CS 164, Fall 2020 **CS 164: Homework #5** **P. N. Hilfinger**

Due: Friday, 23 October 2020

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., having values), I is an identifier, and the T_i are subtypes of E_0's static type. 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 to come up with a static typing rule for this expression.

For example, one might write

```
N + case x := head(shapeList) in
        Rectangle : width(x);
        Circle    : radius(x);
    esac
```

assuming that `shapeList` is a list of `Shapes` and types `Rectangle` and `Circle` are subtypes of `Shape`. The static type ascribed to `x` differs in the two clauses. For example, `width` might be defined only on `Rectangle` and not on `Shape`. The construct is type-safe because the language guarantees that we never execute the `Rectangle` branch unless `x` is a `Rectangle`, so that we are justified in giving `x` the more specific static type `Rectangle` in that clause only. On the other hand, it would be illegal to write

```
N + case x := head(shapeList) in
        Rectangle : width(x);
        Circle    : radius(x);
        Elephant  : 0;
    esac
```

if `Elephant` is not a subtype of `Shape`. Finally, the type of the case conformity clause is the smallest supertype of all the exressions E_i, $i \geq 1$.

Fill in the skeleton file `case_conformity.py` to provide a typing function for these expressions. The file `case_conformity_check.py` provides Python routines for assisting with writing type-checking rules. There is no need to know the rest of this language to do this.

2. Show how the type rules from slide 18 of Lecture 12 work to determine the type of `r` in

```
def r(p, i, L) = if i == [] then i else r p (p i (hd L)) (tl L) fi
```

That is, show the type equations that result from applying the rules from that slide, the types of the subexpressions, and the solution to the equations. Use `'r, 'p, 'i,` and `'L` as the type variables representing the types of the corresponding identifiers. Put your response in the skeleton file `hw5.txt`.

3. We've touched on overloading in lecture, but not gone into any depth, 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, 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 that maps 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))`, it's 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`.