Functional Programming

6 Functional Reduction

6 Functional Reduction

Monoids

Reduction with Monoids

Lifting Monoids

Parallel Reduction

Monoids

A monoid is an algebraic structure

$$\text{Monoid} = (M, \oplus, e)$$

- M is a set of elements
- $\oplus : M \oplus M \rightarrow M$ is an associative binary inner operation on elements of M
  $(a \oplus b) \oplus c = a \oplus (b \oplus c)$
- e is the identity element for operation $\oplus$
  $a \oplus e = a$ and $e \oplus a = a$

Examples Monoids

- $(\mathbb{Z}, +, 0)$, $(\mathbb{R}, +, 0)$
- $(\mathbb{Z}, *, 1)$, $(\mathbb{R}, *, 1)$
- $(2^A, \cup, \{\})$
- (*String*, +, "")

commutative:
$a \oplus b = b \oplus a$

Reduction

also called **folding**

Monoids allow reduction operations

- Reduction from left

  $((((e \oplus a) \oplus b) \oplus c) \oplus d)$

  iterative version

  - start with identity value e
  - combine current result with next element using operation $\oplus$

- Reduction from right

  $(a \oplus (b \oplus (c \oplus (d \oplus e))))$

  recursive version

  - start with identity value e
  - combine it with last element, then result with previous ...

- Parallel reduction

  $(e \oplus a \oplus b) \quad \oplus \quad (e \oplus c \oplus d)$

  - split elements
  - each split start with identity value e and reduce elements in split
  - combine results from splits

Monoid in Scala

Implementation of Monoid by Scala trait

```
trait Monoid[M] {
  val zero: M
  def op(a: M, b: M) : M
}
```

Companion object with factories and concrete Monoids

```
object Monoid {
  def apply[M](z: M, operator: (M, M) => M) : Monoid[M] =
    new Monoid[M] {
      override def op(a: M, b: M): M = operator.apply(a, b)
      override val zero: M = z
    }
  val intPlusMonoid : Monoid[Int] = Monoid(0, (a, b) => a + b)
  val intTimesMonoid : Monoid[Int] = Monoid(1, (a, b) => a * b)
  val doublePlusMonoid : Monoid[Double] = Monoid(0.0, (a, b) => a + b)
  val doubleTimesMonoid : Monoid[Double] = …
  val stringMonoid : Monoid[String] = …
  def listMonoid[A] : Monoid[List[A]] = …
  def setMonoid[A] : Monoid[Set[A]] = …
}
```

6 Functional Reduction

Monoids

Reduction with Monoids

Lifting Monoids

Parallel Reduction

Reduction with Monoids

Reducing lists

- **reduceRight**: recursive

```
def reduceRight[A](as: List[A])(monoid: Monoid[A]): A =
  as match {
    case Nil => monoid.zero
    case (hd::tl) => monoid.op(hd, reduceRight(tl)(monoid))
  }
```

```
import Monoid.*
val lst = List(1, 2, 3, 4, 5)

val sum = reduceRight(lst)(intPlusMonoid)

val str = reduceRight(lst.map(_.toString))(stringMonoid)
```

- **reduceLeft**: iterative

```
def reduceLeft[A](as: List[A])(monoid: Monoid[A]): A = {
  var r = monoid.zero
  for (a <- as) {
    r = monoid.op(r, a)
  }
  r
}
```

```
import Monoid.*

val lst = List(1, 2, 3, 4, 5)

val sum = reduceLeft(lst)(intPlusMonoid)

val str = reduceLeft(lst.map(_.toString))(stringMonoid)
```

Scala Language Feature: Context Parameters

often called **givens**

Context parameters allow providing values from the context

- without providing it as an actual parameter in a method call
- but providing it **implicitly** by a **value of given type**

How it works

- Define a parameter as **using**

```
def methWithUsing(x: Int)(using s: SomeType)
```

- Define a variable as **given**

```
given someVal : SomeType = SomeType("some value")
```

- Compiler inserts **given value** for the **using parameter** based **on type**

```
methWithUsing(1)
```

(someVal) implicitly

Scala Language Feature: Context Parameters

Example: Prompts in user interactions

- Define **Prompt** class

```
case class Prompt(prompt: String)
```

Must have specific type for givens

- Define method **readLine** with using parameter

```
def readLine(using p: Prompt) : String = {
  print(p.prompt)
  scn.nextLine
}
```

- Define given value

```
given p: Prompt = Prompt("Your input: ")
```

- use given value in subsequent method calls

```
val input1 = readLine
val input2 = readLine
val input3 = readLine
```

```
Your input: Hallo
Your input: World
Your input:
```

Where compiler looks for givens:

- In current scope
  - ☐ define givens in current scope
  - ☐ use imports for bringing givens into scope
- In companion object

Reduction with Monoids with Context Parameter

- Define context parameter for monoid with **using**

```
  def reduceRight[A](as: List[A])(using monoid: Monoid[A]): A =
    as match {
      case Nil => monoid.zero
      case (hd :: tl) => monoid.op(hd, reduceRight(tl)(monoid))
    }
```

- Define default monoids for data types as

```
object Monoid {
  ...
  given intPlusMonoid : Monoid[Int] = of(0, (a, b) => a + b)
  val intTimesMonoid : Monoid[Int] = of(1, (a, b) => a * b)
  given doublePlusMonoid : Monoid[Double] = of(0.0, (a, b) => a + b)
  val doubleTimesMonoid : Monoid[Double] = of(1.0, (a, b) => a * b)
  given stringMonoid : Monoid[String] = of("", (a, b) => a + b)
  given listMonoid[A] : Monoid[List[A]] = of(List(), (l1, l2) => l1.appendedAll(l2))
  given setMonoid[A] : Monoid[Set[A]] = of(Set(), (s1, s2) => s1 ++ s2)
}
```

default monoids as givens

Reduction with Monoids with Context Parameter

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

Call reduce with given

- **intPlusMonoid**: Given monoid for **Int**

```
val sum1 = reduceRight(lst)
```

(**intPlusMonoid**) implicitly

- **stringMonoid**: Given monoid for **String**

```
val str = reduceRight(lst.map(e => e.toString))
```

(**stringMonoid**) implicitly

- **intTimesMonoid**: explicitly

```
val prod1 = reduceRight(lst)(using Monoid.intTimesMonoid)
```

(**intTimesMonoid**) explicitly

- **customMonoid**: as new given monoid for String

```
given customMonoid : Monoid[String] = Monoid("", (s1, s2) => s"$s1 $s2")

val str3 = reduceRight(lst.map(e => e.toString))
```

(**customMonoid**) implicitly

Reducible: Providing Reduction Capabilities

Reducible[A]: Trait for structures which are reducible

```
trait Reducible[A] {

  def reduceMap[B](mapper : A => B)(using monoid: Monoid[B]) : B

  def reduce(using monoid: Monoid[A]) : A = reduceMap( a => a )
}
```

Companion object with factories

```
object Reducible {
  def apply[A](as: Iterable[A]): Reducible[A] =
```

Exercise 6

```
  def apply[A](tree: BinTree[A]) : Reducible[A] = ...
}
```

6 Functional Reduction

Monoids

Reduction with Monoids

Lifting Monoids

Parallel Reduction

Lifting Monoids

```scala
trait Monoid[M] {
  val zero: M
  def op(a: M, b: M) : M
}
```

From Monoid[A] to Monoid[Optional[A]]

```scala
object Monoid :
  def optionMonoid[A](using elemMonoid: Monoid[A]) : Monoid[Option[A]] =
    Monoid(None, (optA, optB) => {
      (optA, optB) match {
        case (None, None) => None
        case (Some(a), None) => optA
        case (None, Some(b)) => optB
        case (Some(a), Some(b)) => Some(elemMonoid.op(a, b))
      }
    })
  ...
```

- identity value **zero** = **None**
- combiner function **op**:
  - if both values are None => None
  - if one is None and the other Some(a) => Some(a)
  - if both are Some(a) and Some(b) => combine a and b with Monoid for elements and return it in a Some

Lifting Monoids

```
trait Monoid[M] {
  val zero: M
  def op(a: M, b: M) : M
}
```

From Monoid[A] to Monoid[Optional[A]]

```
object Monoid :
  def optionMonoid[A](using elemMonoid: Monoid[A]) : Monoid[Option[A]] =
    Monoid(None, (optA, optB) => {
      (optA, optB) match {
        case (None, None) => None
        case (Some(a), None) => optA
        case (None, Some(b)) => optB
        case (Some(a), Some(b)) => Some(elemMonoid.op(a, b))
      }
    })
  ...
```

Example: List of Option with points of students, None if not submitted

- each map represents results from one assignment

```
val listOfOptions = List(Some(7), None, Some(2), None, Some(8), Some(6))
val listOfOptionSum =
  Reducible(listOfOptions).reduce(using Monoid.optionMonoid(using Monoid.intPlusMonoid))
```

- with context parameters

```
given optionIntMonoid : Monoid[Option[Int]] = Monoid.optionMonoid
val listOfOptionSum2 = Reducible(listOfOptions).reduce
```

```
Some(23)
```

Lifting Monoids

```
trait Monoid[M] {
  val zero: M
  def op(a: M, b: M) : M
}
```

From Monoid[A] to Monoid[Map[A]]

```
object Monoid :
  def mapMonoid[K, V](using vMonoid: Monoid[V]) : Monoid[Map[K, V]] =
    Monoid(Map(), (mapA, mapB) => {
      var mapR = mapA
      for ((k, v) <- mapB) {
        if mapA.contains(k) then mapR = mapR.updated(k, vMonoid.op(v, mapA(k)))
        else mapR = mapR.updated(k, v)
      }
      mapR
    })
```

- identity value **zero = empty Map**
- combiner function **op** combining **mapA** and **mapB**
  - start with first map **mapA**
  - iterate over entries of **mapB**
    - if both maps contain entry with key => combine values of entries with with Monoid for elements
    - otherwise add entry from **mapB**

Lifting Monoids

```
trait Monoid[M] {
  val zero: M
  def op(a: M, b: M) : M
}
```

From Monoid[A] to Monoid[Map[A]]

```
object Monoid :
  def mapMonoid[K, V](using vMonoid: Monoid[V]) : Monoid[Map[K, V]] =
    Monoid(Map(), (mapA, mapB) => {
      var mapR = mapA
      for ((k, v) <- mapB) {
        if mapA.contains(k) then mapR = mapR.updated(k, vMonoid.op(v, mapA(k)))
        else mapR = mapR.updated(k, v)
      }
      mapR
    })
```

Example: List of maps with points of students

- each map represents results from one assignment

```
val assnResults : List[Map[String, Int]] =
  List(
    Map("Hans" -> 6, "Fritz" -> 7, "Anna" -> 9),
    Map("Hans" -> 8, "Anna" -> 7),
    Map("Fritz" -> 6, "Anna" -> 8)
  )
```

- compute total points for students

```
val totalPoints =
  Reducible(assnResults)
     .reduce(using Monoid.mapMonoid(
                        using Monoid.intPlusMonoid))
```

```
Map(Hans -> 14, Fritz -> 13, Anna -> 24)
```

Lifting Monoids

```
trait Monoid[M] {
  val zero: M
  def op(a: M, b: M) : M
}
```

From Monoid[A] to Monoid[Map[A]]

```
object Monoid :
  def mapMonoid[K, V](using vMonoid: Monoid[V]) : Monoid[Map[K, V]] =
    Monoid(Map(), (mapA, mapB) => {
      var mapR = mapA
      for ((k, v) <- mapB) {
        if mapA.contains(k) then mapR = mapR.updated(k, vMonoid.op(v, mapA(k)))
        else mapR = mapR.updated(k, v)
      }
      mapR
    })
```

Example: List of maps with courses of students in semesters

- each map represents results from one assignment

```
val coursesTaken =
  List(
    Map("Hans" -> Set(SW1, Math1, IS2),
        "Fritz" -> Set(SW1, BS, Math1)),
    Map("Hans" -> Set(SW2, Math2, BS),
       "Fritz" -> Set(SW2, IS2, Math2))
)
```

- compute set of all courses for students

```
val allCoursesTaken =
  Reducible(coursesTaken).reduce(
       using Monoid.mapMonoid(using Monoid.setMonoid))
```

```
Map(Hans -> HashSet(IS2, BS, Math1, Math2, SW1, SW2),
    Fritz -> HashSet(IS2, BS, Math1, Math2, SW1, SW2))
```

6 Functional Reduction

Monoids

Reduction with Monoids

Lifting Monoids

Parallel Reduction

Parallel Reduction

- Reduction by split and combine
- with Fork-Join-Pool

```
trait ParReducible[A] extends Reducible[A] {
  val THRESHOLD = 7
  def size : Int
  def split: (ParReducible[A], ParReducible[A])

  def parReduceMap[B](mapper : A => B)(using monoid: Monoid[B]) : B = {
    class Task(parAs: ParReducible[A]) extends RecursiveTask[B] {
      override def compute() : B = {
        if (parAs.size <= THRESHOLD) then {
          parAs.reduceMap(mapper)
        } else {
          val (as1, as2) = parAs.split
          val task1 = new Task(as1)
          val task2 = new Task(as2)
          task1.fork()
          task2.fork()
          monoid.op(task1.join(), task2.join)
        }
      }
    }
    val task = new Task(this)
    ForkJoinPool.commonPool.invoke(task)
  }

  def parReduce(using Monoid[A]) : A = parReduceMap(a => a)
}
```

Parallel Reduction

- Create parallel Reducible from Iterable

```
object ParReducible {

  def apply[A](as: Iterable[A]) : ParReducible[A] =
    new ParReducible[A] {

      def size = as.size

      def split: (ParReducible[A], ParReducible[A]) = {
        val (as1, as2) = as.splitAt(as.size / 2)
        (apply(as1), apply(as2))
      }

      override def reduceMap[B](mapper: A => B)(using monoid: Monoid[B]): B = {
        var b = monoid.zero
        for (a <- as) {
          b = monoid.op(b, mapper(a))
        }
        b
      }

      override def toString: String = as.toString()
    }
}
```

Parallel Reduction

- Application to list of numbers from 1 to 50

```
val ns = (1 to 50).toList

given m : Monoid[Int] = Monoid.intPlusMonoid
val sump = ParReducible(ns).parReduceMap( i => i )
```

- Trace
  - **split**: splitting of list
  - **seq**: sequential reduction

```
split: (List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25), List(26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, ... )
split: (List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12), List(13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25))
split: (List(26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37), List(38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50))
split: (List(1, 2, 3, 4, 5, 6), List(7, 8, 9, 10, 11, 12))
split: (List(26, 27, 28, 29, 30, 31), List(32, 33, 34, 35, 36, 37))
seq: (List(1, 2, 3, 4, 5, 6))
seq: (List(26, 27, 28, 29, 30, 31))
split: (List(38, 39, 40, 41, 42, 43), List(44, 45, 46, 47, 48, 49, 50))
seq: (List(7, 8, 9, 10, 11, 12))
seq: (List(38, 39, 40, 41, 42, 43))
seq: (List(32, 33, 34, 35, 36, 37))
split: (List(13, 14, 15, 16, 17, 18), List(19, 20, 21, 22, 23, 24, 25))
seq: (List(13, 14, 15, 16, 17, 18))
seq: (List(19, 20, 21, 22, 23, 24, 25))
seq: (List(44, 45, 46, 47, 48, 49, 50))
sump = 1275
```

Reductions in Scala Collections

Scala provides two types of methods for reduction:

- **fold** – uses initial value

both iterative

```
def foldLeft[B](z: B)(op: (B, A) => B): B
def foldRight[B](z: B)(op: (A, B) => B): B
```

```
val totalLength = words.foldLeft(0) {
  (length, word) => length + word.length
}
```

```
def fold[A1 >: A](z: A1)(op: (A1, A1) => A1): A1
```

```
val sum = numbers.fold(0) {(x, y) => x + y }
```

- **reduce**: uses first value as start
  - ☐ throws exception if empty

```
def reduceLeft[B >: A](op: (B, A) => B): B
def reduceRight[B >: A](op: (A, B) => B): B
def reduce[B >: A](op: (B, B) => B): B
```

```
val max: Int=
  numbers.reduce {
    (m, n) => if (n > m) n else m
  }
```

  - ☐ returns **Option** with **None** if empty

```
def reduceRightOption[B >: A](op: (A, B) => B) : Option[B]
def reduceLeftOption[B >: A](op: (B, A) => B): Option[B]
def reduceOption[B >: A](op: (B, B) => B): Option[B]
```

```
val optMax: Option[Int] =
  numbers.reduceOption {
    (m, n) => if (n > m) n else m
  }
```

Reductions in Java Streams: reduce

reduce with identity value, accumulator and combiner

- **accumulator**: sequential reduction with mapping of elements
- **combiner**: for combining results from parallel executions

```
public interface Stream<E>
  <U> U reduce(U identity,
               BiFunction<U, ? super T, U> accumulator,
               BinaryOperator<U> combiner)
```

Equivalent to **reduceMap** from **Reducible:**
- **identity** and **combiner** as in Monoid
- **accumulator** implements mapping elements and combining with current result

reduce with identity and combine function

- result value is same type as elements
- **combine** used for accumulation and combining results from parallel executions

```
  E reduce(E identity, BinaryOperator<E> combine)
```

reduce with identity and combine function

- starts with first element
- returns **Optional**

```
  Optional<E> reduce(BinaryOperator<E> combine);
```