Maia

Composble, type-level web API design

by **Joseph Abrahamson**

me@jspha.com | github.com/tel | @sdbo

http://github.com/MaiaOrg

Maia is a *API construction kit* for building composable APIs "at the type level".

Two big tricks

Composble APIs leads to API reuse and organic growth.

Type-level API specifications many things can't ever go wrong.

Maia guarantees safe passage between well-typed fortresses

1. Problem

2. Typing a solution

3. Tradeoffs and status

Why bother with types anyway?

$$P(a, b, c) \propto \texttt{P[A, B, C]}$$

$$(\text{some proof}) \vdash P(a, b, c)$$

(some program): `P[A, B, C]`

```
Int      Unit      String => String      Any
```

The best way to unlock and communicate the meaning of a program is to give it an interesting type.[1]

[1] Joseph Tel Abrahamson, Lambda Conf 2017, right now

```
def serve[YourApi <: Api](h: Handler[YourApi]):
  Request[YourApi] => Response[YourApi]
```

A problem

REST APIs are tough to manage

Do you write down your API spec? Do you maintain that documentation? Is it *complete*?

How do you know? Has your server or client drifted?
How do those teams communicate?

Or: Have you ever written

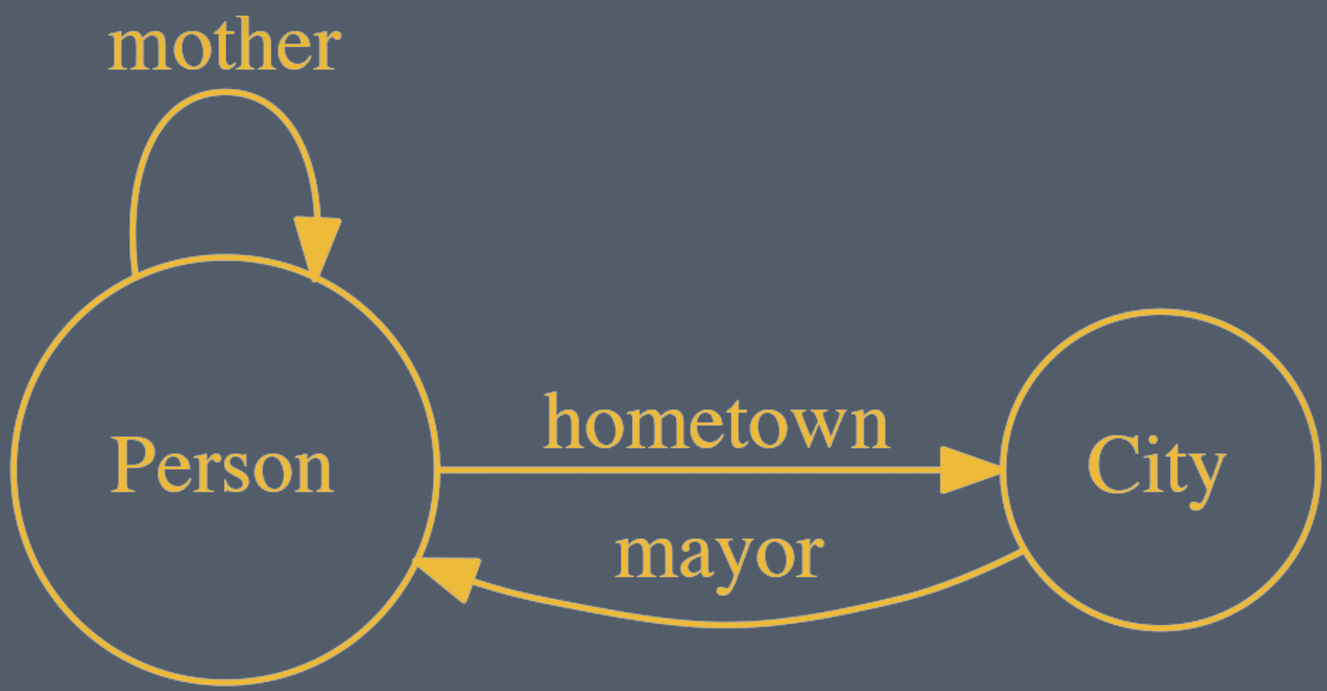
`api/path/path/data_specifically_for_the_user_profile_view.json`?

Have you ever wanted to?

REST APIs are complex, underspecified, and often don't solve client use cases around data fetching

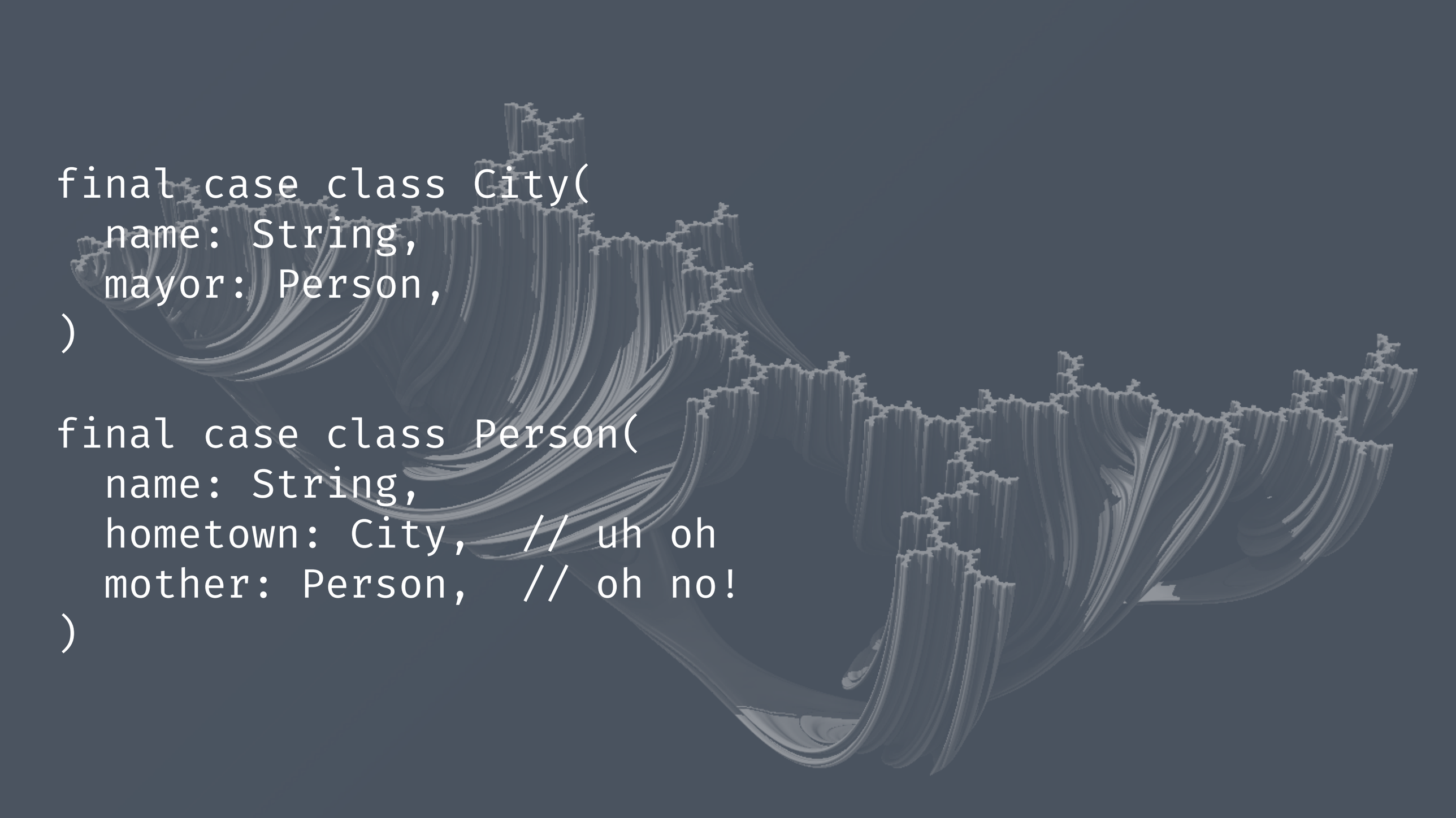
A solution

A new API architecture, typed


mother
Person
hometown
mayor
City

```
final case class City(
  name: String,
  mayor: Person,
)

final case class Person(
  name: String,
  hometown: City,  // uh oh
  mother: Person,  // oh no!
)
```



```
final case class City(
  name: String,
  mayor: Person,
)

final case class Person(
  name: String,
  hometown: City,  // uh oh
  mother: Person,  // oh no!
)
```

Request and response separation

```
final case class CityRequest(
  name: Boolean,
  mayor: Option[PersonRequest]
)

final case class PersonRequest( // <3
  name: Boolean,
  hometown: Option[CityRequest],
  mother: Option[PersonRequest]
)
```

```
val cityReq = CityRequest(name = true)

val personReq = PersonRequest(
  name = true,
  hometown = Some(cityReq),
  mother = Some(PersonRequest(name = true))
)
```

```
final case class CityResponse(
  name: Option[String],
  mayor: Option[PersonResponse]
)

final case class PersonResponse( // <3 <3 <3
  name: Option[String],
  hometown: Option[CityResponse],
  mother: Option[PersonResponse]
)
```

```
val cityResp = CityResponse(
  name = Some("Atlanta")
)

val personResp = PersonResponse(
  name = Some("Joseph"),
  hometown = Some(cityResp),
  mother = Some(PersonResponse(name = "Christine"))
)
```

We can go crazy: some ideas to go further

Handlers

```
import fs2.Task

final case class PersonHandler(
  name: Task[String],
  hometown: Task[CityResponse],
  mother: Task[Person]
)

def serve(h: PersonHandler):
  PersonRequest => PersonResponse = ???
```

Local errors

```
sealed trait ApiError
final case object UnknownRelation extends ApiError

final case class PersonResponse(
  ...
  mother: Option[Either[UnknownRelation, PersonResponse]]
  ...
)
```

Multiplicities

```
final case class PersonResponse(
  ...
  friends: Option[Vector[PersonResponse]]
  ...
)
```

Arguments

```
final case class CityHandler(
  ...
  mayor: Year => Task[Either[ApiError, PersonResponse]]
  ...
)
```

A rich set of tools for building composable, typed APIs...

… with loads of boilerplate >_<

Abstracting over templates

```
final case class PersonRequest(
  name: Boolean,
  hometown: Option[CityRequest],
  mother: Option[PersonRequest]
)
```

```
final case class PersonResponse(
  name: Option[String],
  hometown: Option[CityResponse],
  mother: Option[PersonResponse]
)
```

```
final case class Person...(
  name: ... String ...,
  hometown: ... City ...,
  mother: ... Person ...
)
```

```
final case class Person[F[_]](
  name: F[String],
  hometown: F[City],
  mother: F[Person]
)
```

```
final case class Person[X <: Dsl](
  name: X#Atom[String],
  hometown: X#Obj[City],
  mother: X#Obj[Person]
)
```

```
trait Dsl {
  type Atom[A]
  type Obj[T[_ <: Dsl]]
}
```

```
sealed trait Request extends Dsl {
  type Atom[A] = Boolean
  type Obj[T[_ <: Dsl]] = Option[T[Request]]
}

object Request extends Request
```

```
sealed trait Response extends Dsl {
  type Atom[A] = Option[A]
  type Obj[T[_ <: Dsl]] = Option[T[Response]]
}

object Response extends Response
```

```
val personReq = PersonRequest(
  name = true,
  hometown = Some(cityReq),
  mother = Some(PersonRequest(name = true))
)

val personResp = PersonResponse(
  name = Some("Joseph"),
  hometown = Some(cityResp),
  mother = Some(PersonResponse(name = "Christine"))
)
```

```
val personReq = Person[Request](
  name = true,
  hometown = Some(cityReq),
  mother = Some(PersonRequest(name = true))
)

val personResp = Person[Response](
  name = Some("Joseph"),
  hometown = Some(cityResp),
  mother = Some(PersonResponse(name = "Christine"))
)
```

```
package object maia {
  type Request[T[_ <: Dsl]] = T[form.Request]
  type Response[T[_ <: Dsl]] = T[form.Response]
}

val personReq = Request[Person](...)
val personResp = Response[Person](...)
```

```
def serve[T[  <: Dsl](h: Handler[T]):   // :D
  Request[T] => Response[T] = ???
```

Implementing type-level libraries

Field Spec language

```
final case class Toplevel[X <: Dsl](
  ...
  getPerson: X#ObjK[Person, HasArg[Person.PID], NoErr, One],
  getAllPersons: X#ObjK[Person, NoArg, NoErr, Many]
  ...
)
```

Type-Level Functions

```
sealed trait Response extends Dsl {
  ...
  type ObjK[T[_ <: Dsl], As <: ArgSpec, Es <: ErrSpec, C <: Size] =
      As#Provide[Es#Provide[C#Coll[maia.Response[T]]]]
  ...
}
```

Implicit resolution

```
def combineRequests[T[_ <: Dsl]](r1: Request[T], r2: Request[T])
  (implicit combiner: typelevel.MergeRequests[T]): Request[T] =
  combiner(r1, r2)
```

Status of Maia

Thank you!

Questions?

http://github.com/MaiaOrg | @sdbo

Challenges

→ Injectivity

→ Inference

→ IntelliJ

→ Boilerplate

Injectivity

→ Type functions "lose information"

→ Need to create explicit types parameterized with all of the needed information to retain it

→ Easy to do in Haskell, but "bare" type functions are more important in Scala (compounded with inference)

Inference

→ Scala inference isn't as good as Haskell

→ It's a constant struggle build an API which minimizes the number of explicit types the user must provide

IntelliJ

→ The "de facto" Scala IDE

→ Uses its own compiler chain which isn't 100% compatible with Scala

→ Many features Scala will compile IntelliJ will complain at. This is *terrible UX* so I had to avoid them.

→ Current Maia API is less expressive than Scala *could* offer so that IntelliJ plays nice.

Boilerplate

→ Writing Maia features is a a giant exercise in boilerplate

→ Actual logic code gets multiplied by 2 or 3 to do implicit-based type induction

→ Largely seems to be caused by bad Scala inference (similar Haskell code does not require this, types and kinds are almost always inferred)