TYPED ONTOLOGY

IN APPLICATIONS

Arseniy Zhizhelev, Primetalk / zhizhelev@primetalk.ru

PLAN

- intro
- data representation **needs**
- simple domain model
- ontology model
- various data representations
- technical details
- Pros & Cons
- architectural influence

Data representation needs

- sparse/incomplete/incorrect data representation
- events about isolated properties
- property runtime metainformation

Single data representation does not work for various layers of application

Simple domain model (case classes)

```scala
sealed trait DomainEntity

case class Person(
  name: String,
  address: Address,
  dob: LocalDate
) extends DomainEntity

case class Address(
  postalIndex: String,
  street: String
) extends DomainEntity
```

Simple domain model (Ontology)

```
sealed trait DomainEntity
abstract final class Person  extends DomainEntity
abstract final class Address extends DomainEntity
```

```
object person extends Schema[Person] {
  val name        = property[String]
  val address     = property[Record[Address]]
  val dob         = property[LocalDate]
}

object address extends Schema[Address] {
  val postalIndex = property[String]
  val street      = property[String]
}
```

Type of property

```
object person extends Schema[Person] {
  val name: Property[Record[Person], String] =
    property[String]
}
```

- phantom types `Record[+_]`, `Person`
- `name` can only be used in descendants of `Person`

```
trait Schema[T] {
  def property[B](
    implicit name: sourcecode.Name
  ): PropertyId[Record[T],B]
}
```

- *Note: we use nice macro by Li Haoyi to capture the name of the property.*

Ontology*: cake pattern

```
sealed trait DomainEntitySchema[T <: DomainEntity]
  extends Schema[T] {
  val id          = property[String]
}
sealed trait PersonSchema[T <: Person] extends Schema[T] {
  val name        = property[String]
  val address     = property[Record[Address]]
  val dob         = property[LocalDate]
}
object person2 extends Schema[Person]
  with PersonSchema[Person]
  with DomainEntitySchema[Person]
```

```
sealed trait AddressSchema[T <: Address] extends Schema[T]
  with DomainEntitySchema[T] {
  val postalIndex = property[String]
  val street      = property[String]
}
object address2 extends AddressSchema[Address]
```

Data example

```
val alice1 = Person(
  name = "Alice",
  address = Address(
    postalIndex = "123456",
    street = "Blueberry street, 8"
  ),
  dob = ??? // - unknown, need some fake value
)
val alice2 = person.record(
  person.name := "Alice",
  person.address := address.record(
    address.postalIndex := "12345",
    address.street := "Blueberry street, 8"
  )
)
```

Data representations

- typed map;

```
case class TypedMap[A](
  map: Map[PropertyId[Record[A],_], _])
```

- json;

```
case class JObjectRecord[A](jobject: JObject
```

- case classes + lenses;

```
@Lenses
case class Person(name: String  address: Addr
  dob: LocalDate) extends DomainEntity
```

- HList;
- database row;
- ...

Some implementation details: abstraction layers

- **data storage**
  - typed map, typed json
- **domain ontology**
  - common language definition
- **meta**
  - tools to define ontology (like PropertyId) and data representations
- **meta-of-meta**
  - tools to define meta layer; universal tools available to all data representations

Limitations of case classes

- sparse data
  - sets of case classes + per field data copying;
  - optional fields;
  - fields of type `Try`/`Either`/`\/`.
- events are unrelated case classes + manual event application
- runtime metainformation = annotations + reflection

Advantages of ontology data representations

- sparse data
  - typed maps and json ~ `Map`-like;
  - property conversion errors.
- property-related events (by it's `PropertyId`);

```
case class NewPropertyValue[A,B](
  instanceId: Id[A],
  propertyId: PropertyId[Record[A], B],
  newValue: B)
```

- metainformation - in map or in a custom `PropertyId`
- variety of data representations

Architectural perspective of the ontology

- domain model - in a central place
- ontology = unified domain language
- Easy meta programming (including macros)
- DB schema inference
- Natural event sourcing
- Automatic data transformation
- UI hints
- Domain constraints

Typed ontology (conclusion)

- Single source of domain model
- Compile time type safe JSON
- Wide range of applications

Appreciation to Sponsors

- Lambda Conf committee
- Cotiviti Labs
- An anonymous volunteer

Questions?

Arseniy Zhizhelev, Primetalk / zhizhelev@primetalk.ru