Object-Oriented Design	Lecture 11
CS 3500 Fall 2010 (Pucella)	Friday, Oct 22, 2010

11 Parameterized Classes

Last time, we advocated subtyping from multiple types as a way to get more reuse out of client code (the `Colored` example), and because it leads to the use of "standardized interfaces" to provide functionality in a generic way.

We continue on this track here, by showing how we can use parameterized classes in combination with traits to enable more code reuse. We focus on the `Stream` standardized interface introduced last time:

```
trait Stream[A] {

  def hasElement ():Boolean
  def head ():A
  def tail ():Stream[A]
}
```

Here are some useful functions on streams: `print()` to print the elements of a stream, and `sumInt()` to sum the elements of an integer stream. I've put them in a module `Stream`, and we will be adding to it below.

```
object Stream {
  def print[A] (st:Stream[A]):Unit =
    if (st.hasElement()) {
      println("  " + st.head());
      print(st.tail())
    } else
      ()

  def sumInt (st:Stream[Int]):Int =
    if (st.hasElement())
      st.head() + sumInt(st.tail())
    else
      0
}
```

Last time, we saw an implementation of the LIST ADT with support for `Stream`s. That defined lists of integers. If we wanted to define lists of strings, say, then we'd need to define a new ADT. That's stupid. We can generalize our List ADT to work with an arbitrary element type. It's just like we did for the parameterized trait `Stream[T]` — we just need to define a parameterized List[T] ADT:

```
CREATORS
  empty :      () -> List[T]
  cons :       (T, List[T]) -> List[T]

ACCESSORS
  isEmpty :    () -> Boolean
  first :      () -> T
  rest :       () -> List[T]
  length :     () -> Int
  append :     (List[T]) -> List[T]
  find :       (T) -> Boolean
  // stream operations
  hasElement : () -> Boolean
  head :        () -> T
  tail :        () -> Stream[T]
```

with the usual specification – the only difference is in the signature, which is parameterized by a type `T`. (Note also that I've left out `isEqual()` – equality in the presence of parameterization is somewhat subtle, so I'll come back to it later.

Here is the code obtained from the usual Specification Design Pattern, except taking parameterization into account:

```
object List {

  def empty[T] ():List[T] = new ListEmpty[T]()

  def cons[T] (t:T, L:List[T]):List[T] = new ListCons[T](t,L)


  // EMPTY LIST REPRESENTATION
  //
  private class ListEmpty[T] () extends List[T] {
    def isEmpty ():Boolean = true
    def first ():T =
      throw new RuntimeException("empty().first()")
    def rest ():List[T] =
      throw new RuntimeException("empty().rest()")
    def length ():Int = 0
    def append (M:List[T]):List[T] = M
    def find (f:T):Boolean = false

    // canonical methods?

    override def toString ():String = ""

    // stream functions
    def hasElement ():Boolean = false
    def head ():T =
      throw new RuntimeException("empty().head()")
    def tail ():Stream[T] =
      throw new RuntimeException("empty().tail()")
  }


  // CONS LIST REPRESENTATION
  //
  private class ListCons[T] (n:T, L:List[T]) extends List[T] {
    def isEmpty ():Boolean = false
    def first ():T = n
    def rest ():List[T] = L
    def length ():Int = 1 + L.length()
    def append (M:List[T]):List[T] = List.cons(n,L.append(M))
    def find (f:T):Boolean = { (f == n) || L.find(f) }


    override def toString ():String = n + " " + L.toString()

    // stream functions
    def hasElement():Boolean = true
    def head():T = n
    def tail():Stream[T] = L
  }
}


abstract class List[T] extends Stream[T] {

  def isEmpty ():Boolean
  def first ():T
  def rest (): List[T]
  def length ():Int
  def append (M:List[T]):List[T]
  def find (f:T):Boolean
  // isEqual?
}
```

A couple of things to note. First, when defining the abstract class `List`, we specify the *type parameter*. This is the name by which the type parameter that we use when we actually use `List` will be referred to in the body of `List`. For instance, we see that `List[T]` will be a subtype of `Stream[T]`, for that same `T`. Thus, when we talk about `List[Int]`, it is a subtype of `Stream[Int]`, and when we talk about `List[String]`, it is a subtype of `Stream[String]`.

Second, the implementation classes are also parameterized, which makes sense because they need to subtype `List[T]`, which is itself parameterized.

Third, the creators are generic methods – that's because there's a single creator function for all lists, so it has to work no matter the element type of the list. Thus, when we invoke `cons()`, we get to specify the kind of list we are working on: `List.cons[Int](1,L)` will tack on integer 1 to the integer list `L` and produce an integer list, while `List.cons[String]("hello",L')` will tack on string `"hello"` to list `L'`.

Here are some examples:

```
val L1:List[Int] = List.cons(33,List.cons(66,List.cons(99,List.empty())))

println("L1 = " + L1)
println("Elements L1 = ")
Stream.print[Int](L1)
println("Sum L1 = " + Stream.sumInt(L1))

val L2:List[String] =

List.cons[String]("hello",
                  List.cons[String]("world",
                                    List.empty[String]))
println("L2 = " + L2)
println("Elements L2 = ")
Stream.print[String](L2)
```

with output:

```
L1 = 33 66 99
Elements L1 =
  33
  66
```

```
    99
  Sum L1 = 198
  L2 = hello world
  Elements L2 =
    hello
    world
```

Recall that we do not usually need to explicitly specify a type when calling generic method — the type checker will often figure out the right type for us. Accordingly, from now on, I will leave out the type argument on calls to generic methods, with the understanding that I can always toss them in if the type checker gets confused.

Here's another example of a parameterized ADT, binary trees. Those also provide a nice example of nontrivial stream (which is good because streams for lists are somewhat unexiting).

Here is the BinTree[T] ADT:

```
  CREATORS
    empty :        () -> BinTree[T]
    node :         (T,BinTree[T],BinTree[T]) -> BinTree[T]

  OPERATIONS
    isEmpty :      () -> Boolean
    root :         () -> T
    left :         () -> BinTree[T]
    right :        () -> BinTree[T]
    size :         () -> Int
```

with specification

$$\texttt{empty().isEmpty()} = \textit{true}$$
$$\texttt{node(}v\texttt{,}l\texttt{,}r\texttt{).isEmpty()} = \textit{false}$$
$$\texttt{node(}v\texttt{,}l\texttt{,}r\texttt{).root()} = v$$
$$\texttt{node(}v\texttt{,}l\texttt{,}r\texttt{).left()} = l$$
$$\texttt{node(}v\texttt{,}l\texttt{,}r\texttt{).right()} = r$$
$$\texttt{empty().size()} = 0$$
$$\texttt{node(}v\texttt{,}l\texttt{,}r\texttt{).size()} = 1 + l\texttt{.size()} + r\texttt{.size()}$$

I will not describe the specification of the stream operations, but will simply say that they should deliver all the elements of the tree, in some order. (It's that last bit, the "in some order" one, that is a pain to specify.)

Here is the code obtained from the Specification Design Pattern, with the stream operations added in as well:

```
object BinTree {

  def empty[T] ():BinTree[T] = new Empty[T]()

  def node[T] (n:T, l:BinTree[T], r:BinTree[T]):BinTree[T] =
    new Node[T](n,l,r)

  private class Empty[T] extends BinTree[T] {

    def isEmpty ():Boolean = true
    def root ():T =
      throw new RuntimeException("BinTree.empty().root()")
    def left ():BinTree[T] =
      throw new RuntimeException("BinTree.empty().left()")
    def right ():BinTree[T] =
      throw new RuntimeException("BinTree.empty().right()")
    def size ():Int = 0

    // canonical methods?
    override def toString ():String = "-"

    // stream methods
    def hasElement ():Boolean = false

    def head ():T =
      throw new RuntimeException("BinTree.empty().head()")

    def tail ():Stream[T] =
      throw new RuntimeException("BinTree.empty().tail()")
  }

  private class Node[T] (n:T, l:BinTree[T], r:BinTree[T])
                                                 extends BinTree[T] {

    def isEmpty ():Boolean = false
    def root ():T = n
    def left ():BinTree[T] = l
    def right ():BinTree[T] = r
    def size ():Int = 1 + l.size() + r.size()

    // canonical methods?
    override def toString ():String = n + "[" + l + "," + r + "]"

    // stream methods
    def hasElement ():Boolean = true

    def head ():T = n

    def tail ():Stream[T] = new Pair[T](l,r)
  }

  private class Pair[T] (fst:Stream[T],snd:Stream[T]) extends Stream[T] {

    // stream methods
    def hasElement ():Boolean = {
      fst.hasElement() || snd.hasElement()
    }

    def head ():T =
      if (fst.hasElement())
        fst.head()
      else
        snd.head()

    def tail ():Stream[T] =
      if (fst.hasElement())
        new Pair[T](fst.tail(),snd)
      else
        snd.tail()
  }
}

abstract class BinTree[T] extends Stream[T] {

  def isEmpty ():Boolean
  def root ():T
  def left ():BinTree[T]
  def right ():BinTree[T]
  def size ():Int
}
```

Just as in the `List[T]` case, the parameterization carries over to the implementation classes, and the creators are generic methods.

What's more interesting is the implementation of the stream operations. The stream operations for `empty()` are straightforward, but for `node()`, not so much. The `head()` of the stream clearly should be the root of the tree, but what about the `tail()`? We want to return a stream that will deliver all the elements of the tree without its root. One way to do that is to have `tail()` return a new tree surgically altered so that it doesn't have its root anymore. You can come up with a way to do that. An alternative, which is what I used here, is to realize that since `tail()` only needs to return a `Stream[T]`, there is no reason why it should return a binary tree. It can return anything that is a subtype of `Stream[T]`, and as long as that anything can deliver the rest of the elements stored in the tree (without the root), it's good enough for us. So I use an auxiliary class `Pair` that basically takes two streams (which will be the left and right subtrees of the node under consideration) and deliver all the elements of the first stream follower by all the elements of the second stream.

Here is an example:

```
  val T = BinTree  // convenient abbreviation for BinTree module

  val T1:BinTree[Int] = T.node(33,
                               T.node(66,
                                      T.node(99,T.empty(),T.empty()),
                                      T.empty()),
                               T.node(11,
                                      T.node(22,T.empty(),T.empty()),
                                      T.node(44,T.empty(),T.empty())))
  println("T1 = " + T1)
  println("Elements T1 = ")
  Stream.print(T1)
  println("Sum T1 = " + Stream.sumInt(T1))
```

which yields:

```
  T1 = 33[66[99[-,-],-],11[22[-,-],44[-,-]]]
  Elements T1 =
    33
    66
    99
    11
    22
    44
  Sum T1 = 275
```

Let's look at the `sumInt` function a bit more carefully. It works only on streams of integers (and subtypes, of course). Intuitively, though, there are other classes with a concept of "summing elements". Let's illustrate this using pairs of integers, where the sum of (x_1, y_1) and (x_2, y_2) is just pointwise addition, $(x_1 + x_2, y_1 + y_2)$.

Let's define a simple ADT for pairs of integers, and it's simple enough that I won't bother writing down the specification. (Exercise: write it!):

```
object PairInt {

  def create (f:Int,s:Int):PairInt = new PairImpl(f,s)

  private class PairImpl (f:Int,s:Int) extends PairInt {

    def first ():Int = f
    def second ():Int = s

    def add (i:PairInt):PairInt =
      new PairImpl(f+i.first(),s+i.second())

    // canonical methods?

    override def toString ():String = "(" + f + "," + s + ")"
  }
}

abstract class PairInt {

  def first ():Int
  def second ():Int
  def add (p:PairInt):PairInt
}
```

Straightforward enough. Let's write a variant of `sumInt()` that works with pairs of integers:

```
  def sumPairInt (st:Stream[PairInt]):PairInt =
    if (st.hasElement())
      st.head().add(sumPairInt(st.tail()))
    else
      PairInt.create(0,0)
```

Let's try it out:

```
  val L4:List[PairInt] =
    List.cons(PairInt.create(1,2),
              List.cons(PairInt.create(5,6),
                        List.cons(PairInt.create(8,9),
                                  List.empty())))
  println("L4 = " + L4)
  println("Elements L4 = ")
  Stream.print(L4)
  println("Sum L4 = " + Stream.sumPairInt(L4))
```

which yields the expected output:

```
L4 = (1,2) (5,6) (8,9)
Elements L4 =
  (1,2)
  (5,6)
  (8,9)
Sum L4 = (14,17)
```

Functions `sumInt` and `sumPairInt` are doing pretty much the same thing, but they look a bit different because they use different methods to add streams elements (`+` versus `add()`) and use a different base value. We can make them more similar by first requiring that `sumInt` and `sumPairInt` take the base value as an extra argument, and by defining a new class to represent integers that have an `add()` operation on them – think of them as just a box you can put around an integer to provide it with an `add()` method.

```
object AInt {

  def create (i:Int):AInt = new AIntImpl(i)

  private class AIntImpl (i:Int) extends AInt {
    def int ():Int = i
    def add (a:AInt):AInt = new AIntImpl(i+a.int())
    override def toString ():String = i.toString()
  }
}

abstract class AInt {

  def int() : Int
  def add(a:AInt):AInt
}
```

Here are the new `sumAInt()` and `sumPairInt()` functions:

```
  def sumAInt (base:AInt, st:Stream[AInt]):AInt =
    if (st.hasElement())
      st.head().add(sumAInt(base,st.tail()))
    else
      base

  def sumPairInt (base:PairInt, st:Stream[PairInt]):PairInt =
    if (st.hasElement())
      st.head().add(sumPairInt(base,st.tail()))
    else
      base
```

We can now work with lists of `AInt`s:

```
  def i(n:Int):AInt = AInt.create(n)   // abbreviation for creator

  val L5:List[AInt] =
    List.cons(i(33),List.cons(i(66),List.cons(i(99),List.empty())))

  println("L5 = " + L5)
  println("Elements L5 = ")
  Stream.print(L5)
  println("Sum L5 = " + sumAInt(i(0),L5))
```

with output

```
  L5 = 33 66 99
  Elements L5 =
    33
    66
    99
  Sum L5 = 198
```

Now, both `sumAInt()` and `sumPairInt()` look exactly the same, as far as the implementation of the function is concerned. Only the signatures are different. Presumably, we should be able to generalize into a function `sum()`. Intuitively, `sum()` should be able to work with any stream of values of a type that has an `add()` operation defined. This is the same situation we had last time with `Colored`: we need to define a new trait to represent the fact that a type has an `add()` method that expects a value of that same type as the class, and return a result of the same type as the class itself. Let's call that trait `Addable`:

```
trait Addable[A] {
  def add (a:A):A
}
```

Since both `AInt` and `PairInt` implement a suitable `add()` method, they can both be made subtypes of `Addable`, at the appropriate type:

```
abstract class AInt extends Addable[AInt] {

  def int() : Int
  def add(a:AInt):AInt
}


abstract class PairInt extends Addable[PairInt] {

  def first ():Int
  def second ():Int
  def add (p:PairInt):PairInt
}
```

Let's define a function `sum()` that works with any stream of addable values:

```
def sum[T <: Addable[T]] (u:T,st:Stream[T]):T =
  if (st.hasElement())
    st.head().add(sum(u,st.tail()))
  else
    u
```

Note what this says: `sum()` is happy to work with any stream as long as it is a stream of `T`s where `T` has an operation `add()` of the right type — this is what the `[T <:  Addable[T]]` constraint means).

As an example, here is `sum` working on a list of `AInt`s like above:

```
def i(n:Int):AInt = AInt.create(n)

val L6:List[AInt] =
  List.cons(i(33),List.cons(i(66),List.cons(i(99),List.empty())))

println("L6 = " + L6)
println("Elements L6 = ")
Stream.print(L6)
println("Sum L6 = " + Stream.sum(i(0),L6))
```

with output:

```
L6 = 33 66 99
Elements L6 =
  33
  66
  99
Sum L6 = 198
```

That's great. Can we push this further? We cannot do more for `sum()`, since it's as reusable as it can be. But we can generalize things elsewhere. In particular, our pairs are pathetic. Let's first define pairs of `AInt`s to illustrate the potential for generalization there:

```
object PairAInt {

  def create (f:AInt,s:AInt):PairAInt = new PairImpl(f,s)

  private class PairImpl (f:AInt,s:AInt) extends PairAInt {

    def first ():AInt = f
    def second ():AInt = s

    def add (i:PairAInt):PairAInt =
      new PairImpl(f.add(i.first()),s.add(i.second()))

    // canonical methods?
    override def toString ():String = "(" + f + "," + s + ")"
  }
}

abstract class PairAInt extends Addable[PairAInt] {

  def first ():AInt
  def second ():AInt
  def add (p:PairAInt):PairAInt
}
```

This is a simple variant of the `PairInt` example from earlier, except using `AInt`s instead of `Int`s. Here is an example using a binary tree of pairs of `AInt`s:

```
def p (i:Int,j:Int):PairAInt = PairAInt.create(AInt.create(i),
                                                AInt.create(j))
         // a convenient abbreviation for the creator
```

```
  val T2:BinTree[PairAInt] =
    T.node(p(1,2),
           T.node(p(30,40),T.empty(),T.empty()),
           T.node(p(500,600),
                  T.node(p(7000,8000),
                         T.empty(), T.empty()),
                  T.empty()))

  println("T2 = " + T2)
  println("Elements T2 = ")
  Stream.print(T2)
  println("Sum T2 = " + Stream.sum(p(0,0),T2))
```

with output:

```
  T2 = (1,2)[(30,40)[-,-],(500,600)[(7000,8000)[-,-],-]]
  Elements T2 =
    (1,2)
    (30,40)
    (500,600)
    (7000,8000)
  Sum T2 = (7531,8642)
```

Now, if you look at the `PairAInt` class, two things stick out. First, it should be easy to parameterize — instead of a pair of two values, each of type `AInt`, make it a pair of two values, one of type `T` and one of type `U`, and make those parameters to the class. Second, the `add()` method in `PairAInt` calls the `add()` methods of the values appearing as first and second elements of the pair. So in order for the code above to make sense in general, not only will our pairs be `Addable`, they will also require you to give them two types `T` and `U` that are themselves `Addable` — otherwise the type checker will complain when type checking `add()` in our pair implementation that it's trying to call an `add()` method on a type that is not guaranteed to have it.

Here is the resulting ADT, called APair[T,U]:

```
object APair {

  def create[T<:Addable[T],U<:Addable[U]] (f:T,s:U):APair[T,U] =
    new APairImpl[T,U](f,s)

  private class APairImpl[T <: Addable[T],U <: Addable[U]] (f:T,s:U)
                                                  extends APair[T,U] {
```

```
    def first ():T = f
    def second ():U = s

    def add (i:APair[T,U]):APair[T,U] =
      new APairImpl(f.add(i.first()),s.add(i.second()))

    // canonical methods?
    override def toString ():String =
      "(" + f.toString() + "," + s.toString() + ")"
  }
}

abstract class APair[T <: Addable[T],U <: Addable[U]]
                                         extends Addable[APair[T,U]] {

  def first ():T
  def second ():U
  def add (p:APair[T,U]):APair[T,U]
}
```

The types tell the whole story. Look at the type for the abstract class:

```
abstract class APair[T <: Addable[T],U <: Addable[U]]
                                       extends Addable[APair[T,U]] {
```

This says that we're defining an abstract class `APair` that has two parameters, `T` and `U`, each that must be a subtype of `Addable`, and that is itself a subtype of `Addable`. Each use of `Addable` needs to specify the type at which it should be instantiated.

This works and compiles, and gives us a lot of flexibility to create structure on which we can reuse our `sum()` functions (which is of course just a stand-in for some application code that requires access to the content of an aggregate structure using a stream).

Here is the same example as `T2` above, except rewritten using `APair`s of `AInt`s:

```
val p0:APair[AInt,AInt] = APair.create(i(0),i(0))
val p1:APair[AInt,AInt] = APair.create(i(1),i(2))
val p2:APair[AInt,AInt] = APair.create(i(30),i(40))
val p3:APair[AInt,AInt] = APair.create(i(500),i(600))
val p4:APair[AInt,AInt] = APair.create(i(7000),i(8000))

val T3:BinTree[APair[AInt,AInt]] =
  T.node(p1,
         T.node(p2,T.empty(),T.empty()),
          T.node(p3,
                 T.node(p4,
                        T.empty(), T.empty()),
                 T.empty()))
println("T3 = " + T3)
println("Elements T3 = ")
Stream.print(T3)
println("Sum T3 = " + Stream.sum(p0,T3))
```

with output:

```
T3 = (1,2)[(30,40)[-,-],(500,600)[(7000,8000)[-,-],-]]
Elements T3 =
  (1,2)
  (30,40)
  (500,600)
  (7000,8000)
Sum T3 = (7531,8642)
```

But we can also do things beyond what we could easily do before, like working with trees of pairs of an addable integer and another pair, itself a pair of two addable integers:

```
val pp0:APair[AInt,APair[AInt,AInt]] = APair.create(i(0),p0)
val pp1:APair[AInt,APair[AInt,AInt]] = APair.create(i(1),p1)
val pp2:APair[AInt,APair[AInt,AInt]] = APair.create(i(2),p2)
val pp3:APair[AInt,APair[AInt,AInt]] = APair.create(i(3),p3)
val pp4:APair[AInt,APair[AInt,AInt]] = APair.create(i(4),p4)

val T4:BinTree[APair[AInt,APair[AInt,AInt]]] =
  T.node(pp1,
         T.node(pp2,T.empty(),T.empty()),
         T.node(pp3,
                T.node(pp4,
                       T.empty(), T.empty()),
                T.empty()))
println("T4 = " + T4)
println("Elements T4 = ")
Stream.print(T4)
println("Sum T4 = " + Stream.sum(pp0,T4))
```

with output:

```
T4 = (1,(1,2))[(2,(30,40))[-,-],(3,(500,600))[(4,(7000,8000))[-,-],-]]
```

```
Elements T4 =
  (1,(1,2))
  (2,(30,40))
  (3,(500,600))
  (4,(7000,8000))
Sum T4 = (10,(7531,8642))
```