Some notes on functionalization of mashup composition

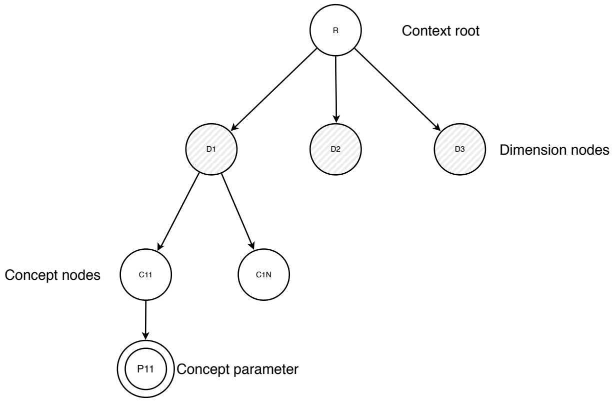
Vittorio Zaccaria

September 23, 2016

Contents

1 Background

It is well understood that the CDT model allows for a declarative expression of the type of a context c which we will broadly assume to be values/parameters associated with the situational needs in which the query must be answered.

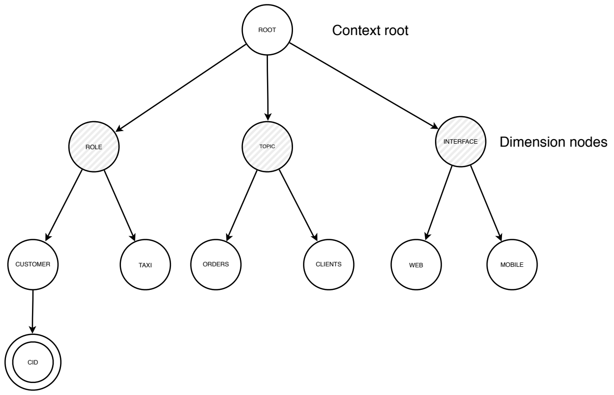


Each dimension D_d has one value of type indicated by one of children $C_{d,i}$, also called **conceptual nodes**.

Each children C can have one or more parameters and provides a **view**, i.e. some way of extracting interesting data from a particular data store (being it a database or a online service).

2 Operation implementation

Let us try to produce an implementation of the following CDT:



We can identify the following dimensions which can be seen as instances of an enumerative type:

```
data Dimension   = Role | InterestTopic | Interface
```

and the following concepts:

```
data Concept = Customer Int
               | Restaurant
               | Orders Int
               | Food
               | Web
               | SmartPhone
```

Finally we define a context as an array of concepts with their own values:

```
data Context     = Ctx [ Concept ] deriving (Show)
```

2.1 Describing the tree

We can thus see the CDT as a tree of nodes (`Tree NodeData`), where each node can be one of three things:

```
data NodeData    = D Dimension | C (Context -> Maybe View) | Root
```

Each node, in fact, associates a concept with a view which we implement as a function (`Context -> Maybe View`). For example, the node associated with the `Customer` concept looks in the context to see whether there is a request for a specific context value; in our example queries are just sql-like strings:

```
customerView :: Context -> Maybe View
customerView (Ctx []) = Nothing
customerView (Ctx (Customer n:_)) = Just $ E $ "select customers where id=" ++ show n
customerView (Ctx (_:xs)) = customerView (Ctx xs)
```

2.2 Working with the tree

First of all, we should define views as a monoid with a unit (`mempty`) and an operator (`mappend`) to join them; in our case the operator joins query strings:

```
instance Monoid View where
  mempty              = Empty
  mappend Empty x     = x
  mappend y Empty     = y
  mappend (E x) (E y) = E (x ++ " doubleintersection " ++ y)
```

The actual view creation can be seen as a `fold` operation of views that each concept node generates given the current context. In the following, function `f` is used to transform each node into either a valid view or the empty view.

```
getViews :: Foldable t => Context -> t NodeData -> View
getViews ctx = foldMap (f ctx) where
    f cx (C c)   = fromMaybe mempty (c cx)
    f _          = mempty
```

2.3 Example operation

Given:

```
context = Ctx [ Web, Customer 3 ]
cdt = Node Root [
        dim Role [ leaf customerView ],
        dim InterestTopic [],
        dim Interface [ leaf webView ]
  ]
```

getting the views gives:

```
λ> getViews context cdt
E "select customers where id=3 doubleintersection select _ where type=web"
```