Experience Report: Haskell Relational Record

Kei Hibino Shohei Murayama
Asahi Net, Inc.
k.hibino@asahinet.com
shohei.murayama@asahinet.com

Kazuhiko Yamamoto
IIJ Innovation Institute Inc.
kazu@iij.ad.jp

Abstract

HaskellDB is an embedded, domain-specific language that provides composability and type-safety for SQL. In spite of such excellent features, use of HaskellDB in real-world database operations discloses its drawbacks, including column name collisions and unclear semantics of aggregate queries. To solve these issues, we implemented Haskell Relational Record (HRR), which has support for outer joins and type-propagated placeholders as well as provides semantically-clear and conflict-free composability. HRR supports structured projections, which corresponds to nested, standard Haskell records. This paper describes the key ideas of HRR and reports on our experience developing and using it.

General Terms Languages

Keywords SQL, Relational Database, Type-Safety, Composable Query, Outer Join, Aggregation, Placeholder, Nested Record.

1. Introduction

Asahi Net, Inc. is a Japanese Internet service provider with about 572,000 residential customers, as of March 2015. Its business model is to minimize labor expenses by automating operations, while keeping a small share of the market. For this purpose, Asahi Net has maintained many hand-written SQL [4] statements in Java, Perl, and other programming languages.

It is well-known that SQL statements represented in strings are error-prone and difficult to maintain due to lack of composability. In 2012, the authors tried to use HaskellDB [1, 5], which provides excellent features such as type-safety and composability. If queries expressed in HaskellDB can be compiled, generated SQL statements are always valid. There is no need to run the programs to test SQL statements by connecting databases. Since HaskellDB queries can be composed, large queries can be built from well-tested, small queries.

Unfortunately, we soon faced issues with HaskellDB. For instance, the semantics of aggregate queries are unclear, and column names in composed queries conflict. We initially tried to resolve the issues in HaskellDB, but it was difficult to integrate our new ideas, and we therefore decided to develop our own system from scratch.

The result is called Haskell Relational Record (HRR), a next-generation implementation of HaskellDB. Like HaskellDB, HRR features type-safety and composability, but its composability is semantically-clear and conflict-free. HRR has been in use at Asahi Net since March 2013, and more than two years of production use demonstrates its stability and usability. We have also released HRR as open-source software[1].

This paper describes the design and implementation of HRR and reports on our experience using it in production. Section 2 illustrates some issues with HaskellDB, and Section 3 shows our solutions in HRR. Section 4 and Section 5 describe some advanced features of HRR and our experience in developming and using it, respectively. Related work is in Section 6 and our conclusion is in Section 7.

2. HaskellDB Issues

Asahi Net uses 2,027 unique SQL statements, of which 1,375 are `SELECT` statements. Note that it is difficult to count SQL statements in untyped programming languages such as Perl, so we only counted them in Java and Haskell. This paper focuses on `SELECT` queries, which we have found are the most challenging. When we tried to express them in HaskellDB queries, we faced the following issues:

- Limited expression ability of projections
- No outer join support
- Column name conflicts
- Partial support for placeholders
- Unclear aggregation semantics

2.1 Limited expression ability of projections

HaskellDB originally used TRex [2], which was only available in Hugs, and later switched to its own *extensible records* (or *heterogeneous lists*) for portability. For HaskellDB developers, extensible records are an elegant solution because this single mechanism can express both types of tables and types of projection queries. For example, consider a table `Employee`:

```
CREATE TABLE Employee (
  id      INTEGER     NOT NULL
, name    VARCHAR(32) NOT NULL
, dept_id INTEGER     NOT NULL);
```

The type of this table can be expressed using extensible records as follows:

```
(RecCons Id (Expr Int)
 (RecCons Name (Expr String)
  (RecCons Dept_id (Expr Int) RecNil)))
```

The type of selections of two columns can be expressed as well:

```
(RecCons Id (Expr Int)
 (RecCons Name (Expr String) RecNil))
```

[1] http://khibino.github.io/haskell-relational-record/

Haskell Symposium 2015, September 3–4, 2015, Vancouver BC Canada.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM [to be supplied]. . . $15.00.
http://dx.doi.org/10.1145/[to be supplied]

Extensible records can only express flat structure, which is sufficient for representing SQL projections. In some cases, however, nested structures are preferable, as they can generalize the results of SQL queries and preserve information on table structure. For example, suppose that we have another table called `Department`:

```
CREATE TABLE Department (
  dept_id   INTEGER     NOT NULL
, dept_name VARCHAR(32) NOT NULL);
```

We might want to use the following `MemberInfo` type as a projection on the two joined tables:

```
data Member = Member { mem_name :: String
                     , mem_dept :: String }
data Ids = Ids { id_emp :: Int, id_dept :: Int }
data MemberInfo = MemberInfo Member Ids
```

2.2 No outer join support

HaskellDB supports inner joins and cross joins. The following is an example of an inner join between `Employee` and `Department`, whose modules are imported qualified as `E` and `D` respectively:

```
do e <- table employee
   d <- table department
   restrict (e ! E.dept_id .==. d ! D.dept_id)
   project $ E.name << e ! E.name
           # D.dept_name << d ! D.dept_name
```

However, HaskellDB does not provide (left, right, or full) outer joins, which are common in real-world database operations. At Asahi Net, 5.2% of the `SELECT` statements are outer joins.

2.3 Column name conflicts

With HaskellDB, programmers are responsible for ensuring that column names do not conflict. It is not feasible, however, to make column names unique among all tables of a real-world database. Moreover, conflicts occur even with a single table when self-joins are used. For example, consider the following table, which contains an identifier, a name, and the identifier of the person's mother:

```
CREATE TABLE Person (
  id        INTEGER     NOT NULL
, name      VARCHAR(32) NOT NULL
, mother_id INTEGER     NOT NULL);
```

The following is a HaskellDB query for obtaining a pair of a person and his/her mother. Note that the corresponding module is imported as `P`:

```
do p1 <- table person
   p2 <- table person
   restrict (p1 ! P.mother_id .==. p2 ! P.id)
   project $ P.name << p1 ! P.name
           # P.name << p2 ! P.name
```

HaskellDB produces the following SQL statement from this query:

```
SELECT name1 as name,
       name1 as name
  FROM (SELECT id as id2,
               name as name2
          FROM Person as T1) as T1,
       (SELECT name as name1,
               mother_id as mother_id1
          FROM Person as T1) as T2
 WHERE ((mother_id1) = (id2))
```

The column name `name` in the top level `SELECT` conflicts. This SQL would return a valid result, but reusing the query would result in unexpected behavior.

2.4 Partial support for placeholders

Placeholders are used to parameterize SQL statements, and they are essential for SQL statement reusability. At Asahi Net, 85.7% of the `SELECT` statements use placeholders. Unfortunately, HaskellDB only has partial support for placeholders. Consider the following HaskellDB query, which takes one parameter, indicated by the `param` keyword:

```
do e <- table employee
   restrict $ e ! E.name .==. param (constant "")
   project $ E.id << e ! E.id
           # E.name << e ! E.name
```

This HaskellDB query is converted to the following SQL statement:

```
SELECT id, name
  FROM Employee as T1
 WHERE ((name) = (?))
```

Unfortunately, HaskellDB does not provide a way to use this query by specifying the parameter.

2.5 Unclear aggregation semantics

Aggregate queries are also essential for real-world database operations. At Asahi Net, 17.2% of the `SELECT` statements use aggregation. HaskellDB provides aggregate queries, but the semantics are unclear[2]. For instance, consider the following query, which uses aggregation:

```
countMembers = do
    e <- table employee
    d <- table department
    restrict (e ! E.dept_id .==. d ! D.dept_id)
    project $ D.dept_id << d ! D.dept_id
            # members << count (e ! E.id)
```

`members` is an integral column. This HaskellDB query is converted into the intended SQL statement:

```
SELECT dept_id2 as dept_id,
       COUNT(id1) as members
  FROM (SELECT dept_id as dept_id2
          FROM Department as T1) as T1,
       (SELECT id as id1,
               dept_id as dept_id1
          FROM Employee as T1) as T2
 WHERE ((dept_id1) = (dept_id2))
 GROUP BY dept_id2
```

Now consider a second query that uses the first one:

```
do m <- countMembers
   project $ members << m ! members
```

This HaskellDB query is converted into the following SQL statement:

```
SELECT COUNT(id1) as members
  FROM (SELECT dept_id as dept_id2
          FROM Department as T1) as T1,
       (SELECT id as id1,
               dept_id as dept_id1
          FROM Employee as T1) as T2
 WHERE ((dept_id1) = (dept_id2))
```

This projection from an aggregate table form is semantically incorrect because the `GROUP BY` clause is lost. Note that inlining the first query into the second one has the same issue.

[2] This issue was originally reported in HaskellDB issue 22 on GitHub.

3. Solutions in HRR

HRR is designed using several components, including *relations* and *queries*. Relations are composable, while queries are a final representation of SQL `SELECT` statements. Their types are `Relation p r` and `Query p r`, respectively, where `p` and `r` are phantom types for placeholders and the results of SQL queries, respectively. A relation can be converted into a query using the following function:

```
relationalQuery :: Relation p r -> Query p r
```

A query can then be translated into an SQL statement and sent to a database system using the following function:

```
runQuery :: (IConnection conn
            ,ToSql SqlValue p
            ,FromSql SqlValue a) =>
            conn -> Query p a -> p -> IO [a]
```

Currently, HRR uses HDBC[3] as a database abstraction layer. `IConnection` is a typeclass, defined in HDBC, that represents connections to database systems. The third parameter, `p`, holds any placeholder values.

A relation is defined for each table in the database via bootstrapping, as is discussed in Section 4.1. For example, the following relation, which selects employees from the `Employee` table, is automatically created:

```
employee :: Relation () Employee
```

Relations can be built using other relations. For instance, the following is a relation that selects employees from the `Employee` table whose identifier is less than 10:

```
initialEmp :: Relation () Employee
initialEmp = relation $ do
  e <- query employee
  wheres $ e ! E.id' .<. value 10
  return e
```

The `relation` function restricts the type of the inner *build monad* to `QuerySimple`:

```
relation :: QuerySimple (Projection Flat r)
         -> Relation () r
```

`QuerySimple` is a state monad that stores information on joining, correlation naming, filtering, ordering, etc. A `Projection` represents a type of SQL *expression*. It has two phantom types. The first parameter indicates whether or not expressions use aggregation operations, and `Flat` indicates that aggregate operations are not used. The second parameter indicates the type of the SQL query result.

The `query` function converts a relation to a build monad, where `m` is specialized to `QuerySimple` in the above example:

```
query :: (MonadQualify ConfigureQuery m, MonadQuery m)
      => Relation () r
      -> m (Projection Flat r)
```

The `initialEmp` relation is converted to the following SQL:2011[4] statement.

```
SELECT ALL T0.id AS f0, T0.name AS f1, T0.dept_id AS f2
  FROM employee T0
 WHERE (T0.id < 10)
```

3.1 Structured projections

We decided to use standard records (data types with field labels) [6] to express the structured projections discussed in Section 2.1. Abstraction layers for database systems give and take a flat structure of projections. In HDBC, the interface is essentially lists of strings, which should be converted to and from nested records. To capture this characteristic, we introduced *projection paths*, which map between flat data types and structured ones.

For example, let $\pi_{xs} : A \triangleright B$ denote a projection path. "nth leaf type of record A" indicates the nth element of the flattened fields sequence of record A in depth-first order. From the record point of view, this function converts record A to record B. From the list point of view, it creates a new list from the original list by enumerating indices of the original one (xs). Note that the length of xs is equal to the width of B, which is the number of SQL columns. Let $p_{xs} : \mathcal{P}\, A$ denote a projection for record A, corresponding to the list of single SQL expressions xs.

3.1.1 Typing and reduction rules

The essential rules of typing projection paths and projections are as follows:

$$\frac{}{A \vdash \pi_{[0..w-1]} : A \triangleright A} \text{ (T-id)}$$

(w is the width of type A)

$$\frac{}{A, B \vdash \pi_{[x_0,..,x_{w-1}]} : A \triangleright B} \text{ (T-sel)}$$

($0 \leq x_n < v, v$ is the width of type A)
($0 \leq n < w, w$ is the width of type B)
(nth leaf type of record B is x_nth leaf type of record A)

$$\frac{A, B, C \vdash \pi_{xs} : A \triangleright B \quad \pi_{ys} : B \triangleright C}{A, B, C \vdash \pi_{xs} \odot \pi_{ys} : A \triangleright C} \text{ (T-compose)}$$

$$\frac{B, C \vdash p_{xs} : \mathcal{P}\, B \quad \pi_{ys} : B \triangleright C}{B, C \vdash p_{xs} \ \mathbf{idx}\ \pi_{ys} : \mathcal{P}\, C} \text{ (T-index)}$$

$$\frac{A, B, C \vdash \pi_{xs} : A \triangleright B \quad \pi_{ys} : A \triangleright C}{A, B, C \vdash \pi_{xs} \oplus \pi_{ys} : A \triangleright (B, C)} \text{ (T-pair)}$$

$$\frac{B, C \vdash p_{xs} : \mathcal{P}\, B \quad p_{ys} : \mathcal{P}\, C}{B, C \vdash p_{xs} \oplus p_{ys} : \mathcal{P}(B, C)} \text{ (T-P-pair)}$$

Emphasized words here indicate infix operators. The following are the essential reduction rules of projection paths and projections:

$$\frac{\pi_{xs} \odot \pi_{ys}}{\pi_{[xs!!y|y \leftarrow ys]}} \text{ (R-compose)} \qquad \frac{p_{xs}\ \mathbf{idx}\ \pi_{ys}}{p_{[xs!!y|y \leftarrow ys]}} \text{ (R-index)}$$

$$\frac{\pi_{xs} \oplus \pi_{ys}}{\pi_{xs+\!\!+ys}} \text{ (R-pair)} \qquad \frac{p_{xs} \oplus p_{ys}}{p_{xs+\!\!+ys}} \text{ (R-P-pair)}$$

Here is an example of T-index, where *MI* indicates `MemberInfo`:

$$\frac{p_{[\text{“}Bob\text{”},\text{“}Op\text{”},\text{“}7\text{”},\text{“}3\text{”}]} : \mathcal{P}\, MI \quad \pi_{[2,3]} : MI \triangleright Ids}{p_{[\text{“}Bob\text{”},\text{“}Op\text{”},\text{“}7\text{”},\text{“}3\text{”}]}\ \mathbf{idx}\ \pi_{[2,3]} : \mathcal{P}\, Ids}$$

This is reduced according to R-index as follows:

$$\frac{p_{[\text{“}Bob\text{”},\text{“}Op\text{”},\text{“}7\text{”},\text{“}3\text{”}]}\ \mathbf{idx}\ \pi_{[2,3]}}{p_{[\text{“}7\text{”},\text{“}3\text{”}]}}$$

3.1.2 Implementing projection paths

Since Haskell data manipulation operations are much richer than those of SQL, we need to restrict them to the projections of HRR relations. For this purpose, we use an abstract data type `Pi a b` to express $\pi_{xs} :: A \triangleright B$ and provide a set of manipulation operators. The following implements T-compose, T-pair, T-P-Pair, and T-index, respectively, where `(!)` is a specialized version of the HRR implementation:

```
(<.>) :: Pi a b -> Pi b c -> Pi a c
(><) :: ProjectableApplicative p =>
        p a -> p b -> p (a, b)
(!) :: Projection c a -> Pi a b -> Projection c b
```

[3] http://hackage.haskell.org/package/HDBC

Pi and Projection c are instances of ProjectableApplicative. The following rule holds:

```
a ! p1 ! p2 = a ! (p1 <.> p2)
```

Here is an example of HRR relations using projection paths, where E.name', D.deptId', etc. are auto-generated projection paths:

```
employeePair :: Relation () ((String,String),(Int,Int))
employeePair = relation $ do
  e <- query employee
  d <- query department
  on $ e ! E.deptId' .=. d ! D.deptId'
  return $ (e ! E.name' >< d ! D.deptName') ><
           (e ! E.id' >< d ! D.deptId')
```

The following example shows that projection paths are composable, where fst' and snd' are standard projection paths for pairs provided in HRR:

```
emps :: Relation () (String, Int)
emps = relation $ do
  e <- query employeePair
  return $ e ! fst' ! fst' >< e ! (snd' <.> fst')
```

3.1.3 Applicative-like style

So far, we can only build nested structures using pairs. Addition of the following rules enables a style that is similar to applicative style[7] for building arbitrarily nested records:

$$\frac{A, B, C \vdash T : B \to C \quad \pi_{xs} : A \triangleright B}{A, B, C \vdash T \langle\$\rangle\, \pi_{xs} : A \triangleright C} \text{ (T-map)}$$

$$\frac{B, C \vdash T : B \to C \quad p_{xs} : \mathcal{P}\, B}{B, C \vdash T \langle\$\rangle\, p_{xs} : \mathcal{P}\, C} \text{ (T-P-map)}$$

(T is record data constructor)
(the width of type B = the width of type $C = \mathit{length}\ xs$)

$$\frac{A, B, C \vdash \pi_{xs} : A \triangleright (B \to C) \quad \pi_{ys} : A \triangleright B}{A, B, C \vdash \pi_{xs} \langle * \rangle\, \pi_{ys} : A \triangleright C} \text{ (T-ap)}$$

$$\frac{B, C \vdash p_{xs} : \mathcal{P}\,(B \to C) \quad p_{ys} : \mathcal{P}\, B}{B, C \vdash p_{xs} \langle * \rangle\, p_{ys} : \mathcal{P}\, C} \text{ (T-P-ap)}$$

(the width of type $B = \mathit{length}\ ys$)
(the width of type $C = \mathit{length}\ xs + \mathit{length}\ ys$)

$$\frac{T \langle\$\rangle\, \pi_{xs}}{\pi_{xs}} \text{ (R-map)} \qquad \frac{T \langle\$\rangle\, p_{xs}}{p_{xs}} \text{ (R-P-map)}$$

$$\frac{\pi_{xs} \langle * \rangle\, \pi_{ys}}{\pi_{xs+ys}} \text{ (R-ap)} \qquad \frac{p_{xs} \langle * \rangle\, p_{ys}}{p_{xs+ys}} \text{ (R-P-ap)}$$

We cannot simply use the Haskell Applicative class because T in T-map is limited to record data constructors. Here is an example of HRR relations using applicative-like style to return MemberInfo:

```
memberInfo :: Relation () MemberInfo
memberInfo = relation $ do
  e <- query employee
  d <- query department
  on $ e ! E.deptId' .=. d ! D.deptId'
  let mem = Member |$| e ! E.name' |*| d ! D.deptName'
      ids = Ids |$| e ! E.id' |*| d ! D.deptId'
  return $ MemberInfo |$| mem |*| ids
```

3.2 Unique column names

To avoid column name conflicts as described in Section 2.3, unique column names are generated when HRR relations are converted into SQL statements. The SQL specification only allows one or two level references (e.g. C or T.C), so column names do not conflict if the following conditions are satisfied:

- Column names of a projection specified to SELECT are unique.
- Table names specified to FROM are unique.

For column names, HRR generates unique labels based on the width of the record type corresponding to a projection. For table names, HRR's state monad increments a counter when joining. Here is the HRR representation of the example in Section 2.3:

```
noConflicts :: Relation () (String, String)
noConflicts = relation $ do
  p1 <- query person
  p2 <- query person
  on $ p1 ! P.motherId' .=. p2 ! P.id'
  return $ (,) |$| p1 ! P.name' |*| p2 ! P.name'
```

This HRR relation is converted to the following SQL statement:

```
SELECT ALL T0.name AS f0, T1.name AS f1
  FROM person T0 INNER JOIN person T1
    ON (T0.mother_id = T1.id)
```

3.3 Supporting outer joins

To handle outer joins, as discussed in Section 2.2, HRR provides the queryMaybe operator, which has a Maybe result type, in order to express nullability:

```
queryMaybe :: (MonadQualify ConfigureQuery m
              ,MonadQuery m)
           => Relation () r
           -> m (Projection Flat (Maybe r))
```

The combinations of query and queryMaybe express inner joins, left outer joins, right outer joins, and full outer joins. Here is an example of a right outer join:

```
outerJoin = relation $ do
  e <- queryMaybe employee
  d <- query department
  on $ e ?! E.deptId' .=. just (d ! D.deptId')
  return $ (,) |$| e |*| d
```

This HRR relation results in:

```
SELECT ALL T0.id AS f0, T0.name AS f1, T0.dept_id AS f2,
           T1.dept_id AS f3, T1.dept_name AS f4
  FROM employee T0 RIGHT JOIN department T1
    ON (T0.dept_id = T1.dept_id)
```

3.4 Type-propagated placeholders

Here is an HRR representation of the example in Section 2.4:

```
paramQuery :: Relation String (Int, String)
paramQuery = relation' . placeholder $ \ph -> do
    e <- query employee
    wheres $ e ! E.name' .=. ph
    return $ (,) |$| e ! E.id' |*| e ! E.name'
```

placeholder takes a function which takes a projection to express placeholders. Each element of the projection must be used exactly once, in the right order. It is the programmer's responsibility to abide by this rule. This HRR relation is converted into the following SQL statement:

```
SELECT ALL T0.id AS f0, T0.name AS f1
  FROM employee T0
 WHERE (T0.name = ?)
```

HRR relations with placeholders can be run using runQuery, as explained in the beginning of this section.

As described previously, HRR uses a phantom type for placeholders. Placeholder type information is carried to upper layers, where it is actually used. To explain this mechanism, consider the following HRR relation equivalent to paramQuery:

```
paramQuery2 :: Relation String (Int, String)
paramQuery2 = relation' $ do
  e <- query employee
  (ph', ()) <- placeholder $ \ph ->
      wheres $ e ! E.name' .=. ph
  return (ph', (,) |$| e ! E.id' |*| e ! E.name')
```

`placeholder` returns a pair containing a dummy value that holds the type of the placeholders and the result of the function in the first argument. The monad then returns both the dummy value and a final result. The `relation'` function has the following type:

```
relation' ::
    QuerySimple (PlaceHolders p, Projection Flat r)
 -> Relation p r
```

The first parameter `p` of `Relation` carries the type information of `p` in `PlaceHolders`.

3.5 Clear aggregation semantics

The key to providing clear aggregation semantics as discussed in Section 2.5 is type separation of non-aggregate and aggregate build monads. In addition to `QuerySimple`, HRR provides the `QueryAggregate` state monad for expressing aggregate build monads. This monad allows use of operators corresponding to `GROUP BY`/`HAVING` as well as ordering restrictions. Aggregate build monads can be converted into a relation using the following function:

```
aggregateRelation ::
     QueryAggregate (Projection Aggregated r)
  -> Relation () r
```

Note that the `Aggregated` type parameter restricts the results to aggregates. The following is an HRR representation of the example in Section 2.5, where `countMembers2` is *reused* in `count2`:

```
countMembers2 :: Relation () (Int, Int)
countMembers2 = aggregateRelation $ do
  e <- query employee
  d <- query department
  on $ e ! E.deptId' .=. d ! D.deptId'
  gDeptId <- groupBy $ d ! D.deptId'
  return $ (,) |$| gDeptId |*| count (e ! E.id')

count2 :: Relation () Int
count2 = relation $ do
  m <- query countMembers2
  return $ m ! fst'
```

`QueryAggregate` is used in `countMembers2`, while `QuerySimple` is used in `count2`. This separation is the key to preserving correct semantics of table forms. `count2` is converted to the following SQL statement, which uses `GROUP BY`:

```
SELECT ALL T2.f0 AS f0
  FROM (SELECT ALL T1.dept_id AS f0, COUNT(T0.id) AS f1
          FROM employee T0 INNER JOIN department T1
            ON (T0.dept_id = T1.dept_id)
         GROUP BY T1.dept_id) T2
```

4. Advanced Features

4.1 Bootstrapping via Template Haskell

HRR relations representing tables can be defined manually, but hand-written definitions are error-prone and cannot capture table schema changes, so HRR can auto-generate them using Template Haskell[9]. When HRR relations are compiled, table schemas are obtained from a database system, and HRR table relations and records are automatically defined.

We recommend defining one table per module in order to avoid column name conflicts. In the future, using OverloadedRecordFields[4] will make it possible to include multiple declarations of tables and records in one module.

For automatic code generation, it is necessary to map Haskell data types to those in a database system. HRR provides default mappings for each supported database system, currently DB2, PostgreSQL, SQLite, MySQL, Microsoft SQL Server, and OracleSQL. To ease introduction of HRR into products, mappings can be customized.

4.2 Window functions

HRR also provides SQL window functions, which are used in our database operations. The following is an example that obtains a projection of a department sequential number and an employee name:

```
rankOfName :: Relation () (Int, String)
rankOfName = relation $ do
  e <- query employee
  d <- query department
  on $ e ! E.deptId' .=. d ! D.deptId'
  let seqNo = rowNumber `over` do
        partitionBy $ d ! D.deptId'
        orderBy (e ! E.name') Asc
  return $ (,) |$| seqNo |*| e ! E.name'
```

In this example, `over` has the following signature:

```
over :: SqlProjectable (Projection c)
     => Projection OverWindow a -> Window c ()
     -> Projection c a
```

In the `Window` monad, only operators corresponding to `PARTITION BY`/`ORDER BY` can be used. The first parameter of `Window` is a phantom type for aggregation information that restricts partition column references in this monad. This HRR relation results in the following statement:

```
SELECT ALL ROW_NUMBER() OVER
            (PARTITION BY T1.dept_id
                 ORDER BY T0.name ASC) AS f0,
            T0.name AS f1
  FROM employee T0 INNER JOIN department T1
    ON (T0.dept_id = T1.dept_id)
```

4.3 Direct SQL embedding

Programmers may want to use database-system-dependent SQL code fragments that are not supported by HRR, so HRR provides a way to embed SQL code fragments directly. The following is an example of an HRR relation that uses the `substr` function to select employees whose name starts with "A":

```
substr :: (SqlProjectable p, ProjectableShowSql p)
       => p String -> p Int -> p Int -> p String
substr s begin len = unsafeProjectSql $
  "substr(" <>
  unsafeShowSql s     <> ", " <>
  unsafeShowSql begin <> ", " <>
  unsafeShowSql len   <> ")"

employeeA :: Relation () Employee
employeeA = relation $ do
  e <- query employee
  wheres $ substr (e ! E.name') (value 1) (value 1)
           .=. value "A"
  return e
```

This HRR relation is converted to the following statement:

```
SELECT ALL T0.id AS f0,
           T0.name AS f1,
           T0.dept_id AS f2
  FROM employee T0 WHERE (substr(T0.name, 1, 1) = 'A')
```

[4] https://ghc.haskell.org/trac/ghc/wiki/Records/OverloadedRecordFields

In our experience, this feature helped ease the introduction of HRR into our products.

5. Experience

5.1 Schema changes

When a database schema changes, the compiler finds code that must be changed as well, thanks to Haskell's strong type system. In our experience, this breaks down psychological barriers that make schema changes daunting.

5.2 Functional programming to SQL

HRR brings modular programming to SQL. For instance, the functionality of `filter` and `sortBy` is implemented by `wheres` and `orderBy` in HRR. Moreover, structured projections, relations, and build monads are all first class: they can be passed to and returned from functions, and they can be bound to variables. Even placeholders can use structured projections. Such features allow the use of functional programming style with SQL, which is not possible when using hand-written SQL statements in strings.

The following HRR relation is an example of functional programming style. It takes a list of filters on projections and applies them to the result projection of another HRR relation:

```
filters :: [Projection Flat MemberInfo ->
            Projection Flat (Maybe Bool)]
        -> Relation () MemberInfo
filters fs  = relation $ do
  mi <- query memberInfo
  sequence_ [ wheres (f mi) | f <- fs ]
  return mi
```

5.3 Validity of generated SQL statements

The composability of HRR relations enables HRR users to write complex relations without hesitation, even though such relations are converted into complex SQL statements. HRR users sometimes feel that it is difficult to validate such resulting statements. It would be very beneficial to prove the validity of HRR conversions, but we currently do not know how to achieve this.

5.4 Phantom types and GADTs

Recently, GADTs[8] are popular for implementing embedded, domain-specific languages in Haskell. GADTs are suitable if the final internal structures in the target domain are known. Since we implemented HRR incrementally, we were unable to make use of GADTs. Phantom types gave us more flexibility. We are still unsure if we can replace phantom types with GADTs even after HRR matures.

5.5 Monad comprehension

Monad comprehensions are generalized list comprehensions that support several notations, including database queries[3]. Since HRR relations are monads, they can be expressed using monad comprehensions. For instance, the `initialEmp` example can be written as follows:

```
relation $ [e | e <- query employee
              , () <- wheres $ e ! E.id' .<. value 10]
```

We prefer `do` notation, however, because use of monad comprehensions is syntactically awkward (note the binding to unit above) and has mismatched semantics. One example of mismatched semantics is the extended keyword `group by` which cannot be used for aggregate relations in HRR because the semantics of the keyword assume that computations are on Haskell data, while actual aggregations are done by database systems.

6. Related Work

Opaleye is another next-generation implementation of HaskellDB. It supports record-based projections and outer joins as well as provides semantically-clear composability. To provide semantically-clear aggregate queries, each column must have exactly one level of aggregation or must appear in the `GROUP BY` clause (but not both). Opaleye makes use of *profunctors* to satisfy this condition. While HRR uses monads to build queries, Opaleye uses arrows. Thanks to the profunctor approach, writing queries in Opaleye is like programming with the list arrow in Haskell. As of this writing, PostgreSQL is the only database system supported.

In join products, Opaleye automatically combines sub-query `WHERE` clauses into an outer `WHERE` clause, which prevents capture of the projection of the left sub-query in the join-product. HRR does not do this, as it is not easy to combine `WHERE` clauses of join-product sub-queries that involve union-like operations (`UNION`, `EXCEPT`, and `INTERSECT`), which HRR supports. This is a trade-off between Opaleye's type-safety and HRR's simplicity and expresiveness of SQL statements. One of our priorities in future work is to match the type-safety of Opaleye by implementing semantics-preserving query transformation.

7. Conclusion

This paper describes Haskell Relational Record (HRR), an embedded, domain-specific language that provides composability and type-safety to SQL statements. HRR overcomes issues of predecessors by providing structured projections, unique column naming, outer joins, type-propagated placeholders, and clear aggregation semantics. It also supports automatic schema code generation, direct SQL embedding, and window functions. We have used HRR in our products for more than two years and have confirmed its stability and usability.

Acknowledgments

We would like to deeply thank Travis Cardwell for thoroughly reviewing an early draft of this paper. We would like to express our gratitude to Michael Snoyman, Andres Löh, and Tom Ellis for their feedback.

References

[1] B. Bringert and A. Höckersten. Student Paper: HaskellDB Improved. In *Proceedings of Haskell Workshop*, 2004.

[2] B. R. Gaster and M. P. Jones. A Polymorphic Type System for Extensible Records and Variants. *Technical Report NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham*, 1996.

[3] G. Giorgidze, T. Grust, N. Schweinsberg, and J. Weijers. Bringing Back Monad Comprehensions. In *Proceedings of Haskell Symposium*, 2011.

[4] *Information technology - Database languages - SQL.* ISO/IEC 9075 standard, 2011.

[5] D. Leijen and E. Meijer. Domain Specific Embedded Compilers. In *Proceedings of the 2nd Conference on Domain-Specific Languages*, 1999.

[6] S. Marlow et al. *Haskell 2010 Language Report*, 2010.

[7] C. McBride and R. Paterson. Applicative Programming with Effects. *Journal of Functional Programming*, 18, 2008.

[8] T. Schrijvers, S. P. Jones, M. Sulzmann, and D. Vytiniotis. Complete and Decidable Type Inference for GADTs. In *Proceedings of ICFP*, 2009.

[9] T. Sheard and S. P. Jones. Template metaprogramming for Haskell. In *Proceedings of Haskell Workshop*, 2002.