HaskellDB

Advanced Functional Programming

Niels van Hoorn & Eelco Lempsink

Utrecht University

March 28, 2007

Outline

1. Introduction
2. Internals
3. Using HaskellDB
4. Conclusion

HaskellDB

- Obviously, to work with databases
- Developed by Daan Leijen, improved by students from Chalmers
- Implemented as DSEL
  - No SQL strings
  - Typed checked queries
  - Abstracts from implementation
- Unwrapping Relational Algebra
- (Re-)implements a Query Language

Recalling Relational Algebra

SQL

```
SELECT name AS author, title
FROM authors, titles
WHERE authors.id = titles.author_id
```

Relational Algebra

$$\pi_{(author name, title)}(\rho_{author=name}(\sigma_{(id=author_id)}(authors \times titles)))$$

Translating into HaskellDB

Relational Algebra

$$\pi_{(author,title)}(\rho_{author=name}(\sigma_{(id=author_id)}(authors \times titles)))$$

HaskellDB

```
do a <- table authors
   t <- table titles
   restrict (a!id .==. t!author_id)
   project  (author << a!name #
             title  << t!title)
```

Looking closer

Code

```
restrict (a!id .==. t!author_id)
```

Observations

- `restrict` takes an expression as argument
- `(!)` looks up an attribute in a relation
- `(.==.)` compares two expressions

Expressions

Abstract Syntax

```
data PrimExpr = AttrExpr Attribute
              | BinExpr  BinOp PrimExpr PrimExpr
              | UnExpr   UnOp  PrimExpr
              ...

data BinOp = OpEq | OpLt | OpLtEq | ...
```

Combinatory sugar

```
(.==.) :: PrimExpr -> PrimExpr -> PrimExpr
(.==.) x y = BinExpr OpEq x y
```

Type Embedding

Phantom type

```
data Expr a = Expr PrimExpr
```

Typing combinators

```
(.==.) :: Eq a => Expr a -> Expr a -> Expr Bool
(.==.) (Expr x) (Expr y) = Expr (BinExpr OpEq x y)

(.+.) :: Num a => Expr a -> Expr a -> Expr a
(.+.) (Expr x) (Expr y) = Expr (BinExpr OpPlus x y)
```

Type-safe Attributes

- Recall: `a!id .==. t!author_id`
- `(!)` looks up an attribute in a relation, such as a table

Data Types

```
(!) :: Rel -> Attr -> Expr a

data Rel   = Rel   Assoc
data Attr  = Attr  Attribute
data Table = Table TableName Assoc
```

Type Embedding

Lookup

```
(!) :: Rel r -> Attr r a -> Expr a

data Rel r     = Rel   Assoc
data Attr r a   = Attr  Attribute
data Table r   = Table TableName Assoc
```

What we would like to have

```
(!) :: (name ∈ r) => Rel r -> Attr name a -> Expr a
```

Records

- In `r` a 'record' type is stored
- A record contains the types of a relation or database row

Example

```
data Table r = Table TableName Assoc

authors :: Table
  ((RecCons Id (Expr Int)
     (RecCons Name (Expr String)
       (RecCons Address (Expr String) RecNil))))

data RecNil          = RecNil
data RecCons fn a r = RecCons a r
```

Example of Table type

But this type is very lengthy...

```
authors :: Table
    ((RecCons Au_id (Expr Int)
      (RecCons Au_lname (Expr String)
       (RecCons Au_fname (Expr String)
        (RecCons Phone (Expr String)
         (RecCons Address (Expr String)
          (RecCons City (Expr String)
           (RecCons State (Expr String)
            (RecCons Zip (Expr String)
             (RecCons Contract (Expr Bool) RecNil)
            )))))))))
```

Datatypes for database representation

```
data DBInfo = DBInfo { dbname :: String
                     opts   :: DBOptions
                     tbls   :: [TInfo] }
```

```
data TInfo  = TInfo   { tname :: String
                      cols  :: [CInfo] }
```

```
data CInfo  = CInfo   { cname :: String
                      descr :: FieldDesc }
```

```
type FieldDesc = (FieldType, Bool)
```

Ways of creating a database representation

Three solutions

- Creating representation from database
- Creating database from representation
- Creating representation manually

```
dbToDBSpec :: Bool -> String
           -> Database -> IO DBInfo
dbSpecToDatabase :: Database -> DBInfo -> IO ()
dbInfoToModuleFiles :: FilePath -> String
                    -> DBInfo -> IO ()
```

Getting the tables

Recalling the example

```
do a <- table authors
   t <- table titles
   restrict (a!id .==. t!author_id)
   project  (author << a!name #
             title  << t!title)
```

```
table :: Table r -> Query (Rel r)
```

The Query monad

The Query monad

- Contains the relational algebra expression that is being build
- Keeps track of currently bound variables
- Creates unique identifiers for the variables

Query example

```
authorsWithTitles :: Query (Rel r)
authorsWithTitles =
        do a <- table authors
           t <- table titles
           restrict (a!id .==. t!author_id)
           project  (author << a!name #
                     title  << t!title)
```

```
titlesFromCalifornia =
        do at <- authorsWithTitles
           restrict (at!state .==. constant "CA")
           project  (author << at!name #
                     title  << at!title)
```

Demo

Executing and showing the result of Queries

```
options = PostgreSQLOptions "localhost" "pubs" "" ""
postgresqlConnect :: PostgreSQLOptions ->
                        (Database -> IO a) -> IO a
withDB q = postgresqlConnect options q
showWithQuery showRow q  = withDB $ \db ->
          do rows <- query db q
             putStr (unlines (map showRow rows))
```

Conclusion

- Good
- Clean
- Fun