Compossible

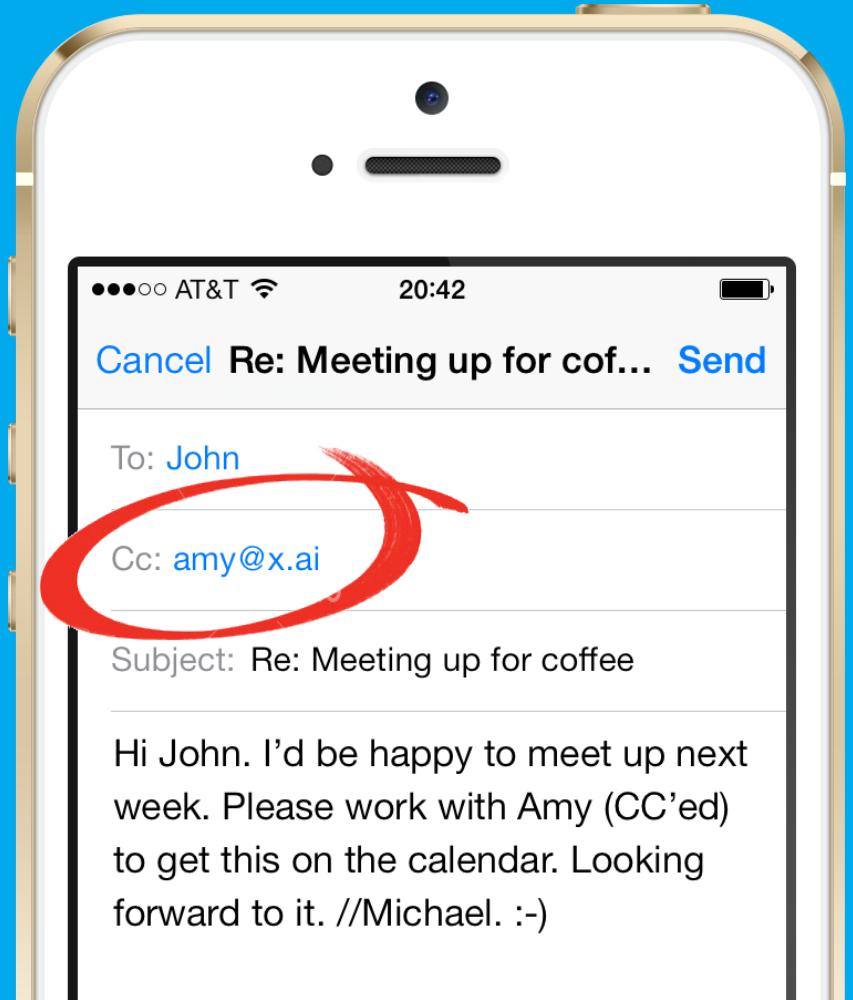
Extensible Records and Type-Indexed Maps

Jan Christopher Vogt / @cvogt

SCALADAYS MARCH 16TH - 18TH SAN FRANCISCO

x.ai

a personal assistant who schedules meetings for you



~~Scala Records~~

Compossible

Extensible Records and Type-Indexed Maps

Talk contents

- why care?
- features by examples
- implementation under the hood
- other implementations

Extensible records

Anonymous classes on steroids

Extensible Records

"Anonymous case classes", similar to `new{...}`

- ad-hoc structured data
- type-safe
- no reflection

Concise structural data transformations!

- like .copy(...) but also ++, etc.

Concise transformations

Pain

```
val e = Email(body=...)

EmailWithHeaders(
  body = e.body,
  headers = ...
)
```

Solution

```
val r = Record
(body=...)

r ++ Record
(headers=...)
```

Operations

Record creation

```
// Scala
val s = new{
  def name = "Chris"
  def age = 99 }
```

```
// Compossible
val r = Record(
      name = "Chris",
      age = 99 )
```

Human writable, familiar types

```
// Scala

s: { def name: String
     def age:  Int }
```

```
// Compossible
r: Record[
   { def name: String
     def age:  Int }
]
```

Non-reflective field access

```
// Scala

// reflective call
s.name
```

```
// Compossible

// no reflective call
r.name
```

No 22 field limit

```
Record(
   f1= …,
   …
   f1000 = …,
   …
)
```

Copy

```
val r = Record(name="Chris", age=99)

val r2 = r.copy(name="Miguel")

r2.name // "Miguel"
r2.age  // 99
```

Merge / Extend

```
val r1 = Record( name="Chris" )
val r2 = Record( age=99 )
val r3 = r1 ++ r2

r3.name
r3.age
```

From case class

```
case class Person(name: String)
val p = Person("Chris")
val r = Record.from(p) ++ Record(age=99)
r.name
r.age
```

To case class

```
case class Person(name: String, age: Int)

val r = Record(age=99, name="Chris")

r.to[Person]
```

Magic conversions (opt-in)

```
import compossible.conversions._
case class Person(name: String)
val p: Person = Record(name="Chris")
val r = p ++ new {def age=99}
r.name
r.age
```

Easy transformations

```
case class Person(name: String)
case class AgedPerson(
  name: String, age: Int)

val p = Person("Chris")
val p2 = (p ++ Record(age=99))
         .to[AgedPerson]
```

Select

```
val r = Record(
  name="Chris", age=99, address="NYC"
)

val r2 = r(select name & age)
r2.name
r2.address // type-error
```

Blackbox fallback (code completion)

```
val r = Record(new{
  def name = "Chris"
  def age = 99
})
val r2 = r.select[{def name: String}]
r.name
r.age // type-error
```

Json deserialization (adhoc and 22+)

```
val r = Json.parse( jsonString ).as[
   Record[{
      def name: String
      def age: Int
   }]
]
```

TODO

Structural pattern matching

```
val r = Record(name="Chris", age=99)
r match {
  case Record(name) =>
  case Record(age) =>
  case Record(age,name) =>
  case Record(name,age) =>
}
```

Structurally match classes

```
val p = Person(name="Chris", age=99)
p match {
  case Record(name) =>
  case Record(age) =>
  case Record(age,name) =>
  case Record(name,age) =>
}
```

Rename

```
val r = Record(
  age=99
)

val r2 = r(rename age to years)
r2.years
r2.age // type-error
```

Remove

```
val r = Record(
  name="Chris", age=99
)

val r2 = r(remove name)
r2.name // type error
```

Slick projections

```
// before
PersonTable.map(p => (p.name,p.age))
           .filter( _._2 > 18)

// using records:
PersonTable.map(_(select name & age))
           .filter( _.age > 18)
```

Patterns

Class-like

```
type Person = Record[{
  def name: String
  def age: Int
}]

implicit class PersonMethods(val p: Person) extends AnyVal{
  def isMinor = p.age <= 18
}
```

Other implementations

What about Shapeless records?

- most mature & complete implementation
- based on HLists & String singleton types
  - extensive use of implicits
  - clean theory
  - full code completion (in theory)
  - compile time overhead
  - verbose, non-human writable types
  - less familiar than new{ def name: String }

What about Scala Records (EPFL)?

- no transformation at the moment
- type-safe copy-less views on dynamic data
- supports specialization
- based on structural refinement types
  - familiar Syntax: Rec{ def name: String }
  - non-human writable types
  - whitebox macros: no IntelliJ support

Project may merge with Compossible Records

Compossible Records

- not stable yet (goal: very soon)
- based on phantom structural types
  - familiar Syntax: Record[{ def name: String }]
  - fast through macros
  - mostly blackbox macros: IntelliJ support
  - whitebox in few places:
    no IntelliJ support for few operations
  - May merge with Scala Records at some point

Type-indexed Maps

Intersection types as Type-Sets

for type-safety, dependencies and effects

Creation and lookup

```
val m: TMap[String] = TMap("Chris")

m[String] // "Chris"
```

Merging

```
val m: TMap[String with Int]
     = TMap("Chris") ++ TMap(99)

m[String] // "Chris"
m[Int]    // 99
```

Let’s implement one

… live coding ...

Dependencies with TMap + Reader

```
def m1: TMap[Database] => …
def m2: TMap[Logger] => …
def m12: TMap[Database with Logger] =>
…
  = for{… <- m1; … <- m2} yield …

m12( TMap(logger) ++ TMap(database) )
```

Type-safe validation

See John Pretty’s talk tomorrow

Problem & TODO

Problem: TMap[+T] and subtyping

```
val m = TMap("Chris") ++ TMap(99)

m[Any] // ???
TMap(5) ++ TMap(6) // ???
m: TMap[Any] // ???

// note: TMap[+T]
```

Solution: `TMap[T]` + conversion macros

```
val m = TMap("Chris") ++ TMap(99)


m[Any] // type-error
TMap(5) ++ TMap(6) // type-error
m: TMap[Any] // type-error
```

Behind the Scenes

Structural, Phantom, Intersection Types

and Macros

Structural types

```
{
  def name: String
  def age: Int
}
```

Phantom Types

```
Record[{
  def name: String
  def age: Int
}]

TMap[Int]
```

Structural types without reflection

```
val r: Record[{def name: String}] = Record(name="Chris")

implicit def materialize[T](r: Record[T]): T = macro ...

r.name
// ==> materialize(r).name
// ==> new{
        val values$ : Map[String, Any] = r.values$
        def name = macro lookup[String]
        …
      }.name
// ==> r.values$("name").asIntanceOf[String]
```

Merging Record: Intersection type

```
val r1: Record[{def name: String}]
val r2: Record[{def age: Int}]

val r3: Record[{def name: String}
           with {def age: Int}]
  = r1 ++ r2
```

Wait, didn't we want this?

```
val r3: Record[{def name: String}
          with {def age: Int}]
        Record[{def name: String
                 def age: Int}]
  = r1 ++ r2
```

Intersection Types

```
type A
type B

(A with B) <:< A
(A with B) <:< B
```

Structural, Intersection Types

```
type A = {def a:Int}
type B = {def b: String}

(A with B) <:< A
(A with B) <:< B
(A with B) =:= { def a:Int
                 def b:String }
```

You wan’it, you got’it

```
val r3:

          Record[{def name: String
                  def age: Int}]
  = r1 ++ r2
```

TMap Reader: Intersection type

```
def m1: TMap[A] => ...
def m2: TMap[B] => ...
def m12: TMap[A] with TMap[B] => ...

  = for{... <- m1; ... <- m2} yield ...
```

TMap Reader: Intersection type

```
def m1: TMap[A] => …
def m2: TMap[B] => …
def m12: TMap[A] with TMap[B] => …
         TMap[A with B] => …
  = for{… <- m1; … <- m2} yield …
```

Intersection types and variance

```
type A                          // note: TMap[+T]
type B

TMap[A with B] <:< TMap[A] with TMap[B]

TMap[A with B] => X
  >:> TMap[A] with TMap[B] => X
```

You wan’it, you got’it

```
def m12: TMap[A with B] => …

  = for{… <- m1; … <- m2} yield …
```

Summary

Records

- ad-hoc type-safe structured data
- concise transformations

TMaps

- type-safe maps of
  values or dependencies or effects
- inference/tracking via merging

Thank you to

- Scala Records team:
  Vojin Jovanovic, Tobias Schlatter, Heather Miller, Hubert Plocinicak

- Guillaume Massé

- Jon Pretty

chris @ human.x.ai

Senior Backend Engineer

we are hiring!

Twitter: @cvogt @xdotai
Github: cvogt/compossible