Squeal

A Deep Embedding of SQL in Haskell

Eitan Chatav

Morphism, LLC

DataKinds

Type level strings

```
:set -XDataKinds
:kind "what is my kind daddy?"
> "what is my kind daddy?" :: GHC.Types.Symbol
```

DataKinds

Type level natural numbers

```
:set -XDataKinds
:kind 90210
> 90210 :: GHC.Types.Nat
```

DataKinds

Type level lists

```
:set -XDataKinds
:kind '[Int,Double,String]
> '[Int,Double,String] :: [*]
```

DataKinds

Type level tuples; named fields

```
:set -XDataKinds -XTypeOperators -XPolyKinds
import GHC.Types
type (:::) (alias :: Symbol) (ty :: kind) = '(alias, ty)
:kind "Beverley Hills" ::: 90210
> "Beverley Hills" ::: 90210 :: (Symbol, Nat)
```

DataKinds

Algebraic data kinds

```
:set -XDataKinds
data PGType = PGbool | PGint4 | PGnumeric | PGtext
data NullityType = Null PGType | NotNull PGType
:kind 'Null 'PGint4
> 'Null 'PGint4 :: NullityType
```

TypeInType

Kind synonyms

```haskell
data ColumnType
  = Optional NullityType
  | Required NullityType

type ColumnsType = [(Symbol,ColumnType)]

type TablesType = [(Symbol,ColumnsType)]
```

Schema DSL

- We now have a simple type level DSL that allows us to declare the schema of a database

```
type Schema =
  '[ "users"  :::
       '[ "id"   ::: 'Optional ('NotNull 'PGint4)
        , "name" ::: 'Required ('NotNull 'PGtext)
        ]
   , "emails" :::
       '[ "id"      ::: 'Optional ('NotNull 'PGint4)
        , "user_id" ::: 'Required ('NotNull 'PGint4)
        , "email"   ::: 'Required ('Null 'PGtext)
        ]
   ]
```

SQL Embedding

Initial and final embedding

- We have a choice for how to embed languages in Haskell, initial or final.
- Initial embedding
  - Use Haskell data types to define an abstract syntax tree
  - Great for precision and correctness
  - Use ADTs for simple unityped syntax trees
  - Use GADTs for complex typed syntax trees
- Final embedding
  - Embed as a final interpretation
  - Great for performance and extensibility
  - In the case of PostgreSQL, a ByteString to be passed to LibPQ
  - Use functions instead of constructors of a datatype
  - Think of it as a type directed pretty printer / string interpolator

SQL Embedding

Sublanguages

- SQL actually consists of a bunch of sublanguages
- The structured query language of SELECT statements
- The data manipulation language of INSERT, DELETE and UPDATE statements
- The data definition language CREATE, DROP and ALTER statements
- There is also a data control language and a transaction control language

SQL Embedding

Structured query language

- How do we type the following queries?

```
SELECT * FROM users
SELECT id AS userId FROM users WHERE name = $1
```

- Each statement outputs columns.
- The second statement demonstrates that statements can have positional input parameters.
- Furthermore, both statements are only sensible in the presence of the schema we declared.

SQL Embedding

Structured query language

```
newtype Query
  (schema :: TablesType)
  (params :: [ColumnType])
  (columns :: ColumnsType)
    = UnsafeQuery { renderQuery :: ByteString }

getUsers :: Query Schema '[]
  '[ "id"   ::: 'Optional ('NotNull 'PGint4)
   , "name" ::: 'Required ('NotNull 'PGtext) ]
getUsers = UnsafeQuery "SELECT * FROM users"

getUserId :: Query Schema
  '[ 'Required ('NotNull 'PGtext) ]
  '[ "userId" ::: 'Required ('NotNull 'PGint4) ]
getUserId = UnsafeQuery
  "SELECT id AS userId FROM users WHERE name = $1"
```

SQL Embedding

Data manipulation language

```
newtype Manipulation
  (schema :: TablesType)
  (params :: [ColumnType])
  (columns :: ColumnsType)
    = UnsafeManipulation
    { renderManipulation :: ByteString }

insertUser :: Manipulation Schema
  '[ 'Required ('NotNull 'PGtext)]
  '[ "userId" ::: 'Required ('NotNull 'PGint4) ]
insertUser = UnsafeManipulation
  "INSERT INTO users (id, name) VALUES (DEFAULT, $1)
   RETURNING id AS userId;"
```

SQL Embedding

Data definition language

- Data definitions are statements that changes the schema

```
newtype Definition
  (schema0 :: TablesType)
  (schema1 :: TablesType)
    = UnsafeDefinition { renderDefinition :: ByteString }
```

SQL Embedding

Data definition language

- Data definitions are statements that changes the schema

```
setup :: Definition '[] Schema
setup = UnsafeDefinition
  "CREATE TABLE users (
    id serial, username text NOT NULL,
    PRIMARY KEY (id)
   );
  CREATE TABLE emails (
    id serial,
    userid integer NOT NULL,
    email text NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY (userid) REFERENCES users (id)
      ON DELETE CASCADE ON UPDATE RESTRICT
  );"
```

SQL Expressions

SQL Expressions

- SQL expressions are the basic atoms of a SQL statement

```
newtype Expression
  (tables :: TablesType)
  (params :: [ColumnType])
  (ty :: ColumnType)
    = UnsafeExpression
    { renderExpression :: ByteString }

type Condition tables params
  = Expression tables params
    ('Required ('NotNull 'PGbool))
```

- We give expressions a rich API by using standard Haskell typeclasses, e.g. Num, Fractional, Floating, IsString

SQL Expressions

- We also need to be able to express column and parameter expressions

```
class KnownSymbol column => HasColumn column columns ty
  | column columns -> ty where
    getColumn
      :: Expression '[table ::: columns] params ty

class KnownNat n => HasParameter (n :: Nat) params ty
  | n params -> ty where
    param :: Expression tables params ty
```

OverloadedLabels

- OverloadedLabels is a new feature of GHC that enables a succinct notation for a value which is completely determined by a type level string

```
:set -XOverloadedLabels -XTypeApplications
import GHC.OverloadedLabels

:type fromLabel @ "foo"
fromLabel @ "foo" :: IsLabel "foo" a => a

:type #foo
#foo :: IsLabel "foo" t => t
```

OverloadedLabels

- OverloadedLabels is a new feature of GHC that enables a succinct notation for a value which is completely determined by a type level string

```
data Alias (alias :: Symbol) = Alias
instance IsLabel alias (Alias alias) where
  fromLabel = Alias

instance HasColumn column columns ty
  => IsLabel column
    (Expression '[table ::: columns] params ty) where
      fromLabel = getColumn
```

Tables

- We also must be able to reference tables from a schema

```
newtype Table
  (schema :: TablesType)
  (columns :: ColumnsType)
    = UnsafeTable { renderTable :: ByteString }

class KnownSymbol table => HasTable table tables columns
  | table tables -> columns where
    getTable :: Table tables columns

instance HasTable table schema columns
  => IsLabel table (Table schema columns) where
    fromLabel = getTable
```

Type Expressions

- Type expressions are used for typecasts and CREATE statements

```
newtype TypeExpression (ty :: ColumnType)
  = UnsafeTypeExpression
  { renderTypeExpression :: ByteString }

bool :: TypeExpression ('Required ('Null 'PGbool))

notNull
  :: TypeExpression (optionality ('Null ty))
  -> TypeExpression (optionality ('NotNull ty))
```

GADTs

Gadgets

- GADTs or “gadgets” are generalized algebraic datatypes
- Like ADTs but with explicit type signatures for constructors
- Have a “generators and relations” feel to them
- Great for precisely typed abstract syntax trees

GADTs

Heterogeneous lists

```
data NP expr xs where
  Nil :: NP expr '[]
  (:*) :: expr x -> NP expr xs -> NP expr (x ': xs)
```

GADTs

Aliased expressions

```
data Aliased expr aliased where
  As
    :: expr x
    -> Alias alias
    -> Aliased expr (alias ::: x)
```

GADTs

FROM clauses

```
data FromClause schema params tables where
  Table
    :: Aliased (Table schema) table
    -> FromClause schema params '[table]
  Subquery
    :: Aliased (Query schema params) table
    -> FromClause schema params '[table]
  CrossJoin
    :: FromClause schema params right
    -> FromClause schema params left
    -> FromClause schema params (left ++ right)
  InnerJoin
    :: FromClause schema params right
    -> Condition (Join left right) params
    -> FromClause schema params left
    -> FromClause schema params (left ++ right)
```

Actions

Representations of Monoids

- An action of a monoid M on X is a function

```
act :: M -> End X
```

- End X is the set of endomorphisms

```
type End x = x -> x
```

- Actions respect the monoid homomorphism laws

```
act mempty = id
act (m2 <> m1) = m2 . m1
```

Actions

Table expressions

- In SQL, a table expression is generated from a FROM clause by the action of WHERE, LIMIT and OFFSET clauses
- WHERE is an action of Condition, AND, TRUE by filtering
- LIMIT is an action of Word64, min, ALL by restricting the number of rows
- OFFSET is an action of Word64, +, 0 by dropping a number of rows
- At least, it would make sense if it were true!

Actions

Table expressions

- Since [X] is the free monoid over X, we can use lists in the definition of a table expression and fold the modifier clauses in our interpreter

```
data TableExpression schema params tables =
  TableExpression
    { fromClause :: FromClause schema params tables
    , whereClause :: [Condition tables params]
    , limitClause :: [Word64]
    , offsetClause :: [Word64]
    }
```

Actions

Table expressions

- Since [X] is the free monoid over X, we can use lists in the definition of a table expression and fold the modifier clauses in our interpreter

```
from relation = TableExpression relation [] [] []

where_ condition = TableExpression
  { whereClause = condition : whereClause }

limit lim = TableExpression
  { limitClause = lim : limitClause }

offset off = TableExpression
  { offsetClause = off : offsetClause }
```

Select

- Now we can define a select combinator for queries

```
select
  :: NP (Aliased (Expression tables params)) columns
  -> TableExpression schema params tables
  -> Query schema params columns
```

Select

- Now we can define a select combinator for queries

```
select (#id `As` #userId :* Nil)
  (from (Table (#users `As` #u))
  & where_ (#name .== param @1)
  & limit 10 & offset 10)
```

```
SELECT id AS userId
  FROM users AS u
  WHERE name = $1
  LIMIT 10 OFFSET 10
```

Select

- Now we can define a select combinator for queries

```
select
  ( #u ! #name `As` #userName :*
    #e ! #email `As` #userEmail :* Nil)
  ( from (Table (#users `As` #u)
    & InnerJoin (Table (#emails `As` #e))
      (#u ! #id .== #e ! #user_id)) )
```

```
SELECT
  u.name AS userName,
  e.email AS userEmail
  FROM users AS u
    INNER JOIN emails as e
    ON u.id = e.user_id
```

Categories

- Recall that definitions are transformations of a schema

```
data Definition
  (schema0 :: TablesType)
  (schema1 :: TablesType)
```

- We can give this a category structure

```
instance Category Definition where
  id = UnsafeDefinition “;”
  def1 . def0 = UnsafeDefinition $
    renderDefinition def0 <> “ “ <> renderDefinition def1
```

Create

- We can define create statements, which change the schema by adding a new table

```
type family Create alias x xs where
  Create alias x '[] = '[alias ::: x]
  Create alias y (x ': xs) = x ': Create alias y xs

createTable
  :: KnownSymbol table
  => Alias table
  -> NP (Aliased TypeExpression) columns
  -> [TableConstraint schema columns]
  -> Definition schema (Create table columns schema)
```

Create

- We can now define our schema from earlier

```
setup :: Definition '[] Schema
setup =
  createTable #users
    ( serial `As` #id :*
     (text & notNull) `As` #name :* Nil )
    [ primaryKey (Column #id :* Nil) ]
  >>>
  createTable #emails
    ( serial `As` #id :*
      (int & notNull) `As` #user_id :*
      text `As` #email :* Nil )
    [ primaryKey (Column #id :* Nil)
    , foreignKey (Column #user_id :* Nil)
      #users (Column #id :* Nil)
      OnDeleteCascade OnUpdateCascade ]
```

Indexed Monads

- Let Fun be the monoidal category of Haskell functors with composition and identity
- An (Atkey) indexed monad is a Fun-enriched category
  - A set of objects called indices $i, j, k, \ldots$
  - For any i, j a functor

```
m i j :: Type -> Type
instance Functor (m i j)
```

  - An identity natural transformation

```
η :: x -> m i i x
```

  - A composition natural transformation

```
μ :: m j k (m i j x) -> m i k x
```

  - Left & right identity and associativity laws

```
μ . η = id
η . fmap μ = id
μ . μ = μ . fmap μ
```

Indexed Monads

LibPQ

```
newtype Connection (schema :: TablesType)

newtype PQ schema0 schema1 m x = PQ
  { runPQ
      :: Connection schema0
      -> m (x, Connection schema1)
  }
```

- PQ is an indexed monad transformer
- For a fixed index, ‘PQ schema schema‘ is a normal monad transformer

Indexed Monads

LibPQ

```
newtype Result columns

define
  :: Definition schema0 schema1
  -> PQ schema0 schema1 IO (Result '[])

manipulateParams
  :: ToParams x params
  => Manipulation schema params columns
  -> x -> PQ schema schema IO (Result columns)

runQueryParams
  :: ToParams x params
  => Query schema params columns
  -> x -> PQ schema schema IO (Result columns)
```

Indexed Monads

LibPQ

```
pqBind
  :: Monad m
  => (x -> PQ schema1 schema2 m y)
  -> PQ schema0 schema1 m x
  -> PQ schema0 schema2 m y

pqThen
  :: Monad m
  => PQ schema1 schema2 m y
  -> PQ schema0 schema1 m x
  -> PQ schema0 schema2 m y
```

Generics

```haskell
newtype Result columns

getRows
  :: FromRow columns y
  => Result columns
  -> IO [y]

class ToParams (x :: Type) (params :: [ColumnType]) where
  toParams :: x -> NP (K (Maybe ByteString)) params

class FromRow (columns :: ColumnsType) (y :: Type) where
  fromRow :: NP (K (Maybe ByteString)) columns -> y
```

Generics

- Using generics-sop, the sum-of-products formulation of generics

```
instance
  ( SListI params
  , IsProductType x xs
  , AllZip ToColumnParam xs params
  ) => ToParams x params where
    toParams
      = htrans
          (Proxy @ToColumnParam)
          (toColumnParam . unI)
      . unZ . unSOP . from
```

Generics

- Using generics-sop, the sum-of-products formulation of generics

```
instance
  ( SListI results
  , IsProductType y ys
  , AllZip FromColumnValue results ys
  , SameFields (DatatypeInfoOf y) results
  ) => FromRow results y where
    fromRow
      = to . SOP . Z
      . htrans
          (Proxy @FromColumnValue)
          (I . fromColumnValue)
```