# Assignment 7:
# Subtyping and Objects

15-312: Foundations of Programming Languages
Jason Reed (jcreed+@cs.cmu.edu)

Out: Thursday, November 4, 2004
Due: Thursday, November 11, 2004 (1:30 pm)

50 points total

## 1 Subtyping products (15 points)

In this problem we explore subtyping for products. We assume we have a subtyping judgment with reflexivity, transitivity, and the primitive coercion $\mathsf{itof} : \mathsf{int} \leq \mathsf{float}$. As discussed in lecture, subtyping for products $\tau_1 \times \tau_2$ is co-variant in both $\tau_1$ and $\tau_2$.

Extending the subtyping for products any further is fraught with difficulties. One attempt is based on the observation that wherever a value of type $\tau_1$ is expected we can supply instead of value of $\tau_1 \times \tau_2$. Similarly, wherever a value of type $\tau_2$ is expected we can supply a value of type $\tau_1 \times \tau_2$. The corresponding coercions would be the first and second projections. That is, we would have

$$\frac{}{\lambda x.\mathsf{fst}(x) : \tau_1 \times \tau_2 \leq \tau_1}\ \mathit{proj}_1 \qquad \frac{}{\lambda x.\mathsf{snd}(x) : \tau_1 \times \tau_2 \leq \tau_2}\ \mathit{proj}_2$$

1. (5 pts) Show with a concrete counterexample that the system with the two rules above is not *coherent*. You should exhibit two types $\tau$ and $\sigma$ and two *different* coercions $f : \tau \leq \sigma$ and $g : \tau \leq \sigma$.

2. (5 pts) Suppose that in reaction to the problem with coherence from part (1) we reject the second subtyping rule ($\mathit{proj}_2$), allowing only the first ($\mathit{proj}_1$). Unfortunately, this does not solve the problem. Exhibit a counterexample to coherence in using only rule $\mathit{proj}_1$ (and the usual rules of subtyping).

3. (5 pts) A pair $\mathsf{pair}(e_1, e_2)$ can be represented by the record $\{1{=}e_1, 2{=}e_2\}$. Then the first and second projection are defined by $\mathsf{fst}(e) = \#1\ e$ and $\mathsf{snd}(e) = \#2\ e$, respectively. Relate depth and width subtyping to the subtyping rules for pairs from Problem 1 and explain why coherence is not violated here.

## 2 Casts in EML (10 points)

Consider the following declarations:

```
module
 class B of ...
 class C extends B of ...
 method foo(C) : int
end
```

The EML language does not have a primitive cast operation. However, in practice many object-oriented programs need a way to find out if an object of static type class B is really an instance of class C, so that they can call functions like foo that are only defined on C.

Note: for the question below, it may be helpful to assume that function cases in EML can accept arbitrary patterns, not just a tuple of the form $(x_1 : C_1, \ldots, x_n : C_n)$.

1. (10 pts) Write a function called `ifC` in EML that has type `B -> (C -> int) -> int option`. If the first argument is really an instance of class C, `ifC` should call the function passed in as the second argument with the first argument as a parameter and return `SOME(i)` where `i` is the result of the function call; otherwise `ifC` should return `NONE`.

## 3 Multiple Inheritance (15 points)

Assume we extend the EML language so that it provides multiple inheritance. Function declarations and function cases are unchanged, as is the semantics for function case lookup. Class declarations now take a list of classes in the extends clause:

$$classdecl \quad ::= \quad [\texttt{abstract}]\ \texttt{class}\ C\ [\texttt{extends}\ C_1, \ldots, C_n]\ \texttt{of}\ \{\overline{l} : \overline{\tau}\}$$

Assume that some program has a structure of the form:

```
module
    class Rectangle
        of { upperLeft:Point, lowerRight:Point }
    class BorderedRectangle extends Rectangle
        of { border:Color }
    class FilledRectangle extends Rectangle
        of { fill:Color }
end
```

Now assume that two programmers, working independently, create the following structures and add them to the program:

```
module
    method draw (Shape) : unit
    extend draw (x : Rectangle) = ...
    extend draw (x : BorderedRectangle) = ...
    extend draw (x : FilledRectangle) = ...
end
module
    class BorderedFilledRectangle
        extends BorderedRectangle,FilledRectangle
        of { }
end
```

The intuitive semantics of multiple inheritance is that `BorderedFilledRectangle` will inherit fields from both of its superclasses, and a function case that applies to a superclass will also apply to `BorderedFilledRectangle`.

1. (5 pts) Consider the global typechecking algorithm discussed in class, Does this algorithm give the right answer in the case of multiple inheritance? What errors, if any, will it report when run on the code above?

2. (5 pts) C++ provides two semantics for multiple inheritance: virtual and non-virtual. In non-virtual inheritance, `BorderedFilledRectangle` would have two copies each of the `upperLeft` and `lowerRight` fields–one inherited from `Rectangle` through `BorderedRectangle`, and one inherited from `Rectangle` through `FilledRectangle`. In virtual inheritance, `BorderedFilledRectangle` would have only one such copy.

   The distinction between virtual and non-virtual inheritance applies only to the case when fields are inherited from the same superclass along different inheritance paths; in all other cases the semantics of virtual and non-virtual inheritance are identical.

   Which is the most reasonable semantics in the example above: non-virtual or virtual inheritance? Explain your answer.

3. (5 pts) Is the semantics (virtual or non-virtual) you chose above always the right one, or are there reasonable examples where you need the other semantics? If there is a reasonable example, show one. If not, argue why there are no such reasonable examples. **[Hint: think "is a" vs. "has a"]**

## 4 Monads (10 points)

Recall the rules that we gave for monads in MinML:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e \div \tau}$$

$$\frac{\Gamma \vdash m \div \tau}{\Gamma \vdash \mathsf{comp}(m) : \bigcirc\tau} \qquad \frac{}{\mathsf{comp}(m)\ value}$$

$$\frac{\Gamma \vdash e : \bigcirc\tau \quad \Gamma, x{:}\tau \vdash m \div \sigma}{\Gamma \vdash \mathsf{letcomp}(e, x.\, m) \div \sigma}$$

1. (2 pts) Write a MinML function $inject$ of type $\bigcirc\tau \to \bigcirc\bigcirc\tau$.
2. (2 pts) Write a MinML function $extract$ of type $\bigcirc\bigcirc\tau \to \bigcirc\tau$.
3. (5 pts) Show that these are inverses of one another, in other words, that $inject(extract(v)) \mapsto^* v$ and $extract(inject(v)) \mapsto^* v$.