CS 164, Spring 2008 **CS 164: Homework #5** **P. N. Hilfinger**

Due: Friday, 20 March 2009

See the files in Unless the problem specifies otherwise, please put your solutions in a file named `hw5.txt` in your `hw5` subdirectory.

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. 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 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), T0, Env) :- ??
```

The implication here is that all the clauses have to produce values of some common type, `T0`. There is no need to know the rest of this language to do this. See the skeleton in `hw5/case_conform.pl`.

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).

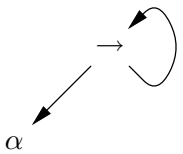
3. Write a legal Python program that simply prints "`static`" and that would also be legal if Python used dynamic scoping, but would print "`dynamic`" instead.

4. Show how the type rules from slide 11 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. The unification algorithm from class does not handle cases such as unifying $\alpha \rightarrow \beta$ against $\alpha \rightarrow \alpha \rightarrow \beta$. To do so would require unifying β against a term that contains itself, so that β would have to have a non-tree-like structure like this:



Still, this kind of type does occur and can be useful. A revised algorithm handles it. In this algorithm, we extend `binding(`P`)` function so that it applies to *all* terms, not just type variables. Initially, all terms (variables or not) are bound to themselves. The abstract match function becomes:

```
match(P, A) {
    while (P != binding(P))
        P = binding(P);
    while (A != binding(A))
        A = binding(A);
    if (P == A)    /* Pointer equality here */
        return true;
    if (P is a type variable)
        binding(P) = A;
        return true;
    }
    if (A is type variable)
        binding(A) = P;
        return true;
    }
    binding(P) = A;
    if (oper(P) != oper(A)
        || length(children(P)) != length(children(A)))
        return false;
    for each child cp in children(P)
```

```
                and corresponding child ca in children(A)
            if (! match (cp, ca))
                return false;
        return true;
    }
```

If `match(P,A)` succeeds, and we replace all nodes in either `P` or `A` with their bindings (following them recursively until we get to unbound terms, as in the program above), we'll get the same structure. Unlike the algorithm in lecture, however, this structure might be a general (even circular) graph. Fill in the skeleton `unify.py` in the `hw5` directory to implement this algorithm, and perform (at least) the simple test indicated. (There isn't much to this problem really, aside from a lot of useful reading and understanding of Python code.)