Comprehending Monoids with Class (Extended Abstract)

Lionel Parreaux
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:

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:

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

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 $M\{ e \mid p \}$, where each $p$ is either a generator $x \leftarrow xs$ or a boolean predicate acting like a guard, and $M$ 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.

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 $M$ 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:

```scala
for { fname <- fileNames
      word <- streamFile(fname).characters.splitOn(' ')
      if word.nonEmpty } yield ( avg(word.length.toDouble),
                                      count().groupBy(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 $(A \Rightarrow R) \Rightarrow As \Rightarrow R$ where $R$ has to be a monoid and $As$ has to be a finite source of 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
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.

<table>
<thead>
<tr>
<th>Canonical Semigroup</th>
<th>Associated Canonical Monoid</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>(NonZero[Nat], +)</td>
<td>(Nat, +, 0)</td>
<td>C</td>
</tr>
<tr>
<td>(List[Int], ++)</td>
<td>(List[Int], ++, Nil)</td>
<td>O F</td>
</tr>
<tr>
<td>(NonEmpty[Set[T]], _ union _)</td>
<td>(Set[T], _ union, Set.empty)</td>
<td>C I F</td>
</tr>
<tr>
<td>(Max[Nat], _ max _)</td>
<td>(Max[Nat], _ max, 0)</td>
<td>C I</td>
</tr>
<tr>
<td>(Max[Int], _ max _)</td>
<td>(Option[Max[Int]], _flatMap(m =&gt; m max _), None)</td>
<td>C I</td>
</tr>
<tr>
<td>(Streamed[T], _ concat _)</td>
<td>(Streamed[T], _ concat, Streamed.empty)</td>
<td>L O</td>
</tr>
<tr>
<td>(Incr[Set[T]], _ concat _)</td>
<td>(Incr[Set[T]], _ concat, Incr.empty)</td>
<td>I L O</td>
</tr>
<tr>
<td>(Map[K,NonZero[Nat]], _ merge _)</td>
<td>(Map[K,NonZero[Nat]], _ merge, Map.empty)</td>
<td>C F</td>
</tr>
</tbody>
</table>

Table: 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.

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 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: 0): Max[0] for types 0 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 semigroup 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.

Footnote 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


