Assignment 7: Subtyping and Objects

15-312: Foundations of Programming Languages
Daniel Spoonhower (`spoons@cs.cmu.edu`)

Out: Thursday, November 6, 2003
Due: Thursday, November 13, 2003 (1:30 pm)

50 points total

1 Subtyping products (10 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 τ_1 and τ_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 τ_1 is expected we can supply instead of value of $\tau_1 \times \tau_2$. Similarly, wherever a value of type τ_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 τ and σ 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 (proj_2), allowing only the first (proj_1). Unfortunately, this does not solve the problem. Exhibit a counterexample to coherence in using only rule proj_1 (and the usual rules of subtyping).

2 Subtyping records (15 points)

Another approach to aggregate values that can be used instead of products is records. For that, we extend the type system by *record types* that we denote by ρ; we use l to denote record labels (not to be confused with memory locations).

$$\begin{array}{rrcl} \text{Types} & \tau & ::= & \ldots \mid \{\rho\} \\ \text{Record Types} & \rho & ::= & \cdot \mid l{:}\tau, \rho \end{array}$$

We extend expressions to allow the formation of records, denoted by r, and also the selection of a field from a record, written as $e.l$ for a record label l.

$$\begin{array}{rrcl} \text{Expressions} & e & ::= & \ldots \mid \{r\} \mid e.l \\ \text{Records} & r & ::= & \cdot \mid l{=}e, r \end{array}$$

We assume that any label occurs at most once within a record or record type. We sometimes use parentheses to enclose record types or record so the scope of the ',' is more clearly visible. Such parentheses are not properly part of the syntax of the language. We have a new typing judgment $r : \rho$, used in the following rules.

$$\frac{\Gamma \vdash r : \rho}{\Gamma \vdash \{r\} : \{\rho\}} \qquad \frac{\Gamma \vdash e : \{\rho\} \quad \rho = \rho_1, l{:}\tau, \rho_2}{\Gamma \vdash e.l : \tau}$$

$$\frac{}{\Gamma \vdash (\cdot) : (\cdot)} \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash r : \rho}{\Gamma \vdash (l{=}e, r) : (l{:}\tau, \rho)}$$

Note that the field selection operation $e.l$ will always yield a unique answer on well-typed records. This is because labels in a record must be unique.

In this notation, the empty record expression corresponds to the unit type. Note that there is a minor ambiguity in that the empty record and its type are both denoted by '$\cdot$', that is, $\vdash \{\cdot\} : \{\cdot\}$. As usual, we omit a leading '$\cdot$' in a record.

We do not give the formal operational semantics, but the elements of a record should be evaluated from left to right, eventually yielding a record all of whose fields are values. On values, the order of the fields in a record are not significant, so we consider $\{l_1{:}\tau_1, l_2{:}\tau_2\} = \{l_2{:}\tau_2, l_1{:}\tau_1\}$.

In the remainder of the this problem you will be asked to design subtyping rules for records validating certain principles. Your rules do not need to show coercions; we are only interested in pure subtyping.

1. (5 pts) *Depth subtyping* expresses that subtyping is co-variant in all the fields of a record. Formalize depth subtyping with one or more rules.

2. (5 pts) *Width subtyping* expresses that we can forget extraneous fields. Formalize width subtyping with one of more rules.

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

3 Casts in EML (10 points)

Consider the following declarations:

```
class B of ...
class C extends B of ...
fun foo : C -> int;
```

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 as C_1,...,x_n as 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`.

4 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:

$$\mathit{classdecl} \;::=\; [\texttt{abstract}]\ \texttt{class}\ C\ [\texttt{extends}\ C^*]$$
$$\texttt{of}\ \{\bar{l} : \bar{\tau}\}$$

Assume that some program has a structure of the form:

```
structure Shapes = struct
    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:

```
structure Draw = struct
    fun draw : Shapes.Shape -> unit;
    extend fun draw (x as Rectangle) = ...
    extend fun draw (x as BorderedRectangle) = ...
    extend fun draw (x as FilledRectangle) = ...
end
structure ExtendedShapes = struct
    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, as well as the modular version of the EML typechecking algorithm.

Do each of these algorithms give the right answer in the case of multiple inheritance? What errors, if any, will each of these algorithms 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"]**