PLRG

CS320 - Scala Seminar

Parametric Polymorphism

Jaemin Hong

hjm0901@gmail.com

November 14, 2018

PLRG

Lists of Integers

```
trait List

case object Nil extends List

case class Cons(head: Int, tail: List) extends List
```

Options of Integers

```
trait Option

case object None extends Option

case class Some(n: Int) extends Option
```

Streams of Integers

```
class Stream(val head: Int, tl: => Stream) {

  lazy val tail = tl

}
```

More Than Integers

- We need lists, options, and streams of other types as well.
- For example,
    - a list of booleans,
    - an option of a function from a string to a string, and
    - a stream of doubles.

Functions for Lists

```
def list_map(l: List, f: Int => Int): List = l match {
  case Nil => Nil
  case Cons(h, t) => Cons(f(h), list_map(t, f)) }

def list_filter(l: List, f: Int => Boolean): List = l match {
  case Nil => Nil
  case Cons(h, t) =>
    if (f(h)) Cons(h, list_filter(t, f)) else list_filter(t, f) }
```

Functions for Options

```
def option_map(opt: Option, f: Int => Int): Option =
  opt match {
    case None => None
    case Some(n) => Some(f(n)) }


def option_flatMap(opt: Option, f: Int => Option): Option =
  opt match {
    case None => None
    case Some(n) => f(n) }
```

Functions for Streams

```
def stream_map(st: Stream, f: Int => Int): Stream =
  Stream(f(st.head), stream_map(st.tail, f))


def stream_map2(
  st0: Stream, st1: Stream, f: (Int, Int) => Int
): Stream =
  Stream(f(st0.head, st1.head), stream_map2(st0.tail, st1.tail, f))
```

More Than Integers

- We need functions for lists, options, and streams of other types as well.
- For example,
  - `list_filter` takes a list of strings,
  - `option_map` from an option of a string to an option of an integer, and
  - `stream_map2` from two streams of integers to a stream of doubles.

The First Solution

- Implement different types and functions for all possible types.
- For example, ...

A List for Each Type

```
trait IList

case object INil extends IList

case class ICons(head: Int, tail: IList) extends IList


trait SList

case object SNil extends SList

case class SCons(head: String, tail: SList) extends SList


...
```

A Function for Each Type

```
def list_map_int_int(l: IList, f: Int => Int): IList =
  case INil => INil
  case ICons(h, t) => ICons(f(h), list_map_int_int(t, f)) }

def list_map_int_str(l: IList, f: Int => String): SList =
  case INil => INil
  case ICons(h, t) => SCons(f(h), list_map_int_str(t, f)) }

...
```

Implementation for Each Type

PLRG

- Does not reuse code.
- Infinitely many types exist.

```
Int

Int => Int

Int => Int => Int

...
```

The Second Solution

- Use the top type.
    - Every value belongs to the top type.
    - In Scala, the top type is `Any`.

The Top Type

```
1: Any

true: Any

"abc": Any

((x: Int) => x): Any


def f(x: Any): Any = x

f(42): Any

f(false): Any
```

Lists of "Any"s

```
trait List

case object Nil extends List

case class Cons(head: Any, tail: List) extends List
```

Lists of "Any"s

```
List(1, 2, 3): List

List("a", "b", "c"): List


List(1, "a", true): List
```

Lists of "Any"s

```
def list_get(l: List, i: Int): Any = ...


val integers = List(1, 2, ...)

list_get(integers, 0) + list_get(integers, 1)  // type error

list_get(integers, 0).asInstanceOf[Int] +

  list_get(integers, 1).asInstanceOf[Int]
```

Lists of "Any"s

```
def list_get(l: List, i: Int): Any = ...


val integers = List("1", 2, ...)

list_get(integers, 0).asInstanceOf[Int] +

  list_get(integers, 1).asInstanceOf[Int]

ClassCastException: String cannot be cast to Integer
```

Using the Top Type

- Is error-prone.
    - A collection may contain a value of a wrong type.
    - User may cast an element to a wrong type.
    - Nothing is checked by the type checker.

"The" Solution

- A function abstracts a term by a term.

```
(1 * 1 * 1)

(2 * 2 * 2)
```

```
f = λx.(x * x * x)

f(1) = (1 * 1 * 1)

f(2) = (2 * 2 * 2)
```

"The" Solution

- Now, abstracts a term by a type.

```
def list_filter(l: List of Int, Int => Boolean): List of Int
def list_filter(l: List of String, String => Boolean):
  List of String


f = λT.(def list_filter(l: List of T, T => Boolean):
          List of T)
f(Int)
f(String)
```

Parametric Polymorphism

- We call such a feature parametric polymorphism, or generics.
- Polymorphism allows using a single implementation with multiple types.
    - Parametric polymorphism
        - Implementation is parametrized by types.
    - Subtype polymorphism
    - Ad-hoc polymorphism

Defining Lists

```
trait List[T]

case object Nil[T] extends List[T]

case class Cons[T](head: T, tail: List[T]) extends List[T]
```

```
error: ';' expected but '[' found.

  object Nil[T]
```

Defining Lists

```
trait List[T]

case class Nil[T]() extends List[T]

case class Cons[T](head: T, tail: List[T]) extends List[T]
```

Creating Lists

```
Nil[Int](): List[Int]

Cons[Int](1, Nil[Int]()): List[Int]

Cons[Int](2, Cons[Int](1, Nil[Int]())): List[Int]



Nil[String](): List[String]

Cons[String]("a", Nil[String]()): List[String]

Cons[String]("b", Cons[String]("a", Nil[String]())): List[String]
```

Using Lists

```
val one: List[Int] = Cons[Int](1, Nil[Int]())

Cons[Int](2, one).head: Int

one match {

  case Nil() => 0

  case Cons(h, t) => h + 3

}
```

Omitting Type Arguments

```
Nil(): List[Int]

Cons(1, Nil()): List[Int]

Cons(2, Cons(1, Nil())): List[Int]



Nil(): List[String]

Cons("a", Nil()): List[String]

Cons("b", Cons("a", Nil())): List[String]
```

Creating Lists

```
def List[T](elems: T*): List[T] = ...
```

```
List[Int](1, 2, 3)

List(1, 2, 3)

List[String]("a", "b", "c")

List("a", "b", "c")
```

Filtering Lists

```
def list_filter(l: List, f: Int => Boolean): List =
  l match {
    case Nil => Nil
    case Cons(h, t) =>
      if (f(h)) Cons(h, list_filter(t, f))
      else list_filter(t, f)
  }
```

Filtering Lists

```
def list_filter[T](l: List[T], f: T => Boolean): List[T] =
  l match {
    case Nil() => Nil()
    case Cons(h, t) =>
      if (f(h)) Cons(h, list_filter(t, f))
      else list_filter(t, f)
  }
```

Filtering Lists

```
list_filter[Int](List(1, 2, 3, 4), _ % 2 == 0)

list_filter(List(1, 2, 3, 4), _ % 2 == 0)

// error: missing parameter type for expanded function


list_filter[String](List("a", "ab", "cd"), _.length == 2)

list_filter(List("a", "ab", "cd"), _.length == 2)

// error: missing parameter type for expanded function
```

Type Inference

- Scala uses local type inference.
- Two possible solutions to help the Scala compiler:
    - Currying
    - Instance methods (object-oriented style)

Currying

```
def f(x: Int, y: Int): Int = x + y

f(1, 2)
```

```
def f(x: Int)(y: Int): Int = x + y

f(1)(2)
```

Filtering Lists

```
def list_filter[T](l: List[T])(f: T => Boolean): List[T] =
  l match {
    case Nil() => Nil()
    case Cons(h, t) =>
      if (f(h)) Cons(h, list_filter(t)(f))
      else list_filter(t)(f)
  }
```

PLRG

Filtering Lists

```
list_filter[Int](List(1, 2, 3, 4))(_ % 2 == 0)

list_filter(List(1, 2, 3, 4))(_ % 2 == 0)
```

```
list_filter[String](List("a", "ab", "cd"))(_.length == 2)

list_filter(List("a", "ab", "cd"))(_.length == 2)
```

Mapping Lists

```
def list_map(l: List, f: Int => Int): List =
  l match {
    case Nil => Nil
    case Cons(h, t) => Cons(f(h), list_map(t, f))
  }
```

Mapping Lists

```
def list_map[T, S](l: List[T])(f: T => S): List[S] =
  l match {
    case Nil() => Nil()
    case Cons(h, t) => Cons(f(h), list_map(t)(f))
  }
```

PLRG

Mapping Lists

```
list_map[Int, Int](List(1, 2, 3))(n => n * n)

list_map(List(1, 2, 3))(n => n * n)


list_map[String, Int](List("a", "ab", "abc"))(_.length)

list_map(List("a", "ab", "abc"))(_.length)


list_map[String, String](List("a", "ab", "abc"))(_.reverse)

list_map(List("a", "ab", "abc"))(_.reverse)
```

Folding Lists

```
def list_foldRight(l: List, n: Int, f: (Int, Int) => Int): Int =
  l match {
    case Nil => n
    case Cons(h, t) => f(h, list_foldRight(t, n, f))
  }
```

Folding Lists

```
def list_foldRight[T, S](l: List[T], n: S)(f: (T, S) => S): S =
  l match {
    case Nil() => n
    case Cons(h, t) => f(h, list_foldRight(t, n)(f))
  }
```

Folding Lists

```
list_foldRight[Int, Int](List(1, 2, 3), 1)(_ * _)

list_foldRight(List(1, 2, 3), 1)(_ * _)


list_foldRight[String, String](List("a", "b", "c"), "")(_ + _)

list_foldRight(List("a", "b", "c"), "")(_ + _)


list_foldRight[Int, String](List(1, 2, 3), "")(_ + _)

list_foldRight(List(1, 2, 3), "")(_ + _)
```

Folding Lists

```
def list_foldLeft(l: List, n: Int, f: (Int, Int) => Int): Int = {
  @tailrec def aux(l: List, inter: Int): Int = l match {
    case Nil => inter
    case Cons(h, t) => aux(t, f(inter, h))
  }
  aux(l, n)
}
```

Folding Lists

```
def list_foldLeft[T, S](l: List[T], n: S)(f: (S, T) => S): S = {
  @tailrec def aux(l: List[T], inter: S): S = l match {
    case Nil() => inter
    case Cons(h, t) => aux(t, f(inter, h))
  }
  aux(l, n)
}
```

PLRG

Folding Lists

```
list_foldLeft[Int, Int](List(1, 2, 3), 1)(_ * _)

list_foldLeft(List(1, 2, 3), 1)(_ * _)
```

```
list_foldLeft[String, String](List("a", "b", "c"), "")(_ + _)

list_foldLeft(List("a", "b", "c"), "")(_ + _)
```

Folding Lists

```
list_foldLeft[Int, List[Int]](List(1, 2, 3), List())(
  (t, h) => Cons(h, t)
)
```

```
list_foldLeft(List(1, 2, 3), List())((t, h) => Cons(h, t))
// type mismatch;  found: Int, required: Nothing
```

```
list_foldLeft(List(1, 2, 3), List[Int]())((t, h) => Cons(h, t))
```

Subtype Polymorphism

- Is defined by substitutability.
    - If every value of type `T` can replace any occurrence of value of type `S`,
    - then `T` is a subtype of `S`, i.e. `T <: S`.

Subtype Polymorphism

- The top type: `Any`
  - `T <: Any` for any type `T`
- The bottom type: `Nothing`
  - `Nothing <: T` for any type `T`
  - No value belongs to `Nothing`.
  - A `throw` expression has type `Nothing`.

Subtype Polymorphism

```
lazy val error: Nothing = throw new Exception
```

```
def f(x: Nothing): Int = x
```

```
def g(x: Nothing): String = x
```

```
def h(x: Nothing): List[Int] = x
```

Subtype Polymorphism

- Users can explicitly declare subtype relationship by using `extends`.

```
class Fruit

class Apple extends Fruit

class Banana extends Fruit
```

- Then, `Apple <: Fruit` and `Banana <: Fruit`, but their converses are wrong.
- `Apple` is not a subtype of `Banana` and vice versa.

Polymorphism

- Parametric polymorphism is mainly used in functional languages.
- Subtype polymorphism is mainly used in object-oriented languages.

- Scala has both!
- To see how two features interact is interesting.

Appending Lists

```
def append[T](l0: List[T], l1: List[T]): List[T] = l0 match {
  case Nil() => l1
  case Cons(h, t) => Cons(h, append(t, l1)) }


append(List(1, 2, 3), List(4, 5, 6))

append(List("a", "b", "c"), List("d", "e", "f"))
```

Appending Lists

```
trait Fruit { val sold: Boolean }


case class Apple(sold: Boolean, radius: Int) extends Fruit

def getApples(n: Int): List[Apple] = ...


val oldApples = getApples(1)

val apples = append(oldApples, getApples(2))
```

Appending Lists

```
case class Banana(sold: Boolean, length: Int) extends Fruit

def getBananas(n: Int): List[Banana] = ...


val bananas = getBananas(2)

val fruits = append(apples, bananas)

// type mismatch;  found: List[Apple], required: List[Fruit]

// type mismatch;  found: List[Banana], required: List[Fruit]
```

Appending Lists

- Currently, `List[T] <: List[S]` iff `T = S`.
- We call it invariance.

```
val fruits = append(
  list_map(apples)(a => a: Fruit),
  list_map(bananas)(b => b: Fruit)
)
```

Covariance

- A list of apples is a list of fruits.
- A list of bananas is a list of fruits.

- It is more intuitive if `List[T] <: List[S]` iff `T <: S`
- We call it covariance.

Covariance

- To make a list covariant, use the + symbol.

```
trait List[+T]

case class Nil[T]() extends List[T]

case class Cons[T](head: T, tail: List[T]) extends List[T]
```

Covariance

```
val oldApples: List[Apple] = getApples(1)

val apples: List[Apple] = append(oldApples, getApples(2))

val bananas: List[Banana] = getBananas(2)

apples: List[Fruit]

bananas: List[Fruit]

val fruits: List[Fruit] = append(apples, bananas)
```

Covariance

- Now, we can use the `object` keyword again by inheriting `List[Nothing]`.

```
trait List[+T]

case object Nil extends List[Nothing]

case class Cons[T](head: T, tail: List[T]) extends List[T]
```

Covariance

```
Cons(1, Cons(2, Nil))

Cons("a", Cons("b", Nil))
```

```
def list_map[T, S](l: List[T])(f: T => S): List[S] = l match {
    case Nil => Nil
    case Cons(h, t) => Cons(f(h), list_map(t)(f)) }
```

Selling Fruits

```
def removeSold(l: List[Fruit]): List[Fruit] =

  list_filter(l)(!_.sold)



removeSold(List(Apple(true, 5), Apple(false, 6)))

removeSold(List(Banana(false, 15), Apple(true, 20)))
```

Selling Fruits

```
list_map(
  removeSold(List(Apple(true, 5), Apple(false, 6)))
)(_.radius)  // error: value radius is not a member of Fruit


list_map(
  removeSold(List(Banana(false, 15), Banana(true, 20)))
)(_.length)  // error: value length is not a member of Fruit
```

Selling Fruits

```
def removeSold(l: List[Fruit]): List[Fruit] =
  list_filter(l)(!_.sold)
def removeSold(l: List[Apple]): List[Apple] =
  list_filter(l)(!_.sold)
def removeSold(l: List[Banana]): List[Banana] =
  list_filter(l)(!_.sold)
```

- Working, but not scalable

Selling Fruits

```
def removeSold[T](l: List[T]): List[T] =

  list_filter(l)(!_.sold)

// error: value sold is not a member of type parameter T
```

- Need to constrain type parameter `T` to be a subtype of `Fruit`.

Upper Bounds

```
def removeSold[T <: Fruit](l: List[T]): List[T] =

  list_filter(l)(!_.sold)
```

- The `<:` symbol constrains type parameter `T` to be a subtype of `Fruit`.
- `Fruit` is an upper bound of `T`.

Upper Bounds

```
list_map(
  removeSold(List(Apple(true, 5), Apple(false, 6)))
)(_.radius)
list_map(
  removeSold(List(Banana(false, 15), Banana(true, 20)))
)(_.length)
removeSold(List(1, 2, 3))
// type argument Int does not conform to type parameter bound T <: Fruit
```

Further Abstraction

```
def list_flatMap[T, S](l: List[T])(f: T => List[S]): List[S]

flatMap = map + flatten



def list_unit[T](t: T): List[T]
```

Further Abstraction

```
scala> list_flatMap(List(1, 2, 3))(
         n => List(n, n * n)
       )
res: List[Int] =
  Cons(1,Cons(1,Cons(2,Cons(4,Cons(3,Cons(9,Nil))))))

scala> list_unit(1)
res: List[Int] = Cons(1,Nil)
```

Further Abstraction

```
def list_map[T, S](l: List[T])(f: T => S): List[S] =

  list_flatMap(l)(t => list_unit(f(t)))
```

```
scala> list_map(List(1, 2, 3))(n => n * n)

res: List[Int] = Cons(1,Cons(4,Cons(9,Nil)))
```

Further Abstraction

```
trait Option[+T]

case object None extends Option[Nothing]

case class Some[T](t: T) extends Option[T]
```

Further Abstraction

```
def option_flatMap[T, S](opt: Option[T])(
  f: T => Option[S]
): Option[S]


def option_unit[T](t: T): Option[T] = Some(t)
```

Further Abstraction

```
scala> option_flatMap[Int, Int](Some(1))(n => Some(n * n))

res: Option[Int] = Some(1)


scala> option_flatMap[Int, Int](Some(1))(n => None)

res: Option[Int] = None


scala> option_flatMap[Int, Int](None)(n => Some(n * n))

res: Option[Int] = None
```

Further Abstraction

```
def option_map[T, S](o: Option[T])(f: T => S): Option[S] =

  option_flatMap(o)(t => option_unit(f(t)))
```

```
scala> option_map(Some(1))(n => n * n)

res: Option[Int] = Some(1)
```

Further Abstraction

```
def list_map[T, S](l: List[T])(f: T => S): List[S] =

  list_flatMap(l)(t => list_unit(f(t)))
```

```
def option_map[T, S](o: Option[T])(f: T => S): Option[S] =

  option_flatMap(o)(t => option_unit(f(t)))
```

Further Abstraction

```
def list_map[T, S](l: List[T])(f: T => S)(
  unit: S => List[S],
  flatMap: List[T] => (T => List[S]) => List[S]
): List[S] = flatMap(l)(t => unit(f(t)))


def option_map[T, S](l: Option[T])(f: T => S)(
  unit: S => Option[S],
  flatMap: Option[T] => (T => Option[S]) => Option[S]
): Option[S] = flatMap(l)(t => unit(f(t)))
```

Type Operator

- What are `List` and `Option`?
- There is no value of type `List` or type `Option`.
- They are type operators, which take a type and return a type.

- A function is a value, which takes a value and returns a value.
- A type operator is a type, which takes a type and returns a type.
- We need a type of a type!

Kinds

- A proper type—a type of a value—is denoted by `*`.

```
Int :: *

String :: *

Int => String :: *

List :: * => *

Option :: * => *

List[Int] :: *
```

Higher-kinded Types

```
def map[M[_], T, S](l: M[T])(f: T => S)(
  unit: S => M[S],
  flatMap: M[T] => (T => M[S]) => M[S]
): M[S] =
  flatMap(l)(t => unit(f(t)))
```

Higher-kinded Types

```
def list_map[T, S](l: List[T])(f: T => S): List[S] =

  map[List, T, S](l)(f)(list_unit, list_flatMap)
```

```
def list_map[T, S](l: List[T])(f: T => S): List[S] =

  map(l)(f)(list_unit, list_flatMap)
```

Higher-kinded Types

```
def option_map[T, S](l: Option[T])(f: T => S): Option[S] =

  map[Option, T, S](l)(f)(option_unit, option_flatMap)
```

```
def option_map[T, S](l: Option[T])(f: T => S): Option[S] =

  map(l)(f)(option_unit, option_flatMap)
```

Possible Abstractions

	by a term	by a type

Abstract a term	lambda abstraction
	`(x: Int) => x`
	`def f(x: Int) = x`
		type abstraction
		`class List[T]`
		`def f[T](t: T) = t`

Abstract a type	dependent types	type operator
		`List, Option`
		`type T[X] = X => X`

PLRG

Summary

- To reuse a single implementation for multiple types, parametrize the implementation by using type parameters.
- We call it parametric polymorphism.
- When parametric polymorphism interacts with subtyping, interesting concepts like variance and bounds of type parameters appear.
- We can abstract types by types and create higher-kinded types.
- The most important thing in Computer Science is **abstraction**!