Minimizing Monad Comprehensions

Ryan Wisnesky

TR-02-11

Computer Science Group
Harvard University
Cambridge, Massachusetts

Minimizing Monad Comprehensions

Ryan Wisnesky
Harvard University
ryan@cs.harvard.edu

March 9, 2011

Abstract

Monad comprehensions are by now a mainstay of functional programming languages. In this paper we develop a theory of *semantic optimization* for monad comprehensions that goes beyond rewriting using the monad laws. A monad-with-zero comprehension `do` $x \leftarrow X; y \leftarrow Y;$ `if` $P(x, y)$ `then return` $F(x, y)$ `else zero` can be rewritten, so as to minimize the number of $\leftarrow$ bindings, using constraints that are known to hold of X and Y. The soundness of this technique varies from monad to monad, and we characterize its soundness for monads expressible in functional programming languages by generalizing classical results from relational database theory. This technique allows the optimization of a wide class of languages, ranging from large-scale data-parallel languages such as DryadLINQ and Data Parallel Haskell to probabilistic languages such as IBAL and functional-logical languages like Curry.

1 Introduction

Languages and systems such as MapReduceMerge [39], Ferry [21], Data Parallel Haskell [9], DyadLINQ [27], PIG [34], Fortress [3] and SciDB [11] are proliferating as Moore's law drives the cost of computing ever lower and the size of data ever larger. Like their predecessors SQL, NESL [7], and Kleisli [38], these declarative, collection-oriented languages and systems lift programming to the level of abstract collections such as sets, bags, lists, and trees. And as the database community discovered long ago, the sheer size of the data processed by these systems demands sophisticated optimization [30]. Simply choosing the right order to iterate over several collections can mean the difference between a query than halts in a few seconds vs a few days. At this scale, the particular properties of the data at hand become important [17].

Although these 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) and *comprehensions* (to model queries) [20, 8]. Although monads have seen great success in providing structure to functional programs [37], sophisticated reasoning about monads using a priori semantic information has traditionally belonged to the realm of database theory. For example, in relational query processing, data integrity constraints capture such semantic information as keys, functional dependencies, inclusions, and join decompositions. These constraints are used as additional rewrite rules during optimization, a process known as *semantic optimization* [1, 36, 13].

For example [1], consider a query over a relation (set of records) $Movies$ with fields title, $\mathsf{director}$, and actor:

$$\begin{array}{l}\texttt{for}\ (m_1\ \texttt{in}\ Movies)\ (m_2\ \texttt{in}\ Movies)\\ \texttt{where}\ m_1.\mathsf{title} = m_2.\mathsf{title}\\ \texttt{return}\ (m_1.\mathsf{director}, m_2.\mathsf{actor})\end{array}$$

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 $\mathsf{title} \rightarrow \mathsf{director}$ (meaning that if $(\mathsf{director} : d, \mathsf{title} : t, \mathsf{actor} : a)$ and $(\mathsf{director} : d', \mathsf{title} : t', \mathsf{actor} : a')$ are records in

$Movies$ such that $t = t'$, then $d = d'$), this query is equivalent to

$$\texttt{for}\ (m\ \texttt{in}\ Movies)$$
$$\texttt{return}\ (m.\mathsf{director}, m.\mathsf{actor})$$

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 Midas [5] and Clio [22] automatically generate large amounts of code. The significant, potentially order-of-magnitude speed-ups enabled by semantic optimization are well-documented in the literature and applied in commercial databases such as DB2 [24]. Our goal in this paper is to introduce semantic optimization to programming languages more generally.

Our `for` $-$ `where` $-$ `return` notation is defined in terms of an arbitrary monad, and the soundness of semantic optimization varies from monad to monad. For example, this equivalence does not hold for lists or bags. Nevertheless, we see semantic optimization as useful not only for large-scale collection processing, but for other computations which can be modeled, at least in part, using monad comprehensions, such as functional-logic programming in Curry [4] and Daedalus [23], as well as probabilistic programming in Haskell [15] and IBAL [35].

Contributions

The key to semantic optimization lies in a classical rewriting system from relational database theory, known as *the chase* [12, 2]. The primary contribution of this paper is to define, axiomatically, the monads for which the chase is sound. We are not the first to do so [36], but we are the first to do so in a way accessible to a functional programming audience. A consequence of our functional programming perspective is a catalog of additional monads for which the chase is sound – the probability monad [15] being chief among them. It is likely that entirely new optimization strategies for probabilistic languages can be developed based on the soundness of the chase.

Related Work

Semantic optimization (which conditionally preserves semantics, subject to constraints) complements non-semantic optimization (which always preserves semantics). Relational algebra has a well-developed theory of non-semantic optimization by minimizing detailed cost models [17], and cost models for monad comprehensions have also been developed [28]. Inductive datatypes (and to a lesser extent, function types [32]) and monads as found in functional programming have a well-developed theory of non-semantic optimization by fold-fusion and deforestation [31, 19, 20, 33, 6, 26]. More recently, practical advances in theorem proving have sparked renewed interest in the duality between program verification and semantic optimization [25].

Outline

We begin by giving an overview of monads from a functional programming perspective in Section 2. We then use monad comprehensions to define queries in Section 3 and constraints in Section 4. In Section 5 we describe the chase, and give sufficient conditions for it to be terminating and sound. We then use the chase to optimize queries in Section 6. In these sections, our running example is intuitively interpreted in the set monad, but in Section 7 we re-examine our example in the probability monad.

2 Monads

In functional programming, a monad consists of a type-constructor M and two operations, `return` $: t \to M\ t$ and `bind` $: M\ t \to (t \to M\ t') \to M\ t'$, such that the following three laws hold:

$$\begin{aligned} \texttt{bind}\ (\texttt{return}\ \ x)\ f \;&=\; fx \\ \texttt{bind}\ m\ \texttt{return} \;&=\; m \\ \texttt{bind}\ (\texttt{bind}\ m\ f)\ g \;&=\; \texttt{bind}\ m\ (\lambda x.\ \texttt{bind}\ (fx)\ g) \end{aligned}$$

A monad with zero has another operation, `zero` $: M\ t$, such that two additional laws hold:

$$\begin{aligned} \texttt{bind}\ \texttt{zero}\ f \;&=\; \texttt{zero} \\ \texttt{bind}\ m\ (\lambda x.\ \texttt{zero}) \;&=\; \texttt{zero} \end{aligned}$$

2.1 Examples

Monads with zeros are often used to model collections. For example, consider lists and sets in Haskell, in so-called "insert presentation"

```
data Ins a = Nil | Cons a (Ins a)

-- list monad
instance MonadZero Ins where
 return x = Cons x Nil
 bind x f = append (map f x)
 zero = Nil

-- set monad
instance MonadZero Ins where
 return x = Cons x Nil
 bind x f = union (map f x)
 zero = Nil
```

Monads are not tied to particular representations. For example, the list and set monad can also be defined using so-called "union presentation":

```
data Un = Empty | Singleton a | Union (Un a) (Un a)
```

Not all collections have zeroes – for example, binary trees do not have a zero. Monads can also be formed from functions; here, state with exceptions forms a monad with zero:

```
type ST s a = (s -> Maybe (a, s))

instance MonadZero (ST Int) where
 return s a = Just (s, a)
 bind c f s = case c s of
                 Nothing -> Nothing
                 Just (s', a') -> f a' s'
 zero = \s -> Nothing
```

Monads are by now an important subject in their own right. We refer the reader to [20, 37] for more details.

2.2 Notation

Monads are often used with so-called do-notation, which in Haskell looks like:

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

Haskell programmers typically first encounter do-notation with Haskell's IO monad, as in the following program which outputs "Hello World":

```
main = do putStr "Hello"
          putStrLn "World"
```

Also popular is *monad comprehension notation*, which works for monads with zero, such as lists and sets:

```
[c | x <- X, P]
=
do x <- X
   if P then return c else zero
```

For example,

```
[x | x <- 1..10 , isEven x] = [2, 4, 6, 8, 10]
```

To emphasize the connection with database theory, we will use $\texttt{for} - \texttt{where} - \texttt{return}$ notation, which we define in the next section. Regardless of the choice of notation, monad comprehensions can be normalized using the monad laws, as described in [20]. An interesting direction for future work would be to use a weaker structure, such as applicative functors, in place of monads in our theory.

3 Queries

We will be focusing on comprehensions which are syntactically *conjunctive queries*. We will assume we are working in a strongly-normalizing typed λ-calculus with first-class records, such as [18]. We will write $(l_1 : e_1, \ldots, l_N : e_N)$ to indicate a record with labels $l_1, \ldots l_N$ and corresponding projections $e_1, \ldots, e_N$. We will assume records contain unique labels and are equated up-to label permutation. For the most part, the specifics of our ambient language will not matter. We will abbreviate (potentially 0-length) 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 $P(\overrightarrow{x})$to indicate a conjunction of predicates over the variables $\overrightarrow{x}$. A *tableau* (plural: tableaux) has the form:

Definition 1 (Tableau)

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

The $\overrightarrow{(x\ \texttt{in}\ X)}$ are called *generators*, and the $\overrightarrow{X}$ are called *roots*. A query is a tableau and an expression $R(\overrightarrow{x})$:

Definition 2 (Query)

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

A query is interpreted as a monad comprehension:

Definition 3 (Query Semantics)

```
do    $x_1 \leftarrow X_1$
      $\ldots$
      $x_N \leftarrow X_N$
      if $P(x_1, \ldots, x_N)$
      then return $R(x_1, \ldots, x_N)$
      else zero
```

For example, the query from the introduction:

```
for ($m_1$ in $Movies$) ($m_2$ in $Movies$)
where $m_1$.title = $m_2$.title
return ($m_1$.director, $m_2$.actor)
```

is interpreted as:

```
do    $m_1 \leftarrow Movies$
      $m_2 \leftarrow Movies$
      if $m_1$.title = $m_2$.title
      then return ($m_1$.director, $m_2$.actor)
      else zero
```

which de-sugars into

```
bind $Movies$ ($\lambda m_1$.
 bind $Movies$ ($\lambda m_2$.
  if $m_1$.title = $m_2$.title
  then return ($m_1$.director, $m_2$.actor)
  else zero ))
```

and, in the set-monad, this becomes

```
union (map $Movies$ ($\lambda m_1$.
 (union (map $Movies$ ($\lambda m_2$.
  if $m_1$.title = $m_2$.title
  then Cons ($m_1$.director, $m_2$.actor) Nil
  else Nil )))))
```

A query can also naturally be interpreted as a function over its roots (here, $Movies$). In this case, to evaluate a query we require values for the roots (here, we require a particular relation $Movies$). We will write $q(I)$ to indicate a query q evaluated at I. The I is usually called an *instance*. Our example query can thus also be regarded as the function:

```
$\lambda Movies$.  do    $m_1 \leftarrow Movies$
                         $m_2 \leftarrow Movies$
                         if $m_1$.title = $m_2$.title
                         then return ($m_1$.director, $m_2$.actor)
                         else zero
```

Extensions

Many extensions to conjunctive queries have been studied in the literature. Two stand out as particularly important:

- It is possible to allow generators to be dependent; for example:

  $$\texttt{for}\ (g\ \texttt{in}\ \mathit{Groups})\ (\mathit{person}\ \texttt{in}\ g)\ \ldots$$

  This allows for nested values; for example, nested relations [36].

- It is possible to interpret queries in *monad algebras*, rather than monads [30]. A monad algebra is an operation of type $M\ t \to (t \to t') \to t'$ obeying certain equations. This more general type (relative to `bind`) allows for aggregation operations; for example, it is possible to write a query to count the number of elements in a list, which is impossible in our system.

We will ignore these extensions for now, but it is likely that our results will hold in these more general settings.

4 Constraints

We are interested in reasoning about monad comprehensions under constraints, so as to exploit the constraints found in, for example, large-scale collection processing and functional logic programming. In this section we formally define the class of constraints we are using; in the next section we show how queries can be rewritten by them (a process referred to as *the chase*); in the section after that, we show how the chase can be used to optimize queries.

4.1 Embedded Dependencies

Our constraints take the form of pairs of tableaux, which define so-called *embedded, implicational dependencies* [1]. Intuitively, one tableaux is universally quantified, and the other existentially:

Definition 4 (Embedded Dependency)

$$\begin{array}{l}\texttt{forall}\ \overrightarrow{(x\ \texttt{in}\ X)}\\ \texttt{where}\ P(\overrightarrow{x})\\ \texttt{exists}\ \overrightarrow{(y\ \texttt{in}\ Y)}\\ \texttt{where}\ B(\overrightarrow{x}, \overrightarrow{y})\end{array}$$

The functional dependency from our example is written:

$$\begin{array}{l}\texttt{forall}\ (x\ \texttt{in}\ \mathit{Movies})\ (y\ \texttt{in}\ \mathit{Movies})\\ \texttt{where}\ x.\mathsf{title} = y.\mathsf{title},\\ \texttt{exists}\\ \texttt{where}\ x.\mathsf{director} = y.\mathsf{director}\end{array}$$

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 a 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., $(\mathsf{x_1} : x_1, \ldots, \mathsf{x_N} : x_N)$.

Definition 5 (Front)

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

Definition 6 (Back)

$$\begin{array}{l}\texttt{for } \overrightarrow{(x \texttt{ in } X)}\ \overrightarrow{(y \texttt{ in } Y)} \\ \texttt{where } P(\overrightarrow{x}) \wedge B(\overrightarrow{x}, \overrightarrow{y}) \\ \texttt{return } \mathcal{L}(\overrightarrow{x})\end{array}$$

We will write $I \models C$ to indicate that constraint C holds of instance I.

Definition 7 (Satisfaction) $I \models C$ *when*

$$front(C)(I) = back(C)(I)$$

We will write $front(R, C)$ and $back(R, C)$ to indicate the queries $front(C)$ and $back(C)$ but whose `return` clauses are R. Continuing with our example, our functional dependency holds of a particular instance $Movies$ when

$$\begin{array}{ll} & \texttt{for } (x \texttt{ in } Movies)\ (y \texttt{ in } Movies) \\ & \texttt{where } x.\mathsf{title} = y.\mathsf{title}, \\ & \texttt{return } (\mathsf{x} : x, \mathsf{y} : y) \\ = & \\ & \texttt{for } (x \texttt{ in } Movies)\ (y \texttt{ in } Movies) \\ & \texttt{where } x.\mathsf{title} = y.\mathsf{title} \wedge x.\mathsf{director} = y.\mathsf{director} \\ & \texttt{return } (\mathsf{x} : x, \mathsf{y} : y)\end{array}$$

For example, in this instance:

title	director	actor
T	D	A
T	D	B

the constraint holds because both sides evaluate to (omitting some record labels to save space):

x	y
(T, D, A)	(T, D, A)
(T, D, A)	(T, D, B)
(T, D, B)	(T, D, A)
(T, D, B)	(T, D, B)

whereas in this instance:

title	director	actor
T	D_1	A
T	D_2	B

the constraint does not hold because the left-hand side evaluates to:

x	y
(T, D_1, A)	(T, D_1, A)
(T, D_1, A)	(T, D_2, B)
(T, D_2, B)	(T, D_1, A)
(T, D_2, B)	(T, D_2, B)

which is not equivalent to the right-hand side:

x	y
(T, D_1, A)	(T, D_1, A)
(T, D_2, B)	(T, D_2, B)

5 The Chase

The chase is a confluent rewriting system that allows queries to be rewritten using constraints [1]. In this section we define the chase and characterize when it is sound and when it terminates. In the next section we show how to use it to optimize queries.

5.1 Homomorphisms

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

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

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

Definition 8 (Homomorphism) .

- *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* `return` *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* – that is, when P_1, P_2 are conjunctions of equalities between paths of the form $v.l$ and R_1 and R_2 are records built from paths – finding homomorphisms is NP-hard. Moreover, in this case there are practical, sound heuristics [13] based on pruning the search space of substitutions to remove candidates that are "obviously wrong" based on a partial variable assignment. In this paper, all our examples are path conjunctive.

For example, consider our $Movies$ query (call it Q_1):

$$\begin{array}{rcl} Q_1 & ::= & \texttt{for } (m_1 \texttt{ in } Movies)\ (m_2 \texttt{ in } Movies) \\ & & \texttt{where } m_1.\textsf{title} = m_2.\textsf{title} \\ & & \texttt{return } (m_1.\textsf{director}, m_2.\textsf{actor}) \end{array}$$

and also the smaller query (call it Q_2) which we will later optimize Q_1 into:

$$\begin{array}{rcl} Q_2 & ::= & \texttt{for } (m \texttt{ in } Movies) \\ & & \texttt{return } (m.\textsf{director}, m.\textsf{actor}) \end{array}$$

There is a homomorphism $h : Q_1 \to Q_2$; namely, the substitution $m_1 \mapsto m, m_2 \mapsto m$. To check this, we first apply h to Q_1:

$$\begin{array}{rcl} h(Q_1) & ::= & \texttt{for } (m \texttt{ in } Movies)\ (m \texttt{ in } Movies) \\ & & \texttt{where } m.\textsf{title} = m.\textsf{title} \\ & & \texttt{return } (m.\textsf{director}, m.\textsf{actor}) \end{array}$$

In $h(Q_1)$ each generator (m `in` *Movies*) appears in Q_2. Moreover, the `where` clause of $h(Q_1)$ is a tautology and hence is entailed by the (empty) `where` clause of Q_2. Finally, the two `return` clauses are equal. As such, the substitution $m_1 \mapsto m, m_2 \mapsto m$ is a homomorphism.

In the set monad, homomorphisms are useful because the existence of a homomorphism $A \to B$ implies that for every I, $B(I) \subseteq A(I)$. Indeed, it is easy to see in this example that $Q_2(I) \subseteq Q_1(I)$ for any I. Later we will make use of a similar property for arbitrary monads to show that the chase is sound.

At this point it is instructive to check that there is no homomorphism $Q_2 \to Q_1$. There are only two candidate substitutions: $m \mapsto m_1$ and $m \mapsto m_2$. Neither of these works because neither of the images of Q_2's `return` clause (either `return` (m_1.director, m_1.actor) or `return` (m_2.director, m_2.actor)) is equivalent to Q_1's `return` clause (`return` (m_1.director, m_2.actor)), even under the equality in Q_1 (m_1.title $= m_2$.title). Because there are not homomorphisms in both directions, these two queries are not equivalent. Indeed, consider the instance:

title	director	actor
T	D_1	A
T	D_2	B

Q_1 evaluates to

director	actor
D_1	A
D_1	B
D_2	A
D_2	B

but Q_2 evaluates to:

director	actor
D_1	A
D_2	B

5.2 The Chase

Now we can define *the chase*. Let

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

$$\begin{array}{lll} Q & ::= & \texttt{for}\ \overrightarrow{(v\ \texttt{in}\ V)} \\ & & \texttt{where}\ O(\overrightarrow{v}) \\ & & \texttt{return}\ R(\overrightarrow{v}) \end{array}$$

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

Definition 9 (Chase Step)

$$\begin{array}{lll} \mathit{chase}(Q, C) & ::= & \texttt{for}\ \overrightarrow{(v\ \texttt{in}\ V)}\ \overrightarrow{(y\ \texttt{in}\ Y)} \\ & & \texttt{where}\ O(\overrightarrow{v}) \land B(\overrightarrow{h(x)}, \overrightarrow{y}) \\ & & \texttt{return}\ R(\overrightarrow{v}) \end{array}$$

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

$$Q \rightsquigarrow \mathit{chase}(Q, C) \rightsquigarrow \mathit{chase}(\mathit{chase}(Q, C), C) \rightsquigarrow \ldots$$

The chase will converge to a unique fixed point [13], provided that 1) C is *acyclic* and 2) we do not take a chase step when there is a homomorphism extending h from $\mathit{chase}(Q, C)$ to Q. The definition of acyclicity is somewhat technical, so we defer it until the end of the section. The chase extends straightforwardly to sets of constraints.

Continuing with our $Movies$ example, we can see that there is a homomorphism $x \mapsto m_1, y \mapsto m_2$ from the front of our constraint:

$$\begin{array}{l}\texttt{forall}\ (x\ \texttt{in}\ Movies)\ (y\ \texttt{in}\ Movies)\\ \texttt{where}\ x.\mathsf{title} = y.\mathsf{title},\\ \texttt{exists}\\ \texttt{where}\ x.\mathsf{director} = y.\mathsf{director}\end{array}$$

to our original query:

$$\begin{array}{l}\texttt{for}\ (m_1\ \texttt{in}\ Movies)\ (m_2\ \texttt{in}\ Movies)\\ \texttt{where}\ m_1.\mathsf{title} = m_2.\mathsf{title}\\ \texttt{return}\ (m_1.\mathsf{director}, m_2.\mathsf{actor})\end{array}$$

Hence, the chase applies, and $\mathit{chase}(Q, C)$ is:

$$\begin{array}{l}\texttt{for}\ (m_1\ \texttt{in}\ Movies)\ (m_2\ \texttt{in}\ Movies)\\ \texttt{where}\ m_1.\mathsf{title} = m_2.\mathsf{title} \wedge m_1.\mathsf{director} = m_2.\mathsf{director}\\ \texttt{return}\ (m_1.\mathsf{director}, m_2.\mathsf{actor})\end{array}$$

At this point, we stop taking chase steps, because we have that $\mathit{chase}(\mathit{chase}(Q, C), C) = \mathit{chase}(Q, C)$ and hence there is a homomorphism $\mathit{chase}(\mathit{chase}(Q, C), C) \rightarrow \mathit{chase}(Q, C)$. Note that in general, it is not enough to check for the syntactic equality of $\mathit{chase}(Q, C)$ and Q to stop the chase, as queries can be equivalent without being syntactically equal. Hence, we must use homomorphisms to detect that the chase has converged.

5.3 Soundness

It is well-known that the chase is not sound for arbitrary monads, and it is not sound for the list and bag monads in particular [36]. Intuitively, the chase adds generators to a query, and adding generators to list and bag comprehensions can add additional tuples to the result; in the set monad, these extra tuples disappear by idempotency. For example, in the list monad, our functional dependency $\mathsf{title} \rightarrow \mathsf{director}$ still holds on this instance:

title	director	actor
T	D	A
T	D	B

but our original and optimized queries are not equivalent; they result in, respectively,

D	A
D	B
D	A
D	B

and

D	A
D	B

Theorem 1 *The chase is sound under the following conditions:*

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

```
    for $\overrightarrow{(u \texttt{ in } U)}\overrightarrow{(v \texttt{ in } V)}$
    $X(\overrightarrow{u}, \overrightarrow{v})$
=
    for $\overrightarrow{(v \texttt{ in } V)}\overrightarrow{(u \texttt{ in } U)}$
    $X(\overrightarrow{u}, \overrightarrow{v})$
```

- **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

```
    for $\overrightarrow{(u \texttt{ in } U)}$
    where $P(\overrightarrow{u})$
    return $E(\overrightarrow{u})$
=
    for $\overrightarrow{(u \texttt{ in } U)}$
    where $P(\overrightarrow{u}) \wedge$
       exists $\overrightarrow{(v \texttt{ in } V)}$ where $\overrightarrow{v} = \overrightarrow{a}$
    return $E(\overrightarrow{u})$
```

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

```
    if exists $\overrightarrow{(a \texttt{ in } A)}$
           where $P(\overrightarrow{a})$
    then return $E$
    else zero
=
    for $\overrightarrow{(a \texttt{ in } A)}$
    where $P(\overrightarrow{a})$
    return $E$
```

- **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,

```
    for $\overrightarrow{(u \texttt{ in } U)}$
    where $P(\overrightarrow{u})$
    return $\mathcal{L}(\overrightarrow{u})$
=
    for $\overrightarrow{(u \texttt{ in } U)}$ $\overrightarrow{(v \texttt{ in } V)}$
    where $P(\overrightarrow{u}) \wedge B(\overrightarrow{u}, \overrightarrow{v})$
    return $\mathcal{L}(\overrightarrow{u})$
```

implies, for any X,

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

$$=$$

$$\begin{array}{l}
\texttt{for } \overrightarrow{(u \texttt{ in } U)}\ \overrightarrow{(v \texttt{ in } V)} \\
\texttt{where } P(\overrightarrow{u}) \wedge B(\overrightarrow{u}, \overrightarrow{v}) \\
X(\overrightarrow{u})
\end{array}$$

We will call monads with such properties *UCLID* (**U**niform, **C**ommutative, **L**ogical, **I**dempotent, **D**istinguishable) monads.

The exact phrasing of the idempotency property varies in the literature, and there are multiple non-equivalent definitions [29]. Our particular phrasing is motivated by the law's use in our soundness proof. One consequence of our axioms is the following more intuitive idempotency law (which is equivalent to the definition of idempotency in [14]:

$$\texttt{for } \overrightarrow{(x \texttt{ in } X)} \texttt{ return } E(\overrightarrow{x}) =$$
$$\texttt{for } \overrightarrow{(x \texttt{ in } X)}\ \overrightarrow{(y \texttt{ in } X)} \texttt{ return } E(\overrightarrow{x})$$

This simpler definition makes it easy to rule out certain monads as being idempotent. For example, lists, bags, trees, state and IO are all non-idempotent.

5.4 UCLID Monads

Our study has so far yielded four UCLID monads: the option monad; the finite map monad (of type $t \to \mathit{option}\ a$, for some fixed t), which is also called the partial reader monad; the set monad; and probability monad. We will defer a discussion of the probability monad until Section 7. In this section, we will discuss the expressive power of sets and embedded dependencies.

The chase is well-understood in the set monad. In fact, the chase is complete for nested sets and finite maps [36], at least for path-conjunctive queries. Proving completeness requires constructing a canonical model set for a given query; because this construction depends on the particular monad in question, generalizing completeness results to other monads seems difficult. Nevertheless, this is an interesting direction for future work. The chase in the set monad can also be used for many purposes besides the optimization techniques described in this paper. For example, it is possible to rewrite queries so as to respect constraints and rewrite constraints by other constraints. Particular results vary slightly depending on whether infinite sets are allowed, but the set monad can be described in many presentations, ranging from insert and union presentations (as in Section 2), to a functional presentation $a \to \mathit{bool}$, or even a B-tree presentation.

Embedded dependencies can express a wide variety of constraints between sets (including virtually all constraints used in modern relational database systems, such as keys, foreign keys, inclusions, and join decompositions). Sets, when coupled with embedded dependencies, can model a wide variety of collections. For example, it is possible to model a function or finite map using a set C and a functional dependency:

$$\begin{array}{l}
\texttt{forall } (x \texttt{ in } X)\ (y \texttt{ in } X) \\
\texttt{where } x.\mathsf{key} = y.\mathsf{key} \\
\texttt{exists} \\
\texttt{where } x.\mathsf{value} = y.\mathsf{value}
\end{array}$$

More refined notions such as injective functions can also be modeled this way. Transitive closure can also be expressed:

$$\begin{array}{l}\texttt{forall}\ (x\ \texttt{in}\ X)\ (y\ \texttt{in}\ X)\\ \texttt{where}\ x.\mathsf{b} = y.\mathsf{a}\\ \texttt{exists}\ \overrightarrow{(z\ \texttt{in}\ X)}\\ \texttt{where}\ z.\mathsf{a} = x.\mathsf{a} \wedge z.\mathsf{b} = y.\mathsf{b}\end{array}$$

In short, sets are ubiquitous, versatile, and expressive, and the ability to reason about them under constraints enables many other optimization and verification techniques.

Sets are by far the most well-understood of the UCLID monads, at least with respect to the chase. Since our starting point is relational database theory, we are somewhat biased towards sets and set-based examples. Nevertheless, our study is just beginning and we are hopeful that generalizing from monads to monad algebras will allow us to find UCLID monad algebras.

5.5 Acyclicity

For arbitrary queries and embedded dependencies, the chase may not terminate, and termination is undecidable. However, for *weakly acyclic* constraints, a condition which can be checked in polynomial time and is common in practice, rewriting does terminate, and does so in polynomial time. In this section we describe the weakest condition yet known that guarantees termination [13]. This section is mostly of interest to implementors and can be skipped on a first reading.

To define weak-acyclicity it is necessary to use so-called *domain-relational* notation instead of our so-called *tuple-relational* notation. The difference in the two notations is what we are quantifying over: tuples, or atomic values. For example, our original query, in tuple-relational style, is:

$$\begin{array}{l}\texttt{forall}\ (x\ \texttt{in}\ \mathit{Movies})\ (y\ \texttt{in}\ \mathit{Movies})\\ \texttt{where}\ x.\mathsf{title} = y.\mathsf{title},\\ \texttt{exists}\\ \texttt{where}\ x.\mathsf{director} = y.\mathsf{director}\end{array}$$

In domain-relational style, assuming title, director, and actor are the only attributes in *Movies*, this becomes:

$$\mathit{Movies}(t, d, a) \wedge \mathit{Movies}(t', d', a') \wedge t = t' \to d = d'$$

One of the disadvantages of the domain-relational style is that queries become monomorphic: every attribute must have a variable. Another downside is that we are restricted to path-conjunctive queries: our `where` clauses must be conjunctions of equalities between paths of the form $v.l$, and our return clauses must be records over paths. For this reason, we prefer tuple-relational style, which is more common in functional programming anyway. Otherwise, the two notations are generally equivalent. Using the domain-relational notation allows us to divide embedded dependencies into two classes: *equality-generating dependencies (egds)*, such as above, of the form

$$\phi(\overrightarrow{x}) \to x_1 = x_2$$

and *tuple-generating dependencies (tgds)*, of the form:

$$\phi(\overrightarrow{x}) \to \exists \overrightarrow{y} \psi(\overrightarrow{x}, \overrightarrow{y})$$

An example tgd, in tuple-relational style, is:

$$\begin{array}{l}\texttt{forall}\ (e\ \texttt{in}\ \mathit{Employees})\\ \texttt{where}\\ \texttt{exists}\ (p\ \texttt{in}\ \mathit{People})\\ \texttt{where}\ e.\mathsf{name} = p.\mathsf{name}\end{array}$$

which becomes, in domain-relational style (assuming name is the only attribute):

$$Employees(e) \rightarrow \exists p \; People(p) \wedge e = p$$

Now we can define:

Definition 10 (Weakly Acyclic Constraints) *Let C be a set of embedded dependencies. Construct a directed graph, called the dependency graph, as follows: (1) there is a node for every pair (R, A) with R a top-level generator (such as Movies) and A an attribute of R; call such a pair a position; (2) add edges as follows: for every dependency $\phi(\overrightarrow{x}) \rightarrow \exists \overrightarrow{y} \psi(\overrightarrow{x}, \overrightarrow{y})$ and for every x that occurs in ψ:*

- *For every occurence of x in ϕ in position (R, A_i):*
  1. *for every occurence of x in ψ in position (S, B_j), add an edge $(R, A_i) \rightarrow (S, B_j)$.*
  2. *in addition, for every existentially quantified variable y and for every occurence of y in ψ in position (T, C_k), add a special edge $(R, A_i) \Rightarrow (T, C_k)$.*

Note that there may be two edges in the same direction between two nodes, if exactly one of the two edges is special. Then the constraints are weakly acyclic if the dependency graph has no cycle going through a special edge. A set of tgds and egds is weakly acyclic if the set of all its tgds is weakly acyclic.

Our movies example is trivially acyclic because functional dependencies are equality generating. But the dependency graph for the tgd is:

$$(Employee, Name) \overset{\rightarrow}{\Rightarrow} (Person, Name)$$

which is acyclic.

6 Tableaux Minimization

We now demonstrate how to minimize queries in the presence of constraints, following [13]. The soundness of this procedure follows from the soundness of the chase. Suppose we are given a query Q and acyclic constraints C. We first chase Q with C to obtain U, a so-called *universal plan*. We then search for subqueries of U (obtained by removing generators from U), chasing each in turn with C and checking for equivalence with U.

6.1 Example - Movies

Start with:

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

$$\begin{aligned} C \quad ::= \quad & \texttt{for } (x \texttt{ in } Movies) \; (y \texttt{ in } Movies) \\ & \texttt{where } x.\mathsf{title} = y.\mathsf{title} \\ & x.\mathsf{director} = y.\mathsf{director} \end{aligned}$$

The chased query – the universal plan – is:

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

We may now proceed with tableau minimization by searching for subqueries of U. Removing the generator (m_1 `in` $Movies$) and replacing m_1 with m_2 in the body of the query gives a smaller query:

$$\begin{array}{lll} Q' & ::= & \texttt{for } (m_2 \texttt{ in } Movies) \\ & & \texttt{return } (m_2.\mathsf{director}, m_2.\mathsf{actor}) \end{array}$$

Now we look for a homomorphism $Q' \to U$. The identity substitution works; the important part here to notice is the `return` clause, where (m_2.director, m_2.actor) is equal to (m_1.director, m_2.actor) precisely because of the equality $m_1.\mathsf{director} = m_2.\mathsf{director}$, which appears in U but not in Q. Note that there is also a homomorphism $U \to Q'$ (namely, $m_2 \mapsto m, m_1 \mapsto m$); hence $U = Q' = Q$.

6.2 Example - Minimization without Constraints

We pause to remark that tableaux minimization can also be done *without* constraints. Indeed, this degenerate case was first proposed in 1977 [10]. Consider the (contrived) query:

$$\begin{array}{l} \texttt{for } (x \texttt{ in } X)\ (y \texttt{ in } X) \\ \texttt{where } P(x) \\ \texttt{return } E(x) \end{array}$$

This minimizes to the equivalent query:

$$\begin{array}{l} \texttt{for } (z \texttt{ in } X) \\ \texttt{where } P(z) \\ \texttt{return } E(z) \end{array}$$

In the top-to-bottom direction, the homomorphism is $x \mapsto z, y \mapsto z$, and in the bottom-to-top direction is $z \mapsto x$.

6.3 Example - Indexing

We conclude this section with an optimization scenario involving a tuple-generating constraint (that is, a constraint with a non-empty `exists` clause). As we remarked in the introduction, a reasonably competent programmer might be able to optimize our $Movies$ query directly, without applying the chase at all. But sometimes constraints are not available to the programmer, such as when indices are generated on the fly. Consider the following query, which in the set monad returns the names of all *People* between 16 and 18 years old:

$$\begin{array}{lll} Q & ::= & \texttt{for } (p \texttt{ in } People) \\ & & \texttt{where } p.\mathsf{age} > 16 \wedge p.\mathsf{age} < 18 \\ & & \texttt{return } p.\mathsf{name} \end{array}$$

Depending on the underlying access patterns, or the whims of a database administrator, a modern relational database management system might transparently index *People* by creating another relation $Children$, such that the following constraint holds:

$$\begin{array}{lll} C & ::= & \texttt{forall } (p \texttt{ in } People) \\ & & \texttt{where } p.\mathsf{age} < 21 \\ & & \texttt{exists } (c \texttt{ in } Children) \\ & & \texttt{where } p.\mathsf{name} = c.\mathsf{name} \wedge p.\mathsf{age} = c.\mathsf{age} \end{array}$$

In order to effectively use this new relation, queries written against *People* must be rewritten, at runtime, to use $Children$. Tableaux minimization provides an automated mechanism to do so. First, we look for a homomorphism $front(C) \to Q$, and discover that the identity substitution works, because $p.\mathsf{age} < 21$ is entailed by $p.\mathsf{age} > 16 \wedge p.\mathsf{age} < 18$. Thus the chase applies and we obtain a universal plan:

$$\begin{aligned} U \quad ::= \quad & \texttt{for}\ (p\ \texttt{in}\ People)\ (c\ \texttt{in}\ Children) \\ & \texttt{where}\ p.\mathsf{age} > 16 \wedge p.\mathsf{age} < 18 \wedge \\ & \qquad p.\mathsf{name} = c.\mathsf{name} \wedge p.\mathsf{age} = c.\mathsf{age} \\ & \texttt{return}\ p.\mathsf{name} \end{aligned}$$

Now, we minimize the universal plan by removing the $(p\ \texttt{in}\ People)$ generator (note that to do so we must replace each occurrence of p with some other well-typed variable, in this case c):

$$\begin{aligned} Q' \quad ::= \quad & \texttt{for}\ (c\ \texttt{in}\ Children) \\ & \texttt{where}\ c.\mathsf{age} > 16 \wedge c.\mathsf{age} < 18 \\ & \texttt{return}\ c.\mathsf{name} \end{aligned}$$

We check that $Q' = U$ by looking for homomorphisms in both directions. The identity substitution is a homomorphism $Q' \to U$, owing to the fact that $p.\mathsf{name} = c.\mathsf{name}$. But at this point there is no homomorphism $U \to Q'$, because there is no substitution h that makes $(h(p)\ \texttt{in}\ People)$ equal to $(c\ \texttt{in}\ Children)$. In fact, C alone is not enough to prove that $Q' = Q$ – there may be extra tuples in $Children$ that do not appear in $People$. But if our index was built correctly we know that an additional constraint holds:

$$\begin{aligned} C' \quad ::= \quad & \texttt{forall}\ (c\ \texttt{in}\ Children) \\ & \texttt{exists}\ (p\ \texttt{in}\ Person) \\ & \texttt{where}\ p.\mathsf{name} = c.\mathsf{name} \wedge p.\mathsf{age} = c.\mathsf{age} \end{aligned}$$

As such, we may chase Q' with C' (using the identity substitution) to obtain an equivalent query:

$$\begin{aligned} Q'' \quad ::= \quad & \texttt{for}\ (c\ \texttt{in}\ Children)\ (p\ \texttt{in}\ Person) \\ & \texttt{where}\ c.\mathsf{age} > 16 \wedge c.\mathsf{age} < 18 \wedge \\ & \qquad p.\mathsf{name} = c.\mathsf{name} \wedge p.\mathsf{age} = c.\mathsf{age} \\ & \texttt{return}\ c.\mathsf{name} \end{aligned}$$

Now we can see that the identity substitution is a homomorphism $Q'' \to U$ (again owing to the fact that $p.\mathsf{name} = c.\mathsf{name}$). We have thus concluded that $Q'' = Q' = Q = U$, as required.

7 Example - Probabilistic Movies

Now that we have, at last, proved the equivalence of our $Movies$ query and smaller $Movies$ query for UCLID monads, we can turn from the set monad to the probability monad. In this section, we define the probability monad, describe the meaning of our $Movies$ query in it, and then demonstrate that semantic optimization still makes sense.

7.1 The Probability Monad

The *probability monad* is used in functional programming as a way to represent non-deterministic computation while preserving referential transparency (similar to how the IO monad is used) [15]. The values in the probability monad are probability distributions, which we will represent as lists of pairs of events and floats, such that the floats sum to one. `binding` m with f means to condition f on m. In Haskell, we have:

```
type P a = [(a, Float)]

instance Monad P where
 return x = [(x, 1.0)]
 bind p f = [(y, px*py) | (x, px) <- p, (y, py) <- f x]
```

This definition is somewhat simplified, because as-written this monad does not have a zero and the list may contain, for example, $(a, 0.1)$ and $(a, 0.1)$ instead of $(a, 0.2)$. Nevertheless, it is relatively straightforward to add a zero by wrapping this monad in the Maybe monad, and to normalize the list of events. Rather than focusing on the details, we will describe a (biased) dice-rolling example:

```
data Dice = One | Two | Three | Four

roll :: P Dice
roll = [(One, 0.25), (Two, 0.25),
        (Three, 0.1), (Four, 0.4)]

win :: Dice -> P Bool
win Three = return True
win Four = return True
win _ = return False

test :: P Bool
test = bind roll win
```

Here, *test* evaluates to

```
[(True, 0.5), (False, 0.5)]
```

as we would expect. Intuitively, the probability monad is idempotent because adding a generator corresponds to taking another sample, and independent samples are exactly that: independent. We can test this by running

```
test' :: P Bool
test' = bind roll (\_ -> test1)
```

which does indeed evaluate to the same thing as *test*. Sophisticated probabilistic behavior can be described using this simple representation; see [15] for details.

7.2 Constraints

What are we to make of our *Movies* example in this monad? First, in this monad instances correspond to *probabilistic databases*; in other words, instances are sets of tuples, where each tuple has a corresponding probability. Here is an example instance:

```
data Movie = Movie { title:: String,
                     director:: String,
                     actor:: String }

movies1 = [ (Movie "T" "D" "A", 0.75),
            (Movie "T" "D" "B", 0.25) ]
```

Our functional dependency holds in $movies_1$; recall that the functional dependency holds when:

$$\begin{aligned}
&\texttt{for}\ (x\ \texttt{in}\ Movies)\ (y\ \texttt{in}\ Movies)\\
&\texttt{where}\ x.\mathsf{title} = y.\mathsf{title}\\
&\texttt{return}\ (\mathsf{x} : x, \mathsf{y} : y)\\
=\;&\\
&\texttt{for}\ (x\ \texttt{in}\ Movies)\ (y\ \texttt{in}\ Movies)\\
&\texttt{where}\ x.\mathsf{title} = y.\mathsf{title} \land x.\mathsf{director} = y.\mathsf{director}\\
&\texttt{return}\ (\mathsf{x} : x, \mathsf{y} : y)
\end{aligned}$$

In $movies_1$, both sides evaluate to

x	y	
(T, D, A)	(T, D, A)	.5625
(T, D, A)	(T, D, B)	.1875
(T, D, B)	(T, D, A)	.1875
(T, D, B)	(T, D, B)	.0625

In a different instance, $movies_2$:

```
movies2 = [ (Movie "T" "D1" "A", 0.75),
            (Movie "T" "D2" "B", 0.25) ]
```

the left hand side evaluates to

x	y	
(T, D_1, A)	(T, D_1, A)	.5625
(T, D_1, A)	(T, D_2, B)	.1875
(T, D_2, B)	(T, D_1, A)	.1875
(T, D_2, B)	(T, D_2, B)	.0625

whereas the right hand size evaluates to `zero`. Intuitively, we might expect the right hand side to evaluate to, in analogy with the non-probabilistic case,

x	y	
(T, D_1, A)	(T, D_1, A)	.5625
(T, D_2, B)	(T, D_2, B)	.0625

However, this is not a probability distribution – the numbers do not sum to one. In any case, the two sides are not equivalent, and the constraint does not hold in $movies_2$.

7.3 Optimization

We are now in a position to verify that our $Movies$ query and optimized sub-query give the same result when evaluated on $movies_1$ but a different result when evaluated on $movies_2$. Recall the two queries are:

```
for ($m_1$ in $Movies$) ($m_2$ in $Movies$)
where $m_1$.title $=$ $m_2$.title
return ($m_1$.director, $m_2$.actor)
```

and

```
for ($m$ in $Movies$)
return ($m$.director, $m$.actor)
```

When run on $movies_1$, both the optimized and unoptimized queries yield:

(D, A)	.75
(D, B)	.25

When run on $movies_2$, the unoptimized query yields:

(D_1, A)	.5625
(D_1, B)	.1875
(D_2, A)	.1875
(D_2, B)	.0625

whereas the optimized query yields:

$$\begin{array}{ll} (D_1, A) & .75 \\ (D_2, B) & .25 \end{array}$$

Thus we can see that tableaux minimization is semantics preserving when the constraint holds, and otherwise not.

Although probabilistic functional programming has developed into a field in its own right, probabilistic data management is only now starting to be explored. As such, the connections between the two areas is still unclear. However, whereas we associate a probability to each tuple, recent work in the database community associates a probability to an entire relation [16]. For now we remark only that this connection was quite unexpected, at it is likely that entirely new optimization methods for probabilistic programming languages can be developed based on this technique.

8 Conclusion

We have barely scratched the surface of how the chase may be used to reason about queries; in particular, we have only described how to use the chase to minimize monad comprehensions. Although we are motivated by the trend towards using functional programming languages in domains rich in constraints, such as large-scale collection processing and functional-logic programming, we believe that this connection between programming language theory and relational database theory is sure to yield additional insights. Indeed, the soundness of the chase for the probability monad appears to have been overlooked until now, and it seems likely that entirely new optimization techniques for probabilistic programming languages can be developed to exploit it.

We are currently working to design a programming language based on monad comprehensions and folds over algebraic datatypes that is a suitable intermediate form for many current large-scale data processing systems. Our compiler for this language performs semantic optimization as described in this paper and can serve as a reference implementation for applying these techniques to other systems. We are confident that the chase can bring the same benefits to a much larger class of systems as it has to relational databases.

References

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

[2] A. V. Aho, C. Beeri, and J. D. Ullman. The theory of joins in relational databases. *ACM Trans. Database Syst.*, 4:297–314, September 1979.

[3] S. Antoy and M. Hanus. Functional logic programming. *Commun. ACM*, 53:74–85, April 2010.

[4] S. Balakrishnan, V. Chu, M. A. Hernández, H. Ho, R. Krishnamurthy, S. X. Liu, J. H. Pieper, J. S. Pierce, L. Popa, C. M. Robson, L. Shi, I. R. Stanoi, E. L. Ting, S. Vaithyanathan, and H. Yang. Midas: integrating public financial data. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 1187–1190, New York, NY, USA, 2010. ACM.

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

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

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

[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. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC '77: Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 77–90, New York, NY, USA, 1977. ACM.

[10] P. e. a. Cudre-Mauroux. A demonstration of scidb: A science-oriented dbms. In *VLDB'09: Proceedings of the 2009 VLDB Endowment*. VLDB Endowment, August 2009.

[11] A. Deutsch, A. Nash, and J. Remmel. The chase revisited. In *PODS '08: Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 149–158, New York, NY, USA, 2008. ACM.

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

[13] L. Erkok. *Value recursion in monadic computations*. PhD thesis, Oregon Health & Science University, 2002. AAI3063791.

[14] M. Erwig and S. Kollmansberger. Probabilistic functional programming in haskell. *Journal of Functional Programming*, 2005.

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

[16] R. Fagin, B. Kimelfeld, and P. G. Kolaitis. Probabilistic data exchange. In *Proceedings of the 13th International Conference on Database Theory*, ICDT '10, pages 76–88, New York, NY, USA, 2010. ACM.

[17] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2008.

[18] 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.

[19] N. Ghani and P. Johann. Monadic augment and generalised short cut fusion. *J. Funct. Program.*, 17(6):731–776, 2007.

[20] 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.

[21] T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. Ferry: database-supported program execution. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 1063–1066, New York, NY, USA, 2009. ACM.

[22] 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*.

[23] J. M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Rec.*, 39:5–19, September 2010.

[24] 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.

[25] F. Henglein. Large-scale sound and precise program analysis: technical persepctive. *Commun. ACM*, 53:114–114, August 2010.

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

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

[28] C. B. Jay, M. I. Cole, M. Sekanina, and P. Steckler. A monadic calculus for parallel costing of a functional language of arrays. In *Euro-Par'97 Parallel Processing, volume 1300 of Lecture Notes in Computer Science*, pages 650–661. Springer, 1997.

[29] D. J. King and P. Wadler. Combining monads. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 134–143, London, UK, 1993. Springer-Verlag.

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

[31] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 124–144, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[32] E. Meijer and G. Hutton. Bananas in space: extending fold and unfold to exponential types. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 324–333, New York, NY, USA, 1995. ACM.

[33] E. Meijer and J. Jeuring. Merging monads and folds for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 228–266, London, UK, 1995. Springer-Verlag.

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

[35] A. Pfeffer. Ibal: A probabilistic rational programming language. In *In Proc. 17th IJCAI*, pages 733–740. Morgan Kaufmann Publishers, 2001.

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

[37] P. Wadler. Comprehending monads. In *Mathematical Structures in Computer Science*, pages 61–78, 1992.

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

[39] 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.

Theorem (Soundess of Chase)
Let

$$\begin{array}{lll} Q & ::= & \texttt{for}\ \overrightarrow{(x\ \texttt{in}\ P)} \\ & & \texttt{where}\ C(\overrightarrow{x}) \\ & & \texttt{return}\ E(\overrightarrow{x}) \end{array}$$

$$\begin{array}{lll} d & ::= & \texttt{forall}\ \overrightarrow{(r\ \texttt{in}\ R)} \\ & & \texttt{where}\ B_1(\overrightarrow{r}) \\ & & \texttt{exists}\ \overrightarrow{(s\ \texttt{in}\ S)} \\ & & \texttt{where}\ B_2(\overrightarrow{r}, \overrightarrow{s}) \end{array}$$

And suppose $h : front(d) \to Q$. Then for every I such that $I \models d$, we have that $Q(I) = Q'(I)$, where

$$\begin{array}{lll} Q' & ::= & \texttt{for}\ \overrightarrow{(x\ \texttt{in}\ P)}\ \overrightarrow{(s\ \texttt{in}\ S)} \\ & & \texttt{where}\ C(\overrightarrow{x}) \wedge B_2(h(\overrightarrow{r}), \overrightarrow{s}) \\ & & \texttt{return}\ E(\overrightarrow{x}) \end{array}$$

Reminder: `for` $\overrightarrow{(a\ \texttt{in}\ A)}\ X(\overrightarrow{a})$ denotes

$$\begin{array}{ll} \texttt{do} & a_0 \leftarrow A_0 \\ & \ldots \\ & a_N \leftarrow A_N \\ & X(a_0, \ldots, a_N) \end{array}$$

Do-notation has the standard definition in terms of monadic bind. Also, `where` $P\ Q$ denotes `if` P `then` Q `else zero`.

Proof. Choose $I \models d$. Recall that $h : \mathit{front}(d) \to Q$ means that $h(\overrightarrow{r \texttt{ in } R}) \subseteq \overrightarrow{x \texttt{ in } P}$ and $C(\overrightarrow{x}) \vdash B_1(h(\overrightarrow{r}))$. We will re-write Q into Q' in 5 steps :

$$\begin{array}{rcl}
Q & ::= & \texttt{for } \overrightarrow{(x \texttt{ in } P)} \\
 & & \texttt{where } C(\overrightarrow{x}) \\
 & & \texttt{return } E(\overrightarrow{x}) \\
C(\overrightarrow{x}) \vdash B_1(h(\overrightarrow{r})) & = & (1) \\
 & & \texttt{for } \overrightarrow{(x \texttt{ in } P)} \\
 & & \texttt{where } C(\overrightarrow{x}) \wedge B_1(h(\overrightarrow{r})) \\
 & & \texttt{return } E(\overrightarrow{x}) \\
h(\overrightarrow{r \texttt{ in } R}) \subseteq \overrightarrow{x \texttt{ in } P} & = & (2) \\
 & & \texttt{for } \overrightarrow{(x \texttt{ in } P)}\ \overrightarrow{(v \texttt{ in } R)} \\
 & & \texttt{where } C(\overrightarrow{x}) \wedge B_1(\overrightarrow{v}) \wedge \overrightarrow{v} = h(\overrightarrow{r}) \\
 & & \texttt{return } E(\overrightarrow{x}) \\
d \text{ holds} & = & (3) \\
 & & \texttt{for } \overrightarrow{(x \texttt{ in } P)}\ \overrightarrow{(v \texttt{ in } R)}\ \overrightarrow{(s \texttt{ in } S)} \\
 & & \texttt{where } C(\overrightarrow{x}) \wedge B_1(\overrightarrow{v}) \wedge B_2(\overrightarrow{v}, \overrightarrow{s}) \wedge \overrightarrow{v} = h(\overrightarrow{r}) \\
 & & \texttt{return } E(\overrightarrow{x}) \\
h(\overrightarrow{r \texttt{ in } R}) \subseteq \overrightarrow{x \texttt{ in } P} & = & (4) \\
 & & \texttt{for } \overrightarrow{(x \texttt{ in } P)}\ \overrightarrow{(s \texttt{ in } S)} \\
 & & \texttt{where } C(\overrightarrow{x}) \wedge B_1(h(\overrightarrow{r})) \wedge B_2(h(\overrightarrow{r}), \overrightarrow{s}) \\
 & & \texttt{return } E(\overrightarrow{x}) \\
C(\overrightarrow{x}) \vdash B_1(h(\overrightarrow{r})) & = & (5) \\
Q' & ::= & \texttt{for } \overrightarrow{(x \texttt{ in } P)}\ \overrightarrow{(s \texttt{ in } S)} \\
 & & \texttt{where } C(\overrightarrow{x}) \wedge B_2(h(\overrightarrow{r}), \overrightarrow{s}) \\
 & & \texttt{return } E(\overrightarrow{x})
\end{array}$$

Note that steps 4 and 5 are the same as 1 and 2, so we will just prove steps 1, 2, and 3.

1. The first step is simple:

$$\begin{array}{rcl}
 & & \texttt{for } \overrightarrow{(x \texttt{ in } P)} \\
 & & \texttt{where } C(\overrightarrow{x}) \\
 & & \texttt{return } E(\overrightarrow{x}) \\
C(\overrightarrow{x}) \vdash B_1(h(\overrightarrow{r})) & = & \\
 & & \texttt{for } \overrightarrow{(x \texttt{ in } P)} \\
 & & \texttt{where } C(\overrightarrow{x}) \wedge B_1(h(\overrightarrow{r})) \\
 & & \texttt{return } E(\overrightarrow{x})
\end{array}$$

   Because $C(\overrightarrow{x}) \vdash B_1(h(\overrightarrow{r}))$, we know that $B_1(h(\overrightarrow{r}))$ evaluates to true whenever $C(\overrightarrow{x})$ does. Hence, $C(\overrightarrow{x})$ is equivalent to $C(\overrightarrow{x}) \wedge B_1(h(\overrightarrow{r}))$, and so this step is semantics-preserving.

2. We must show that

$$h(\overrightarrow{r \texttt{ in } R}) \subseteq \overrightarrow{x \texttt{ in } P} \quad = \quad \begin{array}{l} \texttt{for } \overrightarrow{(x \texttt{ in } P)} \\ \texttt{where } C(\overrightarrow{x}) \wedge B_1(h(\overrightarrow{r})) \\ \texttt{return } E(\overrightarrow{x}) \\ \\ \texttt{for } \overrightarrow{(x \texttt{ in } P)}\ \overrightarrow{(v \texttt{ in } R)} \\ \texttt{where } C(\overrightarrow{x}) \wedge B_1(\overrightarrow{v}) \wedge \overrightarrow{v} = h(\overrightarrow{r}) \\ \texttt{return } E(\overrightarrow{x}) \end{array}$$

We do this in four steps:

$$\begin{array}{rcl} & & \texttt{for } \overrightarrow{(x \texttt{ in } P)} \\ & & \texttt{where } C(\overrightarrow{x}) \wedge B_1(h(\overrightarrow{r})) \\ & & \texttt{return } E(\overrightarrow{x}) \\ \textit{axiom with } h(\overrightarrow{r \texttt{ in } R}) \subseteq \overrightarrow{x \texttt{ in } P} & = & (a) \\ & & \texttt{for } \overrightarrow{(x \texttt{ in } P)} \\ & & \texttt{where } C(\overrightarrow{x}) \wedge B_1(h(\overrightarrow{r})) \wedge \\ & & \quad \texttt{exists } \overrightarrow{(v \texttt{ in } R)} \texttt{ where } \overrightarrow{v} = h(\overrightarrow{r}) \\ & & \texttt{return } E(\overrightarrow{x}) \\ v \notin fv(C(\overrightarrow{x}), B_1(h(\overrightarrow{r}))) & = & (b) \\ & & \texttt{for } \overrightarrow{(x \texttt{ in } P)} \\ & & \texttt{where exists } \overrightarrow{(v \texttt{ in } R)} \texttt{ where } C(\overrightarrow{x}) \wedge B_1(h(\overrightarrow{r})) \wedge \overrightarrow{v} = h(\overrightarrow{r}) \\ & & \texttt{return } E(\overrightarrow{x}) \\ \textit{idempotency} & = & (c) \\ & & \texttt{for } \overrightarrow{(x \texttt{ in } P)}\ \overrightarrow{(v \texttt{ in } R)} \\ & & \texttt{where } C(\overrightarrow{x}) \wedge B_1(h(\overrightarrow{r})) \wedge \overrightarrow{v} = h(\overrightarrow{r}) \\ & & \texttt{return } E(\overrightarrow{x}) \\ \textit{congruence} & = & (d) \\ & & \texttt{for } \overrightarrow{(x \texttt{ in } P)}\ \overrightarrow{(v \texttt{ in } R)} \\ & & \texttt{where } C(\overrightarrow{x}) \wedge B_1(\overrightarrow{v}) \wedge \overrightarrow{v} = h(\overrightarrow{r}) \\ & & \texttt{return } E(\overrightarrow{x}) \end{array}$$

We examine each of these in turn:

(a) We want to show that $\texttt{exists}\ \overrightarrow{(v\ \texttt{in}\ R)}\ \texttt{where}\ \overrightarrow{v} = h(\overrightarrow{r})$ always evaluates to true in this particular context. In the set monad, this is true because $\forall a \in A, \exists a' \in A, a = a'$ is valid. We hence require this as axiom. This axiom also holds for lists and bags.

(b) Here we use case analysis on $C(\overrightarrow{x}($ and $B_1(h(\overrightarrow{r})($. When they are both true, the two sides are obviously equal. When either of them is false, the lhs will be zero, and the rhs is equal to

$$\texttt{where exists}\ \overrightarrow{(v\ \texttt{in}\ R)}\ \texttt{where false return}\ E(\overrightarrow{x})$$

which, by definition of constraints, expands to

$$\texttt{where (return}\ \mathcal{L}(-) = \texttt{for}\ \overrightarrow{(v\ \texttt{in}\ R)}\ \texttt{where false return}\ \mathcal{L}(-))\ \texttt{return}\ E(\overrightarrow{x})$$

This will evaluate to false provided that

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

which we will assume as an axiom.

(c) Here we apply the idempotency axiom to the query below the outer loop. Recall that the axiom states that when $\overrightarrow{a} \notin fv(E)$,

$$\texttt{where exists}\ \overrightarrow{(a\ \texttt{in}\ A)}\ \texttt{where}\ P(\overrightarrow{a})\ \texttt{return}\ E = \texttt{for}\ \overrightarrow{(a\ \texttt{in}\ A)}\ \texttt{where}\ P(\overrightarrow{a})\ \texttt{return}\ E$$

(d) This is a straightfoward consequence of the equality $\overrightarrow{v} = h(\overrightarrow{r})$.

3. We must show that

$$d \text{ holds} \quad = \quad \begin{array}{l} \texttt{for } \overrightarrow{(x \texttt{ in } P)}\ \overrightarrow{(v \texttt{ in } R)} \\ \texttt{where } C(\overrightarrow{x}) \wedge B_1(\overrightarrow{v}) \wedge \overrightarrow{v} = h(\overrightarrow{r}) \\ \texttt{return } E(\overrightarrow{x}) \\ \\ \texttt{for } \overrightarrow{(x \texttt{ in } P)}\ \overrightarrow{(v \texttt{ in } R)}\ \overrightarrow{(s \texttt{ in } S)} \\ \texttt{where } C(\overrightarrow{x}) \wedge B_1(\overrightarrow{v}) \wedge B_2(\overrightarrow{v}, \overrightarrow{s}) \wedge \overrightarrow{v} = h(\overrightarrow{r}) \\ \texttt{return } E(\overrightarrow{x}) \end{array}$$

Recall that

$$d \quad ::= \quad \begin{array}{l} \texttt{forall } \overrightarrow{(r \texttt{ in } R)} \\ \texttt{where } B_1(\overrightarrow{r}) \\ \texttt{exists } \overrightarrow{(s \texttt{ in } S)} \\ \texttt{where } B_2(\overrightarrow{r}, \overrightarrow{s}) \end{array}$$

By definition (the meaning of a constraint is that its front equals its back), this unfolds to

$$\begin{array}{ll} & \texttt{for } \overrightarrow{(r \texttt{ in } R)} \\ & \texttt{where } B_1(\overrightarrow{r}) \\ & \texttt{return } \mathcal{L}(\overrightarrow{r}) \\ = & \\ & \texttt{for } \overrightarrow{(r \texttt{ in } R)}\ \overrightarrow{(s \texttt{ in } S)} \\ & \texttt{where } B_1(\overrightarrow{r}) \wedge B_2(\overrightarrow{r}, \overrightarrow{s}) \\ & \texttt{return } \mathcal{L}(\overrightarrow{r}) \end{array}$$

Assuming the uniformity propery (which holds for sets), we have instead, for any X,

$$\begin{array}{ll} & \texttt{for } \overrightarrow{(r \texttt{ in } R)} \\ & \texttt{where } B_1(\overrightarrow{r}) \\ & X(\overrightarrow{r}) \\ = & \\ & \texttt{for } \overrightarrow{(r \texttt{ in } R)}\ \overrightarrow{(s \texttt{ in } S)} \\ & \texttt{where } B_1(\overrightarrow{r}) \wedge B_2(\overrightarrow{r}, \overrightarrow{s}) \\ & X(\overrightarrow{r}) \end{array}$$

We then use the above (call the above lemma d-holds), and two additional lemmas, called split-generators and split-equalities, to get semantics preservation:

```
                      for $\overrightarrow{(x \texttt{ in } P)}$ $\overrightarrow{(v \texttt{ in } R)}$
                      where $C(\overrightarrow{x}) \wedge B_1(\overrightarrow{v}) \wedge \overrightarrow{v} = h(\overrightarrow{r})$
                      return $E(\overrightarrow{x})$
split-generators  =
                      for $\overrightarrow{(x \texttt{ in } P)}$
                      where $C(\overrightarrow{x})$
                          for $\overrightarrow{(v \texttt{ in } R)}$
                          where $B_1(\overrightarrow{v}) \wedge \overrightarrow{v} = h(\overrightarrow{r})$
                          return $E(\overrightarrow{x})$
split-equalities  =
                      for $\overrightarrow{(x \texttt{ in } P)}$
                      where $C(\overrightarrow{x})$
                          for $\overrightarrow{(v \texttt{ in } R)}$
                          where $B_1(\overrightarrow{v})$
                          where $\overrightarrow{v} = h(\overrightarrow{r})$ return $E(\overrightarrow{x})$
        $d -$ holds  =
                      for $\overrightarrow{(x \texttt{ in } P)}$
                      where $C(\overrightarrow{x})$
                          for $\overrightarrow{(v \texttt{ in } R)}$ $\overrightarrow{(s \texttt{ in } S)}$
                          where $B_1(\overrightarrow{v}) \wedge B_2(\overrightarrow{v}, \overrightarrow{s})$
                          where $\overrightarrow{v} = h(\overrightarrow{r})$ return $E(\overrightarrow{x})$
split-equalities  =
                      for $\overrightarrow{(x \texttt{ in } P)}$
                      where $C(\overrightarrow{x})$
                          for $\overrightarrow{(v \texttt{ in } R)}$ $\overrightarrow{(s \texttt{ in } S)}$
                          where $B_1(\overrightarrow{v}) \wedge B_2(\overrightarrow{v}, \overrightarrow{s}) \wedge \overrightarrow{v} = h(\overrightarrow{r})$
                          return $E(\overrightarrow{x})$
split-generators  =
                      for $\overrightarrow{(x \texttt{ in } P)}$ $\overrightarrow{(v \texttt{ in } R)}$ $\overrightarrow{(s \texttt{ in } S)}$
                      where $C(\overrightarrow{x}) \wedge B_1(\overrightarrow{v}) \wedge B_2(\overrightarrow{v}, \overrightarrow{s}) \wedge \overrightarrow{v} = h(\overrightarrow{r})$
                      return $E(\overrightarrow{x})$
```

We will now justify the first uses of split-generators and split-equalities; the second uses are similar (although in the reverse directions).

To justify split-generators, because both the lhs and rhs begin with `for` $\overrightarrow{(x \texttt{ in } P)}$ we would like that the following equality holds between the "rest" of the lhs and rhs:

```
    for $\overrightarrow{(v \texttt{ in } R)}$
    where $C(\overrightarrow{x}) \wedge B_1(\overrightarrow{v}) \wedge \overrightarrow{v} = h(\overrightarrow{r})$
    return $E(\overrightarrow{x})$
=
    where $C(\overrightarrow{x})$
      for $\overrightarrow{(v \texttt{ in } R)}$
      where $B_1(\overrightarrow{v}) \wedge \overrightarrow{v} = h(\overrightarrow{r})$
      return $E(\overrightarrow{x})$
```

When $C(\overrightarrow{x})$ evaluates to true, the two sides are obviously equivalent. When $C(\overrightarrow{x})$ evaluates to false both sides are zero because of the monad-zero law: for any f,

$$\texttt{for } \overrightarrow{(v \texttt{ in } R)} \texttt{ where false } f = \texttt{zero}$$

To justify split-equalities, because both the lhs and rhs begin with `for` $\overrightarrow{(v \texttt{ in } R)}$ we would like that the following equality holds between the "rest" of the lhs and rhs:

```
      where $B_1(\overrightarrow{v}) \wedge \overrightarrow{v} = h(\overrightarrow{r})$
      return $E(\overrightarrow{x})$
=
      where $B_1(\overrightarrow{v})$
      where $\overrightarrow{v} = h(\overrightarrow{r})$ return $E(\overrightarrow{x})$
```

We reason again by case analysis. Suppose $B_1(\overrightarrow{v})$ is false. Then both sides are zero. Suppose $B_1(\overrightarrow{v})$ is true. Now, suppose $\overrightarrow{v} = h(\overrightarrow{r})$ is false. Then both sides are zero. Suppose $\overrightarrow{v} = h(\overrightarrow{r})$ is true. Then both sides are `return` $E(\overrightarrow{x})$, as required.