COMS W4115
Programming Languages and Translators
Lecture 16: Translation of Statements
March 27, 2013

Lecture Outline

1. Logical rules of inference for type checking
2. Run-time storage organization
3. Translation of assignments
4. arrays
5. Boolean expressions
6. If-statements
7. While-statements

1. Logical Rules of Inference for Type Checking

- Type inference templates
  - We can specify the type inference rule "if expression `e1` has the type `int` and expression `e2` has the type `int`, then the expression `e1 + e2` has the type `int`" with a type inference template of the form

```
âŠ¢ e1: int     âŠ¢ e2: int
_____________________

 âŠ¢ e1 + e2: int
```

- The turnstile symbol âŠ¢ is read "it is provable that" so the template can be read as "if it is provable that `e1` has type `int` and it is provable that `e2` has type `int`, then it is provable that `e1 + e2` has type `int`."
- Templates of this form provide a compact way of expressing the type rules of a language.
- We say a type system is sound if whenever âŠ¢ *e*: *T* then *e* evaluates to a value of type *T*.
- We can apply the type-inference templates by making a bottom-up traversal of the AST. We determine the types of the leaves using information from the symbol table. We can then move up the tree determining the type of the interior nodes from the types of their children by applying the inference rule for the operator at a given interior node.
- Type environments
- A type environment is often needed to determine the type of a variable at a given node in the AST. A type environment is just a mapping from variables to types that is stored in the symbol table. The type environment for each node can be determined by making a top-down pass over the AST respecting the type scoping rules of the language.
- The type inference rules are augmented with the type environment information. For example, if `E` is a type environment, we modify the template to make type inferences within the context of `E`:

```
E âŠ¢ e1: int     E âŠ¢ e2: int
_____________________________

 E âŠ¢ e1 + e2: int
```

- Static type checking can be done by making a top-down pass to compute the type environment for each node followed by a bottom-up pass to check the types at each node.

2. Run-time Storage Organization

- Run-time memory layout
  - Typical memory layout

```
  (low address) Code
                Static data
                Runtime heap
                  â†“

                Free memory

                  â†‘
 (high address) Runtime stack
```

3. Translation of Assignments

- Here is Fig. 6.20 (p. 381), an SDTS generating three-address code for assignments:

```
S â†’ id = E ; { gen(top.get(id.lexeme) '=' E.addr); }
E â†’ E1 + E2  { E.addr = new Temp();
               gen(E.addr '=' E1.addr '+' E2.addr); }
 | - E1      { E.addr = new Temp();
               gen(E.addr '=' 'uminus' E1.addr); }
 | ( E1 )    { E.addr = E1.addr; }
 | id        { E.addr = top.get(id.lexeme); }
```

- Example. Here is the three-address code generated by this SDTS for the assignment statement: `a = b + -c;`

```
t1 = uminus c
t2 = b + t1
a = t2
```

4. Arrays

- Referencing a one-dimensional array
  - In C and Java, array elements are numbered `0, 1,..., n-1` for an array `A` with n elements.
  - Element `A[i]` begins in location `(base + i Ã— w)` where `base` is the relative address of the storage allocated for `A` and `w` is the width of each element.
- Common layouts for multidimensional arrays
  - Row-major order
  - Column-major order
- See Fig. 6.22 (p. 383) for an SDD generating three-address code for assignments with array references.
- Example: three-address code for the expression `c + a[i][j]` assuming the width of an integer is 4

```
t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a[t3]
t5 = c + t4
```

5. Boolean Expressions

- Boolean expressions are composed of boolean operators (&&, ||, !) applied to boolean variables, relational expressions, and other boolean expressions.
- Short-circuit evaluation: Some languages, such as C and Java, do not require an entire boolean expression to be evaluated.
  - Given `x && y`, if x is false, then we can conclude the entire expression is false without evaluating y.
  - Given `x || y`, if x is true, then we can conclude the entire expression is true without evaluating y.
- Numerical encoding
  - In C, the numerical value 0 represents false; a nonzero value represents true.
- Positional encoding
  - The value of a boolean expression can be represented by a position in three-address code, and the boolean operators can be translated into jumps.
  - The expression

```
if (x < 100 || x > 200 && x != y)
  x = 0;
```

can be translated into the following three-address instructions:

```
     if x < 100 goto L2
     ifFalse x > 200 goto L1
     ifFalse x != y goto L1
L2:  x = 0
L1:
```

6. Translation of If-statements

- Boolean expressions often appear in the context of flow-of-control statements such as:
  - If statements
  - If-else statement
- See Figs. 6.36 (p. 402) and 6.37 for SDDs translating these statements with booleans into three-address code.
- For the expression

```
if (x < 100) || x > 200 && x != y)
  x = 0;
```

these SDDs produce the following three-address instructions:

```
    if x < 100 goto L2
    goto L3
L3: if x > 200 goto L4
    goto L1
L4: if x != y goto L2
    goto L1
L2: x = 0
L1:
```

This code can be transformed into the code in Section 5 by eliminating the redundant goto and changing the directions of the tests in the second and third if-statements.

7. Translation of While-statements

- Consider the production `S â†’ while ( B ) S1` for while-statements. The shape of the code for implementing this production can take the form:

```
begin: // beginning of code for S
   code to evaluate B
   if B is true goto B.true
   if B is false goto B.false
B.true:
   code to evaluate S1
   goto begin
B.false:  // this is where control flow will go after executing S
```

- Here is an SDD for this translation (from Fig. 6.36, p. 402):

```
S â†’ while ( B ) S1 {
         begin = newlabel()
         B.true = newlabel()
         B.false = S.next
         S1.next = begin
         S.code = label(begin) || B.code ||
                  label(B.true) || S1.code ||
                  gen('goto' begin)
}
```

8. Practice Problems

1. Use the SDD of Fig. 6.22 (ALSU, p. 383) to translate the assignment `x = a[i][j] + b[i][j]`.
2. Add rules to the SDD in Fig. 6.36 (ALSU, p. 402) to translate do-while statements of the form:

   S â†’ **do** *S* **while** *B*

   Show the code your SDD would generate for the program

```
do
  do
```

```
    assign1
  while a < b
while c < d
```

9. Reading

- ALSU, Sections 6.4 - 6.8, 7.1

---

aho@cs.columbia.edu