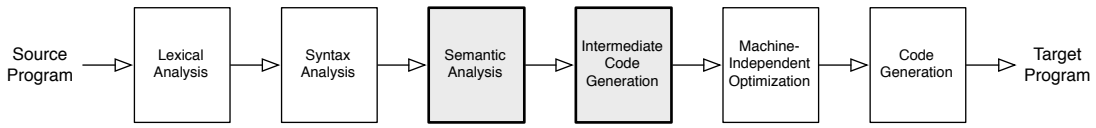
HW 6: Object Types, Subtyping, and TAC

CSCI 434T
Fall, 2007

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 second half of the problem set explores "Intermediate Code Generation", the phase of the compiler that converts our high-level AST representation into a low-level TAC representation. This lowering makes our representation closer to a realistic machine model and will greatly facilitate the generation of assembly code and optimization. After exploring this translation step, we wrap up the week's material by thinking about how objects are represented at run time, focusing on data layout and dynamic dispatch. The last two questions explore how we can optimize dynamic dispatch using semantic information about the class hierarchy. Many optimizations will require building and manipulating control-flow graphs, but this particular one is a great example of a semantics-driven optimization, where analyzing a program's types will enable the compiler to generate more efficient code.

Next week, we will examine the full details of the run-time environment for IC, which describes memory storage organization, calling conventions, and the particulars of the machine to which we will compile IC programs, namely the x86 processor.

There are several optional research papers on the reading list this week. They are both quite interesting and I encourage you to read them and think about the related questions below.

Note: We will discuss this material next Wednesday, Oct. 17, during our meetings. We will also have meetings on Monday, Oct. 22. That time will be spent on any remaining questions from this material, as well as a few additional questions that I may assign next week. The final versions of all questions will be due Wednesday, Oct. 24.

Readings

- "Type Systems," Luca Cardelli, Section 6, pages 28–30. *(Variant types are basically enumerations.)*
- Dragon 6.2 – 6.2.1. *(Skim)*
- Appel, Chapter 14.
- "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis," Jeffrey Dean, David Grove, and Craig Chambers, ECOOP 1995. *(optional, on web)*
- "Language-Based Information-Flow Security," Andrei Sabelfeld and Andrew C. Myers, *IEEE Journal on Selected Areas in Communications*, 2003. *(optional, on web)*

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. Since we didn't have a chance to discuss it last week, please come prepared to look at #6 from HW 5.

(Optional) 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. If you're interested, 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 are the open issues in this area of research?

6. This problem explores how to translate a program represented as an AST into TAC, the lower-level representation that we will then translate into x86 assembly code. The specification of the TAC instruction set for this question is on the web site. As an example, consider the following while loop and its translation:

```
n = 0;
while (n < 10) {
  n = n + 1;
}
```

```
n = 0
label test
t1 = n < 10
t2 = not t1
cjump t2 end
label body
n = n + 1
jump test
label end
```

We would like to develop an automatic way to generate TAC from a program. The automatic method will be a syntax-directed translation, meaning that we will describe it as a function that takes as input a syntactic form from the source language and returns a sequence of TAC instructions that is semantically equivalent. Note that the operands of each TAC instruction are either program variable names (ie, `n`), temporary variable names introduced during translation

(`t2`, `t3`), or constants (`0`, `10`, `1`). Branch instructions refer to label names generated during the translation.

More precisely, we will define a function T such that $T[s]$ is the low-level TAC representation for the high-level statement s. If e is an expression, we denote by the

$$t := T[e]$$

the low-level TAC instructions to compute e and store the result value in the variable `t`.

Expressions with subexpressions will be translated by recursively translating the subexpressions and then generating the code to combine their results appropriately. To make this concrete, suppose e is $e_1 + e_2$, then $t := T[e]$ would be:

```
t1 := T[e1]
t2 := T[e2]
t = t1 + t2
```

where the first two lines recursively translate e_1 and e_2 and store the results of those expressions in new temporary variables $t1$ and $t2$, which are then added together and stored in t. Here are a few other general cases:

e	`t :=` $T[e]$
`v`	`t := v` (variable)
`n`	`t := n` (integer)
`e.f`	`t1 :=` $T[e]$ (field access) `t = t1.f`
e_1`[`e_2`] =` e_3	`t1 :=` $T[e_1]$ (array assignment) `t2 :=` $T[e_2]$ `t3 :=` $T[e_3]$ `t1[t2] := t3`

Note that I generate new temporary names whenever necessary. For more complex expressions, we simply recursively apply rules: `T[ a[i] = x * y + 1 ]` becomes the following:

```
t1 := T[a]                t1 = a              t1 = a              t1 = a
t2 := T[i]                t2 = i              t2 = i              t2 = i
t3 := T[x * y + 1]        t4 := T[x * y]      t6 := T[x]          t6 = x
                     ≡                    ≡   t7 := T[y]      ≡   t7 = y
                                              t4 = t6 * t7        t4 = t6 * t7
                          t5 := T[1]          t5 = 1              t5 = 1
                          t3 = t4 + t1        t3 = t4 + t1        t3 = t4 + t1
```

The translation scheme for statements follows the same pattern: T`[ while e s ]`, for example, is

```
label test
t1 := T[e]
t2 = not t1
cjump t2 end
T[s]
jump test
label end
```

(a) Define T for the following syntactic forms:

- `t :=` $T[e_1$ `*` $e_2]$

- `t := T[`e_1 `||` e_2`]` (where `||` is short-circuit)
- `t := T[`e_1 `&&` e_2`]` (where `&&` is short-circuit)
- `T[ if` e `then` s_1 `else` s_2 `]`
- `T[` s_1`;` s_2`; ...;` s_n `]`
- `t := T[ f(`e_1`, ...,` e_n`) ]`

(b) Note that these translation rules introduce more copy instructions than strictly necessary. For example, `t4 := T[ x * y ]` becomes

```
t6 = x
t7 = y
t4 = t6 * t7
```

instead of the single statement

```
t4 = x * y
```

Describe how you would change your translation function to avoid generating these unnecessary copy statements.

(c) The original rules also use more unique temporaries than required, even after changing them to avoid the unnecessary copy instructions. For example,

`T[ x = x*x+1; y = y*y-z*z; z = (x+y+w)*(y+z+w)]`

becomes the following:

```
t1 = x * x
t2 = t1 + 1
x = t2
t3 = y * y
t4 = z * z
y = t3 - t4
t5 = x + y
t6 = t5 + w
t7 = y + z
t8 = t7 + w
z = t6 * t8
```

Rewrite this to use as few temporaries as possible. Generalizing from this example, how would you change T to avoid using more temporaries than necessary.

This seems like a good idea, but can you think of any reasons why you may prefer the original translation scheme that uses more temporaries?

(d) **(Optional)** Translation of some constructs, such as nested `if` statements, `while` loops, short-circuit and/or statements may generate adjacent labels in the TAC. This is less than ideal, since the labels are clearly redundant and only one of them is needed. Illustrate an example where this occurs, and devise a scheme for generating TAC that does not generate consecutive labels.

7. Appel, 14.3.

8. Appel, 14.4.

9. **(Optional)** The Chambers paper introduced many of the ideas behind the previous two questions. Read that paper and ponder the following:

   - When will CHA be most effective in a program? When will it be less effective?
   - What are the main limitations of CHA?
   - Section 2.2.3 discusses how CHA can be used for dynamically-typed languages like Smalltalk. Can you think of how the techniques outlined for Smalltalk could be used for Java?

- Compare this approach for dynamically-typed languages to the one outlined in "Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches" (on the web page). What is the same? What is different?