Comprehending Monoids with Class (Extended Abstract)

Lionel Parreaux
EPFL, Switzerland

Christoph E. Koch
EPFL, Switzerland

Abstract

The design of embedded database query languages has long relied on monadic comprehension (and specifically list comprehension), a natural foundation for expressing queries over collections of data. We argue that *monoid* comprehension is an interesting alternative foundation for such languages. We show that a generalized version of the monoid comprehension calculus can be naturally encoded in languages with support for type classes, and that this unlocks a new kind of expressive power — among other things, it gives us a grouping construct for free and it allows queries mixing heterogeneous data types (lists, sets, multisets, infinite streams, maps, etc.), while using the type system to statically verify some desirable properties about these queries. We believe that these new directions have the potential of making language-integrated queries more pleasant to use, more expressive, and eventually easier to optimize.

ACM Reference Format:
Lionel Parreaux and Christoph E. Koch. 2018. Comprehending Monoids with Class (Extended Abstract) . In *Proceedings of Type-Driven Development (TyDe'18).* ACM, New York, NY, USA, 3 pages. https://doi.org/

1 Introduction

In functional programming circles, the study of monadic list comprehension and of its generalization to arbitrary monads has attracted persistent interest [1, 4, 5, 7–11]. Below is an example list comprehension that computes the cartesian product of two lists xs and ys and filters each resulting (x, y) pair so that x is greater than y:

$$[\ (x,y) \mid x \leftarrow xs,\ y \leftarrow ys,\ x > y\ ]$$

In this extended abstract, we turn our attention to a different interpretation of comprehension based on monoids rather than monads, and we argue that it is often more appropriate for expressing database queries. The idea is not new, but seems not to have received the attention it deserves from this community. The list comprehension above can be rewritten in monoid comprehension syntax [2] as follows:

$$++\{\ (x,y) \mid x \leftarrow xs,\ y \leftarrow ys,\ x > y\ \}$$

where ++ denotes list concatenation. One major difference of this calculus is that xs and ys are *not* required to be lists — they can be other "collection monoids" (monoids with a *unit* element), and the aggregated result needs not be a list either — it can be another monoid (not necessarily a collection). In general, a monoid comprehension has syntax $\mathcal{M}\{\ e \mid \overline{p}\ \}$, where each p is either a generator $x \leftarrow xs$ or a boolean predicate acting like a guard, and $\mathcal{M}$ is the merge operation denoting the result monoid. Crucially, not all combinations of monoids are allowed. For example, if one generator's right-hand side is a set, the result type cannot be a list, because that would make the semantics of the query dependent on the order in which the set is iterated (which is unspecified). This not only makes query semantics deterministic, but also gives more freedom to the query engine, which has more options for parallelizing the query execution.

TyDe'18, September 27, 2018, St. Louis, Missouri, United States

This notation, introduced by Fegaras and Maier [2, 3] only supports a predefined set of monoids. Though the principles are general and actual languages may add support for more monoids, there is no way to express the *composition* of monoid types from existing types, which is a staple of functional programming with type classes. For example, a tuple of two monoids is also a monoid, and a map where the values form a semigroup is a monoid; but how shall we denote the 'merge' operation $\mathcal{M}$ for such compositions?

Notation being a central tool of thought [6], it is perhaps unsurprising that this restrictive notation has masked the true potential of monoid comprehensions for so long. We can get a sense of their real expressive power by writing them in terms of type classes, which leave monoid instances to be resolved and composed implicitly. In the example below, we demonstrate an embedding in Scala, whose `for`-comprehension syntax is not necessarily monadic and can easily be repurposed to accept a *monoidal* interpretation:

```
for { fname <- fileNames
      word  <- streamFile(fname).characters.splitOn(' ')
      if word.nonEmpty }
  yield ( avg(word.length.toDouble) ,
          count().groupedBy(word.toLowerCase) )
```

This query iterates over all the words contained in a set of files and aggregates the global average word length as well as per-word case-insensitive occurrence counts. This is desugared to a composition of `map`, `flatMap` and `filter`, which we have overloaded to aggregate monoids. For example, one signature of `map` is `(A => R) => As => R` where `R` has to be a monoid and `As` has to be a finite source of `A` elements. The type inferred is `(Option[Avg[Double]], Map[String, NonZero[Nat]])`.

This query cannot be written as a single-pass monadic comprehension: first, we would have to express several traversals, as monadic comprehension does not offer a way to aggregate values "in parallel"; second, we would not be able to mix different collection types as above, requiring explicit conversions; third, we would be creating many more intermediate collections; fourth, we would have to use an ad-hoc

Canonical Semigroup	Associated Canonical Monoid	Properties
(`NonZero[Nat], _ + _`)	(`Nat, _ + _, 0`)	C
(`List[T], _ ++ _`)	(`List[T], _ ++ _, Nil`)	O F
(`NonEmpty[Set[T]], _ union _`)	(`Set[T], _ union _, Set.empty`)	C I F
(`Max[Nat], _ max _`)	(`Max[Nat], _ max _, 0`)	C I
(`Max[Int], _ max _`)	(`Option[Max[Int]], _.flatMap(m => m max _), None`)	C I
(`Streamed[T], _ concat _`)	(`Streamed[T], _ concat _, Streamed.empty`)	L O
(`Incr[Set[T]], _ concat _`)	(`Incr[Set[T]], _ concat _, Incr.empty`)	I L O
(`Map[K,NonZero[Nat]], _ merge _`)	(`Map[K,NonZero[Nat]], _ merge _, Map.empty`)	C F

Table 1. Some example canonical semigroup instances, their associated canonical monoid forms, and their properties. Where C = commutative, I = idempotent, L = lazy, and for data sources O = ordered, F = finite.

implementation of the group-by functionality, whereas all the syntax `x.groupedBy(y)` above does is return a singleton `Map(x -> y)`, which is a monoid, giving us grouping for free. The monadic query would look like like the following:

```
val (lens,wrds) = (for { fname <- fileNames
  wrd <- streamFile(fname).characters.splitOn(' ').toList
 if wrd.nonEmpty }
 yield (wrd.length.toDouble, w.toLowerCase)).unzip;
(average(lens), wrds.groupBy(identity).mapValues(_.size))
```

In the rest of this extended abstract, we briefly describe our generalization of MCC, and its embedding in Scala — to the best of our knowledge the first such typed embedding.

2 Semigroups and Canonical Monoids

Reasoning exclusively about monoids is too restrictive; semigroups (which are like monoids, but do not require a zero element) come up when we know that an aggregation will at least consume one element — that is the case when grouping elements into a map, as each sub-aggregate for a given key will have at least one element, otherwise the key simply would not be in the map.

We determine the semantics of comprehensions based on the monoid and semigroup instances[1] of the types involved in the **`yield`** part of the queries. In order to use non-standard instances (such as product on integers instead of sum), we use zero-overhead new-types such as `Product` and `Max`, with functions `product(x: N): Product[N]` for `Numeric` types `N` and `max(x: O): Max[O]` for types `O` with an `Ordering` instance, etc.

Many aggregation types are semigroups but not monoids; for example, *minimum* on natural numbers or *union* on non-empty sets. In particular, we provide the `NonZero[N]` and `NonEmpty[X]` wrapper types, which are zero-overhead "phantom subtypes" (so that `NonZero[N] <: N` and `NonEmpty[Xs] <: Xs`) that statically add more information to a type — a sort of simple type refinement — and these types are only semigroups when their wrapped type is a monoid. Note that any semigroup can be lifted to a monoid by wrapping it in an `Option` type, where `None` becomes the ad-hoc zero element, but some semigroups actually have more natural monoid generalizations than wrapping them in an `Option` type. For example, the canonical monoid form of `NonZero[Nat]` is `Nat`.

Naturally, it should be illegal to write a comprehension that, for instance, aggregates the minimum age in a list of persons, i.e., `for { p <- persons } yield min(p.age)` (because if `persons` is empty, the result is ill-defined). However, it would make for a poor user experience to flat-out reject such queries and require users to write **`yield`** `Some(min(p.age))`; instead, we defined a type class which automatically lifts a semgroup to its "canonical monoid" when required. In the case above, it will give our query return type `Option[Min[Nat]]`. On the other hand, `count()` has return type `NonZero[Nat]` whose canonical monoid is `Nat`, not `Option[NonZero[Nat]]`, so a query ending with **`yield`** `count()` will have return type `Nat`, while a query ending with **`yield`** `count().groupedBy(k)` (which is really syntactic sugar for the singleton `Map(k -> 1)`) will have return type `Map[K,NonZero[Nat]]`. Table 1 gives some more examples.

3 Heterogeneous Collection Types

In its original formulation, the monoid comprehension calculus of Fegaras and Maier [2, 3] distinguishes between whether the source collection monoids are ordered, may contain repeated elements, or both. This determines which properties the result monoid should have for the query to have the properties alluded to in §1 (well-defined semantics and ability to be parallelized); respectively, it should be: commutative, idempotent, or both. We refine and generalize these notions with more source properties and their associated monoid restrictions, namely: if the source collection is `NonEmpty` the result only needs to be a semigroup; and if the source is *not* known to be finite, then the result monoid must be what we call "lazy" or "incremental" (this allows aggregating streams and defining infinite stream pipelines).

All these conditions and restrictions are enforced statically via Scala's type system, using implicit-based overloading (type classes) together with Scala's mechanism for prioritization of implicit search, so that the most specific (i.e., the less restrictive) **`for`** comprehension interface is selected automatically depending on the types of the source collections.

[1] We use the open-source *cats* functional programming library for Scala, which provides type classes such as `Monoid` and `CommutativeMonoid`, as well as many standard instances (https://github.com/typelevel/cats).

References

[1] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2007. Links: Web Programming Without Tiers. In *Formal Methods for Components and Objects*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 266–296.

[2] Leonidas Fegaras and David Maier. 1995. Towards an Effective Calculus for Object Query Languages. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (SIGMOD '95)*. ACM, New York, NY, USA, 47–58.

[3] Leonidas Fegaras and David Maier. 2000. Optimizing Object Queries Using an Effective Calculus. *ACM Trans. Database Syst.* 25, 4 (Dec. 2000), 457–516.

[4] Mary F. Fernandez, Jérôme Siméon, and Philip Wadler. 2001. A Semi-monad for Semi-structured Data. In *Proceedings of the 8th International Conference on Database Theory (ICDT '01)*. Springer-Verlag, London, UK, UK, 263–300.

[5] Jeremy Gibbons. 2016. Comprehending Ringads: For Phil Wadler, on the Occasion of his 60th Birthday. In *A List of Successes That Can Change the World (LNCS)*, Vol. 9600. Springer, 132–151.

[6] Kenneth E. Iverson. 1980. Notation As a Tool of Thought. *Commun. ACM* 23, 8 (Aug. 1980), 444–465. https://doi.org/10.1145/358896.358899

[7] Simon Peyton Jones and Philip Wadler. 2007. Comprehensive Comprehensions. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop (Haskell '07)*. ACM, New York, NY, USA, 61–72.

[8] Erik Meijer, Brian Beckman, and Gavin Bierman. 2006. LINQ: Reconciling Object, Relations and XML in the .NET Framework *(SIGMOD '06)*. ACM, 706–706.

[9] Phil Trinder. 1992. Comprehensions, a Query Notation for DBPLs. In *Proc. of the 3rd DBPL workshop (DBPL3)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 55–68.

[10] Philip Wadler. 1990. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP '90)*. ACM, New York, NY, USA, 61–78.

[11] LIMSOON WONG. 2000. Kleisli, a functional query system. *Journal of Functional Programming* 10, 1 (2000), 19âĂŞ56.