Semantics, WS 2003:
Solutions for assigment 8

Prof. Dr. Gert Smolka, Dipl.-Inform. Guido Tack

Exercise 8.1: Subtyping and References

(a) $(\lambda r : Ref\,(Unit \to Unit).r := unit; !r\ unit)(ref\,(\lambda x : Unit.x))$

Here, the r in $r := unit$ can get the type $Ref\ Top$, so that the assignment is well-typed because of the covariance of Ref. Of course the application $!r\ unit$ is well-typed even without subtyping. Hence, the whole term is stuck, as $!r\ unit$ reduces to $unit\ unit$.

(b) $(!(ref\ unit))\ unit$

Exercise 8.2: Subtyping and Preservation Let $t = (\lambda x : Top.x)(\lambda x : Top.x)$. Then $\varnothing \mapsto t : Top$, $t \longrightarrow \lambda x : Top.x$, and $\varnothing \mapsto (\lambda x : Top.x) : Top \to Top$. Now $Top \neq Top \to Top$, but $Top :> Top \to Top$, which is enough for the Preservation Property. We allow terms to go down in the subtype hierarchy during reduction.

Exercise 8.3: Natural numbers in Java

(a) $Object = \{\}$
$ObjectRep = \{\}$
$ObjectArg = \{\}$
$classObject = \lambda r : ObjectRep.\lambda f.\lambda\_.\{\}$
$newObject = \lambda r : ObjectArg.fix(classObject\ \{\})unit$

$Nat = \{add : Nat \to Nat\}$
$NatRep = \{\}$
$NatArg = \{\}$
$classNat = \lambda r : NatRep.\lambda f.\lambda\_.\{add = \lambda x : Nat.x\}$
$newNat = \lambda r : NatArg.fix(classNat\ \{\})\ unit$

$Pos = \{add : Nat \to Nat\}$
$PosRep = \{pred : Nat\}$
$PosArg = Nat$
$posRec = fix(\lambda rec.$
$\quad \{\ classPos = \lambda r : PosRep.\lambda f.\lambda\_.\{add = \lambda x : Nat.\ r.pred.add(rec.newPos\ x)\},$
$\quad\ \ newPos = \lambda r : PosArg.fix(rec.classPos\ \{pred = r\})\ unit\})$

$newPos = posRec.newPos$
$classPos = posRec.classPos$

(b) $Cla = \{Object, Nat, Pos\}$
$Object <: Nat, Nat <: Pos$
$typ(Object) = \{\}$
$typ(Nat) = \{add \mapsto Nat \to Nat\ \}$
$typ(Pos) = \{pred \mapsto Nat,$
$\qquad\qquad add \mapsto Nat \to Nat\ \}$
$mth(Nat) = \{add \mapsto \lambda x : Nat.x\}$
$mth(Pos) = \{add \mapsto \lambda x : Nat.\ this.pred.add(new\ Pos\ \{pred \mapsto x\})\}$

(c) We need some form of *type case* for subtraction. In Java, this is available through `instanceof`:

```
class Nat extends Object {
  Nat() {}
  Nat add(Nat y) { return y; }
  Nat sub(Nat y) { return this; }
}

class Pos extends Nat {
  Nat pred;
  Pos (Nat x) { pred = x; }
  Nat add(Nat y) { return pred.add(new Pos(y)); }
  Nat sub(Nat y) {
    if (y instanceof Pos) {
      return pred.sub((Pos) y.pred);
    } else {
      return this;
    }
}
```

In SJ, we can introduce a *case* construct for classes as follows:

Typing rule:

$$\frac{\Gamma \vdash t : C' \quad \Gamma \vdash t_1 : C_1 \to T \quad \Gamma \vdash t_2 : C_2 \to T \qquad C <:_0 C_1 \qquad C' <:_0 C_2}{\Gamma \vdash \mathit{case}\ C\ t\ t_1\ t_2\ :\ T}$$

Reduction:

$$\begin{array}{lll} \mathit{case}\ C\ (\mathit{new}\ C'\ f)\ v_1\ v_2 \longrightarrow & v_1\ (\mathit{new}\ C'\ f) & \text{if } C' <:_0 C \\ \longrightarrow & v_2\ (\mathit{new}\ C'\ f) & \text{otherwise} \end{array}$$

Now we can define *sub* in SJ (these are only the extensions):

$$\begin{array}{l} \mathit{typ}\ \mathit{Nat}\ \mathit{sub} = \mathit{Nat} \to \mathit{Nat} \\ \mathit{typ}\ \mathit{Pos}\ \mathit{sub} = \mathit{Nat} \to \mathit{Nat} \\ \mathit{mth}\ \mathit{Nat}\ \mathit{sub} = \lambda y : \mathit{Nat}.\mathit{this} \\ \mathit{mth}\ \mathit{Pos}\ \mathit{sub} = \lambda y : \mathit{Nat}.\mathit{case}\ \mathit{Pos}\ y\ (\lambda y : \mathit{Pos}.\ (\mathit{this}.\mathit{pred}.\mathit{sub})(y.\mathit{pred}))\ (\lambda y : \mathit{Nat}.\ \mathit{this}) \end{array}$$

Exercise 8.4: Buzzwords The examples for the given buzzwords are taken from the different `counter` classes from chapter 18 in Pierce's book.

(a) *Encapsulation* means that the internal representation of an object is generally hidden from view outside the object's definition. So only the object's own methods can access its fields.
Example: The reference cell holding the value of a counter is only accessible by methods of the counter.

(b) *Multiple representations:* Two objects with the same interface may still have differently implemented methods.
Example: Objects of the class `counterClass` (p. 231) implement the increment operator as

```
inc = λ_:Unit.  r.x := succ(!(r.x)),
```

whereas objects of `setCounterClass` (p. 235) use

```
inc = λ_:Unit.  self.set(succ(self.get unit)).
```

(c) *Subtyping:* The notion of subtyping with objects is the same as before: A class A is a subtype of a class B if objects of A can safely be used whenever an object of B is expected.
Programming languages can create this class hierarchy in two ways: Explicitly (via inheritance) or structurally (as with record subtyping). Example: The `setCounter` class is a subtype of the `counter` class, because it holds for the interfaces of objects of these classes that

```
{get:  Unit→Nat, inc:Unit→Unit, set:Nat→Unit}
     <:  {get:  Unit→Nat, inc:Unit→Unit}.
```

(d) *Inheritance:* Objects which share parts of their interfaces will also often share parts of their methods (i.e. implementations). To avoid code duplication, classes can be derived from another class, and objects of the subclass may use methods from the base class. In addition, inheritance often defines the subtyping on classes. Example: Objects of the `backupCounter` class (p. 233) inherit the methods `get` and `inc` from the `resetCounter` class.

(e) *Overriding:* When a new class is defined using inheritance, it is not necessary to use all methods of the base class, but they may also be redefined. This is called overriding.
Example: The `backupCounter` class overrides the method `reset` of the `resetCounter` class.

(f) *Late binding of `self`* means that `self` is only bound at the latest possible moment: when a method of an object is used. This makes it possible for a method defined in a superclass to access methods defined in a subclass.
Example: The class `setCounterClass` from section 18.10.

(g) *Open recursion* means that all methods of a class can be mutually recursive: A method can call other methods of its class (including the method itself) through `self`. Example: The class `setCounterClass` from section 18.10.

Exercise 8.5: Exponentially larger normal forms We define the terms t_i as follows:

$t_0 = \lambda x.x$

$t_1 = (\lambda x.f\ x\ x)\ t_0$

$t_n = (\lambda x.f\ x\ x)\ t_{n-1}$