Programming with Algebraic Data Types

Jiří Jakeš

March 7, 2017

Scala Taiwan Meetup

Algebraic Data Types

Algebra

- part of mathematics
- mathematical structure (symbols and operations, e. g. $6x^2 + 3x + 5 = 14$)

Algebraic data type

= type formed by combining multiple types

Combining data types

- *product*: "type P is formed by both A and B"
- *coproduct (sum)*: "type C is A or B"

Product in Scala

```scala
type P = (A, B)
```

Product in Scala

```scala
type P = (A, B)

case class P(a: A, b: B)
```

Coproduct (sum) in Scala

```scala
type C = Either[A, B]
```

Coproduct (sum) in Scala

```scala
type C = Either[A, B]

sealed trait C
case class C1(a: A) extends C
case class C2(b: B) extends C
```

Also: tagged union, variant record, choice type, discrimination union, disjoint union

More combining

```scala
sealed trait C
case class C1(a: A, b: B) extends C
case class C2(t: T, u: U, v: V) extends C
```

Alegbraic data types

- types formed by combining multiple types
- potentially recursive coproduct of products

Nomenclature

type

variants

```
sealed trait C
case class C1(a: A, b: B) extends C
case class C2(t: T, u: U, v: V) extends C
```

constructors fields

Pattern matching

```scala
sealed trait C
case class C1(a: A, b: B) extends C
case class C2(t: T, u: U, v: V) extends C

def toString(c: C): String = ???
```

Pattern matching

```scala
sealed trait C
case class C1(a: A, b: B) extends C
case class C2(t: T, u: U, v: V) extends C

def toString(c: C): String = c match {
  case C1(a, b) ⇒ s"C1:$a,$b"
  case C2(t, u, v) ⇒ s"C2:$t,$u,$v"
}
```

Pattern matching

```scala
sealed trait C
case class C1(a: A, b: B) extends C
case class C2(t: T, u: U, v: V) extends C

def toString(c: C): String = c match {
  case C1(a, b) ⇒ s"C1:$a,$b"
  case C2(t, u, v) ⇒ s"C2:$t,$u,$v"
}

val toString: C ⇒ String = {
  case C1(a, b) ⇒ s"C1:$a,$b"
  case C2(t, u, v) ⇒ s"C2:$t,$u,$v"
}
```

```scala
def toString(c: C): String = c match {
  case C1(a, b) ⇒ s"C1:$a,$b"
}
```

```scala
def toString(c: C): String = c match {
  case C1(a, b) ⇒ s"C1:$a,$b"
}
```

```
warning: match may not be exhaustive.
It would fail on the following input: C2(_,_,_)
```

Examples

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

Examples

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

sealed trait Either[A, B]
case class Left[A, B](a: A) extends Either[A, B]
case class Right[A, B](b: B) extends Either[A, B]
```

Examples

```scala
sealed trait List[A]
case class Cons[A](h: A, t: List[A]) extends List[A]
case object Nil extends List[Nothing]

sealed trait Tree[A]
case object Empty extends Tree[Nothing]
case class Leaf[A](v: A) extends Tree[A]
case class Node[A](l: Tree[A], r: Tree[A]) extends Tree[A]
```

Domain models

Modelling customers

- Every customer must have at least one contact
- Contact can be email, address or phone

Contact can be email, adress or phone

```
sealed trait Contact
case class Email( ... ) extends Contact
case class Address( ... ) extends Contact
case class Phone( ... ) extends Contact
```

Every customer must have at least on contact

```
case class Customer(
  name: String,
  primaryContact: Contact,
  otherContacts: List[Contact]
)
```

Every customer must have at least on contact

```
case class NonEmptyList[A](h: A, t: List[A])
```

Every customer must have at least on contact

```scala
case class NonEmptyList[A](h: A, t: List[A])

case class Customer(
  name: String,
  contacts: NonEmptyList[Contact]
)
```

Algebraic data types can help us increase type safety of code.

Algebraic data types can help us increase type safety of code.

Make illegal states unrepresentable.

😟 and 😃

😟 Variants are types (in Scala)

```scala
val a: ??? = List(Some(1), Some(2))
```

😟 Variants are types (in Scala)

```scala
val a: List[Some[Int]] = List(Some(1), Some(2))
```

😟 Variants are types (in Scala)

```scala
val a: List[Some[Int]] = List(Some(1), Some(2))

val b: ??? = List(None, None)
```

😟 Variants are types (in Scala)

```scala
val a: List[Some[Int]] = List(Some(1), Some(2))

val b: List[None.type] = List(None, None)
```

Ideally we want `Some(1)` to be of type `Option[Int]`. Can be "solved" by smart constructor or by explicitly providing type.

😟 Verbose definition

```scala
sealed trait Option[T]
final case class Some[T](x: T) extends Option[T]
final case object None extends Option[Nothing]
```

😟 Variants with types, verbose syntax

```scala
// Scala 3 (Dotty)
enum Option[T] {
  case Some(x: T)
  case None
}

val l: List[Option[Int]] = List(Some(1), Some(2))
```

😃 Accurate model

See customers and contacts.

😃 Self documenting

```
sealed trait Contact
case class Email( ... ) extends Contact
case class Address( ... ) extends Contact
case class Phone( ... ) extends Contact
```

😃 Compiler helps

```
def toString(c: Contact): String = c match {
  case Email( ... ) ⇒ "Email"
  case Address( ... ) ⇒ "Address"
}
```

```
warning: match may not be exhaustive.
It would fail on the following input: Phone(_)
```

😃 Illegal state unrepresentable

You cannot create an invalid value.

😃 Anemic domain model

- Separation between logic and data
- No state (allows easier scalability)
- Easier pickling, ORM
- Modularization
- Simpler testing

JSON Driven Development

```
{ "id": "a01", "car": { "make": "Škoda" } }
{ "id": "a02", "bicycle": { "gears": 24 } }
```

```json
{ "id": "a01", "car": { "make": "Škoda" } }
{ "id": "a02", "bicycle": { "gears": 24 } }
```

```scala
case class Car(make: String)
case class Bicycle(gears: Int)

case class Response(
  id: String,
  car: Option[Car],
  bicycle: Option[Bicycle]
)
```

```json
{ "id": "a01", "car": { "make": "Škoda" } }
{ "id": "a02", "bicycle": { "gears": 24 } }
```

```scala
sealed trait Vehicle
case class Car(make: String) extends Vehicle
case class Bicycle(gears: Int) extends Vehicle

case class Response(id: String, vehicle: Vehicle)
```

ADT & Mathematics

```
type A = (Boolean, Boolean, Boolean, Boolean)
```

```
type A = (Boolean, Boolean, Boolean, Boolean)// 16 inhabitants
type B = (Boolean, Boolean, Boolean)
```

```
type A = (Boolean, Boolean, Boolean, Boolean)// 16 inhabitants
type B = (Boolean, Boolean, Boolean)         // 8 inhabitants
type C = (Boolean, Boolean)
```

```
type A = (Boolean, Boolean, Boolean, Boolean)// 16 inhabitants
type B = (Boolean, Boolean, Boolean)          // 8 inhabitants
type C = (Boolean, Boolean)                   // 4 inhabitants
type D = (Boolean)
```

```
type A = (Boolean, Boolean, Boolean, Boolean)// 16 inhabitants
type B = (Boolean, Boolean, Boolean)         // 8 inhabitants
type C = (Boolean, Boolean)                  // 4 inhabitants
type D = (Boolean)                           // 2 inhabitants
type E = ()
```

```
type A = (Boolean, Boolean, Boolean, Boolean)// 16 inhabitants
type B = (Boolean, Boolean, Boolean)         // 8 inhabitants
type C = (Boolean, Boolean)                  // 4 inhabitants
type D = (Boolean)                           // 2 inhabitants
type E = ()                                  // 1 inhabitant
                // type: Unit, value: ()
```

```
type A = (Boolean, Unit)
```

```
type A = (Boolean, Unit)          // (T, ()), (F, ())
type B = (Boolean, Unit, Unit)
```

```
type A = (Boolean, Unit)             // (T, ()), (F, ())
type B = (Boolean, Unit, Unit)   // (T, (), ()), (F, (), ())
type C = (Boolean, Unit, Unit, Unit)
```

$$\overbrace{\texttt{(Boolean, Boolean, Boolean)}}^{2\times2\times2=8}$$

```
type A = (Boolean, Boolean, Boolean)

type B = (Boolean, Unit, Unit)
```

$$\underbrace{\texttt{(Boolean, Unit, Unit)}}_{2\times1\times1=2}$$

Product corresponds to multiplication.

```scala
sealed trait T
case class A(b: Boolean) extends T
case class B(b: Boolean) extends T
case class C(b: Boolean) extends T
```

```scala
sealed trait T
case class A(b: Boolean) extends T
case class B(b: Boolean) extends T
case class C(b: Boolean) extends T
```

```scala
A(true), B(true), C(true),
A(false), B(false), C(false)
```

```scala
sealed trait T
case class A(b: Boolean) extends T
case class B(b: Boolean) extends T
case class C(b: Boolean) extends T
```

$2 + 2 + 2 = 6$

```scala
A(true), B(true), C(true),
A(false), B(false), C(false)
```

```scala
sealed trait T
case class A(b: Boolean) extends T
case class B(b: Boolean) extends T
case class C(n: Nothing) extends T
```

```scala
sealed trait T
case class A(b: Boolean) extends T
case class B(b: Boolean) extends T
case class C(n: Nothing) extends T
```

```scala
A(true), B(true),
A(false), B(false)
```

Coproduct corresponds to addition.

```
type T = (Boolean, Boolean, Nothing)
```

$$\texttt{type T = (}\overbrace{\texttt{Boolean, Boolean, Nothing}}^{2 \times 2 \times 0 = 0}\texttt{)}$$

Algebraic data types form *semiring* (半環).

Semiring is a set equipped with two binary operations $+$ and $\cdot$, called addition and multiplication, such that:

- addition is commutative with identity element 0

  $$a + b = b + a \qquad a + 0 = a$$

- multiplication has identity element 1

  $$a \cdot 1 = a$$

- multiplication is distributive

  $$a \cdot (b + c) = (a \cdot b) + (a \cdot c)$$

- multiplication by 0 annihilates

  $$a \cdot 0 = 0$$

0	`Nothing`
1	`Unit`
$a + b$	`Either[A, B]` (or sealed trait)
$a \cdot b$	`(A, B)` (or case class)
$1 + 1 = 2$	`Boolean`
$1 + a$	`Option[A]`
$a \cdot (b + c)$	`(A, Either[B, C])`
$a \cdot b + a \cdot c$	`Either[(A, B), (A, C)]`
b^a	`A => B`

Thank you for attention