TYPED ONTOLOGY

IN APPLICATIONS

Arseniy Zhizhelev, Primetalk / zhizhelev@primetalk.ru

PLAN

- intro
- data representation needs
- simple domain model
- shortcomings of case classes
- ontology to the rescue
- data representations - tmap, json, case-classes + lenses, database, etc.
- technical details - meta, metameta
- architectural influence
- conclusion

Data representation needs:

- sparse/incomplete/incorrect data representation
  - db identifiers are not available for new instances;
  - some data might not be available immediately;
  - when parsing data from external sources, we might encounter errors with certain properties.
- events about separate properties: property change/remove/new value; collection changes add/remove; subscription to the events
- associating runtime metainformation (description, type information, predicates, alternative names) with properties

The needs differ in various layers of application. DB needs ids, UI needs UI-control hints, business logic needs constraints.

Simple domain model

```
sealed trait DomainEntity

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

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

Shortcomings of case classes:

- sparse data
  - different sets of case classes in different layers + per field data copying;
  - optional fields;
  - fields of type `Try`.
- events are represented as unrelated case classes with manual event application
- runtime metainformation as annotations + reflection

Ontology to the rescue:

```
sealed trait DomainEntity

abstract final class Person extends DomainEntity

abstract final class Address extends DomainEntity

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

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

Data example:

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

Data representations

- typed map;
- json;
- case classes + lenses;
- HList;
- database row;
- ...

Data representations with ontology:

- sparse data
  - typed maps and json are map-like structures that naturally represent absence;
  - apart from ordinary data, we may keep errors associated with property in the same record.
- it's easy to capture event data about separate properties - just refer to a property by it's ontological `PropertyId`;
- any metainformation can be associated with property either in separate maps, or directly in the `PropertyId` class

As there are various data representations, they can be used in various layers of application to fit their needs.

Architectural perspective of the ontology

- single place for domain model representation
- single domain language for the complete application
- code generation (often with macros in the single compiler pass) of various data representations - database schemas, case classes, etc.
- automatic data transfer/transformation between application layers
- natural event sourcing implementation
- UI hints and constraints

Conclusion

- typed ontology - single source of domain model for application
- compile time safety
- wide range of applications - sparse data representation, event sourcing, etc.