Submitted to Haskell Symposium 2016

Haskell Relational Record

A Pragmatic Embedded System for Type-Safe and Composable SQL Queries

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

Hand-written SQL statement strings are error-prone because SQL syntax is not checked in host programming languages. One solution to this problem is to use an embedded domain-specific language (EDSL) in a strongly-typed programming language, which generates syntactically-correct SQL statements when the code passes type checking. HaskellDB, recognized as the first implementation of such an EDSL, enables building large queries from small, well-tested queries. Unfortunately, HaskellDB is insufficient for real-world business because it does not support some necessary operations, such as outer joins. Moreover, aggregated queries do not keep the structure of the corresponding tables and nested Haskell record types are not allowed as query result types. To solve these problems, we implemented Haskell Relational Record (HRR), a pragmatic EDSL in Haskell that supports a large part of the SQL standard, including outer joins and correlated subqueries. HRR provides query composability yet handles both non-aggregate and aggregate queries correctly. HRR query results can be of any Haskell record type. SQL expressions in queries are typed with nested Haskell records. We show algorithms for primitive operators of record manipulation which preserve correspondence between nested Haskell records and flat SQL structures. HRR's arrow interface provides type-safety—generated SQL statements do not contain syntax errors, type errors, or reference errors. This also holds when using the monad interface, aside from cases where correlated subqueries are joined. More than three years of production use demonstrates stability and usability of HRR.

Categories and Subject Descriptors H.2.3 [*DATABASE MANAGEMENT*]: Languages—Embedded System

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

1. Introduction

SQL (ISO 2011) is the standard query language for relational database systems. Programs often use SQL statements to manipulate data in database systems, but hand-written SQL statement strings are error-prone because SQL syntax is not checked in host programming languages. It is difficult to write complicated SQL statement strings correctly, and they are not reusable. One approach to making SQL statements reusable is the integration of query functionality into programming languages. LINQ (Meijer et al. 2006) is a prominent example, and a more seamless integration is found in SML# (Ohori and Ueno 2011).

Another notable approach, which does not require modification of programming languages, is the use of an embedded domain-specific language (EDSL). HaskellDB (Leijen and Meijer 1999; Bringert and Höckersten 2004) is such an EDSL in Haskell (Marlow et al. 2010), with excellent features. Since HaskellDB queries are *composable*, large queries can be built from well-tested, small queries. HaskellDB queries that compile generate *syntactically-correct* SQL statements.

In 2012, Asahi Net, Inc. attempted to use HaskellDB for its business. Asahi Net is a Japanese Internet service provider with about 587,000 residential customers, as of March 2016. 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 hand-written SQL statements in Java, Perl, and other programming languages. In the Java and Haskell code, 2,622 unique SQL statements are used, of which 1,770 are `SELECT` statements. Note that the count does not include queries in untyped languages, where it is difficult to make an accurate count.

Unfortunately, use of HaskellDB in practical database operations discloses its drawbacks. It is insufficient for real-world business because it does not support some necessary operations, such as outer joins and correlative subqueries. Note that 5.0% and 1.8% of the `SELECT` statements are outer joins and correlative subqueries, respectively, at Asahi Net. Moreover, if aggregate queries are reused, they do not keep the structure of the corresponding *tables*, as described in Section 6.1. Query result types are limited to flat, *heterogeneous lists*, which cannot correspond to nested Haskell records.

We developed a pragmatic EDSL for SQL, called Haskell Relational Record (HRR), from scratch, to help programmers write complicated, real-world SQL statements correctly. Like HaskellDB, HRR provides composability to enable reuse of code fragments at various levels. For example, queries in HRR can be composed using joins, as well as combined using set operations such as union, intersection, and difference. To resolve HaskellDB issues, we designed HRR to support a large part of the SQL standard, including outer joins and correlated subqueries.

HRR provides separate query monads for non-aggregate and aggregate queries. This ensures that aggregate query table structures are maintained even if the queries are reused. Query results can be of any Haskell record type, and query expressions are typed with Haskell records so that they can be reused. To support this, we developed primitive operators which preserve correspondence between nested Haskell record types and flat SQL structures.

If HRR code passes type checking by a Haskell compiler, running the corresponding executable generates syntactically-correct SQL statements. The algorithm to generate SQL statements is straightforward—no optimization is carried out. For instance, a nested query is translated into a nested SQL query. When the HRR arrow (Hughes 1998) interface is used, generated SQL statements are *valid*, where "valid" means that they do not have type errors or reference errors, but they may include errors relating to null. When using the monad interface, this does not hold in cases where correlated subqueries are joined—generated SQL statements contain reference errors. In practice, this is not a serious problem because database systems can always detect the errors.

When HRR queries are compiled by a Haskell compiler, the schema of target tables are obtained and necessary definitions for the queries are automatically generated. The compiler checks that the queries are consistent with the generated definitions. In our experience, this schema capture mechanism breaks down psychological barriers that make schema changes daunting. Currently, HRR supports DB2, PostgreSQL, SQLite, MySQL, Microsoft SQL Server, and OracleSQL. To support such a wide variety of database systems, HRR generates standard SQL statements only, avoiding use of database-system-specific features. We hope that this eases database system migration.

HRR has been in use at Asahi Net since March 2013, and more than three years of production use demonstrates its stability and usability. We have also released HRR as open-source software.[1]

This paper describes HRR, primarily focusing on its type-safety and composability which help programmers to write complicated SQL statements correctly. The contributions of this paper are as follows:

- We provide a pragmatic EDSL for SQL which provides composability. HRR is type-safe when the arrow interface is used, where "type-safe" means that generated SQL statements are syntactically-correct and valid (i.e. with no type or reference errors, but possibly with null errors). This also holds when using the monad interface, except in cases where correlated subqueries are joined.
- We show that type separation of non-aggregate and aggregate queries brings type-safety, and converting these types into a single query type enables query composability.
- We show that SQL expressions in queries are typed with nested Haskell records and provide algorithms for primitive operators of record manipulation which preserve correspondence between nested Haskell records and flat SQL structures.

This paper is organized as follows. Section 2 describes HRR by example, and composability is discussed in Section 3. Section 4 explains how HRR code is translated into SQL statements. Section 5 discuss SQL expressions with Haskell record types. Related work is covered in Section 6, and our conclusion is in Section 7.

2. Basic Concepts by Example

This section explains the basic concepts of HRR by example.[2] This paper focuses on SELECT statements (i.e. SQL queries), which we have found are the most challenging. Throughout this paper, the monad interface of HRR is used for simplicity.

2.1 A Simple Query

We use the following table of employees throughout this paper:

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

To use this table from Haskell, it is necessary to map Haskell records to those in a database system. Haskell records representing tables can be defined manually, but hand-written definitions are error-prone and cannot capture table schema changes. HRR can auto-generate them using Template Haskell (Sheard and Peyton Jones 2002): when HRR code is compiled, table schemas are obtained from a database system and Haskell records are defined automatically. To ease the introduction of HRR into products, mappings can be customized.

If we use SQLite, and file `company.db` contains all of the tables used in this paper, we should prepare the following `Employee.hs` for the `Employee` table:

```
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FlexibleInstances #-}
module Employee where
import Database.Record.TH.SQLite3

$(defineTable "company.db" "employee")
```

The following data type is automatically defined at compile time:

```
data Employee = Employee {
    id     :: !Int
  , name   :: !String
  , deptId :: !Int
  } deriving (Show)
```

2.1.1 SELECT Statement Relations

To work with auto-generated Haskell records, we need additional components to represent SELECT *statements*. In HRR, such components are called *relations*. For the example above, the following relation, which selects employees from the `Employee` table, is also automatically created:

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

`Relation` is a type for relations, with two phantom parameters(Leijen and Meijer 1999). The first is the placeholder type (see Appendix A). The second indicates the result type. Here, `employee` represents a SELECT statement, with no placeholder parameters, that returns `Employee` values. We can print the SELECT statement generated by the `employee` relation:

```
GHCi> print employee
SELECT id, name, dept_id FROM employee
```

A `Relation` value can be converted to an SQL string and sent to a database system using the following function:

```
runRelation :: (IConnection conn
               ,ToSql SqlValue p
               ,FromSql SqlValue a) =>
               conn -> Relation 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. `SqlValue` is a type of SQL value in HDBC. The third parameter, `p`, holds any placeholder values.

The following example demonstrates running `employee` with SQLite:

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

[2] To simplify explanation, this paper uses wrapper modules and definitions exported from the `relational-record-examples` package, available in Hackage.

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

```
GHCi> :load Employee
GHCi> import Database.Relational.Query.SQLite3
GHCi> conn <- connectSqlite3 "company.db"
GHCi> runRelation conn employee ()
[Employee {id = 1, name = "Smith", deptId = 100}
,Employee {id = 20, name = "Parker", deptId = 101}]
```

2.2 A Query with Filtering

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 `n`:

```
import qualified Employee as E
import Database.Relational.Query

earlyEmp :: Int -> Relation () E.Employee
earlyEmp n = relation $ do
  e <- query E.employee
  wheres $ e ! E.id' .<. value n
  return e
```

Each component in this example is described later, and Appendix B shows the auto generated code for `Employee.hs`. Note that `n` is just a parameter of the Haskell function, not a placeholder parameter. Let's choose 10 for `n`. In this case, `earlyEmp` is converted to the following SQL statement:

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

Running `(earlyEmp 10)` results in:

```
GHCi> runRelation conn (earlyEmp 10) ()
[Employee {id = 1, name = "Smith", deptId = 100}]
```

Later, we will look at this filtering example in more detail and use it to highlight some important concepts of HRR.

2.2.1 Query Monads

`SELECT` statements are roughly categorized into non-aggregate and aggregate queries. HRR provides two separate monads, called *query monads*, to build them. This type separation is one of the keys to type-safety. In this paper, we use two kinds of query monads:

- `QuerySimple` — a query monad for non-aggregate queries, which stores information on joining, table/column naming, filtering, ordering, etc.
- `QueryAggregate` — a query monad for aggregate queries, which, in addition to the information stored by `QuerySimple`, stores information on `GROUP BY` and `HAVING`, as well as ordering restrictions

As seen in `earlyEmp`, relations can be reused with the following pattern:

```
relation $ do
  e <- query <RELATION>
  ...
  return e
```

The `query` function converts a relation to an action in the query monad, which is specialized to `QuerySimple` in the above example:

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

`QuerySimple` and `QueryAggregate` are instances of `MonadQuery`. The `relation` function restricts the type of the query monad to `QuerySimple`:

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

Using this pattern, a relation is converted to a query monad and then converted back to a `Relation` for composability, as explained in Section 3.

2.2.2 Projections for SQL Expressions

SQL statements consist of SQL *expressions*. In HRR, SQL expression components are called *projections*. Note that the term "projection" in this paper is different from SQL terminology, where it is used to indicate how to extract columns. The concept of projection is precisely explained in Section 5.

`Projection` is a data type for projections, with two phantom parameters. The first indicates the context of SQL expressions, taking one of the following values:

- `Flat` — the base context
- `Aggregated` — the return value of `GROUP BY` or aggregate functions
- `Exists` — arguments for `EXISTS`.
- `OverWindow` — arguments for `OVER`

The second parameter indicates the Haskell record type of the SQL query result.

2.2.3 Filtering Expressions

Line "`wheres $ e ! E.id' .<. value n`" of the example in Section 2.2 implements the filter to select employees whose identifier is less than `n`. The expressions have the following types:

```
wheres    :: Projection Flat (Maybe Bool)
          -> QuerySimple ()
(.<.)     :: Projection c a -> Projection c a
          -> Projection c (Maybe Bool)
e         :: Projection Flat Employee
(! E.id') :: Projection Flat Employee
          -> Projection Flat Int
value     :: a -> Projection Flat a
```

`wheres` implements the `WHERE` clause. It stores a query condition in the query monad, which is `QuerySimple` in this example. `id'` is an auto-generated accessor to `id` of the `Employee` Haskell record, in the projection level. `(!)` is a function that applies an accessor to a Haskell record. `value` creates an SQL constant expression from a Haskell constant value. `(.<.)` is the less-than operator in the projection level. `(.<.)` returns `Maybe Bool`, which `wheres` takes because SQL boolean is nullable. The signatures of HRR functions used in this paper are shown in Figure 3 and Figure 6.

2.3 Inner Join

In HRR, inner joins are expressed using a pair of `query` actions. To give an example of an inner join, we now introduce a table for departments:

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

The following HRR code selects pairs of employee names and their department names:

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

on implements the ON clause. (><) is an operator that produces a pair while (.=.) is the equal operator in the projection level. This HRR code is converted into the following SQL statement:

```
SELECT ALL T0.name AS f0, T1.dept_name AS f1
  FROM employee T0 INNER JOIN department T1
    ON (T0.dept_id = T1.dept_id)
```

Running this code results in:

```
GHCi> runRelation conn innerJoin ()
[("Smith","Personnel"),("Parker","Admin")]
```

2.4 Outer Join

To handle outer joins, HRR provides the queryMaybe operator, which uses a Maybe result type to express nullability:

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

The combinations of query and queryMaybe express the following four joins:

- query then query — inner joins
- query then queryMaybe — left outer joins
- queryMaybe then query — right outer joins
- queryMaybe then queryMaybe — full outer joins

Here is an example of a left outer join that selects departments and early employees in the department, if they exist:

```
outerJoin :: Relation () (String,Maybe String)
outerJoin = relation $ do
  d <- query department
  e <- queryMaybe employee
  on $ e ?! E.deptId' .=. just (d ! D.deptId')
  on $ e ?! E.id' .<. just (value 10)
  return $ d ! D.deptName' >< e ?! E.name'
```

(?!) is the Maybe version of (!), and just is a function that injects a value into Maybe, in the projection level. The signatures of these functions are shown in Figure 6. This HRR relation is converted into the following SQL statement:

```
SELECT ALL T0.dept_name AS f0, T1.name AS f1
  FROM department T0 LEFT JOIN employee T1
    ON (T1.dept_id = T0.dept_id) AND (T1.id < 10)
```

Running the outerJoin relation results in:

```
GHCi> runRelation conn outerJoin ()
[("Personnel",Just "Smith")
,("Admin",Nothing)]
```

2.5 An Aggregate Query

A SELECT statement with a GROUP BY clause may only select grouping elements and expressions using an aggregate function. Violating this restriction is a common mistake in hand-written SQL. Consider the following (incorrect) example:

```
SELECT name, dept_id, COUNT(id) -- incorrect
  FROM employee
 GROUP BY dept_id
```

Here, name is neither a grouping element nor an expression using an aggregate function. Some database systems return ill-chosen values for this kind of incorrect SELECT statement, and the mistake may go unnoticed. This section explains how such errors can be prevented using query monads.

As explained in Section 2.2.1, QueryAggregate is used for aggregate queries. It can be converted into a relation using the following function:

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

If the query monad is QueryAggregate, the first parameter of Projection must be Aggregated. Type consistency is guaranteed because aggregateRelation requires this combination. Note that relation, described in Section 2.2.1, requires the combination of QuerySimple and Flat.

The following is an example of an aggregate query that returns the number of employees for each department:

```
countMembers :: Relation () (String, Int)
countMembers = aggregateRelation $ do
  e <- query E.employee
  d <- query D.department
  on $ e ! E.deptId' .=. d ! D.deptId'
  gDeptName <- groupBy $ d ! D.deptName'
  return $ gDeptName >< count (e ! E.id')
```

count implements the COUNT aggregate function of SQL. groupBy implements the GROUP BY clause. It takes a grouping element and returns an aggregate-typed key that can be used in the result.

```
groupBy :: Projection Flat r
        -> QueryAggregate (Projection Aggregated r)
```

countMembers is converted to the following SQL statement, which uses GROUP BY:

```
SELECT ALL T1.dept_name 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_name
```

3. Relation Composability

This section discusses relation composability. As described in Section 2.2.1 and Section 2.5, the QuerySimple and QueryAggregate query monads can be converted to relations. Relations are composable regardless of the inner query monads used.

As described later, relations using the arrow interface are always type-safe but those using the monad interface are not. If correlated subqueries are joined in the monad interface, invalid SQL statements are generated. Even though the monad interface does not provide perfect type-safety, HRR users tend to use it because of its simplicity. Fortunately, incorrect correlated subqueries can be always detected as reference errors when executing the corresponding relations.

3.1 Combining

To show combining relations with set operations, consider the following new table for past members:

```
CREATE TABLE PastMembers (
  year      INTEGER     NOT NULL
, dept_name VARCHAR(32) NOT NULL
, headcount INTEGER     NOT NULL
);
```

The PastMembers table contains the number of employees per department, in or before 2015. The following code extracts this information:

```
import qualified PastMembers as P

pastMembers' :: Relation () (Int,(String,Int))
pastMembers' = relation $ do
  p <- query P.pastMembers
  return $ p ! P.year' ><
           (p ! P.deptName' >< p ! P.headcount')
```

The following code counts the current number of employees per department:

```
countMembers2016 :: Relation () (Int,(String,Int))
countMembers2016 = aggregateRelation $ do
  e <- query E.employee
  d <- query D.department
  on $ e ! E.deptId' .=. d ! D.deptId'
  gDeptName <- groupBy $ d ! D.deptName'
  return $ value 2016 ><
           (gDeptName >< count (e ! E.id'))
```

The inner query monads of `pastMembers'` and `countMembers2016` are `QuerySimple` and `QueryAggregate`, respectively. They can be composed with `unionAll`:

```
memberHistory :: Relation () (Int,(String,Int))
memberHistory =
  countMembers2016 `unionAll` pastMembers'
```

This code is converted to the following SQL statement:

```
 SELECT ALL 2016 AS f0, T1.dept_name AS f1,
            COUNT(T0.id) AS f2
   FROM employee T0 INNER JOIN department T1
     ON (T0.dept_id = T1.dept_id)
  GROUP BY T1.dept_name
UNION ALL
 SELECT ALL T2.year AS f0, T2.dept_name AS f1,
            T2.headcount AS f2
   FROM pastmembers T2
```

3.2 Joining

To show nested relations, we reuse `countMembers`, defined in Section 2.5. Here is an example of nesting relations where the query monad of `countMembers2016'` is `QuerySimple` while that of `countMembers` is `QueryAggregate`:

```
countMembers2016' :: Relation () (Int,(String,Int))
countMembers2016' = relation $ do
  m <- query countMembers
  return $ value 2016 >< m
```

This code is converted into the following SQL:

```
SELECT ALL 2016 AS f0, T2.f0 AS f1, T2.f1 AS f2
  FROM (SELECT ALL T1.dept_name 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_name) T2
```

Other systems, such as QUEΛ (Suzuki et al. 2016), convert a nested query to a flat query. As seen in this example, however, HRR translates relations to SQL queries in a straightforward manner—a nested query is left as a nested query. Refer to Section 4 for details.

3.3 Correlation

SQL provides correlated subqueries, which use values from outer queries. The specification allows correlated subqueries to be specified in the `WHERE` clause, the `HAVING` clause, etc., but not in the `FROM` clause. In other words, correlated subqueries must not be joined. In HRR, a correlated subquery can be expressed with a function whose arguments are projections and return value is a relation. The following is an example of a correlated subquery that returns identifiers for early employees in a given department:

```
depEarlyEmp :: Int -> Projection Flat D.Department
            -> Relation () Int
depEarlyEmp n d = relation $ do
  e <- query E.employee
  wheres $ e ! E.id' .<. value n
  wheres $ e ! E.deptId' .=. d ! D.deptId'
  return $ e ! E.id'
```

Here is an example relation using this correlated subquery in the `WHERE` clause:

```
correlative :: Relation () String
correlative = relation $ do
  d <- query department
  exist <- queryList $ depEarlyEmp 10 d
  wheres $ exists exist
  return $ d ! D.deptName'
```

This relation selects names of departments that have at least one early employee. `queryList` converts a relation to a subquery whose return value type is a set of projections. `exists` implements the `EXISTS` clause. They have the following signatures:

```
queryList :: Relation () r
          -> QuerySimple (ListProjection
                           (Projection Exists) r)
exists :: ListProjection (Projection Exists) r
       -> p (Maybe Bool)
```

`correlative` is converted into the following (valid) SQL statement:

```
SELECT T0.dept_name AS f0
  FROM department T0
 WHERE (EXISTS (SELECT T1.id AS f0 FROM employee T1
                WHERE (T1.id < 10)
                  AND (T1.dept_id = T0.dept_id)))
```

Since the correlated subquery is used in the `WHERE` clause, it can refer to the table labeled with `T0` in the `FROM` clause.

3.4 Invalid Correlation

Unfortunately, HRR does not provide perfect type-safety for correlated subqueries if the monad interface is used. Consider the following correlated subquery, which returns a department name for a given employee:

```
empDep :: Projection Flat Employee
       -> Relation () Department
empDep e = relation $ do
    d <- query department
    wheres $ e ! E.deptId' .=. d ! D.deptId'
    return d
```

The following code uses this correlated subquery for joining:

```
wrongJoin :: Relation () (String,String)
wrongJoin = relation $ do
  e <- query employee
  d <- query $ empDep e
  return $ e ! E.name' >< d ! D.deptName'
```

`wrongJoin` is converted into the following (invalid) SQL statement:

```
SELECT ALL T0.name AS f0, T2.f1 AS f1
  FROM employee T0
 INNER JOIN (SELECT ALL T1.dept_id AS f0,
                        T1.dept_name AS f1
              FROM department T1
             WHERE (T0.dept_id = T1.dept_id)) T2
    ON (0=0)
```

The inner `SELECT` refers to the invisible outer table `T0`. Running this code results in the following reference error:

```
GHCi> runRelation conn wrongJoin ()
*** Exception:
  SqlError
  { seState = ""
  , seNativeError = 1
  , seErrorMsg = "... : no such column: T0.dept_id"
  }
```

This error happens because monadic scope is different from that of SELECT statements. A monad implementation therefore does not provide type-safety for correlated subqueries.

3.5 Arrow Interface

Opaleye developers suggested that this restriction can be implemented using arrows. Here is an arrow version of `empDep`:

```
empDepA :: Projection Flat Employee
        -> Relation () Department
empDepA e = relation $ proc () -> do
  d <- query department -< ()
  wheres -< e ! E.deptId' .=. d ! D.deptId'
  returnA -< d
```

The following is an arrow version of `wrongJoin`:

```
wrongJoinA :: Relation () (String,String)
wrongJoinA = relation $ proc () -> do
  e <- query employee -< ()
  d <- query $ empDepA e -< ()
  return -< e ! E.name' >< d ! D.deptName'
```

Only unit is allowed as input to `query` arrows. Since the argument `e` of `empDepA` is out of scope in arrow notation, the Haskell compiler can detect it: "`Not in scope: ‘e’`".

4. Converting Relations to SQL Queries

This section describes the internal structures of HRR and how relations are translated into SELECT statements.

4.1 Monad Transformers for Query Monads

HRR's query monads are implemented with monad transformers (Liang et al. 1995). In addition to a base state monad that is used for labeling, HRR uses the following monad transformers:

- state transformer for `FROM`, to build the join tree
- writer transformer for `WHERE`, to accumulate restrictions
- writer transformer for `GROUP BY`, to accumulate grouping terms
- writer transformer for `HAVING`, to accumulate restrictions
- writer transformer for `ORDER BY`, to accumulate ordering keys

`QuerySimple` utilizes the base, `FROM`, `WHERE`, and `ORDER BY` transformers, while `QueryAggregate` utilizes the base and all of the above transformers. As described previously, `query`, `queryMaybe`, etc. can be used in both non-aggregate and aggregate query monads thanks to the overloading technique described in (Jones 1995).

4.2 Translation Algorithm

`Relation` is defined using `SubQuery` and two phantom type parameters:

```
newtype Relation p r = Relation SubQuery
```

Simplified, `SubQuery` is defined as follows:

```
data SubQuery = Table <auto generated table>
              | Union SubQuery SubQuery
              | Simpl <result columns>
                      <join subqueries tree>
                      [<where condition>]
                      [<order by>]
              | Aggre <result columns>
                      <join subqueries tree>
                      [<where condition>]
                      [<group by>]
                      [<having condition>]
                      [<order by>]
```

`Table` holds an auto-generated primitive relation, and `Union` stores two relations concatenated with `unionAll`. `Simpl` and `Aggre` keep information stored in `QuerySimple` and `QueryAggregate`, respectively. This is why `relation` and `aggregateRelation` can convert `QuerySimple` and `QueryAggregate` to `Relation`, respectively.

When a `Relation` value is used in an aggregate or non-aggregate query monad, the inner structure of the `Relation` value is not necessary; only the parameters are required. This `Relation` value is stored in `<join subqueries tree>` when the outer query is converted to another `Relation` value.

The following `toSQL` function summarizes the algorithm used to convert a `SubQuery` value to a SELECT statement:

```
toSQL :: SubQuery -> String
toSQL (Table tbl) = tableToSQL tbl
toSQL (Union l r) =
     paren (toSQL l)
  ++ "UNION ALL"
  ++ paren (toSQL r)
toSQL (Flat rc jst wc ob) =
     "SELECT"
  ++ intercalateComma rc
  ++ expandJoinProduct jst -- calls toSQL
  ++ expandWhere wc        -- calls toSQL
  ++ expandOrderBy ob
toSQL (Aggregated rc jst wc gb hc ob) =
     "SELECT"
  ++ intercalateComma rc
  ++ expandJoinProduct jst -- calls toSQL
  ++ expandWhere wc        -- calls toSQL
  ++ expandGroupBy gb
  ++ expandHaving hc       -- calls toSQL
  ++ expandOrderBy ob
```

`SubQuery` data structures represent SELECT statements in a straightforward manner, so `toSQL` can simply concatenate stored information. Functions that we have not described, such as `expandWhere`, also concatenate stored information with proper spacing. Note that `expandJoinProduct` traverses `<join subqueries tree>` and calls `toSQL` for each subquery.

4.3 Avoiding Column Name Conflicts

Since column names are unique within a table but are not globally unique, they are prone to conflict. Self join is a basic example. Consider the following `Person` table:

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

A naive SQL statement to obtain a list of mother-child pairs would be:

```
SELECT name, name -- incorrect
  FROM person INNER JOIN person
    ON (mother_id = id)
```

One cannot tell which table each `name` belongs to. Tables should therefore be properly labeled, and column names should be used with proper qualifications. HRR avoids column name conflicts by giving global unique names to tables and giving table-unique names to columns. The concrete algorithm is as follows:

Base case: Assume that table names of existing tables in a database system are globally unique, and assume that all column names are table-unique.

Recursive case: Give global-unique names to tables specified in `FROM` clauses, and give table-unique names to the result elements.

This algorithm ensures table name uniqueness even if the `SELECT` statements are nested. Column names used in `SELECT` statements are specified with qualified names such as `T.C`. Even if `C` is not unique, `T.C` is global unique since `T` is global unique.

Code to obtain a list of mother-child pairs is as follows:

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

This is converted into the following SQL statement, with no column conflict:

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

5. SQL Expressions with Haskell Records

Type errors are another common mistake in hand-written SQL statements. The following is an incorrect SQL example that compares an integer and a string:

```
SELECT name
  FROM employee
 WHERE id = 'Smith' -- incorrect
```

Type errors can be prevented by typing SQL expressions, of course. The correct HRR code is as follows:

```
pair :: Relation () String
pair = relation $ do
  e <- query E.employee
  wheres $ e ! E.name' .=. value "Smith"
  return $ e ! E.name'
```

As described in Section 2.2, types of SQL expressions are represented as `Projection`, and the Haskell type system checks type consistency. This HRR code is converted into the following `SELECT` statement:

```
SELECT ALL T0.name AS f1
  FROM employee T0
 WHERE (T0.name = 'Smith')
```

We designed HRR to allow any Haskell record types to the result type of queries. Since queries are used in other queries as SQL expressions, SQL expressions should be typed with Haskell record types in HRR.

In Haskell, field accessors of records are functions. To compose records, the applicative style (McBride and Paterson 2008) is wildly accepted. Since field accessors are instances of `Reader` monad, the applicative style can be used even with field accessors. The following is an example to swap members of a pair using field accessors in the applicative style:

```
swap :: (a,b) -> (b,a)
swap = (,) <$> snd <*> fst
```

HRR provides such operators of record manipulation. This section describes how to implement primitive operators to compose and decompose Haskell records which correspond to flat SQL structures. In HRR, projections, `(!)` with accessors, and the applicative-like style described below correspond to Haskell records, field accessors, and the applicative style, respectively.

5.1 Composing Projections

HRR provides the applicative-like style that is similar to the applicative style. Consider the following data types:

```
data Member = Member {
    memName :: String
  , memDept :: String
  }
data Ids = Ids {
    idEmp :: Int
  , idDept :: Int
  }
data MemberInfo = MemberInfo {
    miMem :: Member
  , miIds :: Ids
  }
```

Here is an example of HRR relations composing projections in the applicative-like style:

```
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
```

`(|$|)` and `(|*|)` correspond with `(<$>)` and `(<*>)`, respectively. See Figure 3 for their signatures. We cannot simply use the Haskell `Applicative` typeclass because function types to be mapped are limited to Haskell record data constructors only. The above code is converted to the following SQL statement:

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

5.2 Composing Accessors

In addition to projections, accessors are also used in the applicative-like style. Combining accessors and the applicative-like style, we can define type conversion functions which are not unthinkable in the SQL world. The following code automatically generates accessors for `Member`, `Ids`, and `MemberInfo`:

```
$(makeRecordPersistableDefault ''Member)
$(makeRecordPersistableDefault ''Ids)
$(makeRecordPersistableDefault ''MemberInfo)
```

Here is an example of an accessor converting `MemberInfo` back to `Employee`:

```
toEmployee :: Pi MemberInfo E.Employee
toEmployee = E.Employee |$| miIds' <.> idEmp'
                        |*| miMem' <.> memName'
                        |*| miIds' <.> idDept'
```

`Pi` is a type for accessors. "`Pi a b`" indicates that type `b` is extracted from type `a`. `(<.>)` is a composition operator for accessors.

$Int, String, Day \in$ LType		(leaf types)
$X \in$ HRec		(record types)
$A, B, C \in$ HType	$::=$	$Int \mid String \mid Day \mid X \mid (A, B) \mid$
		$A \to B$
		(type of record data constructor)
$\tau \in$ Type	$::=$	$\mathcal{P}A \mid A \triangleright B$
$xs, ys \in$ EList		(Haskell list of SQL expressions)
$ns, ms \in$ IList		(Haskell list of integer indices)
$T \in$ RCons		(record data constructor)
$x \in$ Term	$::=$	$p_{xs} \mid p_{xs} \oplus p_{ys} \mid$
		$T \ \langle\$\rangle \ p_{xs} \mid p_{xs} \ \langle*\rangle \ p_{ys} \mid$
		$\pi_{ns} \mid \pi_{ns} \oplus \pi_{ms} \mid$
		$T \ \langle\$\rangle \ \pi_{ns} \mid \pi_{ns} \ \langle*\rangle \ \pi_{ms} \mid$
		$p_{xs} \ \mathbf{idx} \ \pi_{ns}$
$\Gamma \in$ TEnv	$::=$	$\bullet \mid \Gamma, x : \tau \mid \Gamma, T : A \to B$

$\Gamma \vdash x : \tau$ means the term x is typed as τ under the type environment Γ.

Figure 1. Definitions of notations for typing rules

5.3 Primitive Operators

As describe earlier, `Projection` is a type for projections while `Pi` is that for accessors. Primitive operators to compose and decompose records in HRR are:

- `(<.>)` — composing
- `(!)` — indexing
- `(|$|)` — mapping
- `(|*|)` — applying

In the next three subsections, we give their implementations which preserve correspondence between Haskell record types and flat SQL structures over the record operations.

5.4 Definition of Projections

Precisely speaking, projections in HRR are correspondences between SQL expressions and Haskell data types. Internally, these Haskell data types are represented as a list of SQL expressions. With notations defined in Figure 1, projections can be defined as follows:

$$p_{[\langle exp_1\rangle, \langle exp_2\rangle, ..., \langle exp_w\rangle]} : \mathcal{P}\ A$$

Here, A denotes a Haskell record type, and $\mathcal{P}\ A$ indicates the projection type. p is a constructor that takes a list of SQL expressions and builds a value whose type is $\mathcal{P}\ A$. A is associated with the SQL list $[\langle exp_1\rangle, \langle exp_2\rangle, ..., \langle exp_w\rangle]$, where indices are based on *depth-first* ordering and w is the width of A.

The following projection is for the result type of the SQL statement for `memberHistory` in Section 3.1:

$$p[2016, T1.dept\_name, COUNT(T0.id)] \\ : \mathcal{P}(Int, (String, Int))$$

Virtual expressions are also typed. For example, `e` in `earlyEmp`, defined in Section 2.2, does not appear in the corresponding SQL statement but has the following projection:

$$p[T0.f0, T0.f1] : \mathcal{P}Employee$$

5.5 Definition of π Functions

Hereafter, we call HRR accessors π functions. The following notation is used to represent the π function, to extract Haskell record B from A:

$$\pi_{ns} : A \triangleright B$$

Here, ns is a list whose elements are indices of A. The following is the definition of the π functions with notations defined in Figure 1:

$$\frac{}{\Gamma \vdash \pi_{[n_0,..,n_{w-1}]} : A \triangleright B} \ \text{(T-pi)}$$

$(0 \le n_i \le v - 1, v$ is the width of type $A)$
$(0 \le i \le w - 1, w$ is the width of type $B)$
$(i^{\text{th}}$ leaf of Haskell record B
corresponds with $n_i{}^{\text{th}}$ leaf of Haskell record $A)$

Note that indices are of depth-first order. Figure 2 illustrates T-pi:

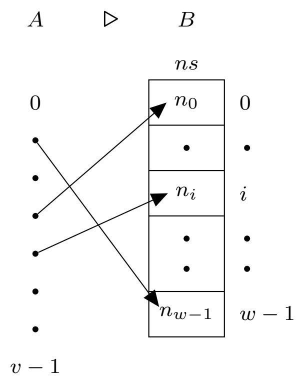

Figure 2. The definition of π. n_i is the i^{th} element of ns, whose value is an index of A.

Here are examples of π functions:

$$\pi_{[1,2]} : (Int, (String, Day)) \triangleright (String, Day)$$
$$\pi_{[0,2]} : (Int, (String, Day)) \triangleright (Int, Day)$$
$$\pi_{[0,2,1]} : (Int, (String, Day)) \triangleright ((Int, Day), String)$$

5.6 Implementation

Figure 3 summaries the signature of the primitive operators. With these signatures, the Haskell type system infers types of phantom parameters according to the typing rules defined in Figure 4. For instance, the signature of `(<.>)` represents T-compose. Figure 3 also contains this correspondence relationship.

T-P-map restricts T (functions to be mapped) to Haskell record data constructors. Data constructors decide the number of parameters, their types and their ordering. Since Haskell records can be nested, this rule is applied recursively. For a given record, this rule creates a tree whose leaf elements are non-record (i.e. single column) types. A record build with T-P-map and T-P-ap has such a tree. A list of non-record leaf elements extracted in depth-first order is associated with a list of SQL expressions. If associations for given records are correct, association for the record build from those records are also correct. In other words, depth-first ordering is preserved. In addition to T-P-map and T-P-ap, T-map and T-ap requires the same preservation of depth-first ordering.

Figure 5 explains the algorithm to implement these primitive operators. All operations are implemented with list-appending and

```
-- Composing (T-compose)
(<.>) :: Pi a b -> Pi b c -> Pi a c
-- Indexing (T-index)
(!)   :: Projection c a -> Pi a b -> Projection c b
-- Mapping (T-P-map and T-map)
(|$|) :: (ProductConstructor (a -> b), ProjectableFunctor p) => (a -> b) -> p a -> p b
-- Applying (T-P-ap and T-ap)
(|*|) :: ProjectableApplicative p => p (a -> b) -> p a -> p b
-- Pairing
(><)  :: ProjectableApplicative p => p a -> p b -> p (a, b)
x >< y = (,) |$| x |*| y
```

Figure 3. The signature of the primitive operators and pairing. `ProductConstructor` restricts types to Haskell record data constructors only. `Projection c` and `Pi a` are instances of both `ProjectableFunctor` and `ProjectableApplicative`.

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

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

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

(T is a Haskell record data constructor)

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

(T is a Haskell record data constructor)

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

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

Figure 4. Typing rules for phantom parameters

$$\pi_{ns} \odot \pi_{ms} \hookrightarrow \pi_{[ns!!m \mid m \leftarrow ms]} \quad \text{(E-compose)}$$

$$p_{xs}\ \mathbf{idx}\ \pi_{ns} \hookrightarrow p_{[xs!!n \mid n \leftarrow ns]} \quad \text{(E-index)}$$

$$T\ \langle\$\rangle\ p_{xs} \hookrightarrow p_{xs} \quad \text{(E-P-map)}$$

$$T\ \langle\$\rangle\ \pi_{ns} \hookrightarrow \pi_{ns} \quad \text{(E-map)}$$

$$p_{xs}\ \langle * \rangle\ p_{ys} \hookrightarrow p_{xs+\!\!+ys} \quad \text{(E-P-ap)}$$

$$\pi_{ns}\ \langle * \rangle\ \pi_{ms} \hookrightarrow \pi_{ns+\!\!+ms} \quad \text{(E-ap)}$$

Figure 5. The algorithms of the primitive operators

list-element-picking by indices. These implementations satisfy the preservation of depth-first ordering. Figure 7 shows execution examples of the primitive operators.

With a projection a and π functions p_1 and p_2, the following rule holds:

$$(a\ \mathbf{idx}\ p_1)\ \mathbf{idx}\ p_2 = a\ \mathbf{idx}\ (p_1 \odot p_2)$$

6. Related Work

This section describes other SQL EDSLs in Haskell.

6.1 HaskellDB

HaskellDB is recognized as the first EDSL for SQL in Haskell. In spite of its excellent features such as composability, we found the following drawbacks:

- no outer join support
- limited expression ability of result types
- incorrect behaviors of aggregate queries

To describe the last issue, consider the following query, which uses aggregation:[4]

```
countMembers2 = 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 <- countMembers2
   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 SQL based on an aggregate query is incorrect because the `GROUP BY` clause is lost. Note that inlining the first query into the second one has the same issue.

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

6.2 Opaleye

Opaleye is another EDSL for SQL in Haskell. It supports result types based on Haskell records and outer joins, and it provides composability. To provide pragmatic 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. 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 expressiveness of SQL statements.

7. Conclusion

We implemented Haskell Relational Record, a pragmatic, embedded domain-specific language for writing SQL statements in Haskell. To support real-world database system operations, it supports a large part of SQL, including outer joins and correlated subqueries, which are not supported by previous works. HRR handles non-aggregate and aggregate queries correctly, by using separate query types. Queries are composable since non-aggregate and aggregate queries are converted into a unified query type. SQL expressions in queries are typed with nested Haskell records. We show algorithms for primitive operators of record manipulation which preserve correspondence between nested Haskell records and flat SQL structures. HRR code is converted into syntactically-correct SQL statements. The arrow interface provides type-safety—generated SQL statements do not contain type or reference errors. If correlated subqueries are joined in the monad interface, generated SQL statements are not valid. In practice, this is not a serious problem because incorrectly joined subqueries can be always detected as reference errors.

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, Tom Ellis, Fumiaki Kinoshita, Hiroyuki Kurokawa, Oleg Kiselyov, and Yukiyoshi Kameyama for their feedback.

References

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

J. Hughes. Generalising Monads to Arrows. *Science of Computer Programming*, 37:67–111, 1998.

Information technology - Database languages - SQL. ISO/IEC 9075 standard, 2011.

M. P. Jones. *Functional Programming with Overloading and Higher-Order Polymorphism*, volume 925 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

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

S. Liang, P. Hudak, and M. Jones. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1995.

S. Marlow et al. *Haskell 2010 Language Report*, 2010.

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

E. Meijer, B. Beckman, and G. M. Bierman. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, 2006.

A. Ohori and K. Ueno. Making Standard ML a Practical Database Programming Language. In *Proceedings of ACM ICFP Conference*, 2011.

T. Sheard and S. Peyton Jones. Template Metaprogramming for Haskell. In *Proceedings of Haskell Workshop*, 2002.

K. Suzuki, O. Kiselyov, and Y. Kameyama. Finally, Safely-Extensible and Efficient Language-Integrated Query. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, 2016.

A. HRR features

In addition to features descibed in this paper, HRR provides the followings:

- Type-propagated placeholders
- Window functions
- Direct SQL embedding
- Insertion, deletion, update with/without correlated subqueries
- Customization of type mapping

B. Code Generation

The following is an overview of the auto-generated code for `Employee.hs`, expanded by Template Haskell:

```
data Employee = Employee {
    id     :: !Int
  , name   :: !String
  , deptId :: !Int
  } deriving (Show)

employee :: Relation () Employee
employee = ...

id' :: Pi Employee Int
id' = ...

name' :: Pi Employee String
name' = ...

deptId' :: Pi Employee Int
deptId' = ...
```

C. HRR operators

Figure 6 summarizes the signatures of HRR operators used in this paper. See also Figure 3.

D. Execution Examples

Figure 7 shows execution examples of the primitive operators.

```
-- Equal
(.=.) :: Projection c a -> Projection c a -> Projection c (Maybe Bool)
-- Greater than
(.<.) :: Projection c a -> Projection c a -> Projection c (Maybe Bool)

-- Injecting to Maybe
just  :: p a -> p (Maybe a)
-- Indexing for Maybe
(?!)  :: Projection c (Maybe a) -> Pi a b -> Projection c (Maybe b)

-- Converting a non-aggregate query to a relation
relation   :: QuerySimple (Projection Flat r) -> Relation () r
-- Converting a aggregate query to a relation
aggregateRelation :: QueryAggregate (Projection Aggregated r) -> Relation () r

-- Joining
query      :: MonadQuery m => Relation () r -> m (Projection Flat r)
-- Joining with Maybe values
queryMaybe :: MonadQuery m => Relation () r -> m (Projection Flat (Maybe r))

-- Converting a relation to a subquery
queryList  :: Relation () r -> QuerySimple (ListProjection (Projection Exists) r)

-- SQL constant
value :: a -> Projection c a
-- SQL ON
on :: MonadQuery m => Projection Flat (Maybe Bool) -> m ()
-- SQL WHERE (specialized for simplicity)
wheres     :: Projection Flat (Maybe Bool) -> QuerySimple ()
wheres     :: Projection Flat (Maybe Bool) -> QueryAggregate ()
-- SQL GROUP BY
groupBy    :: Projection Flat r -> QueryAggregate (Projection Aggregated r)
-- SQL COUNT aggregate function
count      :: Integral b => Projection Flat a -> Projection Aggregated b
-- SQL UNION ALL
unionAll   :: Relation () a -> Relation () a -> Relation () a
-- SQL EXISTS
exists     :: ListProjection (Projection Exists) r -> p (Maybe Bool)
```

Figure 6. The signatures of HRR functions. `QuerySimple` and `QueryAggregate` are instances of `MonadQuery`.

$$\pi_{[1,2]} \odot \pi_{[1]} \xhookrightarrow{\text{E-compose}} \pi_{[[1,2]!!1]} \hookrightarrow \pi_{[2]}$$

$$p_{[1,\text{“}Smith\text{”},T2.f0]}\ \mathbf{idx}\ \pi_{[1,2]} \xhookrightarrow{\text{E-index}} p_{[[1,\text{“}Smith\text{”},T2.f0]!!y|y\leftarrow[1,2]]} \hookrightarrow p_{[\text{“}Smith\text{”},T2.f0]}$$

$$(,)\ \langle\$\rangle\ p_{[T2.f0]} \xhookrightarrow{\text{E-P-map}} p_{[T2.f0]}$$

$$(,)\ \langle\$\rangle\ \pi_{[2]} \xhookrightarrow{\text{E-map}} \pi_{[2]}$$

$$p_{[T2.f0]}\ \langle*\rangle\ p_{[\text{“}Smith\text{”}]} \xhookrightarrow{\text{E-P-ap}} p_{[T2.f0]+\!\!+[\text{“}Smith\text{”}]} \hookrightarrow p_{[T2.f0,\text{“}Smith\text{”}]}$$

$$\pi_{[2]}\ \langle*\rangle\ \pi_{[1]} \xhookrightarrow{\text{E-ap}} \pi_{[2]+\!\!+[1]} \hookrightarrow \pi_{[2,1]}$$

Figure 7. Execution examples of the primitive operators