Functional Design & Parser combinators

Lucas Nouguier

Polytech Montpellier

13 November 2023

Outline

ADTs

An Algrebraic Data Type is a composite type made of other types

Example

```
type List[A]   = Nil | Cons[A]
type Option[A] = None | Some[A]


sealed trait Tree
case class Name() extends Tree
case class Select(qual: Tree, name: Name) extends Tree
case class Method(fun: Tree, args: ArgClauses) extends Tree
case class If(p: Tree, thenp: Tree, elsep: Tree) extends Tree

case class ArgClauses(args: List[List[Tree]]) extends Tree
```

Failure handling

	Nulls	Exceptions	ADTs
Difficulty	Easy	Easy	Less easy
Failure reason	Unknown	Known	Known
Performances	Best	Bad	Better
Expressivity	Bad	Better	Best
Runtime failure	Possible	Possible	No

Benchmark	Mode	Count	Score	Error
No exceptions	avgt	10	0.046	± 0.003
Throw & catch	avgt	10	16.268	± 0.239
Throw & no catch	avgt	10	17.874	± 3.199
Throw w/o stacktrace	avgt	10	1.174	± 0.014

Table: *Comparison & Benchmarks (ms/op)*

Failure handling

	Nulls	Exceptions	ADTs
Difficulty	Easy	Easy	Less easy
Failure reason	Unknown	Known	Known
Performances	Best	Bad	Better
Expressivity	Bad	Better	Best
Runtime failure	Possible	Possible	No

Benchmark	Mode	Count	Score	Error
No exceptions	avgt	10	0.046	± 0.003
Throw & catch	avgt	10	16.268	± 0.239
Throw & no catch	avgt	10	17.874	± 3.199
Throw w/o stacktrace	avgt	10	1.174	± 0.014

Table: *Comparison & Benchmarks (ms/op)*

Parser combinators

Create a complex parser from simple ones

Example

```
val tpeSep: Parser[String]  // ( : ) | (: ) | ( :)
val alphas: Parser[Char]    // [a-zA-Z]
val name: Parser[String]    = alphas.repeat.combineAll

// Parser[Param] — x: Int
(name <* tpeSep).product(name).map(???)

// Parser[Queue[Param]] — // x: A, y: B, z: C
(paramParser <* paramSepParser).repeat
  .combineWith(Queue.empty)(_ :+ _)
  .product(paramParser.orElse(Parser.empty))
  .map(???)
```

Parser combinators — example

Given the following code, which parser can we create?

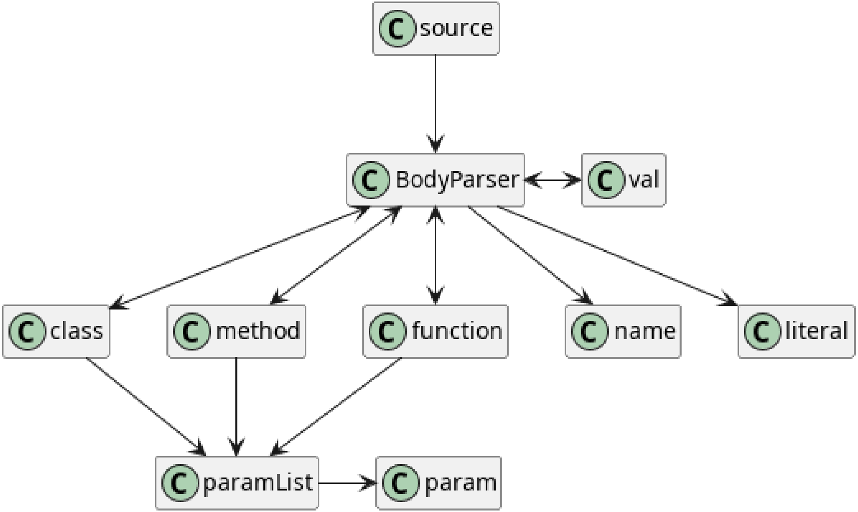
Example

```scala
package scalala

object Main:
  val x = 1
  val y = 2
  val adder = (a: Int, b: Int) => a + b

  def main(args: Array[String]): Unit =
    println(foo(x, y))

  def foo(x: Int, y: Int) = x + y
```

Parser combinators — example

Parser	Description	Example
`name`	A simple name	`x`, `y`, `main`, `foo`
`literal`	A literal value	`1`, `2`
`val`	A value definition	`val x = 1`
`param`	A method parameter	`a:Int`, `args:Array[String]`
`paramList`	A parameter list	`a:Int, b:Int`
`method`	A method definition	`main(args:Array[String]) =`
`function`	A function definition	`adder = (a:Int, b:Int) =>`
`class`	A class definition	`object Main`
`source`	A source file	`package scalala; ...`

Parser combinators — example



General approach

With $\mathcal{P}$ the parser algrebra and $\mathcal{A}$ the ADT algebra. I'll use reification to implement this parser, with 3 kinds of methods:

- Constructors c: $\forall x \notin \mathcal{P}, c(x) \in \mathcal{P}$
- Combinators f: $\forall c_1, c_2 \in \mathcal{P}, f(c_1, c_2) \in \mathcal{P}$
- Interpreters z: $\forall c \in \mathcal{P}, x \notin \mathcal{P}, z(c, x) \in \mathcal{A}$

Example (Reification methods)

```
// Constructor
def string(s: String): Parser[String]
// Combinator
def orElse[A, B](p1: Parser[A], p2: Parser[B]): Parser[A | B]
// Interpreter
Parser[A].parse(input: String): A
```

General approach

For each abstract concept, we'll have a concrete implementation (with a `case class`) and reification methods. For instance, for a simple string parser and a simple mapper:

Example (Reification classes)

```
trait Parser[+A]:
  def map[B](f: A => B): Parser[B] = ParserMap(this, f)
object Parser:
  def string(s: String): Parser[String] = ParserString(s)

case class ParserString(s: String) extends Parser[String]
case class ParserMap[A, B](
  source: Parser[A],
  f: A => B
) extends Parser[B]
```

Outline

Outline

by-value vs by-name

lazy vs eager

Given/using (implicit values)

`using` clause defines a value to be injected by the compiler, based on the expected type among the `given` values

Example

```
given life: Int = 42
def foo(using i: Int): Int = i

val meaningOfLife1 = foo              // : Int = 42
// rewritten as foo(life) by the compiler
val meaningOfLife2 = foo(using life)  // : Int = 42
// we can also explicitly pass the value
```

Variance

Definition

Describe the relation between generic types. Given a generic type $\mathcal{F}$:

Invariant $\forall A, B \quad A \neq B \Leftrightarrow \mathcal{F}[A] \neq \mathcal{F}[B]$

Covariant $\forall A, B \quad A <: B \Leftrightarrow \mathcal{F}[A] <: \mathcal{F}[B]$

Contravariant $\forall A, B \quad A <: B \Leftrightarrow \mathcal{F}[B] <: \mathcal{F}[A]$

It allows a more flexible design, but has some constraints for type-safety.
For a covariant type, we cannot use it's type param as method param type

Example (Covariance constraint)

```
class Foo[+A]:
  def foo[B >: A](x: B) = ??? // cannot use A as param type
```

Variance

Definition

Describe the relation between generic types. Given a generic type $\mathcal{F}$:

Invariant $\forall A, B \quad A \neq B \Leftrightarrow \mathcal{F}[A] \neq \mathcal{F}[B]$

Covariant $\forall A, B \quad A <: B \Leftrightarrow \mathcal{F}[A] <: \mathcal{F}[B]$

Contravariant $\forall A, B \quad A <: B \Leftrightarrow \mathcal{F}[B] <: \mathcal{F}[A]$

It allows a more flexible design, but has some constraints for type-safety.
For a covariant type, we cannot use it's type param as method param type

Example (Covariance constraint)

```
class Foo[+A]:
  def foo[B >: A](x: B) = ??? // cannot use A as param type
```

Outline

Why type classes?

Example (Inheritance)

```scala
trait Encoder      { def encode: String }
trait Combiner[A] { def combine(a: A): String }

abstract class Animal extends Encoder
  with Combiner[Animal]

case class Cat() extends Animal:
  override def encode(): String
  override def combine(b: Animal): String
case class Dog() extends Animal:
  override def encode(): String
  override def combine(b: Animal): String
```

Why type classes?

Example (Composition)

```
trait Encoder[A]  { def encode (a: A): String }
trait Combiner[A] { def combine(a: A, b: A): String }

abstract class Animal(
  encoder: Encoder[Animal],
  combiner: Combiner[Animal]
):
  def encode = encoder.encode(this)
  def combine(b: Animal) = combiner.combine(this, b)
```

Why type classes?

Example (Type class)

```
trait Encoder[-A] { def encode (a: A): String }

abstract class Animal

extension [A](a: A)(using encoder: Encoder[A])
  def encode: String = encoder.encode(a)

given catEncoder: Encoder[Cat] with
  def encode(a: Cat) = "A cat"

aCat.encode // "A cat"
// rewritten as encode(aCat, catEncoder)
```

Type class definition

1. Define a trait (the behavior)
2. Define your methods
3. Define your trait instances
4. (Optional) redefine methods as extension methods

Exercices

1. Define a type class `JsonEncoder` for a `case class Person` with a name, an age and an address
2. Create a `JsonEncoder` for a `List[T]`
3. Create a `JsonEncoder` for an `Option[T]`
4. Try it with a `List[Option[Person]]`

Given syntax help

```
given Type with
  // trait methods implementation
given [A]: Type[A] with
  // trait methods implementation
given [A](using otherGiven: OtherType[A]): Type[A] with
  // trait methods implementation
```

Exercices solution — Part 1

```
case class Person(...)

// 1. TC definition
trait JsonEncoder[-T]:
  def encode(t: T): String

// 3. TC instance for Person
given JsonEncoder[Person] with
  def encode(person: Person): String = ???

// 4. Redefine methods as extension methods
extension [T](t: T)(using encoder: JsonEncoder[T])
  def toJson: String = encoder.encode(t)
```

Exercices solution — Part 2

```
// TC instance for List[T]
given [T: JsonEncoder]: JsonEncoder[List[T]] with
  def encode(list: List[T]): String =
    list.map(_.toJson).mkString("[", ",", "]")

// TC instance for Option[T]
given [T: JsonEncoder]: JsonEncoder[Option[T]] with
  def encode(option: Option[T]): String =
    option.map(_.toJson).getOrElse("null")

List(Some(lulu), None, Some(zozo)).toJson
// : String = [
//   {"name":"lulu",...}
//   null
//   {"name":"zozo",...}
//]
```

Outline

Base types

Definition (Parser & Result ADTs)

```
sealed trait Parser[+A]:
  def parse(input: String): Result[A]
  protected def parse(input: String, index: Int): Result[A]

sealed trait Result[+A]:
  def map[B](f: A => B): Result[B]
  def orElse[B](that: => Result[B]): Result[A | B]

case class Success[+A](...) extends Result[A]
case class Failure extends Result[Nothing]
```

Functor

Problem Need to transform a value inside any kind of (unrelated) data structure

Definition (Functor)

```
trait Functor[F[_]]:
  def map[A, B](fa: F[A])(f: A => B): F[B]
```

- Single abstraction for any generic type
  - Valid types: `Functor[List]`, `Functor[Option]`...
  - Invalid types: `Functor[Int]`, `Functor[Person]`...
- A bit more verbose (can be hidden with extension methods)
- Easy to switch from a data structure to another

Functor

Problem Need to transform a value inside any kind of (unrelated) data structure

Definition (Functor)

```scala
trait Functor[F[_]]:
  def map[A, B](fa: F[A])(f: A => B): F[B]
```

- Single abstraction for any generic type
  - Valid types: `Functor[List]`, `Functor[Option]`...
  - Invalid types: `Functor[Int]`, `Functor[Person]`...
- A bit more verbose (can be hidden with extension methods)
- Easy to switch from a data structure to another

Functor — map implementation

1. New concept → new case class
   - Input: `Parser[A]` & `A => B`
   - Ouput: `Parser[B]`
2. No constructors, one combinator & interpreter

```
trait Parser[+A]:
  def map[B](f: A => B): Parser[B] = ParserMap(this, f)

final case class ParserMap[A, B](
  source: Parser[A],
  f: A => B
) extends Parser[B]:
  override def parse(input: String, index: Int): Result[B] =
    source.parse(input, index).map(f)
```

Functor — map implementation

1. New concept → new case class
   - Input: `Parser[A]` & `A => B`
   - Ouput: `Parser[B]`
2. No constructors, one combinator & interpreter

```
trait Parser[+A]:
  def map[B](f: A => B): Parser[B] = ParserMap(this, f)

final case class ParserMap[A, B](
  source: Parser[A],
  f: A => B
) extends Parser[B]:
  override def parse(input: String, index: Int): Result[B] =
    source.parse(input, index).map(f)
```

Monoid

Problem Need to combine any kind of values

Definition (Monoid)

```
trait Monoid[A]:
  def empty: A
  def combine(a: A, b: A): A
```

- Single abstraction for any type
- Can serve as a AND or OR operation
  - AND: `combine(Parser1,Parser2) = Parser1 then Parser2`
  - OR: `combine(Parser1,Parser2) = Parser1 orElse Parser2`
- Monoids without `empty` are called `Semigroups`

Monoid

Problem Need to combine any kind of values

Definition (Monoid)

```scala
trait Monoid[A]:
  def empty: A
  def combine(a: A, b: A): A
```

- Single abstraction for any type
- Can serve as a AND or OR operation
  - AND: `combine(Parser1,Parser2) = Parser1 then Parser2`
  - OR: `combine(Parser1,Parser2) = Parser1 orElse Parser2`
- `Monoids without empty are called Semigroups`

Monoid — orElse implementation

1. New concept → new case class
   - Input: `Parser[A]` & `Parser[B]`
   - Output: `Parser[A | B]`
2. No constructors, one combinator & interpreter

```
trait Parser[+A]:
  def orElse[B](that: => Parser[B]): Parser[A | B] =
    ParserOrElse(this, that)

final class ParserOrElse[A, B](
  left: Parser[A],
  right: => Parser[B]
) extends Parser[A | B]:
  def parse(input: String, index: Int): Result[A | B] =
    left.parse(input, index)
      .orElse(right.parse(input, index))
```

Monoid — orElse implementation

1. New concept → new case class
    - Input: `Parser[A]` & `Parser[B]`
    - Output: `Parser[A | B]`
2. No constructors, one combinator & interpreter

```
trait Parser[+A]:
  def orElse[B](that: => Parser[B]): Parser[A | B] =
    ParserOrElse(this, that)

final class ParserOrElse[A, B](
  left: Parser[A],
  right: => Parser[B]
) extends Parser[A | B]:
  def parse(input: String, index: Int): Result[A | B] =
    left.parse(input, index)
        .orElse(right.parse(input, index))
```

Semigroupal

Problem Need to operate on multiple values at once, without combining them (e.g. to instanciate a class)

Definition (Semigroupal)

```
trait Semigroupal[F[_]]:
  def product[A, B](fa: F[A], fb: F[B]): F[(A, B)]
```

- Single abstraction for generic type
- Different from `Semigroup`
- Merge two values into one to use later

Semigroupal

Problem Need to operate on multiple values at once, without combining them (e.g. to instanciate a class)

Definition (Semigroupal)

```
trait Semigroupal[F[_]]:
  def product[A, B](fa: F[A], fb: F[B]): F[(A, B)]
```

- Single abstraction for generic type
- Different from `Semigroup`
- Merge two values into one to use later

Semigroupal — product implementation

1. New concept → new case class
   - Input: `Parser[A]` & `Parser[B]`
   - Output: `Parser[(A, B)]`
2. No constructors, one combinator & interpreter

```
trait Parser[+A]:
  def product[T >: A, B](that: Parser[B]): Parser[(T, B)] =
    ParserProduct(this, that)

final case class ParserProduct[A, B](
  left: Parser[A],
  right: Parser[B]
) extends Parser[(A, B)]:
  def parse(input: String, index: Int): Result[(A, B)] =
    left.parse(input, index) match
      case fail: Failure => fail
      case Success(leftResult, _, offset) =>
        right.parse(input, offset).map((leftResult, _))
```

Semigroupal — product implementation

1. New concept → new case class
   - Input: `Parser[A]` & `Parser[B]`
   - Output: `Parser[(A, B)]`
2. No constructors, one combinator & interpreter

```scala
trait Parser[+A]:
  def product[T >: A, B](that: Parser[B]): Parser[(T, B)] =
    ParserProduct(this, that)

final case class ParserProduct[A, B](
  left: Parser[A],
  right: Parser[B]
) extends Parser[(A, B)]:
  def parse(input: String, index: Int): Result[(A, B)] =
    left.parse(input, index) match
      case fail: Failure => fail
      case Success(leftResult, _, offset) =>
        right.parse(input, offset).map((leftResult, _))
```

Semigroupal — Monadic Combinations

When the container is a `Monad`, the `.product()` does a cartesian product (and bypasses 'invalid' values like `None`)

Example (Monadic combination)

```
Semigroupal[List].product(
  List(1, 2),
  List(4, 5),
) // List((1, 4), (1, 5), (2, 4), (2, 5))

Semigroupal[Future].product(
  Future(throw new Exception),
  Future(throw new RuntimeException)
) // Failure(java.lang.Exception)
```

/!\ With `List` & `Future` being `Monads` /!\

Semigroupal — Applicative Combinations

When the container is not a `Monad`, the `.product()` keeps all the values (allows errors accumulation for instance)

Example (Applicative combination)

```
Semigroupal[???].product(
  Validated.invalid(List("Badness")),
  Validated.invalid(List("Fail"))
) // Invalid(List(Badness, Fail))

Semigroupal[???].product(
  Validated.valid("Good"),
  Validated.valid(47),
) // Valid("Good", 47)
```

/!\ With `Validated` <u>not</u> being a `Monad` /!\

Applicative

Problem Need to apply a independent effects to value(s) of a container

Definition (Applicative definition)

```scala
trait Applicative[F[_]]:
  def pure[A](a: A): F[A]
  def ap[A, B](ff: F[A => B])(fa: F[A]): F[B]
```

- Single abstraction for generic type
- Gives access to the `mapN` method, easier to manipulate than `product`

Applicative

Problem Need to apply a independent effects to value(s) of a container

Definition (Applicative definition)

```
trait Applicative[F[_]]:
  def pure[A](a: A): F[A]
  def ap[A, B](ff: F[A => B])(fa: F[A]): F[B]
```

- Single abstraction for generic type
- Gives access to the `mapN` method, easier to manipulate than `product`

Applicative — Example

Reminder: we cannot use `.flatMap`, it is not defined

Example (Usage)

```
val f: (Int, Int) => Int = _ + _
val intList1 = List(5, 10, 15)
val intList2 = List(0, 1)

val adder = intList1.map(i1 => (i2: Int) => f(i1, i2))
// List(i2 => f(5, c), i2 => f(10, c), i2 => f(15, c))
adder.ap(intList2)
// List(5, 6, 10, 11, 15, 16)
```

Monad

Problem Need to chain operations on a same-kind container

Definition (Monad definition)

```
trait Monad[F[_]]:
  def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]
```

- Single abstraction for any generic type
- Continue the chain only on success
- Stop the chain on first failure

Monad

Problem Need to chain operations on a same-kind container

Definition (Monad definition)

```
trait Monad[F[_]]:
  def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]
```

- Single abstraction for any generic type
- Continue the chain only on success
- Stop the chain on first failure

Monad — flatMap implementation

1. New concept → new case class
   - Input: `Parser[A]` & `A => Parser[B]`
   - Output: `Parser[B]`
2. No constructors, one combinator & interpreter

```
trait Parser[+A]:
  def flatMap[B](f: A => Parser[B]): Parser[B] =
    ParserFlatMap(this, f)

final case class ParserFlatMap[A, B](
  source: Parser[A],
  f: A => Parser[B]
) extends Parser[B]:
  def parse(input: String, index: Int): Result[B] =
    source.parse(input, index) match
      case fail: Failure => fail
      case Success(result, input, offset) =>
        f(result).parse(input, offset)
```

Monad — flatMap implementation

1. New concept → new case class
   - Input: `Parser[A]` & `A => Parser[B]`
   - Output: `Parser[B]`
2. No constructors, one combinator & interpreter

```scala
trait Parser[+A]:
  def flatMap[B](f: A => Parser[B]): Parser[B] =
    ParserFlatMap(this, f)

final case class ParserFlatMap[A, B](
  source: Parser[A],
  f: A => Parser[B]
) extends Parser[B]:
  def parse(input: String, index: Int): Result[B] =
    source.parse(input, index) match
      case fail: Failure => fail
      case Success(result, input, offset) =>
        f(result).parse(input, offset)
```

Summary

Functor	transform a value inside a container
Semigroupal	tuple values from any containers
Applicative	apply a independent effects to value(s) of a container
Monad	chain operations on same-kind containers
Monoid	combine values

Type class hierarchy

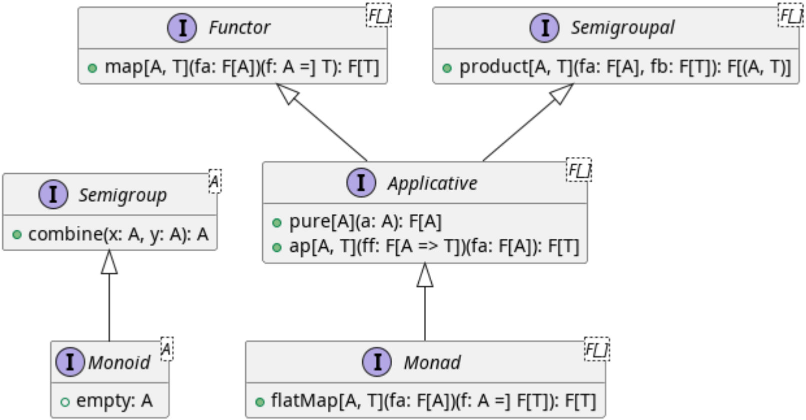


Figure: *Hierarchy*

Outline

Outline

Binary expression ADT

Definition

```
sealed trait Expr:
  def +(that: Expr): Expr

case class Num(value: Int) extends Expr
case class Var(name: String) extends Expr
case class Add(left: Expr, right: Expr) extends Expr
```

Addition parser

Given the following parsers:

number	[0–9]+	plus	[ ]*+[ ]*
variable	[a-zA-Z]+	expr	variable | number

```
val parser0: Parser[Add] =
  (expr, plus, expr).mapN((l, _, r) => l + r)
parser0.parse("x + 1 + a").get // Add(Var("x"), Num(1))

val parser1: Parser[Add] =
  (expr, plus, parser1 orElse expr).mapN((l, _, r) => l + r)
parser1.parse("x + 1 + a").get // NullPointerException

def parser2: Parser[Add] =
  (expr, plus, parser2 orElse expr).mapN((l, _, r) => l + r)
parser2.parse("x + 1 + a").get // StackOverflowError
```

Addition parser

Solution Delay the parser's evaluation

```
// In Parser.scala
object Parser:
  def lzy[A](parser: => Parser[A]) = Delayed(parser)

class Delayed[A](p: => Parser[A]) extends Parser[A]:
  lazy val cached = p
  def parse(input: String, index: Int): Result[A] =
    cached.parse(input, index)

// Addition parser
val parser3: Parser[Add] =
  (expr, plus, lzy(parser3) orElse expr).mapN((l,_,r) => l+r)

parser3.parse("x + 1 + a").get
// : Add = Add(Var("x"), Add(Num(1), Var("a")))
```

Json parser

You can now try to implement a JSON parser:D