Bedelibry: A new framework for digital knowledge management.

Nathan Bedell

September 18, 2019

Abstract

In this paper, I describe the overall design and project structure of my Bedelibry project, discuss some of the problems that this project was meant to solve, give examples of potential use cases, and discuss additional features which may be implemented at a later time.

1 Introduction

When I started this project, the idea was simple. I wanted to have a knowledge base, with facts stored in Prolog in various places, with an API that could query these stores of knowledge. However, since then, I have moved away from using Prolog proper, for several reasons:

1. Prolog is an untyped language, and I want users to be able to specify some kind of schema for the types of queries that they are can make.
2. Storing basic facts in prolog files scattered about the user's system is not a very scalable solution.

To deal with these problems, I decided to design my own system dealing with:

1. Defining and enforcing schemas which define an ontology of *entities*, and *relations* that can hold between those entities and other data types.
2. Defining *datasources*, which describe how to retrieve different facts (i.e. the fact that a certain relation holds between entities).
3.

2 Experiments

2.1 Entities in Haskell

In this section I will describe how an implementation of an "entity framework" might work in Haskell.

```
module Entities where

data Animal = Animal {
  age :: Int,
  weight :: Double,
  name :: String
} -- Note: Instead of a source-to-source translation, we could do:
    deriving(Entity).
  -- which would produce:

data AnimalE = AnimalE {
  ageQ :: Maybe Int,
  weightQ :: Maybe Double,
  nameQ :: Maybe String
}

data Entity = EAnimal

data Prop a b where
  Age    :: Prop Animal Int
  Weight :: Prop Animal Double
  Name   :: Prop Animal String

-- Example

bob = AnimalE {
  ageQ = Just 15,
  weightQ = Just 57,
  nameQ = Just "Bob"
}
```

2.2 Entities in Idris

```
data EntityCommand : List (String, VarType) -> Type where
  NOP : EntityCommand []
  NewEntity : (cmds : EntityCommand xs) -> (s : String) ->
      EntityCommand (xs ++ [(s, Entity)])
  NewDatasource :
        (cmds : EntityCommand xs)
    -> (s : String)
    -> (field : ToyObjectFields)
    -> (IO (assocType field))
    -> EntityCommand (xs ++ [(s, Datasource)])
  Sequence :
       EntityCommand xs
    -> EntityCommand ys
    -> {auto p : Disjoint xs ys}
    -> EntityCommand (xs ++ ys)
  RemoveEntity :
        (cmds : EntityCommand xs)
    -> (s : String)
    -> {auto p : Elem s (map fst xs)}
    -> EntityCommand (?remove s xs)
  MakeQuery :
        (cmds : EntityCommand xs)
    -> (s : ValidEntityRef xs)
```

```
    -> ToyObjectFields
    -> EntityCommand xs
```

2.3 Entities in Scala

```
class Rel(signature: List[EntityType]) extends Identifiable[String]
    {
  def facts: List[Any]
}

trait Identifiable[A] {
  def primaryId:  A
  def rep: A => A
  def identities: List[A]
}

class EntityType(primaryId: String) extends Identifiable[String] {
  def identifies = List(pimaryId)
  def rep        = (x : String) => x
}

// A resource is something like a book, or an article, or a code
    snippet.
trait Resource[A] extends Identifiable[A]

trait FrozenResource[A,T] extends Resource[A] {
  def data: T
  def timestamp: String
}

// A resource that the user has annotated with some kind of
    annotation
// For example, a frozen HTML page that we have highlighted
    something
// in
trait AnnotatedResource[A, Annot, T] extends FrozenResource[A,T] {
  // Content of the annotation
  def annotation: Annot
}

case class Person[A](
    name:        String,
    age:         Int,
    primaryId:   A,
    identities:  List[A],
    rep:         A => A
) extends Identifiable[A]

case class Book[A](
    title:       String,
    authors:     List[Person[A]],
    pages:       Int,
    sections:    List[Section[A]],
    primaryId:   A,
    identities:  List[A],
    rep:         A => A
) extends Resource[A]
```

```
// Stuff for dealing with sections of a book.

sealed trait SectionType
case object Section    extends SectionType
case object Chapter    extends SectionType
case object Subsection extends SectionType

case class Section[A](
    title: String,
    sectionType: SectionType,
    belongsTo: Resource[A],
    subsectionOf: Option[Section[A]],
    primaryId: A,
    identities: List[A],
    rep: A => A
) extends Identifiable[A]

// Stuff for interfacing with databases
sealed trait ContentType
case object PDF extends ContentType
case object Article extends ContentType
case object Query extends ContentType

sealed trait Note
case class Journal(id: Int, datetime: String, content: String)
    extends Note
case class PrologNote(id: Int, datetime: String, content: String)
    extends Note
// Note: In MySQL, contentType can be declared as an Enum
case class WebNote(id: Int, contentType : ContentType, datetime:
    String, content: String)
```