

CSC 447 - Concepts of Programming Languages

Option Type

Instructor: Eric J. Fredericks



Reporting Missing Data

```
1 def reduce[X](xs: List[X], f: (X,X)=>X) : X = xs match {  
2   case Nil => ???  
3   case head :: Nil => head  
4   case head :: tail => f(head, reduce(tail))  
5 } ensuring {  
6   case result => ???  
7 }
```

- ⚠ `case Nil => null`: contract and clients have to distinguish between `null` and a result
- ⚠ `case Nil => throw IllegalArgumentException("Unable to reduce empty list")`: how to write exception contract? how to not force clients distinguish result from exception?



Learning Objectives

- ? How to represent missing results?
 - Identify uses of option types



Option Type

- Principled approach to missing data
- `Option[T]` resembles `List[T]` with length ≤ 1
 - `None` represents absence of data
 - `Some` represents presence
- Example expressions of type `Option[Int]` : `None` , `Some(5)`



Absence of Data

Programming with `null`

```
1 // get subdirectories of directory `name`  
2 def getDirs1(name: String) : List[java.io.File] =  
3   val dir = new java.io.File(name)  
4   val xs = dir.listFiles  
5   if xs == null // dir is not a directory  
6     then null  
7   else xs.toList.filter (_.isDirectory)
```

Programming with optionals

```
1 def getDirs2(name: String) : Option[List[java.io.File]] =  
2   val dir = new java.io.File(name)  
3   val xs = dir.listFiles  
4   if xs == null  
5     then None  
6   else Some(xs.toList.filter (_.isDirectory))
```



Absence of Data: Clients

Client handling null

```
1 def findTemp : List[java.io.File] =  
2   var result: List[java.io.File] = null  
3   var found = false  
4   for s <- List("/temp", "/tmp") if !found do  
5     result = getDirs(s)  
6     found = (result != null)  
7   end for  
8   result  
9 end findTemp
```

```
1 findTemp match  
2   case null => println("No Temporary Directory.")  
3   case result => println(result.length)
```

Client handling option

```
1 def findTemp2 =  
2   getDirs2("/temp") orElse getDirs2("/tmp")
```

```
1 findTemp2  
2   case None => println("No Temporary Directory.")  
3   case Some(result) => println(result.length)
```

- `null` : no compiler support to check for presence of null-checking, cannot have methods
- Map, fold, filter all work on `Option`



Types of Emptiness

- Scala has many values that represent nothing, e.g.,
 - `None` : empty option
 - `Nil` : empty list
 - `null` : reference to nothing
- `Unit` is not an option type
 - `Unit` always has nothing
 - `Unit` nothing is `()`



Nil vs. null

Scala

```
1 def sum (xs : List[Int]) : Int = xs match
2   case Nil => 0
3   case y::ys => y + sum(ys)
```

- Nil : empty list
- null : empty reference

Java

```
1 int sum (Node<Integer> xs)
2   if (xs == null) return 0;
3   else return xs.item + sum(xs.next);
```

- null used to represent emptiness



Nullable Types

- In Scala, we often pretend `null` does not exist
- Recent languages identify `None` and `null`
- [Swift](#)
- [Kotlin](#)
- These languages distinguish *nullable* and *non-nullable* types



Nullable Types

Kotlin nullable versus non-nullable types

- `T?` in Kotlin resembles `Option[T]` in Scala
- `null` is used for `None`
- Types without `?` do not allow `null`

```
1 var a: String = "abc"
2 a = null /* compilation error */
3 a.length /* always safe */
4
5 var b: String? = "abc"
6 b = null /* ok */
7 b.length /* compiler error */
8
9 b!!.length /* may raise exception */
10
11 /* Safe call */
12 b?.length
13 /* Explicitly expanded safe call */
14 if (b != null) b.length else null
15
16 /* Safe call with default */
17 b?.length ?: -1
18 /* Explicitly expanded */
19 val t = b?.length; if (t != null) t else -1
```



Nested Options

- Division by zero: `None` (undefined)

```
1 def safeDivide (n:Int,m:Int) : Option[Int] =  
2   if m == 0 then None  
3   else Some (n/m)
```

- Divide an optional int safely

```
1 val i: Option[Int] = Some(4)  
2 i.map(safeDivide(_,2))      // val res10: Option[Option[Int]] = Some(Some(2))
```

- Avoid nested options

```
1 val i: Option[Int] = Some(4)  
2 i.flatMap(safeDivide(_,2))  // val res10: Option[Int] = Some(2)
```



Summary

- Option types represent absence of data
- Support higher-order functions `map`, etc.
- Enable expressing alternative computation attempts concisely (`orElse`, `getOrElse`)