Elements of Programming Languages
Tutorial 6: Classes, subtyping, and comprehensions
Solution notes

Exercises marked * are more advanced. Please try all unstarrred exercises before the tutorial meeting. Starred exercises are more challenging. Please try all unstarrred exercises before the tutorial meeting.

1. Imperative programming

(a) \( y := x + x \)

\[
\frac{3 + 3 \Downarrow 6}{\sigma, y := x + x \Downarrow \sigma[y := 6]}
\]

(b) if \( x == y \) then \( x := x + 1 \) else \( y := y + 2 \)

\[
\frac{3 \Downarrow 3 \quad 4 \Downarrow 4}{\sigma, y := y + 2 \Downarrow \sigma[y := 6]} \quad \frac{4 + 2 \Downarrow 6}{\sigma, y := y + 2 \Downarrow \sigma[y := 6]} \quad \frac{3 == 4 \Downarrow \text{true}}{\sigma, y := x + 1 \Downarrow \sigma[y := 6]}
\]

(c) (*) while \( x < y \) do \( x := x + 1 \)

\[
\frac{3 < 4 \Downarrow \text{true}}{\sigma, x := x + 1 \Downarrow \sigma[x := 4]} \quad \frac{4 < 4 \Downarrow \text{false}}{\sigma, \text{while } x < y \text{ do } x := x + 1 \Downarrow \sigma[x := 4]}
\]

2. Covariant and contravariant type parameters

Notice that the Box classes have no content — they are just to demonstrate covariance and contravariance.

<table>
<thead>
<tr>
<th>( A )</th>
<th>Any</th>
<th>Nothing</th>
<th>Super</th>
<th>Sub1</th>
<th>Sub2</th>
</tr>
</thead>
<tbody>
<tr>
<td>g1(new Box1[A])</td>
<td>Error</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>g2(new Box1[A]))</td>
<td>Error</td>
<td>OK</td>
<td>Err</td>
<td>OK</td>
<td>Error</td>
</tr>
<tr>
<td>h1(new Box2[A]))</td>
<td>OK</td>
<td>Error</td>
<td>OK</td>
<td>Error</td>
<td>Error</td>
</tr>
<tr>
<td>h2(new Box2[A]))</td>
<td>OK</td>
<td>Error</td>
<td>OK</td>
<td>OK</td>
<td>Error</td>
</tr>
</tbody>
</table>

The OK cases are those where the subtyping relationship holds. The Error cases are those where the relationship doesn’t hold. It may also be helpful to draw a simple lattice diagram (i.e. a tree with Super at the top and Sub1 and Sub2 as children) and show the subsets of the tree corresponding to the types that are valid for each call.

3. Parameterized traits

The trait should look something like this:

```scala
trait Ordered[T] {
  def compare(that: T): Int
  def < (that: T): Boolean = this.compare(that) < 0
  def <= (that: T): Boolean = this.compare(that) <= 0
  def ==( that: T): Boolean = this.compare(that) == 0
  def != (that: T): Boolean = this.compare(that) != 0
  def > (that: T): Boolean = this.compare(that) > 0
  def >= (that: T): Boolean = this.compare(that) >= 0
}
```
4. List comprehensions

(a)

Result = List(2, 3, 4)
List(1, 2, 3).map{x => x + 1}
// or equivalently
List(1, 2, 3).flatMap{x => List(x + 1)}

(b)

Result = List(1)
List(1, 2, 3).filter{x => x % 2 == 0}.map{x => x / 2}
// or equivalently
List(1, 2, 3).flatMap{x => if (x % 2 == 0) {List(x/2)} else {Nil}}

(c) (∗)

Result = List((1, 2), (1, 3), (2, 3))
List(1, 2, 3).flatMap{x => List(1, 2, 3).filter{y => x < y}.map{y => (x, y)}}
// or
List(1, 2, 3).flatMap{x => List(1, 2, 3).flatMap{y => if (x < y) {List((x, y))} else {Nil} }}

5. (∗) Covariant lists

(a) Something like

Cons(1, Cons("abc", Nil))

(b) Something like this:

```scala
abstract class List[+A] {
  def append[C <: A, B <: C](m: List[B]): List[C]
}

case object Nil extends List[Nothing] {
  def append[C <: Nothing, B <: C](m: List[B]): List[C] = m
}

case class Cons[+A](head: A, tail: List[A]) extends List[A] {
  def append[C>: A, B<:C](m: List[B]): List[C] = Cons[C](head, tail.append[C,B](m))
}
```

This is a little tricky. For the Nil case, we need to say that C <: Nothing since A has been instantiated to Nothing. For the Cons case, the type parameters on append seem necessary to make the typechecker happy. The type parameters on Cons are not necessary, but included to help clarify what is going on.

The supertype and subtype bounds are both necessary; if we remove C <: A then we don’t know that we can put A’s into the result list and if we remove B <: C then we don’t know that we can put B’s into the result list.