Zsófia Tóth. *December, 2018*

AP Cheat Sheet

Basics

- List
- useful list operations
  - `List.range(n, m)`
    - where n is start and m is end
  - `List.fill(n)("foo")`
    - where n is the number of times it needs to be filled
  - `List.flatten`
    - `val l: List[Int] = List(List(1,2), List(3,4))`
    - `l.flatten`
      - `List(1, 2, 3, 4)`
- referential transparency (RT)
- higher order functions
- recursion, tail recursion
- compose
- curried functions
- partial curry
  - $f : (A, B, C) => D$
  - `def partialCurry(a: A, b: B)(c: C): D = f(a, b, c)`
    - `val c2d: List[C => D] = map2(la, lb)((a, b) => partialCurry(a, b))`
  - `def applyF(g: C => D, c: C): D = g(c)`
    - `map2(c2d, lc)((c2d, lc) => applyFunc(c2d, c))`
- parametric polymorphism
- pattern matching
- covariant
  - `class Foo[+A]`
    - For some class `List[+A]`, making `A` contravariant implies that for two types `A` and `B` where `A` is a subtype of `B`, then `List[A]` is a subtype of `List[B]`
- contravariant
  - `class Bar[-A]`
    - For some class `Writer[-A]`, making `A` contravariant implies that for two types `A` and `B` where `A` is a subtype of `B`, `Writer[B]` is a subtype of `Writer[A]`
- invariant
  - `class Baz[A]`
  - neither covariant nor contravariant

- construction
  - `::` = List
  - `#::` = Stream
- `map`
  - `def map[B](f: A => B): List[B]`
  - `foldRight` + construction
    - `foldRight(Nil: Empty[B])((h,t) => f(h)::(t))`
- `append`
  - `:::` = List
  - `#:::` = Stream
- `flatMap`
  - `def flatMap[B] (f: A => List[B]) : List[B]`
  - `foldRight` + appending
    - `foldRight(Nil: Empty[B])((h,t) => f(h):::(t))`
  - map can be implemented via $flatMap$
    - $State$
      - `flatMap (a => State.unit(f(a)))`
    - Gen
      - `flatMap (a => Gen.unit(f(a)))`
- `filter`
  - `def filter(f: A => Boolean): List[A]`
    - `foldRight`
- `zipWith`
  - apply a binary op to each element from both lists at the *same* position
  - `def zipWith[A,B,C] (f: (A,B)=>C) (l: List[A], r: List[B]): List[C]`
- semi group

Folding

- Docs
- $reduction$
- fold "is the new" pattern matching with recursion
- z: *initial element*
- op: `f: (A, B) => B`
- foldLeft
  - `def foldLeft[A,B] (z: B) (f: (B, A) => B): B`
  - starts from the ledt side of the list/stream/...
  - `List(1, 2, 3)` = `Cons(1, Cons(2, Cons(3, Nil)))`
  - `f(3, f(2, f(z, 1)))`
    - op is $-$, z is 0
      - $((0-1)-2)-3=-6$
- foldRight

- `def foldRight[A,B] (z: B) (f: (A, B) => B): B`
- starts from the right side of the list/stream/...
- `List(1, 2, 3) = Cons(1, Cons(2, Cons(3, Nil)))`
- `f(1, f(2, f(3, z)))`
  - op is $-$, z is 0
  - $1-(2-(3-0))=2$

Algebraic Data Types

Introduction

How many values do they have ...

- `Nothing`
  - 0
- `Unit`
  - 1
- `Boolean`
  - 2 (*true, false*)
- `Byte`
  - 256
- `String`
  - $many$
- `(Byte, Boolean)`
  - $2 \times 256 = 512$
- `(Byte, Unit)`
  - $1 \times 256 = 256$
- `(Byte, Byte)`
  - $256 \times 256 = 65536$
- `(Byte, Boolean, Boolean)`
  - $256 \times 2 \times 2 = 1024$
- `(Boolean, String, Nothing)`
  - $2 \times many \times 0 = 0$
- `Byte` or `Boolean`
  - $2 + 256 = 258$
- `Boolean` or `Unit`
  - $2 + 1 = 3$
- `Boolean` or `(String, Nothing)`
  - $2 + many \times 0 = 2$

Classes

- Scala

```scala
class ScalaPerson(val name: String, val age: Int)
```

- Java

```java
class JavaPerson {
  final String name;
  final Int age;
}
```

Sum Types

- ADT = **SUM TYPE**
- Sum type
  - This **or** That

```scala
sealed trait Pet
case class Cat(name: String) extends Pet
case class Fish(name: String, color: Color) extends Pet
case class Squid(name: String, age: Int) extends Pet

val bob: Pet = Cat("Bob")
```

- Destructed by pattern matching

```scala
def sayHi(p: Pet): String =
  p match {
    case Cat(n) => "Meow " + n + "!"
    case Fish(n, _) => "Hello fishy " + n + "."
    case Squid(n, _) => "Hi " + n + "."
  }
```

Computations that may return many answers

List

```scala
sealed trait List[+A]
case object Nil extends List[Nothing]
case class Cons[+A](head: A, tail: List[A]) extends List[A]
```

Tree

- binary tree type
  - embeds simple elements of type A in the internal nodes

```scala
sealed trait Tree[+A]
case class Leaf[A] (value: A) extends Tree[A]
case class Branch[A] (left: Tree[A], right: Tree[A]) extends Tree[A]
```

- binary tree type
  - embeds lists of elements of type A in the internal nodes
  - list of tree

```
trait TreeOfLists[+A]
case object LeafOfLists  extends TreeOfLists[Nothing]
case class BranchOfLists[+A] (
    data: List[A],
    left: TreeOfLists[A],
    right: TreeOfLists[A]
) extends TreeOfLists[A]
```

- binary tree type
  - embeds a collection of elements
  - generalized $TreeOfLists$ to use any generic collection `C[_]`

```
trait TreeOfCollections[C[+_], +A]
case class LeafOfCollections[C[+_]] () extends TreeOfCollections[C, Nothing]
case class BranchOfCollections[C[+_], +A](
  data: C[A],
  left: TreeOfCollections[C, A],
  right:TreeOfCollections[C, A]
) extends TreeOfCollections[C, A]
```

Stream

```
sealed trait Stream[+A]
case object Empty extends Stream[Nothing]
case class Cons[+A](h: () => A, t: () => Stream[A]) extends Stream[A]
```

Computations that may fail to return a value

Option

```
sealed trait Option[+A]
case object None extends Option[Nothing]
case class  Some[A](a: A) extends Option[A]
```

Computation that either returns this or that

Either

```
sealed trait Either[+A, +B]
case class Left[A](a: A)  extends Either[A, Nothing]
case class Right[B](b: B) extends Either[Nothing, B]
```

Computations that may fail with an exception

Try

```
sealed trait Try[+A]
case class Success[A](a: A) extends Try[A]
case class Failure[A](t: Throwable) extends Try[A]
```

Options

- Docs
- `Some("value")` and `None`
- for comprehensions
  - `for { } yield ()`
- `map2`
- `sequence`
- `Either[+A, +B]`
  - `Left` ; error
  - `Right` ; result
- `getOrElse`
  - `def getOrElse[B >: A](default: ⇒ B): B`
- `orElse`
  - `def orElse[B >: A](alternative: ⇒ Option[B]): Option[B]`

Streams

- Docs
- infinite streams
  - `from`
  - `to`
  - `fibonacci`
- finite streams
- `unfold`
  - `def unfold[A, S](z: S)(f: S => Option[(A, S)]): Stream[A]`
- lazily evaluation
  - when an expression is passed as a parameter to a function, it is not evaluated before entering the function body
  - evaluated when it is accessed/read the first time inside the function
  - if the result of such expression is never used inside, then it will never be evaluated
- `append`
  - `def append[B >: A](that: => Stream[B]): Stream[B]`
  - lower type bound
  - B is constrained to be a supertype of A
  - allowed to append element of type B to a stream containing elements of type A

        - the result is a stream of B elements
    - `[B <: A]` is an upper type bound
        - B is constrained to be a subtype of A
- library function implementations with `foldRight` and $pattern\ matching$
- $callByValue$
    - default
    - eager evaluation
- $callByName$
    - lazy evaluation
- $callByNeed$
    - lazy evaluation
    - Haskell
    - a memoized variant of call by name where, if the function argument is evaluated, that value is stored for subsequent uses
    - if the argument is side-effect free, this produces the same results as call by name, saving the cost of recomputing the argument

State

- **RNG**
    - `SimpleRNG`
        - class within `RNG` trait
        - takes a seed
        - `nextInt`
            - (Int, RNG)
    - `(A, RNG)`
        - $(value, rng)$
- **RAND**
    - `type RAND[A] = RNG => (A, RNG)`
    - `RAND` type alias for statw
        - `type RAND[A] = State[RNG, A]`
    - `unit`
        - `def unit[A](a: A): Rand[A]`
        - produces: `rng => (value, next)`
    - `map`
    - `map2`
    - `flatMap`
        - bit tricky because of `f: A => RAND[B]`

```scala
rng => {
    val (value, rng2) = ra(rng)
    // f(value) will be a RAND, so pass rng2 to get next value and next state
    val (value2, rng3) = f(value)(rng2)
    (value2, rng3)
}
```

- `sequence`
  - `foldRight`
    - z
      - `unit(Nil: List[A])`
    - op
      - `map2`
- `List.fill(n)(int)`
  - create a list of size n with `RAND[Int]`
  - `val int: Rand[Int] = _.nextInt`
  - same as `rng => rng.nextInt`

- **STATE**
  - make any API purely functional
  - state updates are not RT
    - return the new state along with the value generated
  - this is a more general implementation of `RAND`
    - `type Rand[A] = State[RNG, A]`
  - `case class State[S, +A](run: S => (A, S))`
  - `type State[S, +A] = S => (A, S)`
  - general-purpose functions for capturing states
  - unfold and apply state

```scala
State {
    (s: S) => {
        //must return State[S, B]
    }
}
```

  - `state2stream`
    - `def state2stream[S,A] (state: State[S,A]) (seed: S) :Stream[A]` ; signature
    - implementation:

```scala
state.run(seed) match {
    case(a, s2) => a#::state2stream(state)(s2)
}
```

      - construct a stream from the next value (a)
      - new "seed" to run is the "next" on the same state

Algebraic Design (Functional Design)

- the following 3 chapters have been introduced as part of AD
  - the process of designing purely functional libraries
- parser combinators is heavily focused on AD and contains exercises related to designing a library
- API can be described by and algebra and obey specific $laws$

Par

- purely functional library for parallel and asynchronous computations
- seperate the concern from *describing* a computation to actually *running* it
- *describe*
  - `def unit[A](a: => A): Par[A]`
    - take an unevaluated A
    - return a computation that might evaluate in a seperate *thread*
      - `(es: ExecutorService) => UnitFuture(a)`
        - wrap it in a future task (something that will get evaluated eventually)
- *extract*
  - `def get[A](a: Par[A]): A`
    - get resulting value from a parallel computation
- combine asynchronous computations without waiting for them to finish
  - avoid combining `unit` and `get`
- `type Par[A] = ExecutorService => Future[A]`
  - takes an executor service and wraps A in a future
- `def run[A] (es: ExecutorService) (a: Par[A]) : Future[A] = a(es)`
- $unit$
  - creates computation that immediately results in the value a
- $fork$
  - marks a computation for concurrent evaluation
  - evaluation won't occur until forces by run
- $lazyUnit$
  - wraps it's unevaluated arguement in a Par and marks it for concurrent evaluation
  - `fork(unit(a))`
  - `asyncF[A,B] (f: A => B) : A => Par[B]`
    - async function that takes a function and converts it to that evaluates it result asynchronously
- run
  - fully evaluates given Par
  - spawns parallel computations as requested by $fork$
  - extracts the resulting value
- $map2$
  - combines the result of 2 parallel computations with a $binary$ function
  - does not evaluate the call to f in a seperate logical thread

    - $fork$ is controlling all parallelism
    - `fork(map2(a, b)(f))`
        - if we want the evaluation of f to occur in a seperate thread
- map
    - via `map2(pa, unit(()))((a,_) => f(a))`
    - `map( Par[List[A]] )( List[A] => Boolean/Int/... )`
- $parMap$
    - combines N parallel computations
    - apply a function f to every element in a collection $simultaneously$
- $parFilter$
    - filters in parallel
    - `List[Par[List[A]]]` when `map` and `asyncF`
        - `Nil: List[A]` exists but `Nil: A` does not
        - `map` and apply `sequence()` and `_.flatten`
- $map3$
    - via `map2()`
    - partially apply function
    - partial curry
        - `def partialCurry(a: A, b: B)(c: C): D = f(a, b, c)`
            - `val c2d: C => D = partialCurry(a, b)`
        - `def applyFully(g: C => D, c: C): D = g(c)`
- $sequence$
    - via `foldRight` and `map2()`
- $choiceN$
- $choice$
    - choose between two forking computations based on the result of an initial computation
    - `def choice[A](cond: Par[Boolean])(t: Par[A], f: Par[A]): Par[A]`
- $chooser$
    - `def chooser[A,B](p: Par[A])(choices: A => Par[B]): Par[B]`
- $join$
    - flattens nested $Par[Par[A]]$
    - `def join[A](a: Par[Par[A]]): Par[A]`
- $flatMap$
    - mapping $f: A => Par[B]$ over $Par[A]$ which generates a $Par[Par[B]]$ and then $flattening$ this nested $Par[Par[B]]$ to a $Par[B]$
        - `join(map(p)(f))`

Laws

- `map(unit(x))(f) == unit(f(x))`
- `map(unit(x))(id) == unit(id(x))`
- `map(y)(id) == y`

- `fork(x) == x`

Property Testing

- Scala Check
  - define a property that specifies the behaviour of a method or some unit of code
  - all test data are generated automatically in a random fashion
  - $forAll$
    - creates universally quantified properties directly
      - takes a function as parameter, and creates a property out of it that can be tested with the `check` method
        - `p1.check`
      - function should return `Boolean` or another property, and can take parameters of any types
    - used the most
    - combining existing properties into a new one
      - $p1\&\&p2$
      - $p1||p2$

Generators

- Scala Check's Gen
- `val intList: Gen[List[Int]] = Gen.listOf(Gen.choose(0, 100))`
  - a $generator$ of a list of integers between 0 and 100
- combine $generators$ with `for {} yield()`
- $Arbitrary$ generator
  - `val evenInteger = Arbitrary.arbitrary[Int] suchThat (_ % 2 == 0)`
- homework implemets this with $State$
  - stays RT and pure
  - $Gen.choose(n, m)$ - random number between start and stop

    ```
    def choose (start: Int, stopExclusive: Int): Gen[Int] = {
        Gen(State(rng => RNG.nonNegativeInt(rng) match {
            case (value, next) => ( value % (stopExclusive-start) + start , next )
        }))
    }
    ```

  - $Gen.boolean$ - random `true` or `false`

    ```
    def boolean: Gen[Boolean] = Gen(State(rng => RNG.boolean(rng)))
    ```

  - $Gen.double$ - random double

    ```
    def double: Gen[Double] = Gen(State(rng => RNG.double(rng)))
    ```

  - $Gen.listOfN$
    - using `sequence` and `List.fill(n)(this.sample)`

        - `def sequence[S, A](sas: List[State[S, A]]): State[S, List[A]]`
    - use case of $flatMap$

```
def listOfN (size: Gen[Int]): Gen[List[A]] =
    size.flatMap(n => listOfN(n))
```

    - coin toss (**this** or **that**)

```
def union (that: Gen[A]): Gen[A] =
    Gen.boolean.flatMap(x => if(x) this else that)
```

Properties

- `val prop1 = forAll(intList)(ns => ns.reverse.reverse == ns)`
- `val prop2 = forAll(intList)(ns => ns.reverse.headOption == ns.lastOption)`
    - passing properties
- `val prop3 = forAll(intList)(ns => ns.reverse == ns)`
- Gen is essentially a wrapped State of special kind
- type constraint `[A :Arbitrary]` means that the type A has to implement the $Arbitrary$ trait

Parser Combinators

- AD^2
    - lots of design decisions
    - design chapter

Functional Design (Patterns)

- design patterns

Monoids

- Docs
- abstraction
- set that has
    - $closed, associative$ binary operation
    - $identity$ element
- $closed$
    - $Integers$ are closed with $+$, $-$, $*$, but not $/$
    - $Positive$ numbers are not closed with $-$ (subtraction)
- $identity$ element of ...
    - $+$; addition
        - 0
    - $*$; multiplication

        - 1
- max (-infinity), min (+infinity)
- $||$ (false), $\&\&$ (true)
- $concatenation$, $append$
    - $List$
        - Nil
    - $String$
        - $""$
    - $FingerTree$
        - $Empty$
- $Option$
    - None
    - `orElse`
- $endomonoid$
    - function having the same argument and return type is called an endofunction
- $homomorphism$
    - between monoids M and N, for all values x and y
        - `M.op(f(x), f(y)) == f(N.op(x, y))` => `concat(x.length, y.length) == concat(x, y).length`
            - on $List[A]$ or $String$
- $isomorphism$
    - $homomorphism$ in both directions between two monoids
    - monoid $isomorphism$ between M and N has two $homomorphism$ f and g, where both
        - `f andThen g` and `g andThen f` are identity functions
            - $String$ and $List[Char]$ are $isomorphic$ with concatenation

Foldables

- Docs
- `trait Foldable[F[_]]`
- `F[_]` is the type constructor
    - one arguement
- $foldLeft$
- $foldRight$
- $foldMap$
    - `def foldMap[A, B](as: F[A])(f: A => B)(mb: Monoid[B]) :B`
    - is similar to $fold$ but maps every A value into B and then combines them using the given $Monoid[B]$ instance
- $concatenate$

Functors (=Mappable)

- Docs

- `trait Functor[F[_]]`
- map
  - when a value is wrapped in a structure, a function cannot be applied to that value
  - is parameterized on the type constructor F
- $implicit$ parameter
  - `def mapLaw[A,F[_]] (fn :Functor[F]) (implicit arb: Arbitrary[F[A]]): Prop`
    - `forAll { (fa :F[A]) => fn.map[A,A] (fa) (x=>x) == fa }`
    - it states that when you use this method (mapLaw) there must exist an implicit conversion rule from `F[A]` instances to `Arbitrary[F[A]]` instances
    - scalacheck needs to know that `F[A]` is an instance (or can be made an instance) of Arbitrary in order to be able to generate random instances
  - $F[]$
    - law holds for any type A and a typeconstructor `F[_]`

Monads (=FlatMappable Wrapper)

- Docs
- monad transformer
- wrapper
  - `def unit[A] (x: A): Monad[A]`
- $flatMap$
  - first `map`, then `flatten`
  - To $flatMap$ an object of type `M[A]`, `f: A-> M[B]` is provided,
  - $flatMap$ then uses this function to transform A into $M[B]$, resulting in a $M[M[B]]$, and then it will $flatten$ the whole thing into $M[B]$
- $unit$ and $flatMap$ as a minimal setting
- map, or $map2$ can be defined via $flatMap$
- can extend $Functor$
- $map3monad$ signature
  - `def map3monad[M[_]: Monad, A,B,C,D] (a :M[A], b: M[B], c: M[C]) (f: (A,B,C) => D) :M[D]`
  - `implicitly[Monad[M]]`

Monadic Evaluators

- research paper
- Haskell to Scala
- variation 1 and 2
  - basic evaluator
  - exceptions
  - state
  - output

Lenses

- research paper

Finger Trees

- research paper
  - Haskell
- binary search tree (worst case)
  - $O(n)$
- 2 - 3 finger trees
  - sequence supporting access
    - ends in amortized constant time
    - reduce $O(n)$ to $O(1)$
  - concatenation and splitting
    - $O(log(n))$ in the time of the smaller piece
- balanced
- nodes of Finger Tree - spine
- nodes of 2-3 tree
  - node has 2 or 3 leaves
  - all leaves at the same level
  - `Node a = Node2 a a | Node3 a a a`
- nodes of digits - fingers
- makes use of $laziness$
  - we implemented as an eager data structure
- have an efficient list like data structure
  - can serve as
    - $Sequence$
    - $Priority\ Queue$
    - $Search\ Tree$
    - $Priority\ Search\ Queue$
- Op
  - $reduction$
    - a function that collapses a structure of type $F[A]$ into a single value of type A
      - empty subtrees are replaced by constants
      - intermediate results are combined by using a $binary\ operation$
    - $skewed$ reduction
      - only nested to the *right* or to the *left*
  - dequeue operations
    - $concatenation$
      - polymorphic recursion

            - $addL$
            - $addR$
        - $splitting$
- $isomorphic$ monoid type
    - via **newtype** declaration for each monoid structure of interest
- numerical representation hence $Digit(a)$
- $Digit$
    - `type Digit[A] = List[A]`
    - `object Digit extends Reduce[Digit]`
    - $reduceR$
        - `foldRight`
    - $reduceL$
        - `foldLeft`
    - $toList$
- $Node$
    - `sealed trait Node[+A]`
    - `object Node extends Reduce[Node]`
    - $Node2$
        - `case class Node2[A] (l :A, r :A) extends Node[A]`
    - $Node3$
        - `case class Node3[A] (l: A, m: A, r: A) extends Node[A]`
    - Operations(under `object`)
        - $reduceR$
        - $reduceL$
- $FingerTree$
    - `sealed trait FingerTree[+A]`
    - `object FingerTree extends Reduce[FingerTree]`
    - $Empty$
        - `case class Empty () extends FingerTree[Nothing]`
    - $Single$
        - takes
            - data: A
        - `case class Single[A] (data: A) extends FingerTree[A]`
    - $Deep$
        - takes
            - pr: $Digit[A]$
            - m: $FingerTree[Node[A]]$
            - sf: $Digit[A]$
        - `case class Deep[A] (pr: Digit[A], m: FingerTree[Node[A]], sf: Digit[A]) extends FingerTree[A]`
    - Operations (under `object`)
      - $reduceR$
      - $reduceL$

- $addL$
- $addR$
- $deepL$
- $deepR$
- $toTree$

- `Digit.toTree(FingerTree.toList.listOperation)`
  - `filter`
  - `map`