Scoping Monadic Relational Database Queries

Anton Ekblad
Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden
antonek@chalmers.se

Abstract

We present a novel method for ensuring that relational database queries in monadic embedded languages are well-scoped, even in the presence of arbitrarily nested joins and aggregates. Demonstrating our method, we present a simplified version of *Selda*, a monadic relational database query language embedded in Haskell, with full support for nested inner queries. To our knowledge, Selda is the first relational database query language to support fully general inner queries using a monadic interface.

In the Haskell community, monads are the de facto standard interface to a wide range of libraries and EDSLs. They are well understood by researchers and practitioners alike, and they enjoy first class support by the standard libraries. Due to the difficulty of ensuring that inner queries are well-scoped, database interfaces in Haskell have previously either been forced to forego the benefits of monadic interfaces, or have had to do without the generality afforded by inner queries.

CCS Concepts • **Software and its engineering → Functional languages**; **Data types and structures**; **Domain specific languages**; **Software libraries and repositories**.

Keywords domain-specific languages, haskell, relational databases, scoping

ACM Reference Format:
Anton Ekblad. 2019. Scoping Monadic Relational Database Queries. In *Proceedings of the 12th ACM SIGPLAN International Haskell Symposium (Haskell '19), August 22–23, 2019, Berlin, Germany.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3331545.3342598

Haskell '19, August 22–23, 2019, Berlin, Germany

ACM ISBN 978-1-4503-6813-1/19/08…$15.00
https://doi.org/10.1145/3331545.3342598

1 Introduction

One of the most common tasks in software development is querying databases for information, and SQL-based relational databases in particular. Most languages provide some sort of support for this task, either as standard library functionality or as built-in language primitives. Regardless of its exact implementation, this support often takes the form of an *embedded language*: a high-level interface in which programmers can describe queries directly in the host language, instead of writing raw SQL queries as strings to be passed to the database engine. The embedded language approach has many advantages for database integration: queries are correct by construction, can be type-checked together with the host program, and often allows one to write queries at a higher level of abstraction.

In Haskell, *monads* [18] are the de facto standard interface for embedded languages. They are widely studied by researchers, and thanks to their ubiquity every semi-proficient Haskell programmer is familiar with them. The standard Haskell libraries provide a wide range of functions over monads, and the language itself includes syntactic sugar to make monadic code more readable and concise.

1.1 Monads for Embedded Languages

For the aforementioned reasons, having a monadic interface is often desirable for embedded languages when possible. Using a bespoke abstraction often incurs an unnecessary cost in time and effort for new users to adopt the library or embedded language. Any given Haskell programmer is very likely to already be familiar with the monad abstraction, making the adoption of a monadic embedded language easier.

The same programmer is unfortunately significantly less likely to be familiar with other general abstractions, such as arrows or comonads. When a general abstraction is unfamiliar, its use may instead make the embedded language *harder* to adopt than using a bespoke one, since the programmer now needs to learn new abstractions not directly related to the domain of the problem they want to solve.

There is precedent for using a monadic interface to query collections in Haskell. The built-in list type makes up a monad, where the bind operation is equivalent to the Cartesian product of two lists. As discussed in Sect. 5.1, Haskell also has several monadic embedded languages for querying databases. However, all of these are lacking in generality compared to equivalent non-monadic languages.

```
addresses = do
  (name :*: addr) ← select persons
  city ← leftJoin (λcity → addr .== city) $ do
    (city :*: country) ← select cities
    restrict (country .== "Sweden")
    return city
  return (name :*: city)
```

Figure 1. Obtaining the Swedish address of every person in some database

```
SELECT personName, cityName
  FROM persons
  LEFT JOIN (
    SELECT cityName
    FROM cities
    WHERE cities.country = "Sweden"
  )
  ON persons.address = cityName
```

Figure 2. Obtaining everyone's Swedish address, in SQL

```
illScopedAddresses = do
  (name :*: addr) ← select persons
  city ← leftJoin (const true) $ do
    (city :*: country) ← select cities
    restrict (country .== "Sweden")
    restrict (addr .== city)
    return city
  return (name :*: city)
```

Figure 3. An ill-scoped join

1.2 Scoping Monadic Queries

The difficulty of scoping certain queries is a major hurdle in bringing the expressiveness of monadic embedded database languages up to par. Consider the query in Fig. 1, which selects each person from a table *persons* and associates each person with their home city, but only if that city is located in Sweden. The *inner query* making up the second argument of the *leftJoin* function is joined to the *current result set*, or the Cartesian product of the queries performed so far. It is equivalent to the SQL query in Fig. 2.

While the program from Fig. 1 is not problematic in itself, the fact that the *name* and *addr* identifiers, drawn from the *persons* table, are in scope *inside the body of the join* could enable us to write queries that are not well-scoped. Fig. 3 shows how this query might be written in an ill-scoped manner. The result of the right side of an SQL join operation is not in scope on the left side and vice versa. Thus, the inclusion of an identifier from the scope of the current result set, which makes up the left side of the join, into the body of the right side is a violation of SQL scoping rules.

The same problem arises for aggregate queries, as well as any other type of join operation. To support these operations in a monadic embedding of relational database queries, we need a way to disallow ill-scoped queries like that in Fig. 3, while still allowing queries like the one in Fig. 1.

Contributions In this paper, we extend the expressiveness of monadic database interfaces to cover an arbitrary nesting of joins and aggregate queries. Our contribution consists of (1) a method for ensuring that relational database queries are well-scoped in the presence of inner queries, and (2) *Selda*, a monadic language for database queries embedded in Haskell, demonstrating the use of the aforementioned method.

This paper is based on the work presented in chapter six of the author's PhD thesis [5].

2 A Basic Query Language

Before explaining our method of scoping monadic query languages from Sect. 3 on, we must first briefly describe the query language we are going to scope. For the remainder of this paper, and for this section in particular, it may be helpful to refer to the API listing of our simplified language as given in Fig. 4.

In this paper, we present the *Selda* relational database EDSL, named by an embarrassing pun [13] on *LINQ* [12], from which it draws much of its inspiration. It uses the semantics of the list monad, with the bind operation denoting the Cartesian product of its elements, as its starting point. From there, we extend this basic programming model with projection, restriction, aggregation, join operations, custom type errors, and a framework for safely supporting features specific to some particular database engine without affecting the language's operation on other engines.

The version of Selda presented in this paper is significantly simplified, to focus on the issue of scoping inner queries. The full Selda language further extends the work presented here with datatype-generic tables, type-safe migrations, prepared statements, and a host of other features. Selda is freely available from our website[1].

2.1 The Selda Programming Model

Selda is a backend-agnostic language, meaning that programs written in it can be executed on various different database engines. If a program compiles, it is guaranteed to be executable on any supported backend.

Selda Queries are written in the *Query* monad, which is parameterised over an extra type *s*. This type parameter is discussed at length in Sect. 3 but for now, we will ignore it. Queries are performed over *column expressions* of type *Col*, which also shares this type parameter. As in the list monad, the bind operation denotes the Cartesian product of its two

[1] https://selda.link

arguments. We call this Cartesian product the *current result set*.

Queries are compiled into SQL and executed by a relational database engine using the *query* function. The *Row* class and the type family *FromRow* are responsible for converting the results of a query back to a Haskell-level list of values.

The *Outer* and *UnAggr* type families denote the hoisting of their respective argument types from an *inner* scope to an *outer*. The mechanism by which this is done is explained further in Sect. 3 and 3.2 respectively. For now, it is sufficient to note that these type families are the mechanism by which inner query results — and *only* inner query results — are safely propagated from an inner query to an outer one.

The *Columns* type class denotes any type that is a heterogeneous list of columns, and the *Aggregates* class does the same for *aggregated* columns, which are discussed in Sect. 3.2. These two type classes are the value-level machinery accompanying the *Outer* and *UnAggr* type families, "moving" the actual columns from an inner scope to an outer.

Note that all three type families mentioned here are closed. If one were to open them up for extension, it would be possible to create an instance of *Columns* or *Aggregates* that subverts the scoping restrictions described in Sect. 3, making the language as a whole ill-scoped.

2.2 Result Rows as Heterogeneous Lists

Result rows in Selda are represented as *heterogeneous lists* of columns, much like the *HLIST* data structure by Kiselyov et al. [8]. This allows us to define types and functions inductively over result rows, avoiding the metaprogramming used by many embedded languages to define operations over database results. As we assume all query results to contain at least one column, we define our heterogeneous lists to be non-empty. A heterogeneous list is defined as one or more values interspersed with the *:*:* operator. Note that this means that a plain value of some type is *also* a "list" of a single element. Examples of such lists would be *(1 :*: 2)*, *"foo"*, and *(42 :*: "I'm a little teapot" :*: ())*. The *:*:* operator is defined as follows.

```
data a :*: b where
  (:*:) :: a → b → a :*: b
infixr 1 :*:
```

In Selda, heterogeneous lists are used consistently as a replacement for tuples. Queries receive and return heterogeneous lists, and the *FromRow* type family used by the *query* runner function maps heterogeneous lists over columns to heterogeneous lists over their corresponding Haskell-level types. More formally, *FromRow (Col s a1 :*: ... :*: Col s an)* ≡ *(a1 :*: ... :*: an)*.

2.3 Tables, Expressions and Basic Queries

At the base of every query lie one or more database tables. Selda provides a simple table definition language, where the

```
-- Types and classes
data Table a
data Query s a
data Col s a
data Aggr s a

data Inner s
type family Cols a
type family Outer a
type family UnAggr a
type family FromRow a

class Columns a
class Aggregates a
class Row a
instance Monad (Query s)

-- Executing queries and defining tables
query :: Row a
      ⇒ Database s
      → Query s a
      → IO [FromRow a]

table :: Text
      → ColSpec a
      → Table a

-- Query operations
restrict :: Col s Bool → Query s ()
select :: Columns (Cols s a)
       ⇒ Table a
       → Query s (Cols s a)
inner :: Columns a
      ⇒ Query (Inner s) a
      → Query s (Outer a)

aggregate :: Aggregates a
          ⇒ Query (Inner s) a
          → Query s (UnAggr a)

leftJoin :: Columns a
         ⇒ (Outer a → Col s Bool)
         → Query (Inner s) a
         → Query s (Outer a)
...

-- Expressions
instance Num a ⇒ Num (Col s a)
instance IsString (Col s Text)
(.>), (.<), ... :: Ord a
                ⇒ Col s a
                → Col s a
                → Col s Bool

min_, max_, ... :: Ord a ⇒ Col s a → Aggr s a
count :: Col s a → Aggr s Int
...
```

Figure 4. API of our simplified language

```
persons :: Table
        (   Col s Text
        :*: Col s Int
        :*: Col s Text)
persons = table "persons"
  (   column "name"
  :*: column "age"
  :*: column "city")

getMinorStatus :: Query s (Col s Text :*: Col s Bool)
getMinorStatus = do
  (name :*: age :*: _) ← select persons
  return (name :*: age .< 18)
```

Figure 5. Table definition, projection and expression in Selda

```
getMinors :: Query s (Col s Text)
getMinors = do
  (name :*: age :*: _) ← select persons
  restrict (age .< 18)
  return name
```

Figure 6. Table definition, projection and expression in Selda

Haskell types and SQL names of the table and its columns are given to create a *Table* value. The *Table* type is parameterised over a heterogeneous list of column types.

Tables are queried using the *select* function, which returns a heterogeneous list of expressions corresponding to the table's column types, which may then be pattern matched by the querying computation. Queries may perform *projection* by simply returning the values they want to project.

Columns have the type *Col s a*, and may represent either a column from an actual table, or an *expression* over zero or more such columns. Column expressions may be constructed using the expected operations: addition, subtraction, comparisons, boolean operations, etc. It is important to note that *Col* is an abstract type; a requirement for being able to reify the structure of a monadic computation over columns [17].

Fig. 5 demonstrates how to define a table *persons* and query it to associate each person with a boolean column expression denoting whether the person is a minor.

Queries may be filtered using the *restrict* operation. If a query computation contains a call to *restrict p*, only result rows for which *p* evaluates to true will be kept in the current result set. This is akin to the filtering operation of Haskell's list comprehensions, or the standard library function *filter*. Fig. 6 modifies the query from Fig. 5 to *remove* all non-minors from the result set, instead of merely tagging each person with their "minorness".

```
type family Cols a where
  Cols s (a :*: b) = Col s a :*: Cols s b
  Cols s a         = Col s a

select :: Columns (Cols s a)
       ⇒ Table a
       → Query s (Cols s a)
```

Figure 7. Marking columns with the appropriate scope

3 Inner Queries

Having dispensed with the preliminaries, we now turn to the main contribution of this paper. As we discussed in Sect. 1.2, we want to support general inner queries in aggregates and join operations, while guaranteeing that column expressions from an outer query do not leak into an inner one. To accomplish this, we leverage Haskell's expressive type system. Namely, *phantom types* [3] — purely nominal type parameters — and *closed type families* [4] — the Haskell flavor of functions at the type level rather than at the value level.

3.1 If All Else Fails, Use Type Variables

Recall that the *Query* and *Col* types discussed in Sect. 2.1 are parameterised over an extra type variable *s*.

The reason for this extra parameter, as the attentive reader might have started to suspect by now, is to keep track of the scope of a column expression, ensuring that each database operation only ever interacts with column expressions in "their own" scope. Intuitively, this scope type parameter serves the same purpose as the state thread parameter of Haskell's *ST* monad. This similarity is further discussed in Sect. 5.

Restricting column types In Selda, all operations over columns require that the scope type of the *column* matches the scope type of the *operation*. See for instance the type signature of the *restrict* operation:

```
restrict :: Col s Bool → Query s ()
```

We require any operations which introduce new columns to propagate their own scope variable to those columns. See for instance how this is accomplished for an arbitrary number of columns by the type of the *select* function — the primary source of fresh column expressions in Selda — as shown in Fig. 7.

To differentiate between inner and outer scopes, we assign scopes variables to operations based on their *nesting level* relative to some outermost top-level scope. To keep track of this scope nesting level, we introduce a type *Inner* to our language. A child scope of some scope *s* — that is, a scope nested immediately inside of *s* — is denoted *Inner s*.

This ensures that inner queries can not touch columns from outer queries and vice versa, as long as we manage to ensure that inner queries never "leak" columns from their

```
type family Outer a where
  Outer (Col (Inner s) a :*: b) =
    Col s a :*: Outer b
  Outer (Col (Inner s) a) =
    Col s a
```

Figure 8. Scope hoisting for inner queries

own scope into queries with an outer scope. To this end, we introduce a type family *Outer*, which hoists the type of a heterogeneous list of columns in scope *Inner s*, to the equivalent heterogeneous list type of columns in scope *s*, as shown in Fig. 8.

We then require any inner query *q'* of some parent query *q :: Query s (Outer a)* to have the type *Query (Inner s) a*. We enforce this by ensuring that any function which takes an inner query as an argument forces the *s* parameter of the query to denote a scope *one level deeper* than the current scope. See for instance the types of *inner* and *leftJoin*:

```
inner :: Columns a
      ⇒ Query (Inner s) a
      → Query s (Outer a)

leftJoin :: Columns a
         ⇒ (Outer a → Col s Bool)
         → Query (Inner s) a
         → Query s (Outer a)
```

As all column expressions are parameterised over the scope in which they originated, and as all operations over columns require the scope of the columns to match the present scope, this prevents columns from an outer scope from being used in an inner query. Columns from an inner scope are prevented from escaping to an outer one by the the application of *Outer* type family on the return values of inner queries. As *Outer* is only defined for heterogeneous lists over columns, attempting to return any other type is by definition a type error, as further discussed in Sect. 3.3. This restriction ensures that rogue columns never escape an inner scope. Sect. 5 discusses this property in more detail.

Projecting columns from inner queries But how does this scope hoisting actually happen, and how are inner queries stitched together with its parent query? In Selda, we have opted for the following implementation:

- First, all columns returned by the inner query are gathered into a conventional list. This gathering of columns is the primary responsibility of the *Columns* type class.
- A fresh name is generated for each column returned from the inner query, in the form of a column in the *outer* scope.
- An intermediate, symbolic SQL representation of the inner query is generated.

```
inner :: Columns a
      ⇒ Query (Inner s) a
      → Query s (Outer a)
inner q = do
  (inner_sql, out_cols) ← generateSql q
  out_names ← mapM freshNameFor (toList out_cols)
  let renamed_sql = inner_sql
        { outCols = zip out_names out_cols }
  addSourceQuery renamed_sql
  return (fromList out_names)
```

Figure 9. Pseudocode projecting inner query results

- In the intermediate representation, each returned column is bound to its corresponding generated name (i.e. *SELECT* col_1 *AS* $name_1$, col_2 *AS* $name_2$ ...).
- The list of freshly generated names is converted into a heterogeneous list, to match the return type of *inner*.
- The heterogeneous list of names is returned from *inner*, so that the outer query can refer to the columns projected by the inner query by their new names.

A pseudocode implementation of a shallow embedding of this process is given in Fig. 9.

While our method of scoping is not bound to any particular implementation of the *Query* monad, the steps of gathering the returned columns, binding them to fresh names, and returning the names to the outer scope are crucial.

One might consider augmenting this implementation with certain optimizations — reusing names from inner scopes, or projecting partial expressions over columns in hopes of being able to inline or entirely eliminate them at a later stage, for instance — but this is a relatively complex subject, beyond the scope of this paper.

3.2 Aggregation

Aggregate queries give rise to the same problems with scoping as other inner queries: as aggregation "collapses" its input query, it can not share its input with any non-aggregate query, or even any other aggregation. Fortunately, using the phantom type technique described above solves this problem as well, as long as we give our aggregation function an appropriate type:

```
aggregate :: Aggregates a
          ⇒ Query (Inner s) a
          → Query s (UnAggr a)
```

Note that the type class constraint and return type of this function differ from the ones on, for instance, the *inner* function. In addition to the more general scoping problem, aggregated queries come with a related problem of their own. Consider the following query.

```
allAdults :: Query s (Col s Text)
allAdults = aggregate $ do
  (name :*: age :*: city) ← select persons
  restrict (age .> min_ age)
  return name
```

The equivalent SQL query would be:

```
SELECT name
  FROM persons
  WHERE age > MIN(age)
```

While this query may intuitively make sense — return the names of every person who is older than the youngest person — this is another violation of SQL's scoping rules: `WHERE` clauses must not refer to aggregate expressions. Taking a closer look at the query, we see that there is a good reason for this. Aggregate expressions compute their value over the aggregate of the whole query *after restrictions are applied*. Thus, allowing restrictions to refer to aggregates over the current query would lead to an infinite loop.

At first glance, we might be tempted to solve this problem by scoping all aggregate expressions one level deeper than the current scope:

```
min_ :: Ord a ⇒ Col s a → Col (Inner s) a
```

However, this solution breaks down in the presence of nested inner queries. If a query of scope *s* is able to create column expressions of scope *Inner s*, those column expressions could then be accessed from within any inner query in the same scope. This would effectively make the scoping control we've introduced so far useless!

Instead, we opt to use a different column type entirely for aggregated columns, which we call *Aggr*, and assign appropriate types to our aggregation functions:

```
min_, max_, ... :: Ord a ⇒ Col s a → Aggr s a
count :: Col s a → Aggr s Int
...
```

Similarly, we also need to introduce a new type family — *UnAggr* — to hoist aggregated columns out of their inner scope and turn them into plain columns in the outer query, much like *Outer* does for non-aggregated columns being returned from an inner query:

```
type family UnAggr a where
  UnAggr (Aggr (Inner s) a :*: b) =
    Col s a :*: UnAggr b
  UnAggr (Aggr (Inner s) a) =
    Col s a
```

It should be noted that only aggregate column expressions may be returned from an aggregated inner query, as enforced by the *UnAggr* type family. While some SQL dialects allow non-aggregated expressions to be projected from an aggregate query, the semantics of this are unclear at best. For this reason we can't just use the same type family for *Outer* and *UnAggr*, as this would allow aggregate queries like the following:

```
badInnerReturnType = do
  (name :*: addr) ← select persons
  city ← leftJoin (const true) $ do
    (city :*: country) ← select cities
    restrict (country .== "Sweden")
    return (Just city)
  return (name :*: city)
```

Figure 10. Returning an illegal value from an inner query

```
someAdultWithMinAge :: Query s (Col s Text)
someAdultWithMinAge = aggregate $ do
  (name :*: age :*: city) ← select persons
  return (name :*: min_ age)
```

Returning the minimum age of all persons in the set is all well and good, but pairing it with a non-aggregated name from the set raises precisely the aforementioned concerns about non-aggregate projections. Whose name is it? How does it relate to *min_ age*? Is the relation deterministic? Rather than attempt to provide answers for these questions, we are content to simply disallow this construct, even at the price of an extra type family.

3.3 Unhelpful Type Errors

Producing clear and relevant type errors is a well-known problem of embedded domain-specific languages, and Selda is no exception. The Selda design presented so far is prone to producing error messages which can be hard to understand mainly for two classes of error.

Illegal return values The first class of error occurs when attempting to return a value which is not a heterogeneous list of columns from an aggregate or inner query.

To illustrate, Fig. 10 modifies the example from Fig. 1 to return *Just city* instead of *city*. This is illegal, as there is no *Outer* instance for *Maybe a* — Selda does not know how to decrement the scope counter of a *Maybe*-wrapper value. The program produces the following type error:

```
• Couldn't match type
    'Outer (Maybe (Col (Inner s) Text))'
    with 'Col s Text'
  Expected type:
    Query s (Col s Text)
  Actual type:
    Query s (Outer (Maybe (Col (Inner s) Text)))
• In a stmt of a 'do' block:
  ...
```

The programmer may be able to infer from this error message that returning *Just city* from the inner query was somehow a bad idea. However, *why* this is the case is a complete mystery, frustrating the programmer's efforts to fix the problem.

Scope mismatch errors The second class is the very thing Selda set out to prevent: attempting to refer to a value from

an outer scope in an inner query. This type of error arises either when two Selda expressions from different scopes are combined using some expression-level function, or when some expression is used with a monadic Selda operation from a different scope.

Attempting to compile the ill-scoped query from Fig. 3 does indeed produce a type error as promised, but that type error is less than ideal:

```
• Occurs check: cannot construct
    the infinite type: s ~ Inner s
  Expected type: Col (Inner s) Text
    Actual type: Col s Text
• In the second argument of ‘(.==)’,
    namely ‘addr’
  In the first argument of ‘restrict’,
    namely ‘(city .== addr)’
  In a stmt of a 'do' block:
    restrict (city .== addr)
```

While the type checker's stated expectation to see an expression of type *Col (Inner s) Text* rather than of type *Col s Text* is reasonably clear, again, there is no indication as to *why* this is expected.

3.4 Helpful Type Errors

Fortunately, recent versions of the GHC compiler allows us to specify custom type errors [14], when we are not quite happy with what the compiler provides. Custom type errors are expressed using a simple pretty-printing type-level EDSL, where each error is denoted by a specific, distinct type. When the type checker encounters such a type — for instance, as a superclass constraint on a type class or computed by a type family — it outputs the encoded error instead of treating it as just any other type.

Illegal return values Using this approach to produce more helpful type messages for the first class of error — an illegal return type from an inner query — is relatively straightforward. As we already use the *Outer* type family to restrict which types can be returned, improving the error message is simply a matter of augmenting the type family with one or more cases evaluating to more domain-appropriate type errors.

Fig. 11 augments the *Outer* type family with two new cases, each expressing a custom, scope-related error message. The case matching *Col s a* gives a more helpful explanation of where the programmer went wrong when attempting to return a value which is *not from its own scope*, while the final, catch-all case tells the programmer in no uncertain terms which values are legal to return, should they attempt to return anything but a heterogeneous list of columns.

With this addition, the type error becomes significantly clearer about why the program does not type check:

```
• Only (heterogeneous lists of) row and columns
    can be returned from an inner query.
• In a stmt of a 'do' block:
  ...
```

```
type family Outer a where
  Outer (Col (Inner s) a :*: b) =
    Col s a :*: Outer b
  Outer (Col (Inner s) a) =
    Col s a
  Outer (Col s a) =
    TypeError
      ( 'Text "An inner query can only return"
        ':<>:
        'Text " columns from its own scope."
      )
  Outer a =
    TypeError
      ( 'Text "Only (heterogeneous lists of) columns"
        ':<>:
        'Text " can be returned from an inner query."
      )
```

Figure 11. Custom errors for illegal inner query return types

```
class s ~ t ⇒ Same s t

instance {-# OVERLAPPING #-} Same s s

instance {-# OVERLAPPABLE #-}
  ( s ~ t
  , TypeError
    ( 'Text "An identifier from an outer scope"
      ':<>:
      'Text "may not be used in an inner query."
    )
  ) ⇒ Same s t
```

Figure 12. Type error for mismatched scope parameters

Scope mismatch errors Adding domain-specific error messages to the second class of errors is slightly more involved, as there is no readily apparent place to simply insert a custom type error.

Recall that most expression-level Selda operations have a type similar to $Col\ s\ a \to Col\ s\ a \to Col\ s\ b$, and *restrict* — as a representative of the monadic Selda operations — has the type $Col\ s\ Bool \to Query\ s\ ()$. In all of these types, the scope parameter is explicitly *the same*, so the type checker does the equality checking for us, raising its own, unhelpful, error message whenever it encounters mismatching scope types.

In order to preempt this equality checking, to insert our own error message, we will need to change the type of all such operations to $Same\ s\ t \Rightarrow Col\ s\ a \to Col\ t\ a \to Col\ s\ b$ and $Same\ s\ t \Rightarrow Col\ s\ Bool \to Query\ t\ ()$ respectively, where the *Same* type class is defined as shown in Fig. 12.

In essence, we define our own type equality constraint, but with a custom error message when the constraint is violated.

Whenever the two scope parameters under comparison are the same, the constraint is trivially satisfied. For any other pair of types, however, the *TypeError* instance constraint must be first be satisfied — a condition which can never be fulfilled, but causes the compiler to throw the encoded type error instead.

The decision to add a $s \sim t$ superclass constraint to the *Same* type class merits some discussion. It should be noted that modifying the type signatures of Selda operations as described above could lead to type inference of literals, constants and other Selda expressions without a fixed scope becoming effectively impossible, as every part of a Selda expression would have its own scope variable, independent of the rest of the expression. Adding equality constraint as a superclass to *Same* removes this independence, allowing the compiler to assume that the types involved are indeed equal. However, unlike simply using the same scope variable for all parts of a function's signature — as we did before — this trick forces the type checker to try to solve *other* constraints — such as the *TypeError* constraint in the mismatch case — before deciding that the equality constraint does not hold. An additional benefit of this superclass, is that it prevents the user from creating instances of *Same* for types which are not identical.

With these changes, the type errors caused by scope mismatches become clearer:

```
• An identifier from an outer scope
    may not be used in an inner query.
• In the first argument of ‘restrict’,
    namely ‘(city .== addr)’
  In a stmt of a 'do' block: restrict (city .== addr)
```

However, it should be noted that this increase in clarity of error messages comes at the expense of clarity of type signatures. While a type signature like $Col\ s\ a \to Col\ s\ a \to Col\ s\ b$ is crystal clear about the the parameters' scope variables having to be identical, parsing this information from the modified type signature $Same\ s\ t \Rightarrow Col\ s\ a \to Col\ t\ a \to Col\ s\ b$ requires a greater degree of understanding and mental legwork from the programmer.

4 Scope and Backend-specific Features

What about the outermost scope? The attentive reader may have noticed at this point, that we have yet to discuss the *outermost* scope of a Selda query. It is all well and good to know that if a query lives in scope *s*, any inner query must live in scope *Inner s*, but what should that outermost scope *s* actually be? Many answers to this question are conceivable. As discussed further on in Sect. 5, similar approaches have used an existentially quantified *s* to guarantee uniqueness of scope even on a global level. Another option would be to leave the outermost scope undefined; any scope identifier is fine.

```
class JSONBackend s where
  lookup :: Col s JSON → Text → Col s (Maybe JSON)

instance JSONBackend (Inner s) ⇒ JSONBackend s where
  lookup json key = unsafeHoist (lookup json key)

unsafeHoist :: Col (Inner s) a → Col s a
```

Figure 13. The underpinnings of backend-specific JSON functionality.

Backends with differing capabilites In the case of our method, we recall that Selda is a multi-backend language, as noted in Sect. 2.1, and that any program that compiles should be executable on all supported backends. This would seem to preclude any sort of backend functionality, restricting our language to the lowest common denominator features set of all supported backends. This would be a shame, however, as many database engines natively support unique, powerful features, to give them an edge over the competition: XML and JSON parsing, arbitrary precision integers, and even embedded general purpose programming languages, to mention just a few.

To allow a multi-backend language like Selda to safely support such features, we need to allow type checking of queries making use of backend-specific language extensions when executed on a backend which supports them, while disallowing their use on less capable backends.

To accomplish this, recall the type signature of the *query* function from Fig. 4:

```
query :: Row a
      ⇒ Database s
      → Query s a
      → IO [FromRow a]
```

In particular, note that the *Database* type of its first argument is parameterised over a type variable *s*. This type variable is then used as the base scope of the query to be executed. By allowing each supported backend to provide a phantom type symbolising this particular backend and its own database initialisation function, we effectively give each backend a way to identify itself to the type checker.

Each backend must then provide a function to acquire a database handle parameterised over its corresponding type, such as *sqliteOpen :: FilePath → IO (Database SQLite)*, giving the user a handle to a newly opened SQLite database, or *pgOpen :: ConnectionString → IO (Database PostgreSQL)*, should the user want to connect want to connect to a PostgreSQL database instead.

By piggybacking on the scope parameter, we can now write type classes representing various forms of backend-specific functionality for the more powerful backends to implement. An example of such functionality, implementing lookups over JSON columns, is given in Fig. 13.

Note that we are not done simply by declaring the *JSONBackend* type class, however. As the whole point of this work is to enable monadic database interfaces to take advantage of inner queries, it would be silly to only allow backend-specfic functionality in the outermost scope. For this reason, each such type class should implement a recursive instance like the one we give for *JSONBackend*, to "rise up" from an arbitrary level of scope nesting in search for the outermost scope. Backends, then, only need to give an implementation for their own phantom type — such as *instance JSONBackend PostgreSQL* — to provide their functionality.

Isn't that dangerous? In this example, the recursive instance is implemented using a function *unsafeHoist*, which allows columns to be hoisted one level up without any of the usual checks. While horrifically unsafe in the general case, this particular use case is completely fine as it is only used to find the base scope for purposes of backend capability checking. Similarly, backend-specific functionality normally needs to be implemented using unsafe constructs, as backend-specific modifications to the generated SQL are obviously unsafe in the general case.

Luckily, this adventure in unsafe extension implementation is only necessary once per backend and extension it supports. Once implemented, users may rest assured that any query with a *JSONBackend* constraint on its scope parameter — which any query making use of *lookup* will require — will only compile when used on a database from a backend implementing *JSONBackend* for its representative phantom type.

5 Discussion and Related Work

As discussed in Sect. 1.1, there are several advantages to using a monadic interface when embedding a language into Haskell: good library support, syntactic sugar, and programmer familiarity. However, the disadvantage is relatively obvious: not all languages are equally suited to a monadic interface, and even the languages that are, may require a less obvious implementation than when using another abstraction. As the current state of the art shows, this can be said to be the case for database query languages. However, while the method presented in this paper may be slightly trickier to implement than some other relational database abstraction, it does not complicate the language for the end user. Instead, it enables our language to reap all of the aforementioned advantages of monadic interfaces.

Relation to the ST monad Haskell-savvy readers may have picked up on the fact that our method is strikingly similar to the method introduced by Launchbury and Peyton Jones [9] to allow computations with a referentially transparent interface to safely use mutable references internally. This is accomplished by introducing a monad *ST s a*, in which references of type *STRef s a* may be created and mutated. Like with our method, functions in the monad are parameterised over a state thread *s* (e.g. *newSTRef :: ST s (STRef s a)*), to ensure that computations can only ever interact with references created in the same state thread. Computations in the monad are executed using a function *runST :: (forall s. ST s a) → a*, which ensures that each stateful computation is executed in its own state thread. Just like with inner queries, with state threads we have multiple separate computations in the same monad, which must not be allowed to freely interact with one another.

Unfortunately, this method does not work for inner queries. While the ST monad is intended to stop any *and all interaction* between state threads, our language must allow interaction between queries *in a controlled manner*. As we saw in Sect. 3, the body of an inner query returns a list of columns, where each column has the type *Col (Inner s) a*. This list is then converted into a list of columns of type *Col s a* by the function (i.e. *inner*, etc.) used to execute it. Using the method by Launchbury and Peyton Jones, the *inner* function would instead have had the type *(forall s. Query s a) → Query s' a*. Looking at this type, we can see that it will not allow us to return any type *a* which references *s*. The *s* type variable is bound by a *forall* quantifier local to the function's first argument. Thus, no type referencing *s* could be referred to in the return type (or even other arguments) of *inner*, as *s* is no longer in scope at this point.

Note that, just as this method is not applicable to our problem, our method is not applicable to state threads. Our method can not make use of existential quantification to ensure the uniqueness of each state thread, but instead relies on the top level query to fix the "base" *s*. This means that any application of this method which must be callable from referentially transparent code would be unsafe, as there would be no way to guarantee the global uniqueness of state threads.

Relation to type-level natural numbers Our method essentially encodes the nesting level of the current scope as a type-level natural number, with the *s* determined by the *query* runner function as the zero value, and each consecutive application of the *Inner* type constructor acting as an application of the successor function. This may sound worrisome at first: it is definitely the case that two separate nested queries may have the same nesting level. How, then, can we be sure that their scopes don't leak into each other? The answer lies in the return type of the inner query functions. By only allowing lists of columns to be returned, decrementing the nesting level of each column by one, we ensure that no column is able to escape without having its nesting level properly decremented. Note that this property relies on the *Outer* type family *only* having instances for types which we know how to safely hoist from an inner scope to an outer: *Col s a* and *Col s a :*: b*.

If a column of type *Col s (Col s' a)* could be constructed and inspected, for instance, further measures would be required

to ensure that the nesting level of the inner column would not escape its scope.

5.1 Related Work

The area of relational database EDSLs is an active one, in academia as well as in industry. Arguably, the most well-known such language is *LINQ* [12]. Embedded in the .NET framework, LINQ allows SQL-like queries to be made over not only databases, but any suitable collection. LINQ uses a more restricted, streaming interface which avoids the problem of scoping inner queries entirely, by only allowing inner queries in contexts where no identifiers from the outer scope are bound. In addition, LINQ makes extensive use of runtime errors to voice its displeasure with the aberrant queries that may arise from mixing *entirely separate* queries, rather than preventing such unions by way of types, making it rather less ambitious in the correctness department. *ScalaQL* [16] provides a similar interface for Scala, while the *Opaleye* EDSL [6] uses arrows and profunctors to provide the same for Haskell. Similarly, the *Esqueleto* Haskell EDSL [11] provides a continuation-based interface to database queries. Silva and Visser [15] describe an ingenious ad hoc embedding of database queries into Haskell, capable of supporting functional dependencies and other advanced features. Augustsson and Ågren [2] provide perhaps the most debuggable database EDSL to date, using user-defined type errors to great effect in their Haskell encoding of relational algebra.

All of the aforementioned approaches support fully general inner queries, but do so at the price of forgoing a monadic interface. There exists several monadic database EDSLs however. *HaskellDB*, perhaps the grandfather of database EDSLs, provides a monadic interface to database queries, but does not support inner queries due to the difficulty of typing them [10]. *Haskell Relational Record* [7] is a more recent attack on the problem, which provides a monadic interface which supports joins and aggregation, but only over database tables, and not over inner queries. *Beam* [1] used to take the same approach as Haskell Relational Record, but adopted our scoping model during the preparation of this paper, following the initial release of our Selda library. This gives us additional confidence in the applicability of our method to the monadic SQL scoping problem.

While Selda does not strictly enable any queries to be written that were not possible before, it does expand the set of queries that can be written using a monadic interface over what is possible with the current state of the art.

6 Conclusions and Future Work

We have presented a monadic interface to relational database queries. We improve upon the state of the art by leveraging the host language's type system to ensure that even fully generic inner queries are well-scoped. To our knowledge, ours is the first monadic relational database EDSL to accomplish this.

The Selda database EDSL resulting from this work serves as a solid and flexible base for further exploration of the design space of strongly typed database abstractions, including more specific applications such as migrations, optimisation, and higher order database schemas.

While our method fulfils its original design goal — enabling a monadic database EDSL in the spirit of the list monad — applying our method to other problems remains as future work. One possible such application would be in the area of cache-aware computations, where the hierarchical nature of our scoping method could provide static guarantees regarding prefetching and data access in a hierarchy of caches. Another possible application worth investigating is EDSLs utilising region-based memory management, where our method might provide a framework for safely and efficiently moving data into and out of memory regions as needed.

Acknowledgments

Many thanks to Mauro Jaskelioff, Tom Ellis, Koen Claessen, Michał Pałka, and Jonathan Skårstedt, for their valuable insights, feedback and discussion.

This work was partially funded by the Swedish Foundation for Strategic Research (SSF) under the project Octopi (Ref. RIT17-0023), as well as the Wallenberg Artificial Intelligence, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

[1] Travis Athougies. 2016. Beam. http://travis.athougies.net/projects/beam.html.

[2] Lennart Augustsson and Mårten Ågren. 2016. Experience Report: Types for a Relational Algebra Library. In *Proceedings of the 9th International Symposium on Haskell (Haskell 2016)*. ACM, New York, NY, USA, 127–132. https://doi.org/10.1145/2976002.2976016

[3] James Cheney and Ralf Hinze. 2003. *First-class phantom types.* Technical Report. Cornell University.

[4] Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. 2014. Closed Type Families with Overlapping Equations. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM.

[5] Anton Ekblad. 2018. *Functional EDSLs for Web Applications.* PhD Thesis. Chalmers Institute of Technology.

[6] Tom Ellis. 2014. Opaleye. https://github.com/tomjaguarpaw/haskell-opaleye.

[7] Key Hibino, Shohei Murayama, Shohei Yasutake, Sho Kuroda, and Kazu Yamamoto. 2015. Haskell Relational Record. http://khibino.github.io/haskell-relational-record/.

[8] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. 2004. Strongly Typed Heterogeneous Collections. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell (Haskell '04)*. ACM, New York, NY, USA, 96–107. https://doi.org/10.1145/1017472.1017488

[9] John Launchbury and Simon L. Peyton Jones. 1994. Lazy Functional State Threads. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*. ACM, New York, NY, USA, 24–35. https://doi.org/10.1145/178243.178246

[10] Daan Leijen and Erik Meijer. 1999. Domain Specific Embedded Compilers. In *Proceedings of the 2Nd Conference on Domain-specific Languages (DSL '99)*. ACM, New York, NY, USA, 109–122. https://doi.org/10.1145/331960.331977

[11] Felipe Lessa. 2012. Esqueleto. http://hackage.haskell.org/package/esqueleto.

[12] Erik Meijer, Brian Beckman, and Gavin Bierman. 2006. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD '06)*. ACM, New York, NY, USA, 706–706. https://doi.org/10.1145/1142473.1142552

[13] Shigeru Miyamoto. 1986. The Legend of Zelda. https://en.wikipedia.org/wiki/Link_(The_Legend_of_Zelda).

[14] Alejandro Serrano and Jurriaan Hage. 2017. Type Error Customization in GHC: Controlling Expression-level Type Errors by Type-level Programming. In *Proceedings of the 29th Symposium on the Implementation and Application of Functional Programming Languages (IFL 2017)*. ACM, New York, NY, USA, Article 2, 15 pages. https://doi.org/10.1145/3205368.3205370

[15] Alexandra Silva and Joost Visser. 2006. Strong Types for Relational Databases. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell (Haskell '06)*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/1159842.1159846

[16] Daniel Spiewak and Tian Zhao. 2009. ScalaQL: language-integrated database queries for scala. In *International Conference on Software Language Engineering*. Springer, 154–163.

[17] Josef David Svenningsson and Bo Joel Svensson. 2013. Simple and Compositional Reification of Monadic Embedded Languages. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, New York, NY, USA, 299–304. https://doi.org/10.1145/2500365.2500611

[18] Philip Wadler. 1995. Monads for functional programming. In *International School on Advanced Functional Programming*. Springer, 24–52.