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
- Typically, the intermediate representation (IR) is built from AST and manipulated while optimizing the code

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

```
x := y * z + q / r
```
```
t1 := y * z
t2 := q / r
x := t1 + t2
```
```
for i := 0 to 10 do
  i := i + 1
  goto loop
end
```
```
i := 0
loop:
  if i < 10 goto done;
  i := i + 1
  goto loop;
done:
```
```
x := a[i]
```
```
t1 := i * t1
x := * (a + t1)
```
Available operations

- var := constant
- var := var
- var := unop var
- var := bop var
- var := proc(var, ...)
- var := &var
- var := *(var + constant)
- *(var + constant) := var
- if var goto label
- goto label
- label:
- return var
- return
generally one operation per statement, not arbitrary expressions, etc.

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

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

Another pseudo-example

```cpp
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;
}
```

ICG for variable references

- Two cases
  - if we want l-value, compute address
  - if we want r-value, load value at that address

```cpp
Name LVRef::codegen(SymTabScope* s) {
  ST* ste = s->lookup(_ident, foundScope);
  if (ste->isConstant) {
    Name dest = new Name;
    emit(dest := ste->value());
    return dest;
  }
  return LVRef::codegen(s);
}
```

r-value

```cpp
Name VarRef::codegen(SymTabScope* s) {
  int offset;
  Name base = codegen_address(a, offset);
  Name dest = new Name;
  emit(dest := *base + offset);
  return dest;
}
```
I-value

```c
l-value

Name VarRef::codegen_address(STS* s, int4 offset)
{
    STE* ste = s->lookup(_ident, foundScope);
    if (!ste->isVariable()) {
        // fatal error
    }
    Name base = s->getFPOf(foundScope);
    offset = ste->offset();
    // base + offset = address of variable
    return base;
}
```

Compute address of frame containing variable

```c
Compute address of frame containing variable

Name SymTabScope::getFPOf(foundScope) {
    Name curFrame = FP;
    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

```c
ICG for assignments

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

ICG for function calls

```c
ICG for function calls

Name FunCall::codegen(SymTabScope* s) {
    forall arguments, from right to left {
        if (arg is byValue) {
            Name name = arg->codegen(s);
            emit(push name);
        } else {
            int offset;
            Name base = arg->codegen_addr(s, offset);
            Name ptr = new Name;
            emit(ptr := base + offset);
            emit(push ptr);
        }
    }
    ... continued
```

ICG for function calls, con't

```c
ICG for function calls, con’t

s->lookup(_ident, foundScope);
Name link = s->getFPOf(foundScope);
emit(push link); // callee’s static link
emit(call _ident)

Name result = new Name;
emit(result := RET0);
return result;
```

Accessing call-by-ref params

```c
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, int4 offset){
    ste = s->lookup(_ident, foundScope);
    Name base = s->getFPOf(foundScope);
    offset = ste->offset();
    if (ste->isFormalByRef()) {
        Name ptr = new Name;
        emit(ptr := *(base + offset));
        offset = 0;
    }
    return ptr;
    return base;
```

returning two things
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:
MyArray[i][j];

ICG for if statement

void IfStmt::codegen(SymbolTable* s) {
Same t = _test->codegen(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);
_else_stmts->codegen(s);
emit(done_lab);
}

ICG for while statement

ICG for break 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:
e1 and e2 →
t0 := 0
t1 := e1
iffalse t1 goto 1
0 := e2
1: //value in t0

Example
module main;
var z:int;
procedure p(var q:int);
var b: int;
begin
b := 1 + 2;
b := b + 5;
g := q + 1;
b := a[4][0];
if b > 1 then b := 0 end;
begin
p(s);
end;
end main.

\[
t_1 := 5
*(fp+z_{offset}) := t_1
\]
Prototype compiler structure

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

Target Code Generation

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

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
  - 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
  - addl.1 d2, d3

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

- IR code
  - \( t_1 := t_1 + 1 \)

- Target code for MIPS
  - add $1, d1, d1

- Target code for SPARC
  - add.d $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
  - \( *sp := t_1 \)

- 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.
  - 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 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
    - 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 eval
  - 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, need to make decisions
  - What about loadstore 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

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

Example

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

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

CodeGen for assignments

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

- void AssignStmt::codegen(SyntabScope* s, RegisterBank* rb) {
  int offset;
  Reg base = _value->codegen_address(s, rb, offset);
  emit(*base + offset) := result;
  rb->freeReg(base);
}

CodeGen for if statements

- void IfStmt::codegen(SyntabScope* s, RegisterBank* rb) {
  Reg reg = _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]btcodegen(s, rb);
  }
  TheAssembler->insertLabel(elseLabel);
}
**Example**

```
module main;
var z: int;
procedure p(var z: int);
var b: int;
begin
  b := 1 + 2;
b := b + 3;
q := b + li;
b := a[4][8];
if b > 0 then b := 0 end
end;
begin
  z := 5;
p(z);
end main.
```

**Example, con't**

```
module main;
var z: int;
procedure p(var z: int);
var b: int;
begin
  b := 1 + 2;
b := b + 3;
q := b + li;
b := a[4][8];
if b > 0 then b := 0 end
end;
begin
  z := 5;
p(z);
end main.
```
Example

module main;
var z: int;
procedure p(var q: int);
var b: int;
begin
b := b + 2;
q := q + 1;
b := a[4][8];
if b > 1 then b := 0
end p;
b := 1 + 2;
b := b + z;
q := q + 1;
b := a[3][8];
if b > 1 then b := 0
end p;
z := 3;
p(z);
end main.

Example

module main;
var z: int;
procedure p(var q: int);
var b: int;
begin
b := b + 2;
b := b + 4;
q := q + 2;
b := a[4][8];
if b > 1 then b := 0
end p;
b := 1 + 2;
b := b + z;
q := q + 1;
b := a[3][8];
if b > 1 then b := 0
end p;
z := 3;
p(z);
end main.

Example

module main;
var z: int;
procedure p(var q: int);
var b: int;
begin
b := b + 2;
b := b + 4;
q := q + 2;
b := a[4][8];
if b > 1 then b := 0
end p;
b := 1 + 2;
b := b + z;
q := q + 1;
b := a[3][8];
if b > 1 then b := 0
end p;
z := 3;
p(z);
end main.

Example

module main;
var z: int;
procedure p(var q: int);
var b: int;
begin
b := b + 2;
b := b + 4;
q := q + 2;
b := a[4][8];
if b > 1 then b := 0
end p;
b := 1 + 2;
b := b + z;
q := q + 1;
b := a[3][8];
if b > 1 then b := 0
end p;
z := 3;
p(z);
end main.