EPFL

Week 11 (Monday): Functional Interfaces with Imperative Implementations

CS-214 Software Construction

Implementing a Functional Interface

Consider a program P(...f...) that uses some class f such that

- f is pure, implemented without using mutation
- f is inefficient

e.g. f counts the number of elements in a list, or computes a mathematical function

We would like to replace f with f' such that:

1. f' is more efficient than f (that's why we do this)
2. f' uses mutation and other effects (to achieve efficiency)
3. f'(x) always returns the same value as f(x), for all x (no matter where it is called from P and how many times). Even though f' is internally impure, we cannot tell that from outside; f' hides its effects.

By 3., if P(...f...) was working correctly, so will P(...f'...), and more efficiently so.

This also generalizes to replacing a class with pure methods with more efficient one.

Caching Patterns: LazyCell, Memoization

Reminder: Parameterless Function Values

```
scala> val x =  () => {println("Evaluating x"); 42}
val x: () => Int = Lambda$1306/0x00007efcd049c410@419f0ea

scala> x()
Evaluating x
val res0: Int = 42

scala> x()
Evaluating x                // <- evaluation happens every time
val res1: Int = 42
```

Reminder: Lazy Values

```
scala> lazy val x = {println("Evaluating lazy val x"); 42}
lazy val x: Int

scala> x
Evaluating lazy val x
val res0: Int = 42

scala> x
val res1: Int = 42        // <- evaluation happens only once
```

Lazy Fields and LazyCell Class

```
class LazyCell[+A](init: => A):
  lazy val get = init

val lc = LazyCell({println("Evaluating x"); 42})
```

Syntactic variation of the constructor (`⇒ A` is similar to `() ⇒ A`):

```
class LazyCell[+A](init: () => A):
  lazy val get = init()

val lc = LazyCell(() => {println("Evaluating"); 42}) // LazyCell@291cbe70
lc.get ==> Evaluating
           42
lc.get ==> 42     //  <- evaluation happens only once, no println 2nd time
```

Lazy Cell Use: LazyList is a List with a LazyCell Tail

```
type LazyList[+A] = LazyCell[ListState[A]]

trait ListState[+A]
object Empty extends ListState[Nothing]
case class Cons[+A](head: A, tail: LazyList[A]) extends ListState[A]
```

Laziness refers to list structure: laziness of the tail field

Summary: Lazy Head vs Lazy Tail

Eager list of lazy cells: `List[LazyCell[Int]]`:

```
def loop: Int = 1 + loop // stack overflows if executed
val ll = LazyCell(loop) // LazyCell@31531d0d
val lst = List(ll, ll, ll) // returns immediately
lst.length // 3
lst.head // LazyCell@31531d0d
lst.head.get // stack overflow
```

```
"lazy list of non-cell Int-s:"
val llst = 42 #:: loop
              #:: LazyList()
llst.head      // 42
llst.length    // stack overflow
llst.tail      // LazyCell@...
llst.tail.head // stack overflow
```

```
"lazy list of lazy cells:"
val lhll = LazyCell(42) #:: LazyCell(loop) #::
           #:: LazyList()
lhll.head.get      // LazyCell(42)
lhll.length        // 2
lhll.tail.head     // LazyCell@...
lhll.tail.head.get // stack overflow
```

Lazy Cell Implementation using Mutation

```
class LazyCell[+A](val init: () => A):
  private var cached: Option[A] = None
  def get: A =
    cached match
      case Some(a) => a
      case None =>
        cached = Some(init())
        cached.get
```

It implements this purely functional interface:

```
class LazyCell[+A](val init: () => A):
  def get: A = init()
```

With `private` we know that the only assignment to `cached` happens inside `get`.

Correctness Theorem for LazyCell

Let `init` be a pure expression (e.g. application of a pure function to some arguments, it also cannot println) that evaluates to some value `v`.

Given

```
val lc = LazyCell(init)
```

then in any program execution, all subsequent calls to `lc.get` will evaluate to that value `v`.

- only holds when fields are private—we must use this property in proof

Proof: An Object Invariant for LazyCell

```
class LazyCell[+A](val init: () => A):
  private var cached: Option[A] = None

  def get: A =
    cached match
      case Some(a) => a
      case None =>
        cached = Some(init())
        cached.get
```

Lemma: every `LazyCell` object satisfies the following object invariant:

```
cached == None || cached == Some(init())
```

Proof: induction on the length of subsequent program execution.

Idea of the Proof of An Object Invariant

```
cached == None || cached == Some(init())
```

We only do the proof for sequential programs. By assumption `init()` always denotes the same value.

Let `lc` be the object created.

We consider a step of an arbitrary execution (one state change at a time):

- when `lc` is created, `cached` $=$ `None` by the initial value
- if the step does not modify `cached`, invariant remains true
- if the step modifies `cached`, it must be a call to `lc.get` (because `cached` is private)
  - if initially `cached`=`None` then after assignment `cached` $=$ `Some(init())`
  - if initially `cached` $\neq$ `None` then nothing is modified, so the invariant continues to hold

This proves the object invariant.

Documenting Invariants using a `valid` method

```
class LazyCell[+A](val init: () => A):
  private var cached: Option[A] = None
  def valid: Boolean =
    cached == None || cached == Some(init())   // object invariant
  def get: A = {
    require(valid)
    cached match
      case Some(a) => a
      case None =>
        cached = Some(init())  // assert(valid)
        cached.get
  } ensuring(res => valid && res == init())
```

From the invariant (`valid`) we know that `a` in `Some(a)` equals `init()`.

LazyCell(init) behaves like init() but more efficient. It is *observationally pure*.

Generalizing From LazyCell to Cached Function

```
case class CachedFunction[-A,+B](val f: A => B): // not just ()=>B
  private var cache: Map[A,B] = Map()
  def apply(a: A): B =
    cache.get(a) match
      case Some(b) => println(f"Cache hit: $a -> $b"); b
      case None =>
        val b = f(a)
        cache.update(a,b)
        b

val csin = CachedFunction(math.sin)
val x1 = csin(0.4) // 0.3894183423086505
val x2 = csin(0.4)
Cache hit: 0.4 -> 0.3894183423086505
// 0.3894183423086505
```

Invariant (`valid` method) of CachedFunction

```
case class CachedFunction[-A,+B](val f: A => B):
  private var cache: Map[A,B] = Map()

  def valid: Boolean = cache.keys.forall(a => cache.get(a) == Some(f(a)))

  def apply(a: A): B = {
    require(valid) // only for specification, executing it would ruin performance
    cache.get(a) match
      case Some(b) => b
      case None =>
        val b = f(a)
        cache.update(a,b)
        b
  } ensuring(res => valid && res == f(a)) // just for proofs
```

Making cache field private Does Not Prevent Exposure

Correctness relies on `cache` being modifiable only within CachedFunction.

Adding `getCache` method would expose mutable map and break correctness:

```
case class CachedFunction[-A,+B](val f: A => B):
  private var cache: Map[A,B] = Map()

  def valid: Boolean = cache.keys.forall(a => cache.get(a) == Some(f(a)))

  def getCache: Map[A,B] = cache  // type system does not prevent this

  def apply(a: A): B = {
    ...
  }
```

This is one reason why having aliasing (multiple references to mutable state) makes reasoning about programs difficult.

Fibonacci Function

```
def fib(n: Int): Int =
  if n == 0 then 0
  else if n == 1 then 1
  else fib(n - 1) + fib(n - 2)
```

What is it's complexity as function of n?

- proportional to the number it returns
- note that $fib(n)$ grows exponentially, $fib(n) \geq 2^{n/2}$, for $n \geq 6$
- thus, function takes exponential time (try $fib(44)$)

Does this also take long?

```
val cf = CachedFunction(fib)
cf(44)
```

Yes, as slow as before. Also, only the value for 44 is cached, not e.g. for 42.

Memoization: Caching Recursive Calls of fib

```
def fib(n: Int): Int =
  if n == 0 then 0
  else if n == 1 then 1
  else
    fib(n - 1) + fib(n - 2)
```

$\Longrightarrow$

```
def fib(n: Int): Int =
  if n == 0 then 0
  else if n == 1 then 1
  else
    memo_fib(n - 1) + memo_fib(n - 2)

def memo_fib(a: Int): Int =
  cache.get(a) match
    case Some(b) => b
    case None =>
      val b = fib(a)
      cache.update(a, b)
      b
```

This gives linear-time computation. Can we automate it? Use higher-order functions!

Parameterize fib by Calls to Given Function

$\Longrightarrow$

```
def fib(n: Int): Int =
  if n == 0 then 0
  else if n == 1 then 1
  else
    memo_fib(n - 1) + memo_fib(n - 2)

def memo_fib(a: Int): Int =
  cache.get(a) match
    case Some(b) => b
    case None =>
      val b = fib(a)
      cache.update(a, b)
      b
```

```
def fibR(rec: Int => Int, n: Int): Int =
  if n == 0 then 0
  else if n == 1 then 1
  else
    rec(n - 1) + rec(n - 2)

def memo_fib: Int => Int = (a:Int) =>
  cache.get(a) match
    case Some(b) => b
    case None =>
      val b = fibR(memo_fib, a)
      cache.update(a, b)
      b
```

We call fibR the recursor for fib (see week 4 exercises *Abstracting over recursion*)

Parameterize Memo Function

```
def fibR(rec: Int => Int, n: Int) : Int = ...

def memo(H: (Int => Int,Int) => Int): Int => Int =
  val cache: Map[Int,Int] = Map()
  def rec(a: Int): Int =
    cache.get(a) match
      case Some(b) => b
      case None =>
        val b = H(rec,a)
        cache.update(a, b)
        b
  rec
// fibR and memo are independent of each other: we disentangled mutual recursion
// To put them together, we call:
def fib(x:Int) = memo(fibR)(x) // caches also the intermediate values, linear time
```

`memo` is independent of `fib`, works for all `Int` $\Rightarrow$ `Int` function descriptions.

Generic Memo

Replace `Int ⇒ Int` a with generic `A ⇒ B`

```
def memo[A,B](H: (A => B,A) => B): A => B =
  val cache: Map[A,B] = Map()
  def rec(a: A): B =
    cache.get(a) match
      case Some(b) => b
      case None =>
        val b = H(rec,a)
        cache.update(a, b)
        b
  rec
```

`memo` takes a recursor H and creates a memoized recursive function f such that, for all x,

$$H(f)(x) = f(x)$$

Avoiding the Checks: Dynamic Programming

Our memoized solution goes from larger values back to smaller ones, then uses the map to prevent descending again down same paths.

We can improve performance if we first solve smaller sub-problems, then move to lager ones.

- this does not improve theoretical complexity
- we usually need to specialize it for a given recursive function
- we avoid doing checks, because we know the result will be in the Map

Instead of a map, we often use arrays if we need to compute functions on integers.

See exercises for for Fibonacci function and choose function!

Example: Floyd–Warshall Algorithm

Given a directed graph with non-negative distances on edges (more generally: no negative weight cycles), find the shortest distance for every pair of edges.

Challenge: there may be exponentially many paths, even if we exclude loops.

Suppose nodes are the numbers 0, 1, ..., N-1 and the distances are d(from,to)

path(from,to,k): length of a shortest path that only uses as additional nodes 0,...,k-1.

No need to consider paths with loops. A path either contains *k* or not:

```
def path(from: Int, to: Int, k: Int): Int =
  if k = 0 then d(from, to)
  else
    min(path(from, to, k - 1),  // paths not going through k
        path(from, k, k-1) + path(k, to, k-1)) // path to k, path from k
```

Compare to the exercise to compute binomial coefficients `choose(n,k)`

Analyzing Recursive Definition

```
def path(from: Int, to: Int, k: Int): Int =
  if k = 0 then d(from, to)
  else min(path(from, to, k - 1),
           path(from, k, k-1) + path(k, to, k-1))
```

- does the function terminate? Yes, third argument decreases.
- what is its complexity? Exponential due to 3 recursive calls.
- what would be the size of table to store using memoization? N^3
  - if memo removes entries from table, is the correctness still preserved?

Alternative: dynamic programming. Store only two matrices ($2N^2$):

- path(from,to,k-1) for all from,to
- path(from,to,k) being computed currently
- then, increment k until you reach N

Dynamic Programming for Shortest Path

```
var k = 0
while k < N do
  p = updateDistances(p, k)
  k += 1

def updateDistances(p: Graph, k: Int): Graph =
  val newP: Graph = p; var from = 0
  while from < N do
    var to = 0
    while to < N do
      newP(from)(to) = min(p(from)(to),
                           p(from)(k) + p(k)(to))
      to += 1
    from += 1
  newP
```

Exceptions

Program That Uses Division

```
def recip100(v: Int): Int =
  100 / v

def f(x: Int, y: Int): Int =
  recip100(x) + recip100(y)
```

`f` crashes if `x == 0` or `y == 0`

Indeed, it is not clear what it should return in such case.

Let us rewrite it to use `Option[T]`.

Using Option

```
def recip100(v: Int): Option[Int] =
  if v == 0 then None
  else Some(100 / v)

def f(x: Int, y: Int): Option[Int] =
  recip100(x) match
    case None => None
    case Some(vx) =>
      recip100(y) match
        case None => None
        case Some(vy) => Some(vx + vy)
```

The results are clearly specified now. Function `f` became much longer.

Using Exceptions

```
class ReciprocalOfZero extends Exception
def recip100(v: Int): Int =
  if v == 0 then throw new ReciprocalOfZero
  else 100 / v

def f(x: Int, y: Int): Int =
  recip100(x) + recip100(y)
```

Function `f` looks is same as before. A caller of `f` may get an exception.

Evaluating Exceptions

Once we have exceptions in language, an expression evaluates to a success (S), e.g.

```
recip100(1)
==>
  if 1 == 0 then throw new ReciprocalOfZero else 100 / 1
==>
  100 / 1 ==> S(100)
```

or a failure (F), indicating the exception thrown:

```
recip100(0)
==>
  if 0 == 0 then throw new ReciprocalOfZero else 100 / 0
==>
  throw new ReciprocalOfZero ==> F(ReciprocalOfZero)
```

Scala Exceptions (Similar to Ones in Java)

Key constructs: `throw` and `catch`. Rules to evaluate them:

```
throw r ==> F(r)

S(x) catch cases ==> S(x)
F(e) catch cases ==> cases(e)
```

For normal operations, exceptions escalate

```
S(x) + S(y) ==> S(x + y)
F(r) + S(e) ==> F(r)
S(x) + F(r) ==> F(r)
```

`x`, `y` are values, `r` exception value, `e` expression

Using Exceptions Internally and Hiding It

```
class ReciprocalOfZero extends Exception
def recip100(v: Int): Int =
  if v == 0 then throw new ReciprocalOfZero
  else 100 / v

def f(x: Int, y: Int): Option[Int] =
  try
    Some(recip100(x) + recip100(y))
  catch
    case _:ReciprocalOfZero => None
```

Caller need now know about exceptions. `None` is not very informative.

Try Type: Store More Information than Option (as seen in labs)

```
sealed abstract class Try[+A]
case class Success[+A](value: A) extends Try[A]          // like Some(value)
case class Failure(exc: Throwable) extends Try[Nothing]  // like None

object Try:
  def apply[A](e: => A): Try[A] =
    try Success(e)
    catch
      case exc => Failure(exc)
```

Hiding Exceptions Using Try Type

```
class ReciprocalOfZero extends Exception
def recip100(v: Int): Int =
  if v == 0 then throw new ReciprocalOfZero
  else 100 / v

def f(x: Int, y: Int): Try[Int] =
  Try(recip100(x) + recip100(y))
```

Caller need now know about exceptions.

Information on exception is kept.

The body of `f` remains concise.

Functionally Composing Try Values Explicitly

```
def recip100(v: Int): Try[Int] =
  if v == 0 then Failure(new ReciprocalOfZero)
  else Success(100 / v)

def f(x: Int, y: Int): Try[Int] =
  recip100(x) match
    case Failure(s) => Failure(s)
    case Success(vx) =>
      recip100(y) match
        case Failure(s) => Failure(s)
        case Success(vy) => Success(vx + vy)
```

To abstract the propagation of failures, define method on Try[A] taking function

- `A ⇒ Try[B]` - what to do on success, the body of `case Success( ... ) ⇒`

This is flatMap. Try is a collection storing v of Success(v)

```
sealed abstract class Try[+A]:
  def flatMap[B](onSuccess: A => Try[B]) =
    this match
      case Failure(e) => Failure(e)
      case Success(v) => onSuccess(v)

def recip100(v: Int): Try[Int] =
  if v == 0 then Failure(new ReciprocalOfZero)
  else Success(100 / v)

def f(x: Int, y: Int): Try[Int] =  // shorter thanks to flatMap
  recip100(x).flatMap: vx =>
    recip100(y).flatMap: vy =>
      Success(vx + vy)
```

Use of Exception: Break statements

```
import scala.util.boundary, boundary.break

def firstIndex[T](xs: List[T], elem: T): Int =
 boundary:
   for (x, i) <- xs.zipWithIndex do
     if x == elem then break(i)
   -1
```

Break jumps outside of boundary, returning given value

- boundary introduces a `given` label
- break throws an exception for that label
- boundary catches it

Instead of `return e` use `break(e)` with function wrapped into a `boundary`

Representing control flow

Representing Positions within One Function

Scala offers exceptions as effects for richer control flow and we could use them internally, then catch them. What if we want more general control-flow?

State of a program during execution is given by

- the values stored in its variables
- where it is currently in execution (program counter, pc)

If we want to have full control of where to go in our piece of code, we can rewrite program to use a program counter representation.

- introduce a program counter (pc) integer variable
- rewrite function to use a single loop:
  - examine pc
  - do the corresponding statement
  - update pc

Example: A Program with Multiple Loops

Rewrite this nested while loop using program counter

```
var i = 0
var j = 0
while i < 10 do
  j = 0
  while j < i do
    f(i,j)
    j += 1
  i += 1
```

Program Points and Transformed Program with Single Loop

```
var i, j = 0
while  "1"
    i < 10 do
  j = 0
  while "2"
   j < i do
    "3"
    f(i,j)
    "4"
    j += 1
  "5"
  i += 1
"6"
```

```
var i, j = 0
var pc = 1
while pc != 6 do
  pc match
    case 1 => if i < 10 then {
                j = 0; pc = 2
              } else pc = 6
    case 2 => if j < i then pc = 3
              else pc = 5
    case 3 => f(i,j); pc = 4
    case 4 => j += 1; pc = 2
    case 5 => i += 1; pc = 1
```

To implement break and such, assign pc to the desired value.

Flow Across Functions: Tail Recursion <=> Loops

```
def whileDo =
  if condition then
    command
    whileDo
```

$\Longleftrightarrow$

```
while condition do
  command
```

We can transform one into another.

Control Flow with General Recursion

Consider evaluator:

```
def eval(expr: Expr): Int =
  expr match
    case Const(i) => i
    case Minus(e1, e2) =>
      val v1 = eval(e1)
      val v2 = eval(e2)
      v1 - v2
```

How many copies of `v1` must program remember when computing:

```
eval(Minus(Const(10), Minus(Const(5),
                            Minus(Const(4), Const(1)))))
```

Compiled program uses **stack** for this (an array; add and remove at the end)

Knowing Where to Return

```
def eval(expr: Expr): Int =
  expr match
    case Const(i) => i
    case Minus(e1, e2) =>
      val v1 = eval(e1)
      val v2 = eval(e2)
      v1 - v2
```

When returning from a deep expression with `eval`, program needs to know whether to go back to

- place after `val v1 = eval(e1)` , or
- place after `val v2 = eval(e2)`

Stack can also keep track of the program counter to return to. Also stores result.

Resuming at the Correct Place using Stack

To represent a call:

- push values of local variables to the stack
- push on pc stack the place after the call
- set pc to entry value and the variables to initial values

At the end of function, restore pc

After call, recover the result and local variables

```
case class Stack[T](var content: List[T] = List()):
  def isEmpty: Boolean = content.isEmpty
  def push(v: T): Unit = content = v :: content
  def pop: T =
    val res = content.head
    content = content.tail
    res
```

Transformed Non-Recursive Function

```
def eval(expr: Expr): Int =
  var exprStack = Stack[Expr]()
  var resStack, pcStack = Stack[Int]()

  var expr0 = expr
  var pc = 1

  while !(pcStack.isEmpty && pc == 4) do
    ...
```

Places in the Function Become Values of pc

```
def eval(expr: Expr): Int =
  "1"
  expr match
    case Const(i) => i


    case Minus(e1, e2) =>
      val v1 = eval(e1)
      "2"

      val v2 = eval(e2)
      "3"
      v1 - v2
  "4"
```

```
while !(pcStack.isEmpty && pc == 4) do
  pc match
    case 1 =>
      expr0 match
        case Const(i) =>
          resStack.push(i)
          pc = 4
        case Minus(e1, e2) =>
          pcStack.push(2)    // later to 2
          exprStack.push(e2) // remember e2
          expr0 = e1; pc = 1 // start call
    case 2 =>
      pcStack.push(3) // later continue to 3
      expr0 = exprStack.pop // recover e2
      pc = 1
```

Computing v1 - v2

```
v1 - v2
```

```
case 3 =>
  val v2 = resStack.pop
  val v1 = resStack.pop
  resStack.push(v1 - v2)
  pc = 4
case 4 =>
  if !pcStack.isEmpty then
    pc = pcStack.pop
```

Result will be on top of resStack.

For more: see exercises.