SOLUTION NOTES – Semantics of Programming Languages 2005 – Paper 6 Question 7 (PMS)

This question is based on the material in Chapter 8 of the notes.

(a) The typing rule for subsumption and the typing and reduction rules for downcasts $(T)e$ should be given, together with examples using records. Subsumption requires no runtime checks and guarantees that no record field accesses will be stuck, whereas downcasts require a runtime type check (which may fail) at the downcast point to establish the same guarantee for later computation. There is no need to give the record subtype rules.

(b) The subtype rule should be stated with examples illustrating the contravariant and covariant premises.

(c) The standard encoding of objects as records of methods should be explained, with record subtyping expressing structural object subtyping. Abstracting these records on object state records permits class-style reuse of method definitions.

a) For subsumption, there is a type rule

$$\frac{\Gamma \vdash e : T \quad T <: T'}{\Gamma \vdash e : T'}$$

allowing part of the structure of e to be disregarded. For example, one could ~~type~~ derive

$$\frac{\vdash \{x = 3, y = \text{false}\} : \{x : \text{int}, y : \text{bool}\} \qquad \{x : \text{int}, y : \text{bool}\} <: \{x : \text{int}\}}{\vdash \{x = 3, y : \text{false}\} : \{x : \text{int}\}}$$

There is no term expression form for subsumption here, and no runtime check is required – static typing guarantees that field accesses from records will always succeed.

Down-casting, on the other hand, has an expression form

$$e ::= \ldots \mid (T)e$$

with ~~semantic rules~~ type rule

$$\frac{\Gamma \vdash e : T'}{\Gamma \vdash (T)e : T}$$

allowing $(T)e$ to be of type T, statically, irrespective of the type of e, and operational semantics

$$\frac{e \longrightarrow e'}{(T)e \longrightarrow (T)e'} \qquad (T)v \longrightarrow v \text{ if } \vdash v : T$$

$\searrow$ error otherwise

in which a runtime check that the resulting value dynamically has type T is required. That check suffices to ensure expressions cannot become stuck at any non-downcast point.

b) The rule

$$\frac{T_1' <: T_1 \qquad T_2 <: T_2'}{T_1 \to T_2 \;<:\; T_1' \to T_2'}$$

allows the argument to have more structure than required, eg

$$\frac{\{x:int,\ y:bool\} <: \{x:int\} \qquad string <: string}{\{x:int\} \to string \;<:\; \{x:int,\ y:bool\} \to string}$$

~~and the result~~
(this is contravariant) and extra structure in the result to be discarded, eg

$$string \to \{x:int,\ y:bool\} \;<:\; string \to \{x:int\}$$

c) A simple object can be expressed as a record of functions, eg

$$\{\ method_1 = fn\ x:int \Rightarrow x+1$$
$$method_2 = fn\ y:bool \Rightarrow y\ \}$$

This is a pure object – one with some internal state would be expressed e.g.

```
let val
      field₁ = ref 17
in
      { get = fn (): unit => !field₁ ,
        set = fn x : int => field₁ := x }
```

Record subtyping allows the presence of unrequired fields to be neglected.

By abstracting objects on their internal states we can build simple classes, using structural subtyping to give a subclass relationship. For example:

~~class₁ = fn state : { field₁ : int, .., fieldₙ : int } =>~~
~~let val x = ref state~~
~~in { method₁ = fn (): unit => (!x).field₁ }~~

~~class₂ = fn state : { ... } =>~~

```
class₁ = fn state : { field₁ : int ref, ... } =>
           { method₁ = fn x:T => ! state.field₁
             ... }

class₂ = fn state : { field₁ : int ref, ... } =>
           { method₁ = (class₁ state).method₁
```

Note the subtype relationship (using the covariant side of the function subtype rule) and the explicit use of the superclass.