CSE401: Target Code Generation

David Notkin
Autumn 2000

Target Code Generator

- 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

Difficult 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
  - `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

Example

- IR code
  - `t1 := t1 + 1`
- Target code for MIPS
  - `add $1,$1,1`
- Target code for SPARC
  - `add %1,1,%1`
- Target code for 68k
  - `add.l #1,d1` **or**
  - `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 $1,0($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.

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

- Registers are *much* faster than memory
- Must use registers in load/store RISC machines
- 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
  - SP, FP, return address, …
  - 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
    - For example, loading a spilled value from memory to operate upon it
- Allocatable registers
  - Remaining registers free for the allocator to allocate (PL/0 on MIPS, $8-$25)

What variables can be put in registers?

- Temporary variables: easy to allocate
  - Defined and used exactly once, during expression evaluation
    - So the allocator can free the register after use
  - 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 in order to free register
  - Can easily run out of registers, so need to make decision about which variables get the registers
  - What about assignments to a local through a pointer?
  - What about the debugger?

What about globals in registers?

- Really hard, but doable as 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 record 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 record that `reg` is free

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 `Name`s are equivalent to allocating registers in PL/0

Example

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

vs.

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


Another ICG pseudo-example

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

Codegen example

```
Reg BinOp::codegen(SymTabScope* s, RegisterBank* 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;
}
```


Codegen example

```
void IfStmt::
        codegen(SymTabScope* s, RegisterBank* rb) {
  Reg test = _test->codegen(s, 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);
}
```



```
void CallStmt::
        codegen(SymTabScope* s, RegisterBank* rb) {
  for (int i = _args->length() - 1; i >= 0; i--) {
    Expr* arg = _args->fetch(i);
    Reg areg = arg->codegen(s, rb);
    TheAssembler->push(areg);rb->freeReg(areg);
  }
  SymTabScope* enclScope;
  SymTabEntry* ste = s->lookup(_ident, enclScope);
  …
  Reg staticLink = s->getFPOf(enclScope, rb);
  TheAssembler->push(staticLink);
  rb->freeReg(staticLink);
  rb->saveRegs(s);
  TheAssembler->call(_ident);
  rb->restoreRegs(s);
  TheAssembler->popMultiple((_args->length() + 1) *
                        TheAssembler->wordSize());
}
```


Next lecture

- Run-time system calls
- Beginning of optimization
  - There is none in PL/0 (either basic or extended)

Another example

```
void AssignStmt::
       codegen(SymTabScope* s, RegisterBank* rb) {
  int offset;
  Reg base = _lvalue->codegen_address(s, rb, offset);

  Reg result = _expr->codegen(s, rb);

  TheAssembler->store(result, base, offset);

  rb->freeReg(base);
  rb->freeReg(result);
}
```