CSC 347 - Concepts of Programming Languages

Option Type

Instructor: Stefan Mitsch

Reporting Missing Data

```scala
def reduce[X](xs: List[X], f: (X,X)=>X) : X = xs match {
  case Nil => ???
  case head :: Nil  => head
  case head :: tail => f(head, reduce(tail))
} ensuring {
  case result => ???
}
```

- ⚠ `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?

Learning Objectives

❓ How to represent missing results?

- Understand option type

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 def getDirs1 (dirName : String) : List[java.io.File] =
2   val dir = new java.io.File (dirName)
3   val xs = dir.listFiles
4   if xs == null
5   then null
6   else xs.toList.filter (_.isDirectory)
```

Programming with optionals

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

Absence of Data: Clients

With null

```
1 def printNumTemp1 () =
2   var result : List[java.io.File] = Nil
3   var found = false
4   for s <- List("/temp", "/tmp"); if !found do
5     result = getDirs1(s)
6     found = (result != null)
7   if found
8   then println(result.length)
9   else println("No Temporary Directory.")
```

With option pattern matching

```
1 def printNumTemp2 () =
2   getDirs2("/temp") orElse getDirs2("/tmp") match
3     case None => println("No Temporary Directory.")
4     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`

`Option` vs. `null`

- An option is a type that may have something or nothing
- Scala has many values that represent nothing
  - `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

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

- `Nil` : empty list
- `null` : empty reference

Java

```java
int sum (Node<Integer> xs)
  if (xs == null) return 0;
  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`

```
var a: String = "abc"
a = null /* compilation error */
a.length /* always safe */

var b: String? = "abc"
b = null /* ok */
b.length /* compiler error */

b!!.length /* may give Null Pointer Exception */

b?.length                          /* Safe call */
if (b != null) b.length else null /* expanded */

b?.length ?: -1                                /* Elvis operator */
val t = b?.length; if (t != null) t else -1   /* expanded */
```

Java Optional

- Java 8 added `java.util.Optional`
- The intended use is narrow; for library methods:

Map and FlatMap on Option

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

// .lift(i) safely accesses element at index i
// in bounds: Some(element)
// out of bounds: None replaces IndexOutOfBoundsException
val a : Option[Int] = List(11,21,31).lift(2)
a.map(safeDivide(_,2))     // val res10: Option[Option[Int]] = Some(Some(15))
a.flatMap(safeDivide(_,2)) // val res11: Option[Int] = Some(15)
```

Summary

- Option types represent absence of data
- Can be processed like lists
- Enable expressing alternative computation attempts concisely