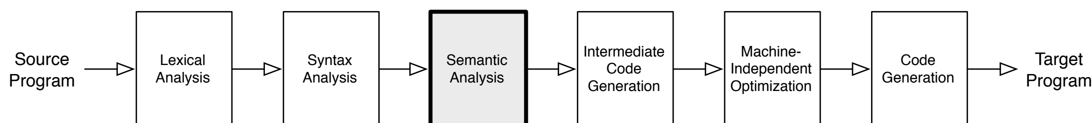
HW 5: Scopes and Type Systems

CSCI 434T
Fall, 2007

Overview



This week, we will examine the symbol table management required for the IC language and explore type systems as a foundation for describing and reasoning about type checking. The first reading is an article from the *Encyclopedia of Computer Science* on type systems written by Luca Cardelli, a very influential researcher. (Cardelli designed and implemented the first ML compiler and helped to develop much of the formal theory of object-oriented programming.) This article covers the basic theory and formalism used to describe type systems.

The third reading, from Kim Bruce's book, touches on many problems specific to designing and type checking object-oriented languages. It should be mostly background reading and will serve as a OOP refresher as you implement IC.

Readings

- "Type Systems," Luca Cardelli, 1997.
  1. *Sections 1–3 (stop after Table 9), and Section 9.*
  2. *Even though you will not need to use it in any detail, this paper does assume familiarity with the λ-calculus as a basic syntax for a programming language. You may wish to refer back to your CS 334 notes or Mitchell book for a quick refresher as you begin to read this. The Wikipedia entry for lambda calculus may also be sufficient.*
- Dragon, Ch 6.3–6.3.2, 6.5.

  Skim.
- *Foundations of Object-Oriented Languages*, Kim Bruce, 2002. Chapters 2–3.

  This is background reading — You can skim it quickly, or use it as a reference for OOP-specific items as the come up in later problem sets and the programming projects.

Exercises

1. This question applies the symbol table discussion from last week's problem set to write the design of the `symtab` package for your IC compiler. This package will contain two main classes: a `SymbolTable` class that encodes all symbol information for a scope, and a Visitor that will traverse the AST, constructing the symbol tables and annotating each AST node with the symbol table corresponding to its scope, as described in the PA 2 handout.

   Please be sure to download the latest version of the IC **Specification, which was modified to simplify some aspects of the scoping rules.**

   Your design need not be long or contain every detail, but it should address the following items:

- Define the general interface to the `SymbolTable` class. Describe what each method does, and sketch the internal representation details. Include a description of what information is stored along-side each symbol in the tables (ie, your `SymbolTable` is really a map from identifier to what?)
- Describe the operations performed by the scope-building visitor upon entry and exit to each form of scope (*e.g.* , global, class, method, block).
- How will the visitor create a new symbol entry when it encounters a declaration?
- When, how, and where will the scoping rules be enforced?
- Sketch the contents of your `SymbolTable` structure after processing the following program.

```
 1:    class A {
 2:      int x,u;
 3:      void f(int y, int z) {
 4:        int x;
 5:        y = x + this.x;
 6:        g(y-u);
 7:      }
 8:      void g(int y) { }
 9:    }
10:
11:    class B extends A {
12:      int a,b,c;
13:      void g(int y) {
14:        if (y > 0) {
15:          int a = 2;
16:          f(y, x);
17:        }
18:      }
19:    }
```

- For each occurrence of CLASSID and ID in the IC grammar, indicate whether or not it is a defining occurrence that introduces a new name into the current scope. For those that are not, indicate the scope in which the defining occurrence of that name would occur. Use this characterization to indicate how your compiler will resolve the following symbols:
  - the first `x` on line 5.
  - `this` on line 5.
  - the second `x` on line 5.
  - `u` on line 6.
  - `x` on line 16.
- In what ways, if any, will you change the AST package to support construction of the symbol tables?

2. Using the type system of Cardelli's paper, prove that

   (a) $\emptyset, x : Nat \vdash succ\ (succ\ x) : Nat$

   (b) $\emptyset, x : Nat \vdash if\ true\ then\ x\ else\ 0 : Nat$

3. For each of the following IC constructs, state whether it is well-typed in some well-formed typing context. If so, give the most general typing context in which the construct is well-typed and write the corresponding proof tree. If the construct is not well-typed in any type context, explain why. (Note that, unlike in the Cardelli system, you do not need to prove that environments and types are well-formed. You should be able to convince yourself that any type or environment that you mention is well-formed, however.)

(a) `(new int[x.length])[x[2]]`

(b) `if (x == v[x] && y == "true") x = y;`

(c) `((a == b) == c) && (a == (b + "c"))`

(d) `f(x)[x.length] = y[2]`

(e) `if (x == a[b[x]] && y) y = b[c[x]];`

4. This problem concerns the typing and the translation of `for` loop constructs.

   (a) Suppose we extend IC to support for loops that operate over ranges of integer values:

   ```
   for (x from e1 to e2) s
   ```

   where `x` is an integer field or local variable (that has been declared before), and `e1` , `e2` are the loop bounds. The loop body is executed for all values of `x` in the range from `e1` to `e2`. Write a typing rule that ensures the safe execution of this loop.

   (b) It is difficult to reason about for loops when the execution of the loop body might change the iteration variable or the loop bounds. Describe a semantic check that would ensure this never happens.

   (c) Suppose we further augment IC with extended for loops, in a fashion similar to those in Java 1.5. An extended for loop in IC will have the following form:

   ```
   for (T x : e) s
   ```

   Here, `x` is a newly declared local variable of type `T` that iterates over all of the elements of the array `e`, and `s` is the loop body. Write an appropriate typing rule for this construct.

5. Suppose we extend IC with tuples of the following form. A tuple type is written as a sequence of types in parentheses. For example, the type `(int,bool,string)` represents a 3-tuple. The individual elements of the tuple can be accessed (i.e., read or written) in a manner similar to array elements. For example, if `x` has type `(int,bool,string)`, the expression `x[0]` has type `int`, `x[1]` has type `bool`, and `x[2]` has type `string`. Tuples are unlike arrays in that the index expression must be a constant. For simplicity, we assume that tuples don't contain class types; this ensures that different tuples cannot be subtypes of each other.

   (a) Explain why is it necessary to require that the index of a type expression must be a constant.

   (b) Write additional typing rules in the static semantics of IC for expressions and statements to support tuples.

   (c) Consider the types `T1 =(int,(int,int)[])` and `T2 =(int,(int,int))[]`. Consider a variable `x` having either type `T1` or type `T2`. Write an expression which type-checks and has the same type in both cases; and an expression which type-checks if `x` has type `T1`, but doesn't if `x` has type `T2`. We require that `x`, 0, and 1 are the only variables and constants in your expressions.

   (d) Syntactically, the tuple element access expression looks like an array element access expression. Will this create problems for type checking? Explain briefly.

6. Consider a C-like language that manipulates pointers. Statements and expressions have the following syntax:

$$
\begin{array}{lcl}
e & \rightarrow & n \mid x \mid \&x \mid *e \\
s & \rightarrow & x = e \mid x = \texttt{malloc}() \mid *x = e
\end{array}
$$

   where n is an integer constant, x is a variable, and `malloc()` allocates an integer or a pointer on the heap (according to the declared type of `x`), and then returns a pointer to that piece of data. The only types are pointers and integers, but pointers can be multi-level pointers. The syntax for types is:

$$
T \rightarrow \texttt{int} \mid T*
$$

(a) Write typing rules for all of the assignment statements. Use judgments of the form $E \vdash s$ for statements, and judgments of the form $E \vdash e : T$ for expressions.

(b) Now let's extend the types in this language with two type qualifiers `taint` and `trust`. Tainted data represents data that the program received from external, untrusted sources, such reading from the standard input or reading from a network socket. All of the other data is `trust`ed. Perl and other languages use tainting to, for example, prevent certain forms of security attacks on web scripts.

To model tainting, we extend the set of statements with a `read()` statement that reads an untrusted integer value from an external source:

$$e \quad \rightarrow \quad ... \mid \texttt{read}()$$

The syntax for qualified types is:

$$\begin{array}{rcl} T & \rightarrow & Q\ R \\ R & \rightarrow & \texttt{int} \mid T\ * \\ Q & \rightarrow & \texttt{taint} \mid \texttt{trust} \end{array}$$

For instance, `trust ((taint int) *)` represents a trusted pointer to a tainted location, and `taint ((taint int) *)` denotes a tainted pointer to a tainted location.

Write appropriate typing rules for expressions n , `x`, $\&x$, $*e$, and `read()` for programs with qualified types. Also write a rule for `malloc`.

(c) We want to prohibit the flow of values from untrusted sources into trusted portions of the memory. However, we want to allow flows of values from trusted locations to tainted locations. We can achieve this by defining an appropriate subtyping relation $\leq$ between qualified types. Fist, we define an ordering between qualifiers:

$$Q \preceq Q' \text{ iff } Q = \texttt{trust} \text{ or } Q = Q'$$

We then use the subtyping rule:

$$\text{[SUBTYPE]} \quad \frac{Q \preceq Q'}{QR \leq Q'R}$$

along with the standard assignment rule in the presence of subtyping:

$$\text{[ASSIGN]} \quad \frac{E \vdash x : T \qquad E \vdash e : T' \qquad T' \leq T}{E \vdash x = e}$$

to enforce the desired control over trusted values. For instance, these rules would make it possible to type-check this code fragment:

```
taint int x;
trust ((trust int) *) y;
y = malloc();
x = *y;
```

Prove that the above program type-checks by showing the proof trees for each of the two assignments.

(d) Write the remaining rule for indirect assignments `*x = e`. Illustrate the use of this rule on a small program.

(e) Consider the following, more general subtyping rules:

$$\text{[SUBTYPE 1]} \quad \frac{Q \preceq Q'}{Q\texttt{int} \leq Q'\texttt{int}} \qquad\qquad \text{[SUBTYPE 2]} \quad \frac{Q \preceq Q' \qquad T \leq T'}{Q(T*) \leq Q'(T'*)}$$

Are these rules sound? If yes, argue why. If not, show a program fragment that type-checks, but yields a type error at run time.