CSC 347 - Concepts of Programming Languages

Option Type

Instructor: James Riely

Reporting Missing Data

```scala
def find[X](xs: List[X],
            p: X=>Boolean) : X =
  xs match {
    case Nil => ???
    case x :: rest =>
      if p(x) then x
      else find(rest, p)
  } ensuring {
    case result => ???
  }
```

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

⚠ `case Nil => null` : contract and clients have to distinguish between `null` and a result

⚠ `case Nil => throw IllegalArgumentException(...)` : exceptions for expected behavior bad, how to write exception contract?

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 def find[X](xs: List[X],
2              p: X => Boolean): X =
3   xs match
4     case Nil => null.asInstanceOf[X]
5     case x :: _ if p(x) => x
6     case _ :: rest      => find(rest, p)
7 end find
```

Programming with optionals

```
1 def find[X](xs: List[X],
2              p: X => Boolean): Option[X] =
3   xs match
4     case Nil             => None
5     case x :: _ if p(x) => Some(x)
6     case _ :: rest.      => find(rest, p)
7 end find
```

Absence of Data: Clients

Find `x>5`, if found return `"found x"`, else find `x>3`, if found return `"found x"`, else print `"not found"`

```
1 val xs = List(1, 2, 3, 4, 5)
```

Client handling null

```
1 find(xs, _ > 5) match
2   case null => find(xs, _ > 3) match
3     case null => "not found"
4     case n    => s"found $n"
5   end match
6   case n    => s"found $n"
7 end match
```

Client handling option

```
1 find(xs, _ > 5) orElse find(xs, _ > 3) match
2   case None    => "not found"
3   case Some(n) => s"found $n"
4 end match
```

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

Nested Options

Safe division returns on zero: `None` (division undefined)

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

```
1 def safeDivide (n:Int,m:Int) : Option[Int] =
2   Option.when(m!=0)(n/m) // Option.unless(m==0)(n/m)
```

- Divide an optional int safely

```
1 val i: Option[Int] = Some(4)
2 i.map(safeDivide(_,2))     // val res: Option[Option[Int]] = Some(Some(2))
3 i.map(safeDivide(_,0))     // val res: Option[Option[Int]] = Some(None)
```

- Avoid nested options

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

Types of Emptiness

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
- `None` : empty option
- `Unit` : always has `()`

Java

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

- `null` : used to represent emptiness
- `null` : empty reference
- `Optional.empty()` : empty option
- `void` : no return value

Nullable Types

- In Scala, we often pretend `null` does not exist
- Recent languages identify `None` and `null`: e.g. 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`

```kotlin
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 raise exception */

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

/* Safe call with default */
b?.length ?: -1
/* Explicitly expanded */
val t = b?.length; if (t != null) t else -1
```

Summary

- `Option` type represents presence/absence of data
- `Either` type represents alternative results
- Support higher-order functions `map`, etc.
- Enable expressing alternative computation attempts concisely (`orElse`, `getOrElse`)