Scoping Monadic Relational Database Queries

Anton Ekblad

Chalmers University of Technology, 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 in 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.

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* [16] 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 another abstraction often incurs an

unnecessary cost in time and effort for new users 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 significantly less likely to be familiar with other abstractions. When a general abstraction is unfamiliar, its use may instead make the embedded language *harder* to adopt, since the programmer also needs to learn new abstractions not directly related to the domain of the problem they are trying 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. 4.1, Haskell also has several monadic embedded languages for querying databases, all of which are lacking in generality compared to equivalent non-monadic languages, however.

1.2 Scoping monadic queries

The lack of expressiveness in other monadic languages comes from a difficulty of scoping certain queries. 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.

```
addresses = do
  (name :*: addr) <- select persons
  city <- leftJoin (\city -> addr .== city) $ do
    (city :*: country) <- select cities
    restrict (country .== "Sweden")
    return city
  return (name :*: city)
```

Fig. 1. Obtaining the Swedish address of each person in a database

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

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

Fig. 2. Obtaining the Swedish address of each person, in SQL

```
illScopedAddresses = do
  (name :*: addr) <- select persons
  city <- leftJoin (\city -> addr .== city) $ do
    (city :*: country) <- select cities
    restrict (country .== "Sweden")
    restrict (city .== addr)
    return city
  return (name :*: city)
```

Fig. 3. An ill-scoped join

database queries, we need a way to disallow ill-scoped queries like that in Fig. 3, while still allowing queries like the one in 1.

Contributions With 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.

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. The language presented in this paper, dubbed *Selda* as an embarrassing pun [12] on *LINQ* [11], 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, and many other features. The version of Selda presented in this paper is significantly simplified, to focus on the issue of scoping inner queries. The full version is freely available from the project website.[1]

[1] https://selda.link

2.1 The Selda programming model

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 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 its associated type *FromRow* is responsible for converting the results of a query back to a Haskell-level list of values. The *Columns* type class denotes any type that is a heterogeneous list of columns, and the *Aggregates* class implements the same restriction for *aggregated* columns, which are discussed in Sect. 3.2. The latter two classes are necessary to ensure that only values from the Selda language itself are propagated through queries.

Note that all three classes are closed, and have no user-accessible methods. 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.

For reference, the API of our simplified language is given in Fig. 4.

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 [7]. 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, *FromRows (Col s a1 :*: ... :*: Col s an) ≡ (a1 :*: ... :*: an)*.

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

data Inner s
type family Outer a

class Columns a
class Aggregates a
class Row a where
  type FromRow a
instance Monad (Query s)

-- Executing queries and defining tables
query :: Row a => Database -> Query s a -> IO [FromRow a]
table :: Columns a => Text -> a -> Table a

-- Query operations
select :: Table a -> Query s a
restrict :: Col s Bool -> Query s ()
inner :: Columns a => Query (Inner s) a -> Query s (Outer a)
aggregate :: Aggregates 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)
...

-- Expressions
instance Num a => Num (Col s a)
(.>), (.<), ... :: 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
...
```

Fig. 4. API of our simplified language

2.3 Tables, expressions and basic queries

At the base of every query lies one or more database tables. Selda provides a simple table definition language, where the 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 return a heterogeneous list of tuples 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 [15].

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.

```
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)
```

Fig. 5. Table definition, projection and expression in Selda

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 minor status.

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

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

Fig. 6. Table definition, projection and expression in Selda

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.

Much like Haskell's *ST* monad, all Selda operations over columns require that the scope type variable of the *column* matches the scope variable of the *operation*. This ensures that inner queries can not touch columns from outer queries and vice versa, as long as we manage to ensure that the inner and outer queries can never share the same scope type variable.

To this end, we introduce a type *Inner* to our language to distinguish different scopes, as well as a type family *Outer*, which hoists the type of a heterogeneous list of results in scope *Inner s*, to the equivalent heterogeneous list type of results in scope *s*, as shown in Fig. 7.

```
data Inner a

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

Fig. 7. Scope hoisting for inner queries

Then, any inner query *q'* of some query *q :: Query s (Outer a)* must 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 force the scope of the columns to 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 *Columns* constraint on the return values of inner queries – that they are always heterogeneous lists of columns. This restriction, together with the "scope hoisting" enforced by the *Outer* type family ensures that scopes are never mixed. Sect. 4 discusses this property in more detail.

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 (Outer a)
```

Note that the type class constraint of this function differs from the one 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
...
```

The attentive reader may have noticed that the definition of the *Outer* type family in Fig. 7 is unnecessarily general: why convert *t (Inner s) a* into *Col s a* instead of the more obvious choice of converting *Col (Inner s) a* into *Col s a*? The *Aggr* type provides the answer: *Outer* does not only hoist "plain" column expressions into the outer scope, but aggregated expressions as well.

It should be noted that only aggregate column expressions may be returned from an aggregated inner query, as enforced by the *Aggregates* type class constraint. While some SQL dialects allow non-aggregated expressions to be projected from an aggregate query, the semantics of this are unclear at best.

4 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 [8] 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)* $\rightarrow$ *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 needs 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)* $\rightarrow$ *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 the *forall* quantifier of the *Query s a* type. Referencing *s* in *a* would allow it to escape its scope, as *a* occurs outside the quantifier's scope.

Note that, just like this method is not applicable to our problem, our method is not applicable to state threads. Our method does 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 which must be callable from referentially transparent code would either be unsafe, as there would be no way to guarantee the global uniqueness of state threads, or be forced to use an existentially quantified "base" *s* anyway, which would make the rest of our method no more than unnecessary line noise.

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 inability of the query expression language to nest columns within columns. If a column of type *Col s (Col s' a)* could be

constructed and inspected, further measures would be required to ensure that the nesting level of the inner column would not escape its scope.

4.1 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* [11]. 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 and SQL-like, 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. *ScalaQL* [14] provides a similar interface for Scala, while the *Opaleye* EDSL [5] uses arrows and profunctors to provide the same for Haskell. Similarly, the *Esqueleto* Haskell EDSL [10] provides a continuation-based interface to database queries. Silva and Visser [13] describe an ingenious ad hoc embedding of database queries into Haskell, capable of supporting functional dependencies and other advanced features. Lennartsson and gren [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 [9]. *Beam* [1] and *Haskell Relational Record* [6] are more recent attacks on the problem, which provide monadic interfaces that do support joins and aggregation, but only over database tables, and not over inner queries.

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 the current state of the art.

5 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. 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.

Acknowledgements

Many thanks to Mauro Jaskelioff, Tom Ellis, Koen Claessen, and Micha Paka, for their valuable insights, feedback and discussion.

References

1. Athougies, T.: Beam. http://travis.athougies.net/projects/beam.html (2016)
2. Augustsson, L., gren, M.: Experience Report: Types for a Relational Algebra Library. In: Proceedings of the 9th International Symposium on Haskell. pp. 127–132. Haskell 2016, ACM, New York, NY, USA (2016)
3. Cheney, J., Hinze, R.: First-class phantom types. Tech. rep., Cornell University (2003)
4. Eisenberg, R.A., Vytiniotis, D., Peyton Jones, S., Weirich, S.: Closed type families with overlapping equations. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM (2014)
5. Ellis, T.: Opaleye. https://github.com/tomjaguarpaw/haskell-opaleye (2014)
6. Hibino, K., Murayama, S., Yasutake, S., Kuroda, S., Yamamoto, K.: Haskell relational record. http://khibino.github.io/haskell-relational-record/ (2015)
7. Kiselyov, O., Lämmel, R., Schupke, K.: Strongly typed heterogeneous collections. In: Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell. pp. 96–107. Haskell '04, ACM, New York, NY, USA (2004), http://doi.acm.org/10.1145/1017472.1017488
8. Launchbury, J., Peyton Jones, S.L.: Lazy functional state threads. In: ACM SIGPLAN Notices. vol. 29, pp. 24–35. ACM (1994)
9. Leijen, D., Meijer, E.: Domain specific embedded compilers. In: ACM Sigplan Notices. vol. 35, pp. 109–122. ACM (1999)
10. Lessa, F.: Esqueleto. http://hackage.haskell.org/package/esqueleto (2012)
11. Meijer, E., Beckman, B., Bierman, G.: Linq: Reconciling object, relations and xml in the .net framework. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data. pp. 706–706. SIGMOD '06, ACM, New York, NY, USA (2006), http://doi.acm.org/10.1145/1142473.1142552
12. Miyamoto, S.: The Legend of Zelda. https://en.wikipedia.org/wiki/Link_(The_Legend_of_Zelda) (1986)
13. Silva, A., Visser, J.: Strong types for relational databases. In: Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell. pp. 25–36. Haskell '06, ACM, New York, NY, USA (2006), http://doi.acm.org/10.1145/1159842.1159846
14. Spiewak, D., Zhao, T.: Scalaql: language-integrated database queries for scala. In: International Conference on Software Language Engineering. pp. 154–163. Springer (2009)
15. Svenningsson, J.D., Svensson, B.J.: Simple and compositional reification of monadic embedded languages. In: ACM SIGPLAN Notices. vol. 48, pp. 299–304. ACM (2013)
16. Wadler, P.: Monads for functional programming. In: International School on Advanced Functional Programming. pp. 24–52. Springer (1995)