsoft's Common Intermediate Language

编译原理

2018/11/23

6

Intermediate Code

- Usually two IRs:
 - High-level IR
Language-independent
(but closer to language)
 - Low-level IR
Machine independent
(but closer to machine)

```

graph LR
    C --> HIR
    Fortran --> HIR
    Pascal --> HIR
    HIR --> LIR
    LIR --> Pentium
    LIR --> Java[Java bytecode]
    LIR --> PowerPC
  
```

2018/11/23 7

Several IRs

- Some modern compilers use several IRs (e.g., k=3 in SML/NJ)
 - each IR in later phase is a little closer (to the machine code) than the previous phase.
- AST \Rightarrow IR1 \Rightarrow IR2 $\dots \Rightarrow$ IRk \Rightarrow machine code
 - pros: make the compiler cleaner, simpler, and easier to maintain
 - cons: multiple passes of code-traversal --- compilation may be slow

2018/11/23 8

High-level IR

- Tree node structure very similar to the **AST**
- Contains high-level constructs common to many languages
 - Expression nodes
 - Statement nodes
- Expression nodes for:
 - Integers and program variables
 - Binary operations: $e_1 \text{ OP } e_2$
 - Arithmetic operations
 - Logic operations
 - Comparisons
 - Unary operations: $\text{OP } e$
 - Array accesses: $e_1[e_2]$

2018/11/23 9

High-level IR

- Statement nodes:
 - Block statements (statement sequences): (s_1, \dots, s_N)
 - Variable assignments: $v = e$
 - Array assignments: $e_1[e_2] = e_3$
 - If-then-else statements: **if** c **then** s_1 **else** s_2
 - If-then statements: **if** c **then** s
 - While loops: **while** (c) s
 - Function call statements: $f(e_1, \dots, e_N)$
 - Return statements: **return** e or **return**
- May also contain:
 - For loop statements: **for** $(v = e_1 \text{ to } e_2)$ s
 - Break and continue statements
 - Switch statements: **switch** (e) { $v_1: s_1, \dots, v_N: s_N$ }

2018/11/23 10

Low-Level IR

- Low-level representation is essentially an instruction set for an abstract machine
- Alternatives for low-level IR:
 - Three-address code or quadruples (Dragon Book):

$$a = b \text{ OP } c$$
 - Tree representation (Tiger Book)
 - Stack machine (like Java bytecode)

2018/11/23 11

Three-Address Code

- three-address code**

$$a = b \text{ OP } c$$
- Has at most three addresses (may have fewer)
- Also named **quadruples** because can be represented as:

$$(a, b, c, \text{OP})$$
- Example:

$$\begin{aligned}
 t1 &= b + c \\
 a &= (b+c)*(-e); & t2 &= -e \\
 a &= t1 * t2
 \end{aligned}$$

2018/11/23 12

Three-Address Code

```

for (i = 0; i < 10; ++i) {
    b[i] = i*i;
}

t1 := 0          ; initialize i
L1: if t1 >= 10 goto L2 ; conditional jump
    t2 := t1 * t1    ; square of i
    t3 := t1 * 4     ; word-align address
    t4 := b + t3     ; address to store i*i
    *t4 := t2        ; store through pointer
    t1 := t1 + 1     ; increase i
    goto L1         ; repeat loop
L2:

```

2018/11/23 13

Low IR Instructions

- **Assignment instructions:**
 - Binary operations: $a = b \text{ OP } c$
 - arithmetic: ADD, SUB, MUL, DIV, MOD
 - logic: AND, OR, XOR
 - comparisons: EQ, NEQ, LT, GT, LEQ, GEQ
 - Unary operation $a = \text{OP } b$
 - Arithmetic MINUS or logic NEG
 - Copy instruction: $a = b$
 - Load /store: $a = *b, *a = b$
 - Other data movement instructions

2018/11/23 14

Low IR Instructions

- **Flow of control instructions:**
 - label L : label instruction
 - jump L : Unconditional jump
 - cjump a L : conditional jump
- **Function call**
 - call f(a1, ..., an)
 - a = call f(a1, ..., an)
 - Is an extension to quads
- **IR describes the Instruction Set of an abstract machine**

2018/11/23 15

Example (Three-Address Code)

```

m = 0;
if (c == 0) {
    m = m + n * n;
} else {
    m = m + n;
}

m = 0
t1 = c == 0
cjump t1 falseb
t2 = n * n
m = m + t2
jump end
label falseb
m = m + n
label end

```

2018/11/23 16

IR TREES

2018/11/23 17

Node Types of IR Trees

- **Expression**
 - Binary operation
 - Memory
 - Sequence
 - Name
 - Const
 - Call

2018/11/23 18

Tree Node for Expression

- **Const(i)**
- **Name(n)**
- **Temp(t)**
- **BinOP(o, e1, e2)**
- **Mem(e)**
 - Starting address e
 - Size of the content wordSize
- **ESeq(s, e)**
 - The statement s is evaluated for side effects
 - e is evaluated for a result
- **Call(f, l)**
 - f is evaluated first
 - l is evaluated from left to right

2018/11/23 19

Node Types of IR Trees

- **Statements**
 - Sequence
 - Label
 - Jump
 - Conditional jump
 - Move
 - Expression

2018/11/23 20

Tree Node for Statements

- **Seq(s1, s2)**
 - s2 follows s1
- **Label(n)**
 - address
- **Jump(e, labs)**
 - Goto e, e may be an address expression whose possible values are labs
- **CJump(o, e1, e2, t, f)**
 - One comparison e1 o e2
 - Two outcomes t and f
- **Move(Temp t, e) or Move(Mem(e1), e2)**
 - Left elements of MOVE
 - Temporary, Memory
- **Exp**
 - Evaluate expression and discard the result

2018/11/23 21

TRANSLATION INTO TREES

2018/11/23 22

How To Translate?

- May have nested language constructs
 - Nested if and while statements
- Need an algorithmic way to translate
- **Solution:**
 - Start from the AST representation
 - Define translation for each node in the AST
 - Recursively translate nodes in the AST

2018/11/23 23

Kinds of Expressions

- **Ex:** Expressions that compute values
 - Represent as T_exp
- **Nx:** Expressions that return no value
 - Represent as T_stm
- **Cx:** Expressions with boolean values
 - Represent as conditional jumps

2018/11/23 24

Examples

```

a > b | c < d
Temp_label z = newlabel();
T_stm s1 = T_Seq(
  T_Cjump(T_gt, a, b, NULL, z),
  T_Seq(
    T_Label(z),
    T_Cjump(T_lt, c, d, NULL,
      NULL)
  )
);

```

Temp_label
T_stm
T_Seq
T_Cjump(T_gt
T_Label

2018/11/23

25

Examples (patchList)

```

patchList trues = PatchList(
  &s1->u.Seq.left->u.Cjump.true,
  PatchList(&s1->u.Seq.right-
    >u.Cjump.true,
    NULL)
);
patchList falses = PatchList(
  &s1->u.Seq.right->u.Cjump.false,
  NULL);
Tr_exp e1 = Tr_Cx(trues, falses, s1);

```

u is the tree node in Tiger book

2018/11/23

26

Conversions among Ex, Nx and Cx

- flag := (a > b | c < d)
 - requires conversion of a Cx into an Ex
- Transform Function
 - Static T_exp unEx(Tr_exp e);
 - Static T_exp unNx(Tr_exp e);
 - Static T_exp unCx(Tr_exp e);

2018/11/23

27

Examples

```

Static T_exp unEx(Tr_exp e) {
  switch (e->kind) {
    case Tr_ex: return e->ex;
    case Tr_cx {
      Temp_temp r = Temp_newtemp();
      Temp_label t = Temp_relabel(f = Temp_label);
      doPatch(e->ucx.trues, t); doPatch(e->ucx.falses, t);
      return T_Eseq(T_Move(T_Temp(r), T_Const(1)),
        T_Eseq(e->u.ccx.stm,
          T_Eseq(T_Label(f),
            T_Eseq(T_Move(T_Temp(r), T_Const(0)),
              T_Eseq(T_Label(t), T_Temp(r))))));
    }
    case Tr_nx: return T_Eseq(e->u.nx, T_Const(0));
  }
}

```

2018/11/23

28

Translate simple variables

- For a simple variable **v** declared in the current procedure's stack frame:



2018/11/23

29

Following static links

- Otherwise must follow the static link

```

MEM( + ( CONST kn, MEM( + ( CONST kn-1, ...,
  MEM(+ (CONST k1, TEMP FP)
  )...)))

```

- k1, ..., kn-1** are the various static link offsets in nested functions
- kn** is the offset of **v** in its own frame.

2018/11/23

30

Translate array variables

- In Pascal
 - Array-valued variable stands for contents of the array
- In C
 - It is a pointer
- In Tiger, Java and ML
 - It behaves like a pointer
 - No name array constants
 - Created and initialize by $ta[n]$ of l
- In Tiger
 - Record values are also pointers

2018/11/23 31

Translate subscripting and field selection

- $A[i]$
 - Calculate the address $(i-l)*s + a$
 - l is the lower bound of the index range
 - s is the size of each array element
 - a is the base address of the array elements
 - If a is global, $a-l*s$ can be done at compile time
- $a.f$
 - Add the constant field offset of f to the address a

2018/11/23 32

Structured L-values

- l-value is the result of an expression
 - that can occur on the left of an assignment
- r-value is one
 - that can only appear on the right of an assignment
- l-value denotes a location
- l-value can occur on the right of an assignment
 - It means the contents of the location

2018/11/23 33

Structured L-values

- An integer of pointer value is a “scalar”
 - It has only one component
- Structured l-values
 - structs in C
 - arrays and records in Pascal
 - $MEM(+ (TEMP fp, CONST kn), S)$

2018/11/23 34

Record and Array Creation

- Record creation and initialization
 - $\{ f1 = e1 ; f2 = e2 ; \dots ; fn = en \}$
- A record may outlive the procedure activation that creates it
- It cannot be allocated on the stack
- It must be allocated on the heap

2018/11/23 35

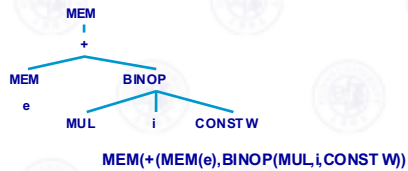
Record and Array Creation

- Call an external memory-allocation function that returns a pointer to an n -word area into a new temporary r
- A series of MOVE trees can initialize offsets $0, 1w, 2w, \dots, (n-1)w$ from r with the translations of expression ei
- The result of the whole expression is r

2018/11/23 36

Translate subscripting and field selection

- Structured l-value can only be represented as an address calculation
- it is very simple to use the MEM node to the l-value before knowing whether it is to be fetched or stored



2018/11/23

37

A Sermon of Safety

- Memory bugs are very common
- Array bound check
 - Time consuming
 - Optimization
- Null check

2018/11/23

38

Translate Expressions

- In these translations, expressions may be nested;
- Translation recurse on the expression structure
- Example: $t = T[(a - b) * (c + d)]$

2018/11/23

39

Translate conditionals

- If e1 then e2 else e3**
 - Treat e1 as a Cx (apply unCx to e1)
 - Treat e2 and e3 as Ex (apply unEx to e2 and e3)
 - Make two labels t and f to which the conditional will branch
 - Allocate a temporary r,
 - after label t, move e2 to r
 - after label f, move e3 to r
 - Both branches should finish by jumping newly created "joint" label

2018/11/23

40

Translate conditionals

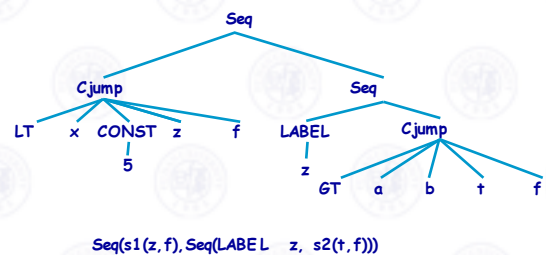
- If e2 and e3 are both "statements" (Nx)
- If e2 or e3 is a Cx
- Treat all of these cases specially
- String comparison
 - Call runtime-system function string Equal

2018/11/23

41

Example

- If (x < 5) a > b; else 0;



2018/11/23

42

Translate strings

- A string literal
 - Constant address of a segment of memory
 - Initialized to the proper characters
- Using the label to references
- Definition of the label, followed by the assembly-language pseudo instruction to reserve and initialize a block of memory to the appropriate characters

2018/11/23 43

Translate strings

- For string literal lit
- Make a new label lab
- Returns the tree T_NAME(lab)
- Puts the assembly language fragment F_string(lab, lit) onto a global list
- All string operations are performed in functions provided by the runtime system
- Heap allocated space for their results
- Return a pointer

2018/11/23 44

Translate strings

- How to represent a string literal
 - Fixed length such as in Pascal
 - Varied length with \0 terminated such as in C
 - One word followed by characters
- How to generate string literal
 - Label definition
 - Pseudo code to generating word constant
 - Pseudo code to generating characters

2018/11/23 45

Translate while loops

test:

```
if not(condition) goto done
body
goto test
```

done:

```
while(e)
{
  s;
}
```

- A break statement simply jump to done
- Done label must be passed as the break parameter

2018/11/23 46

Translate for loops

- Similar to while loop
- Be careful to the overflow

```
e1;
test:
  if not(e2) goto done
  body
  e3
  goto test
done:
```

```
for(e1;e2;e3)
{
  s;
}
```

2018/11/23 47

Translate switch statements

```
switch(e){
  case v1: s1;
  ...
  case vn: sn;
}
```

```
graph TD
    switch[switch] --> e[e]
    switch --> v1[v1]
    switch --> s1[s1]
    switch --> dots[...]
    switch --> vn[vn]
    switch --> sn[sn]
```

2018/11/23 48

Translate function call

- **CALL(NAME lf, [sl, e1, e2, ..., en])**
 - lf is the label for f
 - sl is the static link

2018/11/23 49

Declarations

- **transExp and transDec**
 - Take more arguments
- **transDec returns an extra result**
 - **Tr_exp**
 - Side-effect the frame data structure
 - Additional space
 - A new fragment of tree code

2018/11/23 50

Variable Definition

- **Tr_exp to reflect the initialization**

2018/11/23 51

Function Definition

- **Prologue**
- **Body**
- **Epilogue**

2018/11/23 52

Function Definition

- **Prologue**
 - Pseudo-instructions to announce the beginning of a function;
 - A label definition of the function name
 - An instruction to adjust the stack pointer
 - Instructions to save “escaping” arguments
 - including the static link – into the frame, and to move nonescaping arguments into fresh temporary registers
 - Store instructions to save any callee-saved registers- including the return address register – used within the function.

2018/11/23 53

Function Definition

- **The function body**

2018/11/23 54

Function Definition

- **Epilogue**
 - An instruction to move the return value (result of the function) to the register reserved for that purpose
 - Load instructions to restore the callee-save registers
 - An instruction to reset the stack pointer (to deallocate the frame)
 - A return instruction (Jump to the return address)
 - Pseudo-instructions, as needed, to announce the end of a function

2018/11/23 55

Nested Statements

- Same for statements as expressions: recursive translation
- Example: **if a>b then if c<d then e = f**

2018/11/23 56

IR lowering Efficiency

```

graph TD
    c((c)) --> B1["t1 = c  
fjump t1 Lend1  
t2 = d  
fjump t2 Lend2  
t3 = b  
a = t3  
label Lend2  
label Lend1"]
    c --> d((d))
    d --> B2["fjump c Lend  
fjump d Lend  
a = b  
Label Lend"]
    d --> B2
  
```

2018/11/23 57

Efficient Lowering Techniques

- How to generate efficient Low IR:
 - Reduce number of temporaries
 - Don't use temporaries that duplicate variables
 - Use "accumulator" temporaries
 - Reuse temporaries in Low IR
 - Don't generate multiple adjacent label instructions
 - Encode conditional expressions in control flow

2018/11/23 58