Target Code Generation

Input: intermediate language (IL) or representation (IR)
Intermediate languages/representations:

- three-address code
- AST's + symbol table

Output: target language program
Target languages:

- absolute binary (machine) code
- relocatable binary code
- assembly code
- C

Task of code generator

Bridge the gap between intermediate code & target code

Intermediate code: machine independent
Target code: machine dependent

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 "best" sequence of target machine instructions
with same semantics

"best" = fastest, shortest

Difficulty depends on nature of target 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 not be very efficient
- C: easy, as long as C appropriate for desired semantics
  - can leave optimizations to C compiler

Correctness a big issue, particularly if codegen complex

Example

IR code:

```
t3 := t1 + t2
```

Target code (MIPS):

```
add $3,$1,$2
```

Target code (SPARC):

```
add %1,%2,%3
```

Target code (68k):

```
mov.l d1,d3
add.l d2,d3
```

1 IR instruction may expand to several target instructions

Another example

IR code:

```
t1 := t1 + 1
```

Target code (MIPS):

```
add $1,$1,1
```

Target code (SPARC):

```
add %1,1,%1
```

Target code (68k):

```
add.l #1,d1
```

or

```
inc.l d1
```

Can have choices

- it's a pain to have choices; requires making decisions

Yet another example

IR code:

```
// push x onto stack
sp  := sp - 4
*sp := t1
```

Target code (MIPS):

```
sub $sp,$sp,4
sw $1,0($sp)
```

Target code (SPARC):

```
sub %sp,4,%sp
st %1,[%sp+0]
```

Target code (68k):

```
mov.l d1,-(sp)
```

Several IR code instructions can combine to 1 target instruction
⇒ **hard!**

A final example

Source code:

```
a++; // "a" is a local variable
```

IR code:

Target code:

Instruction selection in PL/0

Do very simple instruction selection,
as part of generating code for AST node

- merged with ICG, because so simple

Interface to target machine: `assembler` class

- function for each kind of target instruction
- hides details of asm format, etc.

Register allocation

Intermediate language uses unlimited temporary variables
- makes ICG easy

Target machine has fixed resources for representing “locals”
- MIPS, SPARC: 31 registers,
  minus SP, FP, RetAddr, Arg1-4, ...
- 68k: 16 registers, divided into data and address regs
- x86: 4(?) general-purpose registers,
  plus several special-purpose registers

Registers *much* faster than memory
Must use registers in load/store RISC machines

Consequences:
- should try to keep values in registers if possible
- must reuse registers for many temp vars
  ⇒ free registers when no longer needed
- must be able to handle out-of-registers condition
  ⇒ **spill** some variables from register to stack locations
- interacts with instruction selection, on CISCs
  ⇒ a real pain

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
- couple of registers kept around for temp values
  - e.g. loading a spilled value from memory in order to operate on it

Allocatable registers
- remaining registers free for register allocator to exploit
  - PL/0 on MIPS: `$8-$25`

Classes of variables

What variables can the allocator try to put in registers?

Temporary variables: easy to allocate
- defined & used exactly once, during expression evaluation
  ⇒ allocator can free up register when used
- usually not too many in use at one time
  ⇒ less likely to run out of registers

Local variables: hard, but doable
- need to determine **last use** of variable in order to free reg
- can easily run out of registers
  ⇒ need to make decision about which variables get regs
- what about assignments to local through pointer?
- what about debugger?

Global variables:
really hard, but doable as a research project

Simple register allocator design

Used in PL/0

Keep set of allocated registers as codegen proceeds
- `RegisterBank` class in PL/0

During codegen, allocate one from set
- `Reg reg = rb->getNew();`
- side-effects reg bank to record that `reg` is taken
- what if no registers available?

When done with register, release it
- `rb->free(reg);`
- side-effects reg bank to record that `reg` is free

Some codegen routines

```
Reg IntLiteral::codegen(Scope* s, RegBank* rb) {
   Reg dest = rb->getNew();
   TheAssembler->loadImmediate(dest, _value);
   return dest;
}

Reg BinOp::codegen(Scope* s, RegBank* rb) {
   Reg r1 = _left->codegen(s, rb);
   Reg r2 = _right->codegen(s, rb);
   rb->free(r1);
   rb->free(r2);
   Reg dest = rb->getNew();
   TheAsm->arith(_op, dest, r1, r2);
   return dest;
}

void AssignStmt::codegen(Scope* s, RegBank* rb) {
   Reg result = _expr->codegen(s, rb);
   int offset;
   Reg base = _lvalue->codegen_addr(s, rb, offset);
   TheAsm->store(result, base, offset);
   rb->free(base);
   rb->free(result);
}
```


Function call codegen routine

```
Reg FnCall::codegen(Scope* s, RegBank* rb) {
   // eval & push arguments
   foreach arg, right to left {
      Reg a;
      if (pass by value) {
         a = arg->codegen(s, rb);
      } else {
         // pass by ref
         int offset;
         Reg base = arg->codegen_addr(s, rb, offset);
         rb->free(base);
         a = rb->getNew();
         TheAsm->arithImmediate(PLUS,a,base,offset);
      }
      TheAsm->push(SP, a);
      rb->free(a);
   }

   ...
```



```
   ...

   // eval & push static link
   Reg link = s->getFPOf(enclosingScope, rb);
   TheAsm->push(SP, link);
   rb->free(link);

   // save any allocated regs across call
   rb->saveRegs(s);

   // call
   TheAsm->call(_ident);

   // restore saved regs
   rb->restoreRegs(s);

   // pop off args & static link
   TheAsm->popMultiple(SP, (#args+1) * sizeof(int));

   // allocate temp reg for result of call
   Reg dest = rb->getNew();
   TheAsm->move(dest, RET0);

   // return result
   return dest;
}
```