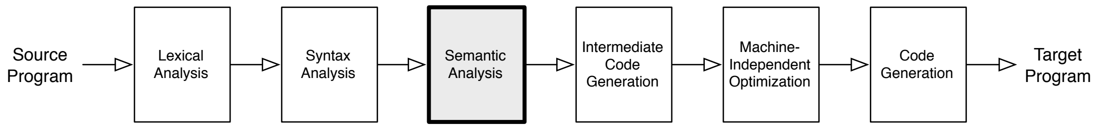
HW 6: Object Types, Subtyping, Type-Based Analysis

CSCI 434T
Fall, 2011

Overview



This week, we wrap up semantic analysis. The readings and first few problems cover subtyping and object types. I hope most of this material will be a quick review of ideas from 334, but it is important to visit them in the context of compiler design as well. The latter part examines several research papers on various uses of types.

We then explore several interesting program analyses described in terms of extended type systems.

Readings

- "Type Systems," Luca Cardelli, Section 6, pages 28–30. *(Variant types are basically enumerations.)*
- "Type-Based Race Detection for Java," Cormac Flanagan and Stephen Freund, *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2000.
- "Language-Based Information-Flow Security," Andrei Sabelfeld and Andrew C. Myers, *IEEE Journal on Selected Areas in Communications*, 2003.

Exercises

1. This question asks you to design the `tc` package for your IC compiler. This package will contain the code to perform the semantic checks outlined in the IC specification. The main class in the package will be a visitor whose job is to annotate each `Expr` node with a type, where `Expr` node is the abstract class from which all expression nodes are derived — your class may be called something different. You will need to extend that class with a `type` field to store the `Type` determined by the typechecker, as well as `setType` and `getType` accessor methods. (Here, I'm using `Type` to refer to the abstract class from which the AST classes representing types are derived — again, your class name may be different.)

   Please come to the tutorial meeting with a design detailed enough to discuss the following items:

   - Draw the AST for the expression `x + 3 == 7 || a[1] > -x` using your AST package. Annotate each node in the tree with the type corresponding to that expression, given the typing environment `E = int x, int[] a`.
   - The typing rules require you to determine whether one type is a subtype of another. How will you implement subtyping for your `Type` objects?
   - Sketch the implementations of the type checker's `visit` method for your AST node class corresponding to each of the following:
     - a unary expression ($!e$ or $-e$).
     - an array access
     - a variable access

    - a field access
    - an assignment statement
- Other than the changes described above, how will you change the `ast` package to support type checking?

2. (a) Cardelli (Section 1) discusses nominal and structural type equivalence, as do Cooper and Torczon (Chapter 4). What is the difference? Which does Java (and IC) use? Both authors describe tradeoffs between them. Do you agree with them? Which is better? Which issues should you worry about?

   (b) Java arrays use the following subtyping rule:

$$\frac{A \leq B}{A[\,] \leq B[\,]}$$

   Demonstrate why this rule causes the type system to be unsound by writing a short program that would cause a run-time type error when executed.

   (c) Suppose I overload a method in a subclass and change the method in the following ways. Which may cause problems at run time if permitted by the type checker? Which would be okay? Explain in a sentence or two, appealing to basic subtyping principles.

   - I change the method's parameter from `String` to `Object` in the subclass.
   - I change the method's return type from `String` to `Object` in the subclass.
   - I change the method's visibility from `private` to `public` in the subclass.
   - I change the method's visibility from `public` to `private` in the subclass.

3. Java's ternary expression $e_1 ?\ e_2 :\ e_3$ evaluates to e_2 if e_1 is true, and e_3 if e_1 is false.

   (a) Are the following expressions and statements well-typed, in the sense that no type error could occur as a result of using them in a program? (Assume that `B` is declared to extend `A`.) For each well-typed ternary expression, what is the most precise type possible? Assume `b` is a boolean variable.

   i. `b ? 10 : 20`
   ii. `b ? 10 : true`
   iii. `A x = b ? new B() : new A()`
   iv. `B x = b ? new A() : new B()`
   v. `b ? null : new A()`

   (b) Extend the IC type system to include this construct. Be sure to assign the most precise type possible to a ternary expression, since this will allow the expression to be used in the most contexts (*e.g.*, `b ? null : null` could be given type `A`, `B`, or `Null`, but the third is the most precise type since `A` and `B` are both supertypes of `Null`.) Show the derivation for (iii) to illustrate how the rule works.

4. Suppose we add interfaces to IC. An interface is declared as illustrated below:

```
interface Moveable {
  void move(int dx, int dy);
}

interface Resizable {
  void resize(int dx, int dy);
}
```

Classes can implement one or more interface:

```
class Rectangle extends Shape implements Moveable, Resizable {
  void move(int dx, int dy) { ... }
  void resize(int dx, int dy) { ... }
}

class Circle extends Shape implements Moveable, Resizable {
  void move(int dx, int dy) { ... }
  void resize(int dx, int dy) { ... }
}
```

We can then declare variables to have interface types in the usual way:

```
Rectangle r = new Rectangle();
Moveable m = r;
m.move(10,10);
...
Resizable rs = r;
rs.resize(-10,2);
```

One interface can also extend another interface, in which case the subinterface "inherits" all methods listed in the superinterface:

```
interface HideableMoveable extends Moveable {
  void hide();
}
...
HideableMoveable hm = ...;
hm.move(10,10);
hm.hide();
```

Describe the semantic checks you would need to add to your IC compiler to ensure that interfaces are used correctly. In particular:

(a) Extend the subtyping rules on page 3 of the IC specification to include interfaces. You may use the letters `I`, `J`, and `K` to denote interfaces, so that you can distinguish an interface names from class names.

(b) Describe the semantic checks one must perform for each interface declaration and each class that implements an interface. How would you need to change the `ast` and `symtab` packages to support interfaces? (A few sentences is sufficient.)

(c) Consider the ternary operator again. Suppose we have the following declarations:

```
boolean b;
Shape s;
Rectangle r;
Circle c;
Moveable m;
Resizable z;
```

If we used the following in a program without typechecking them, which could lead to a type error at run time?

i. `s = b ? s : r`
ii. `m = b ? r : m`
iii. `c = b ? r : c`
iv. `s = b ? r : c`

v. `m = b ? s : c`

(d) Do your rules for ternary expressions from the previous question enable you to check each of these assignments properly? Are there other assignment expressions that your rules could not handle as well? If so, what are the issues and how can you handle them? (A few sentences is sufficient.)

5. [From last week] Read "Type-Based Race Detection for Java" by Flanagan and Freund.

   - The system extends the Java syntax with three items:
     - `guarded_by` modifiers on fields;
     - `requires` modifiers on methods; and
     - classes parameterized by locks.

     What does each syntactic form capture? Give an examples of where each is useful.
   - What guarantee does the type system make about well-typed programs?
   - The main typing judgment is $P; E; ls \vdash e : t$. This is a fairly standard judgment form, except for the inclusion of ls. What is ls? How is it computed, and how is it used?
   - Can you identify any limitations of the approach taken in the paper? (You may wish to think back to some of the issues raised when trying to formulate the type rules for the `for` constructs last week for at least one issue.)

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 $\texttt{malloc}()$ 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 $\texttt{read}()$ statement that reads an untrusted integer value from an external source:

$$e \rightarrow \ldots \mid \texttt{read}()$$

The syntax for qualified types is:

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

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 $\texttt{read}()$ 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:

$$\frac{Q \preceq Q'}{QR \leq Q'R}\ [\textsc{Subtype}]$$

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

$$\frac{E \vdash x : T \qquad E \vdash e : T' \qquad T' \leq T}{E \vdash x = e}\ [\textsc{Assign}]$$

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:

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

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.

7. The Sabelfeld and Myers paper covers the general issue of security and information flow and discusses a number of current research issues regarding how to ensure that confidential information does not accidently leak out of a computation. Please read that paper. Sections I–III are perhaps the most relevant. What does non-interference mean in a security setting and how is it defined? What are the key ideas behind the type system of Section III? Show how to type check the valid and programs at the end of IIIB, and explain why the invalid ones fail to check. What do the authors identify as the open issues in this area of research?