Class 34: Translating Declarations and Expressions

Held: Friday, 18 November 2011

Summary: Today we begin our serious consideration of the back end. In particular, we begin our consideration of the techniques used to translate expressions and assignment statements.

Related Pages:
- EBoard.

Notes:
- No CS extra next week.
- You can find the partially working STAC interpreter in our Examples folder. We’ll discuss it a bit.
- My solution to Phase 4 will be posted tonight.

Overview:
- Translating assignment statements.
- Translating expressions: General issues.
- Translating compound expressions.
- Translating variable expressions.
- Translating constant expressions.
- Translation strategies, revisited.
- Translating array elements.

Translating Assignment

- Typical form: \textit{var := exp}.
- Two alternative techniques:
  - One lets each subexpression choose its own memory location.
  - One tells each subexpression where to put the result.
- The types of the variable and expression should be the same
  - Ideally, we’ve inserted a coercion node already if they’re not.
- Version one: Choose your own memory location
  - 1. Build the code for the expression.
  - 2. Copy the value from the memory location associated with the result to the memory location associated with the variable.
- Version two:
  - 1. Build the code for the expression, telling it to put the result in the correct location.
Complicating Factors

- We want to leave the copy fairly generic at this stage, since the result and the variable can be in memory or registers.
- You may want to think about what it means for the variable and the expression to be compound types, like arrays or records. What copying should you then do?
  - For this reason, some languages disallow direct copying of records and arrays.
- We’ll come back to some of these issues when we deal with procedure calls.
- Note that you can better handle the coercion by changing the structure of the parse exp when you do type checking (to insert a "coerce-to-xxx" node).

Translating Basic Expressions

- Let’s consider how we might translate a variety of kinds of operations, using the two strategies described earlier
  - Version one: Each expression has a location attribute where you can find their result.
  - Version two: Expressions must put their results in a particular address.
- A general philosophy: Premature optimization can hurt. Generate reasonable code to start with and rely on your code optimizer later.

Translating Addition

- Consider an expression of the form $\text{exp}_0 + \text{exp}_1$
- Version one: Each expression describes its own result location.
  - 1. Translate the first argument, remember the location of the result.
  - 2. Translate the second argument, remember the location of the result.
  - 3. Call the appropriate operation, putting the result in an appropriate place.
- How do we do that last step (call the appropriate operation and put the result in the appropriate place)?
  - Some architectures provide three-argument arithmetic operations in which you store the result in one of the three arguments.
  - Other architectures provide two-argument arithmetic operations in which the result is stored in one of the two arguments.
  - Still others require you to work with a special result register, typically called the accumulator.
  - For now, we will use a three-argument version.
  - We can translate any three argument version to a two or one argument version.
    - The code may be less good when translated from three-argument to two-argument form than what we could generate directly in two-argument form.
    - We’ll rely on the “optimizer” to fix this.
- In pseudo-code
/**
 * Generate the code to evaluate the expression in exp and leave the
 * result in result.
 *
 * Assumes that it’s safe to modify result along the way.
 */
Location *
translate_exp (ParseTree *exp)
{
    if (is_constant (exp))
        // ...
    else if (is_add_op (exp))
    {
        Location *left = translate_exp (get_child (exp, 0));
        Location *right = translate_exp (get_child (exp, 2));
        Location *result = genloc ();
        generate_instruction (ADD, result, left, right);
        return result;
    }
    // ...
} // translate_exp

• Alternately, we can build code and address attributes in our parse tree.

exp
  : exp + exp
    { // ...
      $$ .code = concat ($1 .code, $3 .code);
      $$ .address = genloc () ;
      $$ .code .append (new_instruction (ADD, $$ .address, $1 .address, $3 .address));
    }

• Alternately, we can assume that code is generated as we go.

exp
  : exp + exp
    { // ...
      Location *loc = genloc () ;
      Location *left = get_p_attribute ($1 .attributes, "address" ) ;
      Location *right = get_p_attribute ($3 .attributes, "address" ) ;
      set_p_attribute ($$.attributes, "address", loc);
      generate_instruction (ADD, loc, left, right);
    }

• Version two: Each expression must put its result in a specific location.
  1. Decide on a location for the result of the first expression.
  2. Decide on a location for the result of the second expression.
  3. Translate the first expression, telling it to put its result in the first location.
  4. Translate the second expression, telling it to put its result in the second location.
  5. Add the two together, putting the result in the specified location.
In pseudo-code

```c
/**
 * Generate the code to evaluate the expression in exp and leave the
 * result in result.
 *
 * Assumes that it’s safe to modify result along the way.
 */
void translate_exp (ParseTree *exp, Location *result)
{
  if (is_constant (exp))
  {
    generate_instruction (MOV, result, get_attribute (exp, value));
  }
  else if (is_add_op (exp))
  {
    Location *right = genloc ();
    translate_exp (get_child (exp, 0), result);
    translate_exp (get_child (exp, 2), right);
    generate_instruction (ADD, result, result, right);
  }
  // ...
} // translate_exp
```

- Why would we choose one approach rather than the other? Which do you prefer?

**Translating a Variable Expression**

- Consider an expression of the form `id`, where `id` names a variable.
- Version one: Expressions return addresses
  - 1. Memory location: symbols.lookup(id).memloc; Code: none
- Version 2:
  - 1. Code: copy from symbols.lookup(id).memloc to result

**Translating a Constant Expression**

- Consider an expression of the form `constant` (e.g., a numeric constant)
- Version one: Expressions return addresses
  - Option a: Observation: Constants need locations, too
  - Option b: General "locations" include constants
- Version two: Expressions take addresses as parameters
  - Code: Store the constant into result location

**Reflection**

- Which of the two approaches ("generate your own location" or "put the result here") do you prefer?
- Why?
- If we see advantages to both, we can attempt a hybrid strategy.
You are allowed to specify where the result should go.
You can also say “wherever you’d like”.
So the code must return a location.

- If we’re using a parser generator, it’s much easier to use the first approach, because we can do it while parsing.
  - However, our optimization to use fewer memory locations will be more difficult.

### Translating Array Expressions

- Let’s assume for now that we are only dealing with one-dimensional arrays.
- Where in memory is $a[i]$?
  - It’s at location($a$) + $i$ * sizeof($a$element)
- So, we can just generate that.

```
array_ref :
  id _LBRACKET exp _RBRACKET
{
  $$\text{.location} = \text{genloc}();
  \text{generate_instruction} (\text{MOV, R0, CONSTANT(id.address)});
  \text{generate_instruction} (\text{ADD, R0, R0, exp.address})
}
```

- It’s slightly more complicated than that, because we’ll have to play a bit with the address.
- What do we do about multi-dimensional arrays? It’s a similar calculation, just a bit more complicated.
  - The big issue is how you decide to store arrays: row major or column major order.