CSE401: Intermediate Code Generation

David Notkin
Autumn 2000

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
  - Decomposing whole 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 branch and goto statements
- Pointer and address calculations are made explicit

Examples

A.
```
x := y * z + q / r
```

A.
```
t1 := y * z
t2 := q / r
x := t1 + t2
```

B.
```
for i := 0 to 10 do …
end
```

B.
```
i := 0
loop:
  if i < 10 goto done;
  …
  i := i + 1
  goto loop;
done:
```

C.
```
x := a[i]
```

C.
```
t1 := i * 4
x  := *(a + t1)
```


Available operations

- `var := constant`
- `var := var`
- `var := unop var`
- `var := var binop var`
- `var := proc(var, …)`
- `var := &var`
- `var := *(var + constant)`
- `*(var + constant) := var`
- `if var goto label`
- `goto label`
- `label:`
- `return var`
- `return`

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, assigning a new temporary for each result
- Pseudo-code

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


Another pseudo-examples

```
Name BinOp::codegen(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

r-value

```
Reg LValue::codegen(s) {
  int offset;
  Name base = codegen_address(s, offset);
  Name dest = new Name;
  emit(dest := (base + offset));
  return dest;
}

Name VarRef::codegen(s) {
  ste = s->lookup(_ident,foundScope);
  if (ste->isConstant()) {
    Name dest = new Name;
    emit(dest := ste->value());
    return dest;
  }
  return Lvalue::codegen(s);
}
```


l-value

```
Name VarRef::codegen_address(s, int& offset) {
  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

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


ICG for assignments

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


ICG for function calls

```
Name FunCall::codegen(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 on next slide…

con't

```
  s->lookup(_ident,foundScope);
  Name link = s->getFPOf(foundScope);
  emit(push link);

  emit(call _ident)

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


Accessing call-by-ref parameters

- Formal parameter is address of actual, not the value, so we need an extra load statement

```
Name VarRef::codegen_address(s, int& offset)
  ste->s->lookup(_ident,foundScope);
  Name base = s->FPOf(foundScope);
  offset = ste->offset();
  if (ste->isFormalByRef()) {
    Name ptr = new Name;
    emit(ptr := *(base + offset));
    offset = 0;
    return ptr;
  }
  return base;
}
```


ICG for array accesses

- `array_expr[index_expr]`
- Code generated:
  - `a_base := <addr of array_expr>`
    `i:= <value of index_expr>`
    `elem_offset :=`
    `i * <size of element type>`
    `elem_addr := a_base + elem_offset`

ICG for if statement

```
void IfStmt::codegen(s) {
  Name 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);
  emit(else_lab:);
  _else_stmts->codegen(s);
  emit(done_lab:);
}
```


ICG for while statement

ICG for break statement

Short-circuiting

- How to support short-circuit evaluation of `and` and `or`?
- Example
  - ```
    if x <> 0 and y / x > 5 then
       b := y < x;
    end;
    ```
- Treat as control structure, not operator
    ```