Collection Processing with Constraints, Monads, and Folds

Ryan Wisnesky
Harvard University
ryan@cs.harvard.edu

Abstract

We propose an intermediate form based on monad-algebra comprehensions (to represent queries), folds (to represent computation), and setoids over polynomial datatypes (to represent data), suitable for use in collection processing. Such an intermediate form captures, in a uniform way, large fragments of many recent large-scale collection processing languages such as MapReduce, PIG, DryadLINQ, and Data Parallel Haskell, and admits optimization techniques from both programming language theory and relational database theory. We show how to solve four key problems inherent in the naive approach by drawing together recent work from both communities. First, we show how fold fusion can be extended, in an arguably complete way, to exploit the monadic structure of queries. Second, we show how monad comprehensions can be extended to monad-algebra comprehensions, so as to express aggregation and thereby enable comprehension-based optimizations for a wide class of queries. Third, we show how to embed a particular syntactic class of constraints called embedded dependencies into our intermediate form and show how such constraints can be used, for example, to minimize the number of bind operations in a monad comprehension - a process traditionally known as semantic optimization. Finally, we show how to emit proof obligations from our language, so as to ensure that each program is sound with respect to required axioms (such as the monad laws) as well as user-provided invariants (such as ensuring all sets are represented by lists without duplicates).

1. Introduction

Collection processing is a fundamental problem in computer science, and specialized languages for collection processing are almost as old as programming itself. SETL [24] (short for set language) first appeared in 1969 and bears a striking resemblance to mathematical set theory, providing operations such as membership, union, intersection, comprehension, powerset, and quantification. SETL was remarkable for its high level of abstraction: users specified *what* they wanted, not *how* to compute it. Languages that focus on the what over the how are called *declarative*, and by the early 1980s, the success of the relational model firmly established the dominance of the declarative paradigm for collection processing.

Declarative languages are useful for collection processing primarily because of scale. Today it is possible to cheaply process petabytes of data on thousands of networked commodity machines. At this scale it is infeasible for a programmer to completely specify how to compute a given query. On the other hand, the poor asymptotic performance of naively executed declarative queries means that without sophisticated optimization, even simple queries can take entire lifetimes to run. For this reason, the optimization and implementation of queries in a variety of declarative languages on a variety of systems has been extensively studied by the programming language, database, and systems communities over many decades [21].

Although modern collection-oriented languages are invariably declarative, they are not always relational. SQL is a (quasi-relational) mainstay of collection processing, but the past decade has seen an explosion of more expressive languages designed for very large-scale collection processing over clusters of commodity machines. Examples of such systems include MapReduce-Merge [30], Data Parallel Haskell [8], DyadLINQ [20], PIG [22], and Fortress [10], as well as their predecessors NESL [3], and Kleisli [29]. These languages are *functional query languages* [26], which base their syntax and semantics, at least in part, on *purely functional* programming languages.

Functional query languages extend relational query languages by providing more general kinds of data and more expressive operations. Whereas relational algebra provides only relations as data and six operations - join, project, select, union, difference, and rename, functional query languages may provide for additional, nested datatypes such as lists and trees, as well as recursive functions. Although functional query languages vary in the kinds of queries and collections they support, large fragments of these languages can be formalized in a uniform way using *monads* (to model collections), *comprehensions* (to model queries), setoids over polynomial datatypes (to model data), and folds (to model computation) [16].

In arguing for the usefulness of such a formalization, Grust observed [16] that effective intermediate forms for a query languages have the following properties:

- (Ex) Capability of expressing any user query (this affects query operators as well as the type system).
- (Abs) Abstraction from user-level queries while providing a granularity of representation that makes all relevant query language constructs subject to inspection and transformation.
- (Eq) A well-understood equational theory that decides the equivalence preservation of expression transformations.
- (Map) A suitable starting point for a mapping to the access primitives of the underlying persistent store.

Relational algebra meets these criteria, and indeed has proven to be an effective intermediate form for modern relational database systems. A sketch of the design of a good intermediate form for more general functional query languages is the principle contribution of this paper, with additional details and an implementation to follow in the author's forthcoming Ph.D. thesis.

1.1 Motivation

The idea of an intermediate form and query processing pipeline based on folds and setoids over polynomial datatypes was first explored by the relational database community in the 90's during a search for principled extensions to the relational model [5]. However, optimizing folds over setoids is difficult because it is not decidable if a fold respects a setoid. For this reason, the community developed more restrictive languages, like the nested relational calculus (NRC), which could be translated into folds in a principled way and yet still admit the sophisticated optimization developed for relational calculus over the proceeding two decades. Unfortunately, the NRC is not expressive enough to capture the functional query languages in use today. (Besides lacking (Ex), it is also unclear if the NRC possesses (Abs) and (Map) - see [15] for details).

Around the turn of the millenium, Grust picked up the fold-setoid idea and developed a formalism, the Monad Comprehension Calculus (MCC), that provides a theoretical starting point for our

work [15, 16]. The MCC is essentially a simply-typed lambda calculus extended with polynomial datatypes and fold primitives, as well as syntactic sugar (essentially do-notation) for monad comprehensions. It meets the above criteria: (Ex) holds because all primitive recursive functions can be expressed using folds; (Abs) holds because monad comprehensions de-sugar into folds; (Eq) holds because the algebra of programming [2] and equational theory of the λ-calculus are well-understood; and (Map) holds because an implementation need only implement fold, but can potentially implement other special operations (such as join or intersection).

Although it is a good starting point, the MCC is not directly useful for modern collection processing for several reasons:

- (Lack of Fusion) The primary optimization technique for the MCC is well-known *fold-build fusion* [14]. This kind of fusion eliminates intermediate data structures and can be very powerful when it applies, but applicability is limited. For example, it cannot optimize certain uses of the list append function. This is particularly problematic for query processing, where such operations are common. Moreover, fusion applies only to folds, and cannot take advantage of the monadic structure of queries.
- (Lack of Aggregation) Comprehension notation, as developed in the MCC (and in Haskell), cannot be used for aggregation. For example, it is impossible to write a comprehension to compute the sum of a list of integers. Although it is possible to aggregate collections using folds directly, doing so means that comprehension notation is not expressive enough to capture all user-level queries – thereby violating criteria (Ex) and (Abs).
- (Lack of Constraints) The MCC lacks any mechanism for expressing constraints; that is, properties that hold of the particular data at hand, such as key constraints or join decompositions. As years of experience in relational data management demonstrate, such properties are very important at scale [4]. For example, a key constraint can allow what would be a full scan of a set to be replaced with a simple lookup.
- (Lack of Proofs) The MCC (and functional query languages in general) place a lot of trust in the programmer. For example, it is up to the programmer to verify that a monad actually obeys the monad laws, or that a set is correctly represented and manipulated using an underlying list. A compiler can not in general prove that such undecidable properties hold.

Moreover, the MCC lacks programming language features which are particularly useful for collection processing, such as type-inference, row-polymorphism, and extensible records.

1.2 Contributions

The primary contribution of this paper is to show how to address the above challenges. We propose an intermediate form based on monad-algebra comprehensions, setoids over polynomials datatypes, and folds which is suitable for use with a variety of collection-oriented languages. In the decade since the MCC was proposed, each of these problems has been thoroughly studied by either the programming languages or database theory community. Our proposed intermediate form applies these results:

- To address (Lack of Fusion), we show how recent theoretical work on *monadic augment* can be used to optimize queries. The *augment* operation, originally proposed by Gill [14] to cover the problematic case of the non-fusability of list append, generalizes the build operation and is more amenable to fusion. In [13], the authors show that for many monads, the bind operation can be written as an augment, enabling significantly more optimization opportunities.
- To address (Lack of Aggregation), we show how comprehensions, which traditionally are defined in terms of a monad, can instead be defined in terms of a *monad algebra*. This additional generality allows comprehensions to compute aggregations such as the sum of a list of integers. Moreover, our comprehension optimizations apply uniformly to both monad comprehensions and monad algebra comprehensions [23].
- To address (Lack of Constraints) we show how to optimize monadic queries in the presence of so-called embedded, implicational dependencies, a technique originally developed in relational database theory [23]. Constraints induce additional equations between comprehensions that can be exploited later, for example to minimize the number of bind operations in a monad comprehension.
- To address (Lack of Proof) we show how to emit proof obligations which can be solved semi-interactively by the user in the Coq proof assistant. The correctness of the user program, as well as our optimizations, is contingent on the provability of these obligations. Given that many queries take days to run, we do not see the additional effort of semi-interactive proving as an undue burden, especially with recent advances in proof engineering.

Our intermediate form combines elements of relational database theory and programming language theory in fundamentally new and novel ways. We hope that it will serve as a vehicle for further technology transfer between the two areas.

1.3 Related Work

The main alternative to functional query languages are extended relational algebras, which were heavily studied by the database community in the 1990s [6, 28]. Such languages provide nested relations as well as grouping, aggregation, and recursion operations. Operations are chosen so as to be natural extensions of relational operations. This leads to a smooth integration with traditional relational database theory, but most modern collection-oriented languages descend more from the functional query language tradition. See [15] for a comparison of the two approaches.

1.4 Outline

We begin with an overview of our intermediate form in Section 2. This overview should be accessible to anyone with a functional programming background, but Haskell experience is particularly useful. The four areas above are then each addressed in a section: generalized fusion in Section 3, aggregation in Section 4, reasoning under constraints in Section 5, and obligation generation in Section 6. A diagram of our compiler is shown in Figure 1.4. Note that features which are de-sugared before translation into an intermediate form in traditional compilers (such as comprehensions and folds) form the basic concepts involved in our compiler; this suggests that our techniques may apply to early compiler pipeline stages in general purpose languages such as Haskell.

2. Overview

Our formal starting point is Jones' qualified type system of extensible records and variants [12], which can be thought of as a variant of Haskell and is implemented in the TREX extension to Hugs. It has many important properties:

- Strong normalization
- Sound and complete type inference
- Extensible records and variants, equated up-to label permutation

Each of these features is indispensible in collection processing. Rather than describe in detail how records and variants behave in our language, we refer the reader to [12] and simply note that the intended meaning of $(lbl_1 : x, lbl_2 : y)$ is a record with two labels, and $.lbl$ represents the operation of projecting label lbl. We will use standard Haskell syntax whenever possible in this paper.

Unlike Haskell, we disallow recursive function definitions and certain algebraic datatype definitions. Instead we have only one particular kind of recursion: folds over polynomial datatypes. Consider a datatype definition:

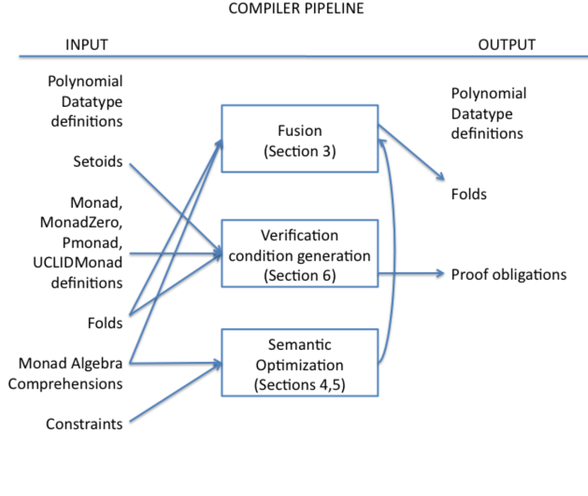

Figure 1. Compiler Pipeline

```
data List a = Nil | Cons A (List A)
```

This creates a type constructor `List` and data constructors `Nil` and `Cons`. Rather than providing case analysis, in our language the only way to process a value of type `List` t is by using a *fold*. This datatype definition creates a `fold` combinator, which is subscripted by the name of the inductive type it is for; subscripts can be readily inferred and will be omitted when possible. The `fold` operation is primitive in our language, with no programmer-visible body. However, `fold` for lists obeys the equations:

```
fold : B -> (A -> B -> B) -> listraw A -> B
fold nil' cons' Nil = nil'
fold nil' cons' (Cons hd tl) =
  cons' hd (fold nil' cons' tl)
```

It is easy to see that folding is essentially replacing the data constructors `Nil` and `Cons` with the functions nil' and $cons'$ of appropriate type; indeed, $\texttt{fold}_X$ is a "spine transformation" or homomorphism between X-algebras. Folds, in combination with records, are sufficient to express all primitive recursive functions over such initial algebras [19]. Here is how to define a count function:

```
count :: List a -> nat
count = fold 0 (\hd tl -> 1 + tl)
```

In addition to `fold`, a datatype declaration also creates a `build` operator, which serves as a kind of inverse to fold. For lists, `build` obeys the following equation:

```
build :: (forall b. b -> (a -> b -> b) -> b) -> List a
build g = g Nil Cons
```

Here, the function g must be parametric in the type b, which is indicated by the placement of the `forall` quantifier in the type. The `build` and `fold` operations are related by the following law, which serves as the basis of fold-build fusion:

$$\texttt{fold}\, n\, c\, (\texttt{build}\, g) \;=\; g\, n\, c$$

The soundness of the fold-build fusion law depends crucially on parametricity [27] holding for our language. We believe this to be the case.

`build` and `fold` exist for any polynomial datatype (using initial algebra semantics). Just like with lists, folds over other datatypes specify a "spine-transformer" recursion scheme for which fusion applies. For details, see [13]. Polynomial datatypes are algebraic datatypes that don't use function spaces, and instead consist solely of products and sums. For example, we must reject the following definitions:

```
data Bad = C (Int -> Bad)

data Bad = Lam (Bad -> Bad)
```

Although it is possible to define a fold over the first datatype, it is unclear how to interpret it as a concrete collection. The second definition has a non-positive use of a function space (that is, the type being defined appears to the left of an arrow), and hence has no fold combinator.

2.1 Quotients

Polynomial datatypes alone are not sufficient for representing *non-free* collections such as sets and bags. Such collections have an equality that is coarser than simple structural equality. Hence to represent such collections, we must quotient a carrier such as `List` by an equivalence relation. Our intermediate form provides support for doing so in a first-class way. As an example, we can obtain sets in insert presentation[1] by quotienting lists with an equivalence relation such that `Cons` 1 (`Cons` 1 `Nil`) is related to `Cons` 1 `Nil`:

```
Set is List quotient by
 (==) :: Eq a => List a -> List a -> Bool
 a == b = ...
 -- a == b when forall x, lookup x a  = lookup x b
```

A lifting operation is generated to lift a function from a carrier to a quotient. At runtime, lifting is a no-op. For example:

```
count' :: Set a -> Nat
count' = lift_Set count
```

Note that `lift` is a family of operations, and has no single type. Rather, its type is the type of its input with the quotient type substituted for the carrier type. Principled ways to lift are discussed in [25]. As we will see later, each use of lift or fold at a quotient type inserts a proof obligation that the underlying equivalence relation is respected. In this example we cannot actually discharge the generated obligation because it is not the case that `count` (`Cons` 1 (`Cons` 1 `Nil`)) = `count` (`Cons` 1 `Nil`).

The Boom Hierarchy

Bags, lists, and sets using *insert presentation*; that is, `Nil` and `Cons` are traditional examples in collection processing because they have a minimal number of constructors. However, to demonstrate the expressiveness of polynomial datatypes and equivalence relations let us first define *union presentation*, which makes use of three constructors, `Empty`, `Singleton`, and `Union`.

```
data Boom A =
   Empty
 | Singleton A
 | Union (Boom A) (Boom A)
```

The fold over `Boom` is (omitting `empty'`, `singleton'`, and `union'` for brevity):

```
fold Empty = empty'
fold (Singleton hd) = singleton' hd
fold (Union l r) = union' (fold l) (fold r)
```

The free structure is binary trees. The *Boom hierarchy* is a family of 16 data structures formed by taking, independently, an equivalence relation containing up to four properties: a unit for `Union`, associativity of `Union`, commutativity of `Union`, and idempotency of `Union`. Some of the data structures in the Boom hierarchy are well-known; for example, associativity of `Union` gives us lists in union presentation. Others are less well-known; for example [7], adding idempotency of `Union` gives us lists that disregard equivalent adjacent sublists. For example, *banana* is equivalent to *bana* is equivalent to *bbbabababana*. The ability to deterministically convert

[1] Insert presentation means that each set is defined as either the empty set or by the insertion of an element into a set. In other words, `Nil` and `Cons`.

structures of one type to another (for example, list to set but not set to list) partially orders the Boom hierarchy. In our intermediate form, only conversions that respect this order generate provable obligations.

2.2 Collection Monads

We will express collections using *monads with zeros*. A monad with zero consists of a type constructor M and three operations, `return` $: t \rightarrow M\ t$, `bind` $: M\ t \rightarrow (t \rightarrow M\ t') \rightarrow M\ t'$, and `zero` $: M\ t$. Our monads are constructed just like in Haskell, using the typeclass mechanism. For example,

```
instance Monad List where
 return :: t -> List t
 return x = Cons x Nil

 bind :: List t -> (t -> List t') -> List t'
 bind x f = concat (map f x)

 zero :: List t
 zero = Nil

--

instance Monad Set where
 return = Singleton
 bind x f = Union (map f x)
 zero = Empty
```

Here, `concat` flattens a list of lists into a single list, and `union` flattens a set of sets into a single set. `map` applies a function to each element of a set or list. Monads with zero must obey five laws (up-to equivalence relations):

```
bind (return x) f = f x
bind m return = m
bind (bind m f) g = bind m (\x -> bind (f x) g)
bind zero f = zero
bind m (\x -> zero) = zero
```

Our compiler emits proof obligations that these properties hold.

Although it is not immediately obvious, binary trees (that is, the free structure of the Boom datatype in the previous subsection) do not have a zero [23], but using an equivalence relation that admits a unit for `Union` does imply a monadic zero.

To make it easier to write monadic expressions, we will use Haskell's standard do-notation:

```
do x <- X
   c
= bind X (\x -> c x)
```

For example, the cartesian product of two sets can be expressed as:

```
do x <- X
   y <- Y
   return (l: x, r: y)
```

Remark By using Haskell-style type-classes, we limit ourselves to associating a single monad definition with each algebraic datatype. In practice, this restriction is infeasible. We have yet to determine a good solution to this problem, but look to [25] for inspiration.

2.3 Comprehensions and Queries

As functional programmers know, monads with zeroes support a particularly compact notation known as *comprehension notation* [4]. To enable powerful optimizations originally developed in database theory, we will be focusing on a particular subset of comprehensions which are *path conjunctive* [23]. A path conjunctive comprehension is a *query* and is intended to be the primary way in which users write computations. They have the the following form. We will abbreviate vectors of variables $x_1, ..., x_N$ as $\overrightarrow{x}$.

Fix a monad with zero M and let $\overrightarrow{X : M\ t}$. We will write $P(\overrightarrow{x})$ and $B(\overrightarrow{x})$ to indicate a conjunction of equalities between paths (sequences of record projections) with head variables $\overrightarrow{x}$. We will write $R(\overrightarrow{x})$ to indicate an arbitrary expression containing (potentially) the variables $\overrightarrow{x}$. A query has the form:

$$\begin{array}{l} \texttt{for}\ \overrightarrow{(x\ \texttt{in}\ X)} \\ \texttt{where}\ P(\overrightarrow{x}) \\ R(\overrightarrow{x}) \end{array}$$

Despite their unfamiliar form, path-conjunctive queries are interpreted in a straightforward way:

$$\begin{array}{ll} \texttt{do} & x_1 \leftarrow X_1 \\ & \ldots \\ & x_N \leftarrow X_N \\ & \texttt{if}\ P(x_1, \ldots, x_N) \\ & \texttt{then}\ R(x_1, \ldots, x_N) \\ & \texttt{else zero} \end{array}$$

An example query is

```
query :: MonadZero M => M (l: t) -> M (l: t) -> M t
query r s = for (x in r) (y in s)
            where x.l = y.l
            return x.l
```

As another example, in the set monad the following query returns (a set of) tuples (d, a) where a acted in a movie directed by d:

```
query :: MonadZero M =>
          M (director: String, actor: String) ->
          M (d: String, a: String)
query movies = for (m1 in movies) (m2 in movies)
               where m1.title = m2.title
               return (d: m1.director, a: m2.actor)
```

In fact, path-conjunctive comprehensions such as these correspond precisely to "select-from-where" queries in SQL and "select-project-join" queries in relational algebra. We may write the above query in SQL as:

```
SELECT m1.director. m2.actor
FROM Movies AS m1, Movies AS m2
WHERE m1.title = m2.title
```

Haskell's list comprehension syntax is also similar. Finally, note that it is possible to allow generators to be dependent, which corresponds to nesting; for example: `for (x in S.x) (y in x) ...`

Having given an overview of the naive monad comprehension calculus, we turn now to the problems with this approach and our solutions.

3. Fusion

Because of the scale at which modern collection processing systems operate, it is extremely important to remove intermediate data structures. At the implementation level, this usually means that operations such as `fold` do not operate on collections as a whole, but instead try to process their elements one-at-a-time. In effect, a graph of folds can become a pipeline of iterators, and this fundamental technique has been a mainstay of collection processing for many years. However, this technique is not a panacea, and it is the job of the intermediate form to reduce the number of folds (and builds) required of any given query. Although the database community has studied this problem for relational languages, in our more general setting we must turn to the programming language community for a suitable theory of fusion.

So-called "short-cut fusion" is traditionally introduced with the following example [13, 14]. Suppose we want to square the numbers in a list and then compute the sum. We would write:

```
sum :: List Int -> Int
sum = fold 0 (+)

map :: (a -> b) -> List a -> List b
map f xs = build (\n c -> fold n (c . f) xs)

sqr :: Int -> Int
sqr x = x * x

sumSqs :: List Int -> Int
sumSqs xs = sum (map sqr xs)
```

Here we are using Haskell abbreviations, writing `(+)` for the addition operation and `(c . f)` for c composed with f. As written, this definition produces an intermediate list, because `sumSqs` unfolds to:

```
sumSqs xs = fold 0 (+)
              (build (\n c -> fold n (c . sqr) xs))
```

Because the intermediate list (constructed by `build`) is immediately consumed by a fold, it is reasonable to expect that the list can be eliminated. In fact it can, using the build-fold fusion rule $\texttt{fold}\, n\, c\, (\texttt{build}\, g) \;=\; g\, n\, c$:

```
sumSqs = \xs -> (\n c. fold n (c . sqr) xs) 0 (+)
       = fold 0 ((+) . sqr)
```

Here it is easy to see that no intermediate sub-list is produced.

Fold-build fusion has been used to great effect in modern functional programming languages such as Haskell, and Grust proposed it as the primary optimization technique for the MCC, but there are some common situations in which it does not apply. Here the traditional examples is append (written as an infix ++):

```
ys ++ xs = fold ys Cons xs
```

Because append is a list producer, to enable fusion we would like to write it in terms of `build`. Without doing so, for example, we cannot apply fold-build fusion to the following:

```
fold z f (map g xs ++ ys)
```

However, writing append using `build` is impossible, as the following naive attempt shows:

```
ys ++ xs = build (\n c -> fold ys Cons xs)
```

This code is incorrect, because ys is a list, but needs to be element type. The problem is also apparent in trying to represent `Cons` as a `build`:

```
ourcons x xs = build (\n c -> c x s)
```

Of course, we can introduce an additional fold:

```
xs ++ ys = build (\n c -> fold (fold n c ys) c xs)
ourcons x xs = build (\n c -> c x (fold n x xs))
```

However, as Gill shows [14], it is sometimes impossible to remove these additional folds, and they introduce significant overhead. Thus Gill introduced a generalization of the `build` operation, called `augment`, for lists in particular:

```
augment : (forall b. a -> (a -> b -> b) -> b)
  -> List a -> List a
augment g xs = g xs Cons
```

The only difference between `build` and `augment` is that `augment` takes an additional argument xs which it uses in place of `Nil`:

```
build g = augment g Nil
```

We can now write our two earlier problematic definitions using `augment` instead of `build`:

```
ys ++ xs = augment (\n c -> fold n c xs) ys
ourcons x xs = augment (\n c -> c x s) xs
```

Fold-augment fusion is then given by the equation:

$$\texttt{fold}\, z\, k\, (\texttt{augment}\, g\, h) \;=\; g\, k\, (\texttt{fold}\, z\, k\, h)$$

Our earlier example can now be fused, and the fold-build rule is a special case of this rule. Fold-augment fusion is quite useful for collection processing, as we would expect operations such as append to abound. Unfortunately, Gill's definition of `augment` was only defined for lists, and hence it cannot be used with arbitrary collections, making it unsuitable as an optimization technique for an intermediate form such as ours. Fortunately, in 2005 Ghani et al were able to show how to generically define an `augment` operation for *parameterized monads* over polynomial datatypes [13]. A parameterized monad is a type class with three operations:

```
class PMonad pm where
 preturn :: a -> pm a c
 (>>!) :: pm a c -> (a -> pm b c) -> pm b c
 pmap :: (c -> d) -> pm a c -> pm a d
```

Obeying five laws:

```
>>! preturn = id
(>>! g) . preturn = g
(>>! (>>! g) . j) = (>>! g) . (>>! j)
pmap g . preturn = preturn
pmap g. (>>! j) = (>>! (pmap g . j)) . pmap g
```

A parameterized monad then induces a monad in a canonical way. This induced monad then has a bind operation which is interdefineable with augment:

```
augment g k = bind (build g) k
```

This is important because it allows fusion to apply to programs of the form `bind (bind x f) g`, which arise (by associativity) from a fairly typical programming patterns (like comprehensions) of the form

```
do x <- X
   y <- Y
   return (f x y)
```

In [13], Ghani further argues that fusion of this form is the best possible.

4. Aggregation

Whereas collection monads and monad comprehensions as described in the overview are likely to be well-known to the functional programmer, *monad algebras* and comprehensions over them probably are not. However, they are quite useful for collection processing [21]. Our interest in them is that they allow us to express aggregation operations using comprehension notation, and moreover, they can be treated in a uniform way using the optimization techniques developed in the next section. Although they can be defined in several different ways, the most straightforward is that a monad algebra is an operation $\texttt{agg} : Mt \rightarrow t$ for a *specific* t that obeys certain equations. For example, adding up all the numbers in a list is a monad algebra for the list monad, but adding up all the numbers in a set is *not* an algebra for the set monad – see [21] for details. The equations that must be respected are

```
agg . return = id
agg . (fmap agg) = agg . join
```

Here `fmap` and `join` have their standard definitions in terms of `return` and `bind`:

```
(fmap f) m = bind m (\x -> return (f x))
join n = bind n id
```

Note that monadic `join` (at each type) is a special case of `agg`. The "Kleisli form" of a monad algebra is an operation (family) $\texttt{loop} : M\ a \rightarrow (a \rightarrow b) \rightarrow b$; in this form, it is easy to see that monadic `bind` (at each type) is a special case of `loop`, by taking $b = M\ c$. This means that every monad has a canonical, "free" monad algebra. This is useful for us because it means that we can re-use our monad comprehension notation, but when translating into folds, we can translate either into the monad's `bind` operation or a more general monad algebra operation `loop`.

This can be made concrete as follows. Write `<--` to indicate `loop` for the summing of a list monad algebra. Then to sum a list using a comprehension, we simply write:

```
do x <-- X
   x
```

To sum a list after adding 1 to each element, we write

```
do x <-- X
   x + 1
```

To sum every pairwise element combination of two lists, we write

```
do x <-- X
   y <-- Y
   x + y
```

Writing aggregations as comprehensions takes some getting used to, and in particular the fact that there can be many monad algebras for a particular monad at a particular type makes indicating which one is intended a verbose endeavor, one for which we do not have a good solution for as yet. Other example monad algebras include logical `and` and `or` for sets of booleans, `max` and `min` for sets of integers (assuming that positive and negative infinity are integers), `sum` for bags of integers, and string concatenation for lists of strings. The monad algebras must also posses a zero-like operation that behaves correctly with respect to the underlying monad's zero.

Remark. Monads bear a resemblance to Monoids; in Haskell, monoids form a typeclass:

```
class Monoid m where
  mempty  :: m
  mappend :: m -> m -> m
```

In addition, `mempty` must be a left and right identity for `mappend`. Containers (as well as other things, like the integers) often form monoids; for example:

```
instance Monoid [a] where
  mempty  = []
  mappend = (++)
```

Monads with a zero and a plus are also monoids (hence, monads have more structure):

```
instance MonadPlus m => Monoid (m a) where
  mempty  = mzero
  mappend = mplus
```

Importantly, monads and monoids operate at different levels: a monad is a type constructor (of kind *type* $\to$ *type* whereas a Monoid is a type (of kind *type*). Fegaras and Maier [11] fabricated a query calculus known as the monoid comprehension calculus based on comprehension notation and monoid-homomorphisms. We are biased towards monads because our optimization techniques are monad-specific, but the exact connection between the monoid and monad comprehension calculi is not well-understood, and is an area for future work.

5. Constraints

At scale, the particular properties of the data at hand become important [4]. This phenomenon is readily apparent in relational database systems, where properties such as key constraints or join decompositions are exploited by a query optimizer. For example [1], consider the following relational query (i.e., in the set monad):

$$\begin{aligned} \mathit{MoviesBig} \quad ::= \quad & \texttt{for}\ (m_1\ \texttt{in}\ \mathit{Movies})\ (m_2\ \texttt{in}\ \mathit{Movies}) \\ & \texttt{where}\ m_1.\texttt{title} = m_2.\texttt{title} \\ & \texttt{return}\ (m_1.\texttt{director}, m_2.\texttt{actor}) \end{aligned}$$

This query returns (a set of) tuples (d, a) where a acted in a movie directed by d. A naive implementation of this query will require a join. However, when Movies satisfies the the functional dependency `title` $\to$ `director` (meaning that if (`director` : d, `title` : t, `actor` : a) and (`director` : d', `title` : t', `actor` : a') are records in Movies such that $t = t'$, then $d = d'$), this query is equivalent to

$$\begin{aligned} \mathit{MoviesSmall} \quad ::= \quad & \texttt{for}\ (m\ \texttt{in}\ \mathit{Movies}) \\ & \texttt{return}\ (m.\texttt{director}, m.\texttt{actor}) \end{aligned}$$

which can be evaluated without a join. (Note that if Movies did not satisfy the functional dependency, the equivalence would not necessarily hold.)

Of course, knowing that the functional dependency holds, a programmer might simply write the optimized query to begin with. But constraints are not always known at compile time, such as when collections are indexed on-the-fly. Moreover, people are not always the programmers: information integration systems such as Clio [17] automatically generate large amounts of code. The significant, potentially order-of-magnitude speed-ups enabled by optimizations of this form are well-documented in the literature and applied in commercial databases such as DB2 [18].

So, our intermediate form must provide a way to express constraints. Moreover, constraints should function as additional re-write rules, over comprehensions, during optimization [1]. In this section we show how to do so. The basic idea is that constraints should have the following syntactic form, called *embedded, implicational dependencies* [1].

$$\begin{aligned} C \quad ::= \quad & \texttt{forall}\ \overrightarrow{(x\ \texttt{in}\ X)}\ \texttt{where}\ P(\overrightarrow{x}) \\ & \texttt{exists}\ \overrightarrow{(y\ \texttt{in}\ Y)}\ \texttt{where}\ B(\overrightarrow{x}, \overrightarrow{y}) \end{aligned}$$

The functional dependency from our example is written:

$$\begin{aligned} & \texttt{forall}\ (x\ \texttt{in}\ \mathit{Movies})\ (y\ \texttt{in}\ \mathit{Movies}) \\ & \texttt{where}\ x.\texttt{title} = y.\texttt{title} \\ & \texttt{exists} \\ & \texttt{where}\ x.\texttt{director} = y.\texttt{director} \end{aligned}$$

Unlike conjunctive queries, which have a straightforward interpretation in a monad with zero, the meaning of an embedded dependency is less clear. We will give the meaning of constraint C using a pair of queries called the *front* and *back* of C. We write $\mathcal{L}(\overrightarrow{x})$ to indicate a record capturing the variables $\overrightarrow{x}$; e.g., $(\mathtt{x_1} : x_1, \ldots, \mathtt{x_N} : x_N)$. The front and back are, respectively :

$$\begin{aligned} \mathit{front}(C) \quad ::= \quad & \texttt{for}\ \overrightarrow{(x\ \texttt{in}\ X)} \\ & \texttt{where}\ P(\overrightarrow{x}) \\ & \texttt{return}\ \mathcal{L}(\overrightarrow{x}) \end{aligned}$$

$$\begin{aligned} \mathit{back}(C) \quad ::= \quad & \texttt{for}\ \overrightarrow{(x\ \texttt{in}\ X)}\ \overrightarrow{(y\ \texttt{in}\ Y)} \\ & \texttt{where}\ P(\overrightarrow{x}) \wedge B(\overrightarrow{x}, \overrightarrow{y}) \\ & \texttt{return}\ \mathcal{L}(\overrightarrow{x}) \end{aligned}$$

When these two queries are equal, the constraint holds. This definition is counter-intuitive, but does match the expected meaning of constraints in the set monad.

5.1 Reasoning about Comprehensions using Constraints

Our eventual goal is to use constraints to re-write comprehensions, using a particular technique known as *the chase*. To do so requires a few more definitions.

A *homomorphism between queries*, $h : Q_1 \to Q_2$

$$\begin{aligned} Q_1 \quad ::= \quad & \texttt{for}\ \overrightarrow{(v_1\ \texttt{in}\ V_1)}\ \texttt{where}\ P_1(\overrightarrow{v_1})\ R_1(\overrightarrow{v_1}) \\ Q_2 \quad ::= \quad & \texttt{for}\ \overrightarrow{(v_2\ \texttt{in}\ V_2)}\ \texttt{where}\ P_2(\overrightarrow{v_2})\ R_2(\overrightarrow{v_2}) \end{aligned}$$

is a substitution mapping the `for` -bound variables of Q_1 (namely, $\overrightarrow{v_1}$) to the `for` -bound variables of Q_2 (namely, $\overrightarrow{v_2}$) that preserves the structure of Q_1 in the sense that

- Each $(h(v_{1_i})\ \texttt{in}\ V_{1_i})$ appears in $\overrightarrow{(v_2\ \texttt{in}\ V_2)}$ (that is, the image of each generator in Q_1 is found in the generators of Q_2).

- $P_1(h(\overrightarrow{v_1}))$ is entailed by $P_2(\overrightarrow{v_2})$ (that is, the images of the conjuncts in Q_1 are a consequence of the conjuncts in Q_2).
- $R_1(h(\overrightarrow{v_1})) = R_2(\overrightarrow{v_2})$, under the equalities in P_2 (that is, the R clauses are equivalent).

For arbitrary predicates P_1 and P_2 and arbitrary expressions R_1 and R_2, finding homomorphisms is undecidable. However, when the queries are path-conjunctive, finding homomorphisms is NP-hard. Moreover, in this case there are practical, sound heuristics [9].

In the set monad, homomorphisms are useful because the existence of a homomorphism $A \to B$ implies that for every I, $B(I) \subseteq A(I)$. In commutative, idempotent monads (which obey some additional structure, which we define at the end of this section) the existence of a homomorphism from B to A and from A to B implies that A and B are equal. We will use this property to show how queries that are interpreted in such monads can be re-written using constraints, using a core technique from database theory called the chase. Let

$$\begin{aligned} C \quad ::= \quad & \texttt{forall}\ \overrightarrow{(x\ \texttt{in}\ X)}\ \texttt{where}\ P(\overrightarrow{x}) \\ & \texttt{exists}\ \overrightarrow{(y\ \texttt{in}\ Y)}\ \texttt{where}\ B(\overrightarrow{x}, \overrightarrow{y}) \end{aligned}$$

$$Q \quad ::= \quad \texttt{for}\ \overrightarrow{(v\ \texttt{in}\ V)}\ \texttt{where}\ O(\overrightarrow{v})\ R(\overrightarrow{v})$$

and suppose there exists a homomorphism $h : front(C) \to Q$. Then a *chase step* is to rewrite Q into $chase(Q, C)$ by adding the image of the existential part of C:

$$\begin{aligned} chase(Q, C) \quad ::= \quad & \texttt{for}\ \overrightarrow{(v\ \texttt{in}\ V)}\ \overrightarrow{(y\ \texttt{in}\ Y)} \\ & \texttt{where}\ O(\overrightarrow{v}) \wedge B(\overrightarrow{h(x)}, \overrightarrow{y}) \\ & R(\overrightarrow{v}) \end{aligned}$$

The chase itself is to repeatedly rewrite Q by looking for homomorphisms from C:

$$Q \rightsquigarrow chase(Q, C) \rightsquigarrow chase(chase(Q, C), C) \rightsquigarrow \ldots$$

The chase will converge to a unique fixed point [9], provided that 1) C is *acyclic* and 2) we do not take a chase step when there is a homomorphism extending h from $chase(Q, C)$ to Q. The definition of acyclicity is somewhat technical and we omit it.

What does the chase buy us? It provides a way to reason about queries under constraints. Consider our movies query again. To get from $MoviesBig$ to $MoviesSmall$, a process known as tableaux minimization [9], we apply the following algorithm. First, we look for homomorphisms from our constraint to $MoviesBig$. We see that there is one, and hence the chase applies. Thus, we may rewrite $MoviesBig$ into a so-called "universal plan":

$$\begin{aligned} U \quad ::= \quad & \texttt{for}\ (m_1\ \texttt{in}\ Movies)\ (m_2\ \texttt{in}\ Movies) \\ & \texttt{where}\ m_1.\texttt{title} = m_2.\texttt{title} \wedge \\ & \qquad m_1.\texttt{director} = m_2.\texttt{director} \\ & \texttt{return}\ (m_1.\texttt{director}, m_2.\texttt{actor}) \end{aligned}$$

Now, to obtain $MoviesSmall$, we remove generators from U to obtain a candidate query q and try to show that $chase(q, C)$ is equivalent to U by looking for homomorphisms in both directions. This process of query minimization is actually complete for finding minimal queries in the set monad [9]. Note that although we have been using monad comprehensions for the example in this section, all the results apply to monad algebra comprehensions [23].

5.2 Monads for which the chase is sound

The re-writing procedure outlined above is only sound for monads with zero that have additional structure, which include sets and probability distributions. The additional axioms we require are:

- **Commutativity**. We require the ability to permute generators as we please.

$$\begin{aligned} & \texttt{for}\ \overrightarrow{(u\ \texttt{in}\ U)}\overrightarrow{(v\ \texttt{in}\ V)} \\ & X(\overrightarrow{u}, \overrightarrow{v}) \\ = \quad & \\ & \texttt{for}\ \overrightarrow{(v\ \texttt{in}\ V)}\overrightarrow{(u\ \texttt{in}\ U)} \\ & X(\overrightarrow{u}, \overrightarrow{v}) \end{aligned}$$

- **Logicality**. We require that `exists` behave "as it should" with respect to `for` . Suppose $\overrightarrow{(a\ \texttt{in}\ V)} \subseteq \overrightarrow{(u\ \texttt{in}\ U)}$. Then

$$\begin{aligned} & \texttt{for}\ \overrightarrow{(u\ \texttt{in}\ U)} \\ & \texttt{where}\ P(\overrightarrow{u}) \\ & \texttt{return}\ E(\overrightarrow{u}) \\ = \quad & \\ & \texttt{for}\ \overrightarrow{(u\ \texttt{in}\ U)} \\ & \texttt{where}\ P(\overrightarrow{u}) \wedge \\ & \quad \texttt{exists}\ \overrightarrow{(v\ \texttt{in}\ V)}\ \texttt{where}\ \overrightarrow{v} = \overrightarrow{a} \\ & \texttt{return}\ E(\overrightarrow{u}) \end{aligned}$$

- **Idempotency**. We require that when $\overrightarrow{a} \notin fv(E)$,

$$\begin{aligned} & \texttt{if exists}\ \overrightarrow{(a\ \texttt{in}\ A)} \\ & \qquad \texttt{where}\ P(\overrightarrow{a}) \\ & \texttt{then return}\ E \\ & \texttt{else zero} \\ = \quad & \\ & \texttt{for}\ \overrightarrow{(a\ \texttt{in}\ A)} \\ & \texttt{where}\ P(\overrightarrow{a}) \\ & \texttt{return}\ E \end{aligned}$$

- **Distinguishability**. We require that `zero` be distinguished from `return`.

$$\texttt{return}\ x \neq \texttt{zero}$$

- **Uniformity** We require that $front(C) = back(C)$ implies $front(R, C) = back(R, C)$ for any R. That is,

$$\begin{aligned} & \texttt{for}\ \overrightarrow{(u\ \texttt{in}\ U)} \\ & \texttt{where}\ P(\overrightarrow{u}) \\ & \texttt{return}\ \mathcal{L}(\overrightarrow{u}) \\ = \quad & \\ & \texttt{for}\ \overrightarrow{(u\ \texttt{in}\ U)}\ \overrightarrow{(v\ \texttt{in}\ V)} \\ & \texttt{where}\ P(\overrightarrow{u}) \wedge B(\overrightarrow{u}, \overrightarrow{v}) \\ & \texttt{return}\ \mathcal{L}(\overrightarrow{u}) \end{aligned}$$

implies, for any X,

$$\begin{aligned} & \texttt{for}\ \overrightarrow{(u\ \texttt{in}\ U)} \\ & \texttt{where}\ P(\overrightarrow{u}) \\ & X(\overrightarrow{u}) \\ = \quad & \\ & \texttt{for}\ \overrightarrow{(u\ \texttt{in}\ U)}\ \overrightarrow{(v\ \texttt{in}\ V)} \\ & \texttt{where}\ P(\overrightarrow{u}) \wedge B(\overrightarrow{u}, \overrightarrow{v}) \\ & X(\overrightarrow{u}) \end{aligned}$$

6. Proofs

In the preceding sections we have tried to indicate the places where a particular property is required to hold but in general there is no way for a compiler to prove it. Verification is needed in the following places:

- At monad, monad algebra, commutative idempotent monad, and parameterized monad definitions, to verify that particular laws hold.
- At equivalence relation definitions, to verify that the provided definition is in fact an equivalence relation.
- At each use of `fold` or `build`, to verify that the operations respect the underlying equivalence relation.

In addition, users can write `assert` and `assume` statements to trigger the explicit generation of an obligation or the addition of a constraint known to hold a priori. This mechanism can only be used with the constraints in the previous section; its purpose is to alert the compiler that additional information is available with which to re-write comprehensions. Here is an example, where we assume a primitive for reading a file as a set, and assuming Haskell-ish monadic IO:

```
main : IO ()
main = do x <- readSet foo
          assume C(x) in
           do print Q(x)
              assert C'(Q(x)) in
              ....
```

Here C and C' are constraints, and Q a query. Generation of obligations stemming from assertions or uses of `fold` and `build` is done by traversing the syntax, placing assumptions in a context. Free variables are universally quantified in the emitted Coq code. This means that our obligations are flow insensitive, and hence may not be provable even though they are "true". For greater precision, we have considered a shallow embedding of our language into Coq, but leave that for future work.

A particular challenge here is coping with extensible, equated up-to-permutation records in Coq. That is, we are translating from a system of qualified types with such records to Coq, which is dependently-typed but lacks such records. During translation, the qualifiers in our language become proof objects guaranteeing, for example, that a projection operation cannot fail. The construction of these objects, we believe, can be performed as part of type-inference, based on the idea of a "principal evidence-passing translation" [12].

7. Conclusion and Future Work

We have proposed an intermediate form based on monad-algebra comprehensions (to represent queries), setoids over polynomial datatypes (to represent data), and folds (to represent computation) suitable for use in collection processing. Such an intermediate form captures, in a uniform way, large fragments of many recent large-scale collection processing languages such as MapReduce, PIG, DryadLINQ, and Data Parallel Haskell. Although we are not the first to propose such an intermediate form, we show how to solve four key problems inherent in the naive approach by applying recent work from both programming language theory and relational database theory. First, we show how fold fusion can be extended to exploit the monadic structure of queries. Second, we show how comprehensions themselves can be extended to allow for aggregation. Third, we show how to embed constraints into our intermediate form and how such constraints can be used, for example, to minimize the number of bind operations in a monad comprehension. Finally, we show how to emit proof obligations from our language, so as to ensure that each program is sound with respect to required properties. Taken together, these improvements pave the way for a compiler based on this intermediate form, which we are currently developing.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] R. Bird and O. de Moor. *Algebra of programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[3] G. E. Blelloch and J. Greiner. A provable time and space efficient implementation of nesl. In *ICFP*, pages 213–225, 1996.

[4] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Rec.*, 23(1):87–96, 1994.

[5] P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theor. Comput. Sci.*, 149(1):3–48, 1995.

[6] P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149:3–48, 1995.

[7] A. Bunkenburg. The boom hierarchy. In *Proc. of the Workshop on Functional Programming, Workshops in Computing*, 1993.

[8] M. M. T. Chakravarty, R. Leshchinskiy, S. P. Jones, G. Keller, and S. Marlow. Data parallel haskell: a status report. In *DAMP 2007*.

[9] A. Deutsch, L. Popa, and V. Tannen. Query reformulation with constraints. *SIGMOD Rec.*, 35:65–73, March 2006.

[10] E. A. et al. The Fortress Language Specification. Technical report, Sun Microsystems, Inc., 2007.

[11] L. Fegaras and D. Maier. Optimizing object queries using an effective calculus. *ACM Trans. Database Syst.*, 25(4):457–516, 2000.

[12] B. R. Gaster and M. P. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, Department of CS, University of Nottingham, November 1996.

[13] N. Ghani and P. Johann. Monadic augment and generalised short cut fusion. *J. Funct. Program.*, 17:731–776, November 2007.

[14] A. J. Gill. Cheap deforestation for non-strict functional languages, 1996.

[15] T. Grust. *Comprehending Queries*. Universitat Konstanz, Ph.D. Thesis, 1999.

[16] T. Grust. *Monad Comprehensions. A Versatile Representation for Queries. In The Functional Approach to Data Management, P.M.D. Gray and L. Kerschberg and P.J.H. King and A. Poulovassilis (eds.).* Springer Verlag, 2003.

[17] L. M. Haas, M. A. Hernndez, L. Popa, M. Roth, and H. Ho. Clio grows up: From research prototype to industrial tool. In *SIGMOD 05*.

[18] Q. heng, J. Gryz, F. Koo, T. Y. C. Leung, L. Liu, X. Qian, and K. B. Schiefer. Implementation of two semantic query optimization techniques in db2 universal database. VLDB '99, 1999.

[19] G. Hutton. A tutorial on the universality and expressiveness of fold. *J. Funct. Program.*, 9(4):355–372, 1999.

[20] M. Isard and Y. Yu. Distributed data-parallel computing using a high-level programming language. In *SIGMOD '09*.

[21] S. K. Lellahi and V. Tannen. A calculus for collections and aggregates. In *CTCS '97*, pages 261–280, London, UK, 1997. Springer-Verlag.

[22] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD 08*.

[23] L. Popa and V. Tannen. An equational chase for path-conjunctive queries, constraints, and views. In *ICDT 99*.

[24] J. T. Schwartz, R. B. Dewar, E. Schonberg, and E. Dubinsky. *Programming with sets; an introduction to SETL*. Springer-Verlag New York, Inc., New York, NY, USA, 1986.

[25] M. Sozeau. A New Look at Generalized Rewriting in Type Theory. *Journal of Formalized Reasoning*, 2(1):41–62, December 2009.

[26] V. Tannen, P. Buneman, and L. Wong. Naturally embedded query languages. ICDT '92, pages 140–154, London, UK, 1992. Springer-Verlag.

[27] P. Wadler. Theorems for free! In *Proceedings 4th Int. Conf. on Funct. Prog. Languages and Computer Arch., FPCA'89, London, UK, 11–13 Sept 1989*, pages 347–359. ACM Press, New York, 1989.

[28] L. Wong. *Querying nested collections*. PhD thesis, Philadelphia, PA, USA, 1994. Supervisor-Buneman, Peter.

[29] L. Wong. Kleisli, a functional query system. *J. Funct. Prog*, 10, 1998.

[30] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD '07*, pages 1029–1040, 2007.