CSE401: Code Generation

Larry Snyder
Spring 2003

Slides by Chambers, Eggers, Notkin, Ruzzo, Snyder and others
© L. Snyder and UW CSE, 1994-2003

Intermediate code generation

- Purpose: translate ASTs into linear sequence of simple statements called intermediate code
  - Can optimize intermediate code in place
  - A later pass translates intermediate code into target code
- Intermediate code is machine-independent
  - Don’t worry about details of the target machine (e.g., number of registers, kinds of instruction formats)
  - Intermediate code generator and optimizer are portable across target machines
- Intermediate code is simple and explicit
  - Decomposes code generation problem into simpler pieces
  - Constructs implicit in the AST become explicit in the intermediate code

PL/0

- Our PL/0 compiler merges intermediate and target code generation for simplicity of coding

Three-address code: a simple intermediate language

- Each statement has at most one operation in its right-hand side
  - Introduce extra temporary variables if needed
- Control structures are broken down into (conditional) branch statements
- Pointer and address calculations are made explicit

Examples

\[
\begin{align*}
  x & := y \times z + q / r \\
  t_1 & := y \times z \\
  t_2 & := q / r \\
  x & := t_1 + t_2
\end{align*}
\]

\[
\begin{align*}
  i & := 0 \\
  \text{loop:} & \\
  \text{if } i < 10 \text{ goto done; } \\
  i & := i + 1 \\
  \text{goto loop; } \\
  \text{done:}
\end{align*}
\]

\[
\begin{align*}
  x & := a[i] \\
  t_1 & := i \times 4 \\
  x & := *(a + t_1)
\end{align*}
\]
Available operations

- `var := constant`
- `var := var`
- `var := unop var`
- `var := var binop var`
- `var := proc(var, ..)`
- `var := * (var + constant)`
- `* (var + constant) := var`

ICG (Intermediate code generation) from ASTs

- Once again (like type checking), we'll do a tree traversal
- Cases
  - expressions
  - assignment statements
  - control statements
- declarations are already done

ICG for expressions

- How: tree walk, bottom-up, left-right, (largely postorder) assigning a new temporary for each result
- Pseudo-code

```c
Name IntegerLiteral::codegen(STS* s) {
  result := new Name;
  emit(result := _value);
  return result;
}
```

Another pseudo-example

```c
Name BinOp::codegen(SymTabScope* s) {
  Name e1 = _left->codegen(s);
  Name e2 = _right->codegen(s);
  result = new Name;
  emit(result := e1 _op e2);
  return result;
}
```

Example

```c
module main;
var z: int;
procedure p(var q: int);
var array[10] of int;
var b: int;
begin
  b := 1 + 2;
  q := q + 1;
  if b < 1 then b := b + 1
end p;
begin
  z := 5;
p(z);
end main.
```

Example

```c
module main;
var z, off; int;
procedure p(var q: int);
var array[10] of int;
var b: int;
begin
  b := 1 + 2;
  q := q + 1;
  if b < 1 then b := b + 1
end p;
begin
  a := 5;
p(a);
call p
```
Two cases

if we want l-value, compute address

if we want r-value, load value at that address
r-value

```cpp
Name LVvalue::codegen(SymTabScope* s) {
    int offset;
    Name base = codegen_address(s, offset);
    Name dest = new Name;
    emit(dest := "base + offset");
    return dest;
}
```

l-value

```cpp
Name VarRef::codegen_address(STS* s, int& offset) {
    ST* ste = s->lookup(_ident, foundScope);
    if (!ste->isConstant()) {
        Name dest = new Name;
        emit(dest := ste->value());
        return dest;
    }
    return LVvalue::codegen(s);
}
```

Compute address of frame containing variable

```cpp
Name SymTabScope::getCodeFrame(SymTabScope* foundScope) {
    Name curScope = ST;
    SymTabScope* curScope = this;
    while (curScope != foundScope) {
        Name newFrame = new Name; // load static link
        int offset = curScope->staticLinkOffset();
        emit(newFrame := "curFrame + offset");
        curScope = curScope->parent();
        curFrame = newFrame;
    }
    return curFrame;
}
```

ICG for assignments

```cpp
AssignStmt::codegen(SymTabScope* s) {
    int offset;
    Name base = _value->codegen_address(s, offset);
    Name result = _expr->codegen(s);
    emit(*base + offset := result);
}
```

ICG for function calls

```cpp
Name FunCall::codegen(SymTabScope* s) {
    for all arguments, from right to left {
        if (arg is byValue) {
            Name name = arg->codegen(s);
            emit(push name);
        } else {
            int offset;
            Name base = arg->codegen_address(s, offset);
            Name ptr = new Name;
            emit(ptr := base + offset);
            emit(push ptr);
        }
    }
    ...continued
```
Accessing call-by-ref params

- Formal parameter is address of actual, not the value, so we need an extra load statement
- Name VarRef::codegen_address(STS* s, int offset);
  int base = s->getFPFromScope();
  offset = s->offset();
  if (s->isFormalByRef()) {
    Name ptr = new Name;
    emit(ptr := * (base + offset));
    offset = 0;
    return ptr;
  }
  return base;

ICG for array accesses

- AST: array_expr[index_expr]
- Code generated:
  (array_b, array_o) := <base, offset of array_expr>
  i := <value of index_expr>
  delta := i * <size of element type>
  (elem_b, elem_o) := (array_b + delta, array_o)
- 2D Arrays? Not really:
  var MyArray array[10] of int;
  MyArray[i][j];

ICG for if statement

- void IfStmt::codegen(SymTabScope* s)
  Label else_lab = new Label;
  emit(if t = 0 goto else_lab);
  _then_stmts->codegen(s);
  Label done_lab = new Label;
  emit(goto done_lab);
  emit(else_lab);
  _else_stmts->codegen(s);
  emit(done_lab);

ICG for while statement

- Short-circuiting of and & or
  - Example
    if x <= 0 and y / x > 5 then
      b := y < x;
    end;
  - Treat as control structure, not operator:
    t0 := 0
    t1 := e1
    if false t1 goto 1
    t0 := e2
    l1: //value in t0

Prototype compiler structure
Target Code Generation

- Input: intermediate representations (IR)
  - Ex: three-address code
- Output: target language program
  - Absolute binary code
  - Relocatable binary code
  - Assembly code
  - C

Task of code generator

- Bridge the gap between intermediate code and target code
  - Intermediate code: machine independent
  - Target code: machine dependent
- Two jobs
  - Instruction selection: for each IR instruction (or sequence), select target language instruction (or sequence)
  - Register allocation: for each IR variable, select target language register/stack location

Instruction selection

- Given one or more IR instructions, pick the “best” sequence of target machine instructions with the same semantics
  - “best” = fastest, shortest
- Correctness is a big issue, especially if the code generator (codegen) is complex

Example

- IR code
  - t3 := t1 + t2
- Target code for MIPS
  - add $3,$1,$2
- Target code for SPARC
  - add $1,$2,$3
- Target code for 68k
  - mov.l d1,d3
  - add.l d2,d3

- Note that a single IR instruction may expand to several target instructions

Difficulty depends on instruction set

- RISC: easy
  - Usually only one way to do something
  - Closely resembles IR instructions
- CISC: hard
  - Lots of alternative instructions with similar semantics
  - Lots of tradeoffs among speed, size
  - Simple RISC-like translation may be inefficient
- C: easy, as long as C is appropriate for desired semantics
  - Can leave optimizations to the C compiler

Example

- IR code
  - t1 := t1 + 1
- Target code for MIPS
  - add $1,$1,1
- Target code for SPARC
  - add $1,$1
- Target code for 68k
  - add.l #1,d1
  - inc.l d1

- Can have choices
  - This is a pain, since choices imply you must make decisions
Example

- IR code (push x onto stack)
  - sp := sp - 4
  - sp := t1
- Target code for MIPS
  - sub sp, sp, 4
  - sw $t1, ($sp)
- Target code for SPARC
  - sub %sp, 4, %sp
  - st %1, [%sp + 0]
- Target code for 68k
  - mov.l d1, -(sp)

Note that several IR instructions may combine to a single target instruction
This is hard!

Instruction selection in PL/0

- Very simple instruction selection
- As part of generating code for an AST node
  - Merged with intermediate code generation, because it's so simple
- Interface to target machine: assembler class
  - Function for each kind of target instruction
  - Hides details of assembly format, etc.
  - Two assembler classes (MIPS and x86), but you only need to extend MIPS

Resource constraints

- Intermediate language uses unlimited temporary variables
  - This makes intermediate code generation easy
- Target machine, however, has fixed resources for representing "locals"
  - MIPS, SPARC: 31 registers minus SP, FP, RetAddr, Arg1-4, ...
  - 68k: 16 registers, divided into data and address registers
  - x86: 4(?) general-purpose registers, plus several special-purpose registers

Register allocation

- Using registers is
  - Necessary: in load/store RISC machines
  - Desirable: since much faster than memory
  - So...
    - Should try to keep values in registers if possible
    - Must reuse registers for many temp variables, so we must free registers when no longer needed
    - Must be able to handle out-of-registers condition, so we must spill some variables to stack locations
    - Interacts with instructions selection, which is a pain, especially on CISCs

Classes of registers

- What registers can the allocator use?
  - Fixed/dedicated registers
  - Claimed by machine architecture, calling convention, or internal convention for special purpose
  - Not easily available for storing locals
  - Scratch registers
  - A couple of registers are kept around for temp values
    - E.g., loading a spilled value from memory to operate upon it
  - Allocatable registers
  - Remaining registers are free for the allocator to allocate
    - (PL/0 on MIPS: $8-$25)

Which variables go in registers?

- Temporary variables: easy to allocate
  - Defined and used exactly once, during expression evaluation
  - So the allocator can free the register after use easily
  - Usually not too many in use at one time
  - So less likely to run out of registers
  - Local variables: hard, but doable
    - Need to determine last use of variable to free register
    - Can easily run out of registers, so need to make decisions
      - What about load/store to a local through a pointer?
      - What about the debugger?
  - Global variables, procedure params, across calls, …:
    - Really hard. A research project?
PL/0’s simple allocator design

- Keep set of allocated registers as codegen proceeds
  - RegisterBank class
- During codegen, allocate one from the set
  - Reg reg = rb->getNew();
  - Side-effects register bank to note that reg is taken
- What if no registers are available?
  - When done with a register, release it
    - rb->free(reg);
    - Side-effects register bank to note that reg is free

---

Example

```pl0
Name IntegerLiteral::codegen(SymTabScope* s) {  
  result := new Name;
  emit(result := _value);  
  return result;
}
```

```
Reg IntegerLiteral::codegen(SymTabScope* s, RegisterBank* rb) {  
  Reg r = rb->newReg();
  TheAssembler->moveImmediate(r, _value);  
  return r;
}
```

---

Connection to ICG

- In the last lecture, the pseudo-code often create a new Name
- Since PL/0 merges intermediate code generation (ICG) with target generation, these new Names are equivalent to allocating registers in PL/0

---

Codegen for assignments

```pl0
AssignStmt::codegen(SymTabScope* s) {  
  int offset;
  Name base = _value->codegen_addr(x, offset);
  Name result = _expr->codegen(x);
  emit(* (base + offset) := result);
}
```

```
void AssignStmt::codegen(SymTabScope* s, RegBank* rb) {  
  int offset;
  Reg base = _value->codegen_address(x, rb, offset);
  Reg result = _expr->codegen(x, rb);
  TheAssembler->store(result, base, offset);
  rb->freeReg(base);
  rb->freeReg(result);
}
```

---

Codegen for if statements

```pl0
void IfStmt::codegen(SymTabScope* s, RegBank* rb) {

  Reg test = _test->codegen(x, rb);
  char* elseLabel = TheAssembler->newLabel();
  TheAssembler->branchFalse(test, elseLabel);
  rb->freeReg(test);
  for (int i=0; i < _then_stmts->length(); i++) {
    _then_stmts->fetch(i)->codegen(s, rb);
  }
  TheAssembler->insertLabel(elseLabel);
}
```

---

Codegen for call statements

```pl0
void CallStmt::codegen(SymTabScope* s, RegBank* rb) {

  for (int i = _args->length(); i >= 0; i--) {
    Reg arg = _args->fetch(i)->codegen(s, rb);
    TheAssembler->push(arg);
  }

  SymTabScope* encScope;
  SymTabEntry* ste = s->lookup(_ident, encScope);
  Reg staticLink = s->getFPOf(encScope, rb);
  TheAssembler->push(staticLink);
  rb->saveRegs(s);
  TheAssembler->call(_ident);
  rb->restoreRegs(s);
  TheAssembler->popMultiple(_args->length() + 1);
}
```
Another example

```c
Name BinOp::codegen(SymTabScope * s) {
Name e1 = _left->codegen(s);
Name e2 = _right->codegen(s);
result = new Name;
emit(result -> e1 _op e2);
return result;
}
Reg BinOp::codegen(SymTabScope * s, RegBank * rb) {
Reg expr1 = _left->codegen(s, rb);
Reg expr2 = _right->codegen(s, rb);
rb->freeReg(expr1);
rb->freeReg(expr2);
Reg dest = rb->newReg();
TheAssembler->binop(_op, dest, expr1, expr2);
return dest;
}
```

Example

```c
lw $8, 0($fp)
li $9, 2
lw $10, 0($fp)
li $11, 1
sub $12, $10, $11
mul $10, $9, $12
add $9, $8, $10
sw $9, 0($fp)
```

Example, con't

```c
lw $8, 0($fp)
li $9, 2
lw $10, 0($fp)
li $11, 1
sub $10, $10, $11
mul $9, $9, $10
add $8, $8, $9
sw $8, 0($fp)
```

Example, con't

```c
module main;
var z : int;
procedure p(var q : int);
var b : int;
begin
b := 1 + 2;
b := b + 3;
b := a[0][0];
if b1 then b := 0 end
end p;
begin
z := 5;
ed while;
```

Example

```c
module main;
var a : int;
procedure p(var q : int);
var array[10] of int;
var b : int;
begin
b := 1 + 2;
b := b + 3;
b := a[0][0];
if b1 then b := 0 end
end p;
begin
z := 5;
ed while;
```
Example

module main;
var z:int;
procedure pr q: int);
var b:int;
begin
  b := 1 + z;
  q := q + 1;
  if b > 1 then b := 0 end
end pr;
begin
  z := 5;
  pr(z);
end main.

Example

module main;
var z:int;
procedure pr q: int);
var b:int;
begin
  b := 1 + z;
  q := q + 1;
  if b > 1 then b := 0 end
end pr;
begin
  z := 5;
  pr(z);
end main.