CSC 347 - Concepts of Programming Languages

Option Type

Instructor: Stefan Mitsch

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 exceptions

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

Programming with optionals

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

Absence of Data: Clients

With exception handling

```
def printNumTemp1 () =
  var result : List[java.io.File] = Nil
  var found = false
  for s <- List("/temp", "/tmp"); if !found do
    try
      result = getDirs1(s)
      found = true
    catch
      case e: java.io.FileNotFoundException => ()
  found match
    case false => println("No Temporary Directory.")
    case true  => println(result.length)
```

With option pattern matching

```
def printNumTemp2 () =
  getDirs2("/temp") orElse getDirs2("/tmp") match
    case None => println("No Temporary Directory.")
    case Some(result) => println(result.length)
```

- 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