CS 164, Spring 2010 **CS 164: Homework #6** **P. N. Hilfinger**

Due: Tuesday, 30 March 2010

1. The Algol 68 language introduced an expression called the *case conformity clause*. Here's one version of it:

`case` I `:=` E_0 `in` T_1`:` E_1`;` T_2`:` E_2`; ...;` T_n`:` E_n`; esac`

where the E_i are expressions (i.e., with values), I is an identifier, and the T_i are types. The idea here is that the program first evaluates E_0, and assigns I its value. If the dynamic type of I is T_i for some i (or a subtype of T_i), the program evaluates E_i and yields its value as the value of the entire clause (it will be a run-time error if no clauses match). If more than one T_i fits, the program chooses one arbitrarily and evaluates it (the expression must type properly regardless of which choice is made). The problem is come up with a static typing rule for this expression. Assume that the AST for the case conformity clause above is represented in Prolog notation as

`case_conform(`$\hat{I}$`,`$\hat{E_0}$`,[case(`$\hat{T_1}$`,`$\hat{E_1}$`),...,case(`$\hat{T_n}$`,`$\hat{E_n}$`)])`

where $\hat{x}$ is the AST for x. So the problem is to find an appropriate replacement for '??' in

```
typeof(case_conform(I,E0,Clauses), T, Env) :- ??
```

The implication here is that all the clauses have to produce values of some common type, `T`. There is no need to know the rest of this language to do this.

Produce two Prolog version of this rule under the following alternative assumptions:

> **Version 1:** Require that all the T_i be subtypes of the static type of E_0.
>
> **Version 2:** Require that at least one clause have a T_i that is a subtype of E_0's type, but allow E_i in clauses for which T_i is *not* a subtype of E_0's type to have any type at all.

So, under version 2, it's OK to write

```
N + case x := head(shapeList) in
       Rectangle : width(x);
       Circle    : radius(x);
       Elephant  : "Hi, there!"
    esac
```

assuming that `shapeList` is a list of `Shapes`, types `Rectangle` and `Circle` are subtypes of `Shape`, and `Elephant` is not. Even though the type of the `Elephant` case is presumably incompatible with that of the other two (presumably numeric), we can ignore it, since the last case can never be taken. Under version 1, however, this **case** expression is illegal.

See the skeletons in `hw6/case_conform1.pl` and `hw6/case_conform2.pl`. To do version 2, you'll need some "impure" Prolog. The following pair of rules (in the order given) define `notok(X)` to succeed if `ok(X)` cannot be satisfied:

```
notok(X) :- ok(X), !, fail.
notok(_).
```

The *cut symbol,* '`!`', basically says, "always succeed, but if you ever have to backtrack past this point, give up on the goal `notok` immediately and don't try any other rules for `notok`." So once you find that `ok(foo)` is satisfiable, you hit the cut symbol and then immediately fail (the goal `fail` has no rules, so that it always fails).

2. In Java, the following is legal:

```
String[] Y;
Object[] X;
...
X = Y;
```

That is, an array of T_1 may be assigned to a variable of type array-of-T_2 as long as T_1 is a subtype of T_2. As it turns out, this rule is unsound in the sense that because of it, certain type errors can only be discovered at execution time, requiring a (somewhat) expensive check that slows down some operations. Give an example of how this can happen (by which I mean an actual Java program).

3. Write a legal Python program that simply prints "`static`" and that would also be legal if Python used dynamic scoping, but would print "`dynamic`" instead.

4. Show how the type rules from slide 15 of Lecture 13 work to determine the types of `Y`, `g`, and `fact` in

```
def Y f = f (Y f)
def g h x = if x = 0 then 1 else h(x-1) * x fi
def fact x = Y g x
```

Assume that '`-`' and '`*`' obey the same rules as '`+`'. (Aside: for obvious reasons, `Y`, the "paradoxical combinator," won't actually work unless this language uses normal-order evaluation, in which expressions are not evaluated until their value is actually used in a primitive operation. However, evaluation is not the point here.)

5. In the project, we don't deal with method overloading, so let's take a shot in the homework. The Python file `overload.py` contains a skeleton that defines a simple AST with two types of node:

Leaf nodes, labeled with an identifier string that names a type.

Call nodes, containing labeled with a function identifier and having 0 or more children (each an AST).

It also defines a type `Signature`, which stands for the type of a function (that is, its argument types and its return type). We'll define an environment as a dictionary mapping function names to lists of function signatures (thus representing sets of overloadings of a given function name).

The idea is to figure out the particular signature to choose for each of the function names in call nodes so as to make all signatures match the argument types.

Fill in the two functions:

resolve1(T, Env): As in Java or C++, require that each signature be selected unambiguously using only the types of the arguments. In other words, given a call such as `f(g(Int))`, the type of `g(Int)` must be determined unambiguosly without reference to the fact that its result will be an argument to `f`.

resolve2(T, Env): As in the Ada language, find signatures for all functions so that the entire AST matches, In other words, given a call such as `f(g(Int))`, its OK to have two possible overloadings of `g` that take `Int` arguments, as long as they have different return types and only one return type fits an overloading of `f`.