COMP 181

Lecture 14

Intermediate representations and code generation

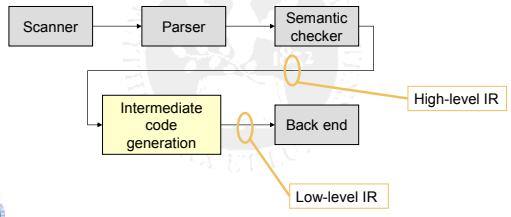
October 19, 2006

Prelude

- Who is Seth Lloyd?
  - Professor of mechanical engineering at MIT, pioneer in quantum computing
- Article in Nature:
  - "*Ultimate physical limits to computation*"
  - Idea: the ultimate laptop
    - Start with 1 kilogram of matter
    - Use speed of light c, quantum scale h, gravity constant G
- System stats:
  - Speed: $2mc^2/\pi h = 5.4 \times 10^{50}$ ops/sec
  - Memory: about 10^{31} bits ≈ 10^{21} GB
  - Bad news: operating temperature is 10^9 Kelvin

Today

- Intermediate representations and code generation



Intermediate representations

- Decisions in IR design affect the speed and capabilities of the compiler
- Some important *IR* properties
  - Ease of generation, manipulation, optimization
  - Size of the representation
  - Level of ***abstraction***: level of "detail" in the IR
    - How close is IR to source code? To the machine?
    - What kinds of operations are represented?
- Often, different IRs for different jobs

  Typically:
  - High-level IR: close to the source language
  - Low-level IR: close to the machine assembly code

Types of IRs

Three major categories

- Structural
  - Graph oriented
  - Heavily used in source-to-source translators
  - Tend to be large

  Examples: Trees, DAGs
- Linear
  - Pseudo-code for an abstract machine
  - Level of abstraction varies
  - Simple, compact data structures
  - Easier to rearrange

  Examples: 3 address code, Stack machine code
- Hybrid
  - Combination of graphs and linear code

  Example: Control-flow graph

High-level IR

- High-level language constructs
  - Array accesses, field accesses
  - Complex control flow

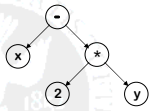
    Loops, conditionals, switch, break, continue
  - Procedures: callers and callees
  - Arithmetic and logic operators

    Including things like short-circuit && and ||
- Often: tree structured
  - Arbitrary nesting of expressions and statements

Abstract Syntax Tree

- AST: parse tree with some intermediate nodes removed

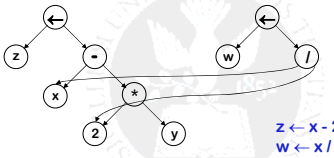
```
x - 2 * y
```



- What is this representation good for?
  - We can reconstruct original source
  - Source-to-source translators
  - Program understanding tools

Directed Acyclic Graph

- A directed acyclic graph (DAG)
  - AST with a unique node for each value



z ← x - 2 * y
w ← x / 2

- Why do this?
  - More compact (sharing)
  - Encodes redundancy

Same expression twice means that the compiler might arrange to evaluate it just once!

Low-level IR

- Linear stream of *abstract* instructions
  - Instruction: single operation and assignment

```
x = y op z        x ← y op z        op x, y, z
```

- Must break down high-level contructs: ***lowering***
  - Example:

```
z = x - 2 * y   ⇒   t ← 2 * y
                    z ← x - t
```

  - Introduce temps as necessary: called ***virtual registers***
- What are advantages?
  - Resembles many machine instruction formats
  - Introduces a new set of names – we'll need these later
  - Fairly compact form

Low-level IR

- Explicit load and store

```
t1 = *p    t1 = load p       move t1, [p]
*p = t2    store [p] = t2    move [p], t2
```

  - Note: we must build addresses using arithmetic
- Simple control-flow

```
label1:
goto label1
if_goto x, label1
```

Jump to label1 if x has non-zero value

Memory model

- What kind of storage do variables represent?

```
x ← y op z
```

  - Possibilities:
    - Values on the stack
    - Values in global variables
    - Temporaries introduced during lowering
- Typically:
  - Temporaries: treated as registers (unlimited #)
  - Other variables: must be explicitly loaded, stored

Memory model

```
z = x - 2 * y   ⇒   t1 ← 2 * y
                    z ← x - t1
```

- Memory operations
  - Often added later, by back-end
  - Explicitly represent variables

IA32 (Intel x86)
%name – a register
$val – a constant
%esp – stack pointer
4(%esp) – indirect + offset

```
move    4(%esp), %eax     // load y
mul     $2, %eax
move    8(%esp), %edx     // load x
sub     %edx, %eax, %ebx  // t1 is ebx
move    %eax, 12(%esp)    // store in z
```

Stack Machine Code

Now, Java VM

- Originally for stack-based computers

```
x - 2 * y
```

```
push x
push 2
push y
multiply
subtract
```

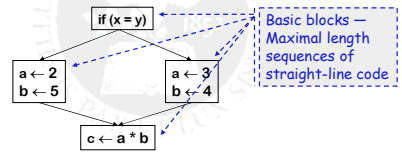
Post-fix notation

- What are advantages?
  - Introduced names are *implicit*, not *explicit*
  - Simple to generate and execute code
  - Compact form – who cares about code size?
    - Embedded systems
    - Systems where code is transmitted (the 'Net)

Control-flow Graph (CFG)

- Models the transfer of control in the procedure
  - Nodes in the graph are ***basic blocks***
  - Within basic block: some linear, low-level linear form
  - Edges in the graph represent control flow

Example:



Static Single Assignment

- Idea:
  - Each variable assigned only once
  - A variable represents a specific value
  - Add on to other forms, mostly CFG

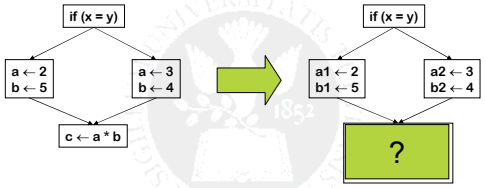
```
x = z + y
x = x + 1
y = x + 2
x = y - 1
```

```
x0 = z0 + y0
x1 = x0 + 1
y1 = x1 + 2
x2 = y1 - 1
```

  - Turns imperative code into functional code

Static Single Assignment

- Problem: what about control flow?



  - Φ functions are selectors
  - Works on loops as well

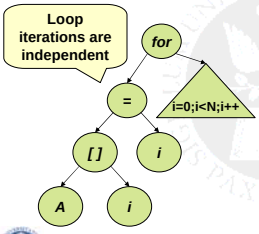
Static Single Assignment

- Advantages
  - Speeds up some program analysis
  - Functional semantics easier to reason about
  - Breaks up live ranges
- Disadvantages
  - Makes some optimizations more complex
  - Doesn't work for C – why?
    - Pointers to local variables: `p = &x; *p = 7;`

IR Trade-offs

```
for (i=0; i<N; i++)
  A[i] = i;
```

Loop iterations are independent



Loop invariant

Strength reduce to temp2 += 4

```
loop:
temp1 = &A
temp2 = i * 4
temp3 = temp1 + temp2
store [temp3] = i
...
goto loop
```

The Rest of the Story…

Representing the code is only part of an *IR*

Other necessary components

- Symbol table (already discussed)
- Constant table
  - Representation, type
  - Storage class, offset
- Storage map
  - Overall storage layout
  - Overlap information
  - Virtual register assignments

Overview

- Kinds of representations
  - Tree or graph
  - Linear
- Level of abstraction
  - High-level: close to the language
  - Low-level: close to machine
  - ***Note***: can represent low-level IR as a tree
- In the compiler:
  - Start with high-level IR, move lower
  - Why?

Towards code generation

```
if (c == 0) {
  while (c < 20) {
    c = c + 2;
  }
}
else
  c = n * n + 2;
```

```
t1 = c == 0
if_goto t1, lab1
t2 = n * n
c = t2 + 2
goto end
lab1:
t3 = c >= 20
if_goto t3, end
c = c + 2
goto lab1
end:
```


Lowering

- How do we translate from high-level IR to low-level IR?
  - HIR is complex, with nested structures
  - LIR is low-level, with *everything* explicit
  - Need a systematic algorithm
- Idea:
  - Define translation for each AST node, *assuming* we have code for children
  - Come up with a scheme to stitch them together
  - Recursively descend the AST

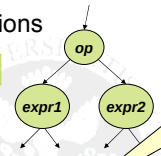
Lowering scheme

- Define a ***generate*** function
  - For each kind of AST node
  - Produces code for that kind of node
  - May call generate for children nodes
- How to stitch code together?
  - Generate function returns a temporary (or a register) holding the result
  - Emit code that combines the results

Lowering expressions

- Arithmetic operations

```
expr1 op expr2
```




```
t1 = generate(expr1)
t2 = generate(expr2)
r = new_temp()
emit( r = t1 op t2 )
return r
```

Generate code for left and right children, get the registers holding the results

Obtain a fresh register name

Emit code for this operation

Return the register to *generate* call above

Lowering expressions

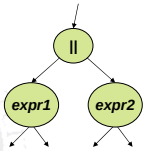
- Scheme works for:
  - Binary arithmetic
  - Unary operations
  - Logic operations
- What about && and ||?
  - In C and Java, they are "short-circuiting"
  - Need control flow…

Short-circuiting ||

- If expr1 is true, don't eval expr2

```
expr1 || expr2
```

```
E = new_label()
r = new_temp()
t1 = generate(expr1)
emit( r = t1 )
emit( if_goto t1, E )
t2 = generate(expr2)
emit( r = t2 )
emit( E: )
return r
```



Details…

```
E = new_label()
r = new_temp()
t1 = generate(expr1)
emit( r = t1 )
emit( if_goto t1, E )
t2 = generate(expr2)
emit( r = t2 )
emit( E: )
return r
```

```
t1 = expr1
r = t1
if_goto t1, E
t2 = expr2
r = t2
E:
. . .
```

```
t3 =
L:
ifgoto
t1 =
```

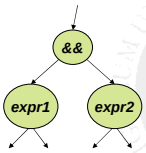

Helper functions

- *emit*()
  - The only function that generates instructions
  - Adds instructions to end of buffer
  - At the end, buffer contains code
- *new_label*()
  - Generate a unique label name
  - Does not update code
- *new_temp*()
  - Generate a unique temporary name
  - May require type information

Order of calls to emit is significant!

Short-circuiting &&

```
expr1 && expr2
```

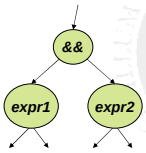



```
N = new_label()
E = new_label()
r = new_temp()
t1 = generate(expr1)
emit( r = t1 )
emit( if_goto t1, N )
emit( goto E )
emit( N: )
t2 = generate(expr2)
emit( r = t2 )
emit( E: )
return r
```


Short-circuiting &&

- Can we do better?

```
expr1 && expr2
```




```
E = new_label()
r = new_temp()
t1 = generate(expr1)
emit( r = t1 )
emit( ifnot_goto t1, E )
t2 = generate(expr2)
emit( r = t2 )
emit( E: )
return r
```

Array access

- Depends on abstraction

```
expr1 [ expr2 ]
```

```
r = new_temp()
a = generate(expr1)
o = generate(expr2)
emit( o = o * size )
emit( a = a + o )
emit( r = load a )
return r
```

Type information from the symbol table

- OR:
  - Emit array op
  - Lower later

Statements

- Simple sequences

```
statement1;
statement2;
. . .
statementN;
```

```
generate(statement1)
generate(statement2)
. . .
generate(statementN)
```

- Conditionals

```
if (expr)
  statement;
```

```
E = new_label()
t = generate(expr)
emit( ifnot_goto t, E )
generate(statement)
emit( E: )
```


Loops

- Emit label for top of loop
- Generate condition and loop body

```
while (expr)
  statement;
```

```
E = new_label()
T = new_label()
emit( T: )
t = generate(expr)
emit( ifnot_goto t, E )
generate(statement)
emit( goto T )
emit( E: )
```


Function call

- Different calling conventions

Why call generate here?

```
x = f(expr1,
      expr2,
      . . .);
```

```
a = generate(f)
foreach expr-i
  ti = generate(expri)
  emit( push ti )

emit( call_jump a )
emit( x = get_result )
```


For loop

- How does “for” work?

```
for (expr1; expr2; expr3)
  statement
```


Assignment

- *Problem*
  - Difference between right-side and left-side
  - Right-side: a value — *r-value*
  - Left-side: a location — *l-value*
- Example: array assignment

```
A[i] = B[j]
```

Store to this location *Load from this location*

Special generate

- Define generate for l-values
  - *lgenerate* returns register contains address
  - Simple case: also applies to variables

```
r = new_temp()
a = generate(expr1)
o = generate(expr2)
emit( o = o * size )
emit( a = a + o )
emit( r = load a )
return r
```

r-value case

```
a = generate(expr1)
o = generate(expr2)
emit( o = o * size )
emit( a = a + o )
return a
```

l-value case

Assignment

- Use **lgenerate** for left-side
- Return *r-value* for nested assignment

```
expr1 = expr2;
```

```
r = generate(expr2)
l = lgenerate(expr2)
emit( store *l = r )
return r
```


At leaves

- Depends on level of abstraction
- ***generate***(v) – for variables
  - All virtual registers: return v
  - Strict register machine: ***emit***(r = load v)
  - Lower level: ***emit***(r = load base + offset)
    - "base" is stack pointer, "offset" from symbol table
  - Note: may introduces many temporaries
- ***generate***(c) – for constants
  - May return special object to avoid r = #

Generation: Big picture

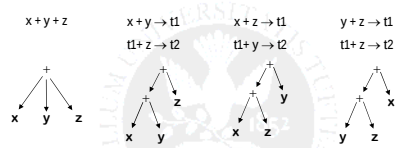
```
Reg generate(ASTNode node)
{
  Reg r;
  switch (node.getKind()) {
  case BIN: t1 = generate(node.getLeft());
            t2 = generate(node.getRight());
            r = new_temp();
            emit( r = t1 op t2 );
            break;
  case NUM: r = new_temp();
            emit( r = node.getValue() );
            break;
  case ID:  r = new_temp();
            o = symtab.getOffset(node.getID());
            emit( r = load sp + o );
            break;
  }
  return r
}
```


Code Shape

- Definition
  - All those nebulous properties of the code that impact performance & code "quality"
  - Includes:
    - Code for different constructs
    - Cost, storage requirements & mapping
    - Choice of operations
  - Code shape is the end product of many decisions
- Impact
  - Code shape influences algorithm choice & results
  - Code shape can encode important facts, or hide them

Code Shape

- An example:



  - What if x is 2 and z is 3?
  - What if y+z is evaluated earlier?

Addition is commutative & associative for integers

- The "best" shape for x+y+z depends on context
  There may be several conflicting options

Code Shape

- Another example – the switch statement
  - Implement it as a jump table
    - Lookup address in a table & jump to it
    - Uniform (constant) cost
  - Implement it as cascaded if-then-else statements
    - Cost depends on where your case actually occurs
    - O(number of cases)
  - Implement it as a binary search
    - Uniform (log n) cost
- Compiler must choose best implementation strategy
  No way to convert one into another

Order of evaluation

- Ordering for performance
  - Using associativity and commutativity
    - Very hard problem
  - Operands
    - op1 must be preserved while op2 is computed
    - Emit code for more intensive one first
- Language requirements
  - Sequence points:
    - Places where side effects must be visible to other operations
  - C examples:

`f() + g()`	may be executed in any order
`f() || g()`	f must be executed first
`f(i++)`	argument to f must be i+1

Code Generation

- Tree-walk algorithm
  - Notice: generates code for children first
  - Effectively, a bottom up algorithm
  - So that means….
- Right! Use syntax directed translation
  - Can emit LIR code in productions
  - Pass registers in $$, $1, $2, etc.
  - Tricky part: assignment

One-pass code generation

```
Goal::= Expr:e  {: RES = e :}
Expr::= Expr:e + Term:t
      {: r = new_temp();
         emit( r = e + t );
         RES = r; :}
    | Expr:e - Term:t
      {: r = new_temp();
         emit( r = e - t );
         RES = r; :}
```

```
Term::= Term:t * Fact:f
      {: r = new_temp();
         emit( r = t * f );
         RES = r; :}
    | Term:t / Fact:f
      {: r = new_temp();
         emit( r = t - f );
         RES = r; :}
```

```
Fact::= ID:i  {: r = new_temp();
                 o = symtab.getOffset(i);
                 emit( r = load sp + o );
                 RES = r; :}
    | NUM:n {: r = new_temp();
               emit( r = $n );
               RES = r; :}
```


Next time…

- Midterm
- Midterm course evaluations
- Then:
  - More code generation
  - Conference report
- Don't forget about PA2