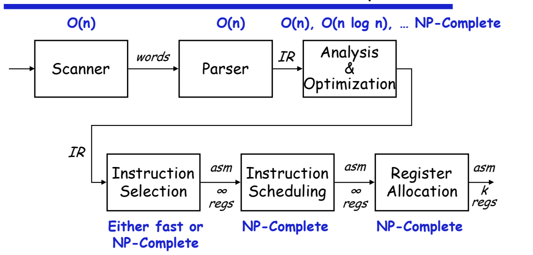
Code Generation

Roadmap (Where are we?)

Last lecture
- Type checking
- Intermediate representations

This lecture
- Code generation
  - → Code shape
  - → Expressions
    - 3-address code
    - Stack code
  - → Assignments

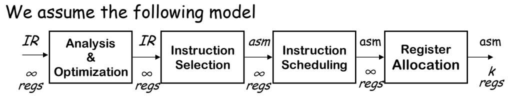
Code Generation: Structure of a Compiler (EaC Ch. 7)



A compiler is a lot of fast stuff followed by some hard problems
- → The hard stuff is mostly in code generation and optimization
- → For superscalars, allocation & scheduling is particularly important

Structure of a Compiler

We assume the following model



- Selection is fairly simple (problem of the 1980s)
- Allocation & scheduling are complex

What about the IR ?
- Low-level, RISC-like IR called ILOC
- Has "enough" registers
- ILOC was designed for compiler backend research

Branches, compares, & labels
Memory tags
Provision for multiple ops/cycle

Definitions

Instruction selection
- Mapping *IR* into assembly code
- Assumes a fixed storage mapping & code shape
- Combining operations, using address modes

Instruction scheduling
- Reordering operations to hide latencies
- Assumes a fixed program *(set of operations)*
- Changes demand for registers

Register allocation
- Deciding which values will reside in registers
- Changes the storage mapping, may add false sharing

These 3 problems are tightly coupled.

The Big Picture

How hard are these problems?

Instruction selection
- Can make locally optimal choices, with automated tool
- Global optimality is (undoubtedly) NP-Complete

Instruction scheduling
- Single basic block ⇒ heuristics work quickly
- General problem, with control flow ⇒ NP-Complete

Register allocation
- Single basic block, no spilling, & 1 register size ⇒ linear time
- Whole procedure is NP-Complete

The Big Picture

Conventional wisdom says that we lose little
by solving these problems independently

Instruction selection

Note: many <u>fuzzy</u> terms here!

- Use some form of pattern matching
- Assume enough registers or target "<u>important</u>" values

Instruction scheduling
- Within a block, list scheduling is "<u>close</u>" to optimal

Optimal for > 85% of blocks

- Across blocks, build framework to apply list scheduling

Register allocation
- Start from virtual registers & map into *k*
- Focus on good priority heuristic

Code Shape

Definition
- All those nebulous properties of the code that impact performance & code "quality"
- Includes code, approach for different constructs, cost, storage requirements & mapping, & choice of operations
- Code shape is the end product of many decisions *(big & small)*

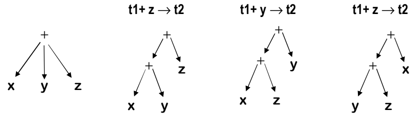
Impact
- Code shape influences algorithm choice & results
- Code shape can encode important facts, or hide them

Rule of thumb: expose as much derived information as possible
- Example: explicit branch targets in ILOC simplify analysis

Code Shape (assuming no rule for associativity)

An example

x + y + z	x + y → t1 t1+ z → t2	x + z → t1 t1+ y → t2	y + z → t1 t1+ z → t2



source level intermediate level

- 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 contextual knowledge
- → There may be several conflicting options

Code Shape

Another example -- the case statement
- 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
- Implement it as a jump table
  - → Lookup address in a table & jump to it
  - → Uniform (constant) cost

Compiler must choose best implementation strategy
No amount of massaging or transforming will convert one into another

Generating 3-address Code

- We'll first target RISC architectures
- Properties
  - → Explicit loads and stores (to registers)
  - → Explicit references to registers
    - Either virtual or physical registers
- Example instructions
  - → load r1, <addr>   // r1 ← value at <addr>
  - → loadi r1, <const>   // r1 ← value of <const>
  - → store r1, <addr>   // <addr> ← r1
  - → move r1, r2   // r1 ← r2
  - → add r1, r2, r3   // r1 ← r2 + r3
  - → sub r1, r2, r3   // r1 ← r2 - r3
  - → mult r1, r2, r3   // r1 ← r2 * r3
  - → neg, r1, r2   // r2 ← -r2
  - → jmp <addr>   // jump to <addr>

Generating 3-address Code for Expressions

The key code quality issue is holding values in registers

- When can a value be safely allocated to a register?
  - → When only 1 name can reference its value
  - → Pointers, parameters, aggregates & arrays all cause trouble
- When should a value be allocated to a register?
  - → When it is both <u>*safe*</u> & <u>*profitable*</u>

Encoding this knowledge into the *IR*
- Use code shape to make it known to every later phase
- Assign a virtual register to anything that can go into one
- Load or store the others at each reference

Relies on a strong register allocator (register-register model)

Generating 3-address Code for Expressions

```
expr(node) {
  int result, t1, t2;
  switch (type(node)) {
     case ×,÷,+,– :
        t1 ← expr(left_child(node));
        t2 ← expr(right_child(node));
        result ← NextRegister();
        emit (op(node), result, t1, t2);
        break;
     case IDENTIFIER:
        t1 ← addr(node);
        result ← NextRegister();
        emit (load, result, t1);
        break;
     case NUMBER:
        result ← NextRegister();
        emit (loadi, result, val(node));
        break;
   }
   return result;
}
```

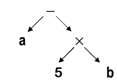
The concept

- Use a simple treewalk evaluator
- Bury complexity in routines it calls
  - addr(), op() & val()
- Implements expected behavior
  - Visits & evaluates children
  - Emits code for the op itself
  - Returns register with result
- Works for simple expressions
- Easily extended to other operators
- Does not handle control flow

Generating 3-address Code for Expressions

```
expr(node) {
  int result, t1, t2;
  switch (type(node)) {
     case ×,÷,+,– :
        t1 ← expr(left_child(node));
        t2 ← expr(right_child(node));
        result ← NextRegister();
        emit (op(node), result, t1, t2);
        break;
     case IDENTIFIER:
        t1 ← addr(node);
        result ← NextRegister();
        emit (load, result, t1);
        break;
     case NUMBER:
        result ← NextRegister();
        emit (loadi, result, val(node));
        break;
   }
   return result;
}
```

Example



Produces

```
Expr(-)
   Expr(a)
      load, r1, addr(a)
   Expr(x)
      Expr(5)
         loadi, r2, val(5)
      Expr(b)
         load, r3, addr(b)
      mult, r4, r1, r3
   sub, r5, r1, r4
```


Extending the Simple Treewalk Algorithm

More complex cases for IDENTIFIER

- What about values in registers?
  - → Modify the **IDENTIFIER** case
  - → Already in a register ⇒ return the register name
  - → Not in a register ⇒ load it as before, but record the fact
  - → Choose names to avoid creating false dependences (*NextRegister()*)
- What about parameter values?
  - → Many linkages pass the first several values in registers
  - → Call-by-value ⇒ just a local variable with "funny" offset
  - → Call-by-reference ⇒ needs an extra indirection
- What about function calls in expressions?
  - → Generate the calling sequence & load the return value
  - → Severely limits compiler's ability to reorder operations

Generating Stack Code

- We'll next target stack code
- Properties
  - → Explicit loads and stores (to stack)
  - → Implicit reference to top of stack
- Example instructions

Java Stack Code	Effect
nop	none
ldc_int c	push constant *c* onto stack
iload index(x)	push local variable X onto stack
istore index(x)	pop stack, store in local variable X
iadd	pop 2 elems off stack, add, push
isub	pop 2 elems off stack, subtract, push
imult	pop 2 elems off stack, multiply, push
ineg	pop stack, negate, push

Generating Stack Code for Expressions

```
expr(node) {
  int result, t1, t2;
  switch (type(node)) {
     case ×,÷,+,– :
        expr(left_child(node));
        expr(right_child(node));
        emit (op(node));
        break;
     case IDENTIFIER:
        emit (iload, addr(node));
        break;
     case NUMBER:
        emit (ldc_int, val(node));
        break;
   }
   return;
}
```

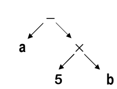
The concept

- Simple treewalk evaluator
- Bury complexity in routines it calls
  - addr(), op() & val()
- Implements expected behavior
  - Visits & evaluates children
  - Emits code for the op itself
  - No return value needed
  - Makes use of implicit stack

Generating Stack Code for Expressions

```
expr(node) {
  int result, t1, t2;
  switch (type(node)) {
     case ×,÷,+,– :
        expr(left_child(node));
        expr(right_child(node));
        emit (op(node));
        break;
     case IDENTIFIER:
        emit (iload, addr(node));
        break;
     case NUMBER:
        emit (ldc_int, val(node));
        break;
   }
   return;
}
```

Example



Produces

```
Expr(-)
   Expr(a)
      iload addr(a)
   Expr(x)
      Expr(5)
         ldc_int val(5)
      Expr(b)
         iload addr(b)
      imult
   isub
```

Extending the Simple Treewalk Algorithm

Adding other operators

- Evaluate the operands, then perform the operation
- Complex operations may turn into library calls
- Handle assignment as an operator

Mixed-type expressions

- Insert conversions as needed from conversion table
- Most languages have symmetric & rational conversion tables

Typical Addition Table

+	Integer	Real	Double	Complex
Integer	Integer	Real	Double	Complex
Real	Real	Real	Double	Complex
Double	Double	Double	Double	Complex
Complex	Complex	Complex	Complex	Complex

Extending the Simple Treewalk Algorithm

What about evaluation order?

- Can use commutativity & associativity to improve code
- This problem is truly hard

What about order of evaluating operands?

- 1^{st} operand must be preserved while 2^{nd} is evaluated
- Takes an extra register that cannot be used for 2^{nd} operand
- Should evaluate more demanding operand expression first

(Ershov in the 1950's, Sethi in the 1970's)

Taken to its logical conclusion, this creates Sethi-Ullman register allocation scheme (ASU p. 564)

Generating Code in the Parser

Need to generate an initial IR form

- Might generate an AST, use it for some high-level, near-source work (type checking, optimization, alias analysis), then traverse it and emit a lower-level IR similar to ILOC

The big picture

- Recursive algorithm really works bottom-up
  - → Actions on non-leaves occur after children are done
- Can encode same basic structure into *ad-hoc* SDT scheme
  - → Identifiers load themselves & stack virtual register name
  - → Operators emit appropriate code & stack resulting VR name
  - → Assignment requires evaluation to an *lvalue* or an *rvalue*

Ad-hoc SDT versus a Recursive Treewalk

```
expr(node) {
  int result, t1, t2;
  switch (type(node)) {
      case ×,÷,+,– :
          t1← expr(left child(node));
          t2← expr(right child(node));
          result ← NextRegister();
          emit (op(node), t1, t2, result);
          break;
      case IDENTIFIER:
          t1← base(node);
          t2← offset(node);
          result ← NextRegister();
          emit (loadAO, t1, t2, result);
          break;
      case NUMBER:
          result ← NextRegister();
          emit (loadI, val(node), none, result);
          break;
      }
      return result;
}
```

```
Goal :    Expr  { $$ = $1; } ;
Expr:     Expr PLUS Term
          { t = NextRegister();
            emit(add,$1,$3,t); $$ = t; }
     |    Expr MINUS Term  {…}
     |    Term { $$ = $1; } ;
Term:     Term TIMES Factor
          { t = NextRegister();
            emit(mult,$1,$3,t); $$ = t; };
     |    Term DIVIDES Factor {…}
     |    Factor { $$ = $1; };
Factor:   NUMBER
          { t = NextRegister();
            emit(loadI,val($1),none, t );
            $$ = t; }
     |    ID
           { t1 = base($1);
             t2 = offset($1);
             t = NextRegister();
            emit(loadAO,t1,t2,t);
            $$ =  t; }
```


Handling Assignment (just another operator)

lhs ← rhs

Strategy

- Evaluate *rhs* to a value *(an rvalue)*
- Evaluate *lhs* to a location *(an lvalue)*
  - → *lvalue* is a register ⇒ move rhs (*register/register copy*)
  - → *lvalue* is an address ⇒ store rhs
- If *rvalue* & *lvalue* have different types
  - → Evaluate *rvalue* to its "*natural*" type
  - → Convert that value to the type of **lvalue*

Unambiguous scalars go into registers

Ambiguous scalars or aggregates go into memory

Handling Assignment

What if the compiler cannot determine the rhs's type ?

- This is a property of the language & the specific program
- If type-safety is desired, compiler must insert a run-time check
- Add a *tag field* to the data items to hold type information

Code for assignment becomes more complex

evaluate rhs
if type(lhs) ≠ rhs.tag
then
convert rhs to type(lhs) *or*
signal a run-time error
lhs ← rhs

This is much more complex than if it knew the types

Handling Assignment

Compile-time type-checking

- Goal is to eliminate both the check & the tag
- Determine, at compile time, the type of each subexpression
- Use compile-time types to determine if a run-time check is needed

Optimization strategy

- If compiler knows the type, move the check to compile-time
- Unless tags are needed for garbage collection, eliminate them
- If check is needed, try to overlap it with other computation
  (*assumes target machine with instruction-level parallelism*)

Handling Assignment (with reference counting)

The problem with reference counting

- Must adjust the count on each pointer assignment
- Overhead is significant, relative to assignment

Code for assignment becomes

evaluate rhs
lhs→count ← lhs→count - 1
lhs ← addr(rhs)
rhs→count ← rhs→count + 1

Plus a check for zero at the end

This adds *1 +, 1 -, 2 loads, & 2 stores*

With extra functional units & large caches, this may become either cheap or free. What about power consumption?