Outline

• Intro to Collection Processing with Functional Query Languages
• Four open problems
• Four solutions
Collection Processing

- Recognized early as an important application domain (SETL, 1960’s)

- Collections are invariably *big*

- Collection languages are invariably *declarative*

- Optimization of declarative queries widely studied
Paradigms

- Relational
  - SQL
  - Datalog
  - Nested Relational Calculus
- Functional
  - MapReduce, PIG
  - SETL, NESL
  - Data Parallel Haskell, DryadLINQ
Functional Query Languages

- Functional Query Languages
  - based in part on pure lambda calculus
  - extend relational languages (usually)
- Rejected in 90’s by DB community in favor of nested relations
- Resurfaced as part of NoSQL movement
- This talk: design a good intermediate form for functional query languages
Naive Approach

• Start with the simply typed lambda calculus
  • Add polynomial datatypes to model data
  • Add folds to model computation
  • Add monads to model collections
  • Add comprehensions to model queries

• We’ll be using Haskell to illustrate
• This approach is re-discovered a lot...
Benefits

- Monad comprehensions de-sugar into folds
- Folds can express all primitive recursion functions
- Folds can be *fused*
- Well-understood equational theory
Drawbacks

• Fusion fails in common situations
• Monad comprehensions cannot express aggregation
• No way to express or use constraints
• With non-free collections (e.g. sets) program soundness is undecidable
This talk

- Fusion fails in common situations
  - Use monadic augment fusion (PL)
- Monad comprehensions cannot express aggregation
  - Use monad algebra comprehensions (DB)
- No way to express or use constraints
  - Add embedded dependencies and chase them (DB)
- With non-free collections (e.g. sets) program soundness is undecidable
  - Emit verification conditions and solve them in Coq (PL)
Basics: Polynomial Data

- Lists in “insert presentation”
  
  ```haskell
  data List a = Nil | Cons a (List a)
  ```

- Fold combinator:
  
  ```haskell
  fold :: b -> (a -> b -> b) -> List a -> b
  
  fold nil' cons' Nil = nil'
  
  fold nil' cons' (Cons hd tl) =
    cons' hd (fold nil' cons' tl)
  
  count :: List a -> Nat
  
  count = fold 0 (\hd tl -> 1 + tl)
  ```

  *Actually, we will use setoids, but I will omit this from the talk...
Fold-Build Fusion

• In addition to fold, a build combinator exists:

\[
\text{build} :: (\forall b. b \rightarrow (a \rightarrow b \rightarrow b) \rightarrow b) \rightarrow \text{List } a
\]

\[
\text{build } g = g \text{ Nil Cons}
\]

• Fold-build fusion:

\[
\text{fold } n \ c \ (\text{build } g) = g \ n \ c
\]
Queries

- Programming directly with folds is tedious.
- Instead, use monads with zeros

```haskell
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
```
Monad Laws

- Monad definitions must obey the laws

\[
\text{bind} \ (\text{return} \ x) \ f = f \ x \\
\text{bind} \ m \ \text{return} = m \\
\text{bind} \ (\text{bind} \ m \ f) \ g = \\
\quad \text{bind} \ m \ (\lambda x \rightarrow \text{bind} \ (f \ x) \ g) \\
\text{bind} \ \text{zero} \ f = \text{zero} \\
\text{bind} \ m \ (\lambda x \rightarrow \text{zero}) = \text{zero}
\]
Do Notation

• Monads let us use do-notation to express queries

\[
\text{do } x \leftarrow X \ c \\
= \text{bind } X (\lambda x \rightarrow c \ x)
\]

• Cartesian product:

\[
\text{do } x \leftarrow X \\
\quad y \leftarrow Y \\
\quad \text{return } (x, y)
\]

• Do notation is parametric in a monad with zero.
Conjunctive Queries

• By further restricting which comprehensions we allow, we end up with *conjunctive queries*.

  \[
  \text{for}(x_1 \text{ in } X_1)\ldots(x_N \text{ in } X_N) \text{ where } P(x_1,\ldots,x_N) \ R(x_1,\ldots,X_n)
  \]

• Interpreted as

  \[
  \text{do } x_1 \leftarrow X_1
  
  \ldots
  
  x_N \leftarrow X_N

  \text{if } P(x_1,\ldots,x_N)
  
  \text{then } R(x_1,\ldots,x_N)

  \text{else zero}
  \]
Example

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

```java
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 SQL (set monad):

```sql
SELECT m1.director, m2.actor
FROM Movies AS m1, Movies AS m2
WHERE m1.title = m2.title
```
Beyond the Naive Approach

- Hopefully you are convinced that the naive approach
  - Can model many collections and computations
  - Captures special cases like SQL
  - Has powerful fusion opportunities

- But problems still remain...
Fusion

• Fold-build fusion is great when it works:
  
  \[ \text{sumSqs} \hspace{1em} = \hspace{1em} \text{fold} \hspace{1em} 0 \hspace{1em} (\hspace{1em} (+) \hspace{1em}) \hspace{1em} \]
  
  \[ (\text{build} \hspace{1em} (\hspace{1em} \text{\textbackslash}n \hspace{1em} c \hspace{1em} \rightarrow \hspace{1em} \text{fold} \hspace{1em} n \hspace{1em} (c \hspace{1em} . \hspace{1em} \text{sqr}) \hspace{1em} \text{xs}))) \]

• Becomes:

  \[ \text{sumSqs} \hspace{1em} = \hspace{1em} \text{fold} \hspace{1em} 0 \hspace{1em} ((+) \hspace{1em} . \hspace{1em} \text{sqr}) \hspace{1em} \]
• But this doesn’t work on append (++)

\[
\text{ys} ++ \text{xs} = \text{fold ys Cons xs}
\]

• 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:

\[
\text{fold z f (map g xs ++ ys)}
\]

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

\[
\text{ys} ++ \text{xs} = \text{build (\n c \rightarrow \text{fold ys Cons xs})}
\]

• This code is incorrect, because \text{ys} is a list, but needs to be element type.
For lists, Gill introduced a generalization of the build operation, called augment,

\[ \text{augment} :: (\forall b. \ a \rightarrow (a \rightarrow b \rightarrow b) \rightarrow b) \rightarrow \text{List} a \rightarrow \text{List} a \]

\[ \text{augment } g \hspace{1pt} x s = g \hspace{1pt} x s \hspace{1pt} \text{Cons} \]

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

\[ \text{build } g = \text{augment } g \hspace{1pt} \text{Nil} \]
Fusion IV

- Fold-augment fusion:
  \[
  \text{fold } z \ k \ (\text{augment } g \ h) = g \ k \ (\text{fold } z \ k \ h)
  \]

- Using augment instead of build allows append to be fused.

- Until 2005, augment was only defined for lists. But Ghani et al showed that for \textit{parameterized monads over polynomial datatypes}, augment always exists and is inter-definable with bind and build:
  \[
  \text{augment } g \ k = \text{bind } (\text{build } g) \ k
  \]
Fusion Conclusion

• Having a generalized augment combinator is a huge win for collection processing, because it allows queries of the form:

```latex
do \ x \leftarrow \ X \\
\quad \ y \leftarrow \ Y \\
\quad \ return \ (f \ x \ y)
```

• to be fused. Ghani further argues that this kind of fusion is complete and the best possible.
Aggregation

- Monad comprehensions cannot express aggregation
- Try summing all the elements of a list L:
  ```haskell
do x <- L
?```
- Problem: the return type of a comprehension is monadic
Monad Algebras

• Unbeknownst to functional programmers, do-notation can be interpreted not just in a monad, but in a monad algebra

• A monad algebra (at t) is given by a function agg
  
  \[ \text{agg} :: \text{Monad } M \Rightarrow M t \rightarrow t \]

• obeying certain equations.

• Summing all the elements in a list is a monad algebra; summing all the elements in a set is not.
Examples

- To sum a list \( X \) using a comprehension, we simply write:

  \[
  \text{do } x \leftarrow X
  \]

- To sum a list \( X \) after adding 1 to each element, we write

  \[
  \text{do } x \leftarrow X
  x + 1
  \]

- To sum every pairwise element combination of two lists \( X, Y \), we write

  \[
  \text{do } x \leftarrow X
  y \leftarrow Y
  x + y
  \]
Aggregation Conclusion

• Writing “aggregation comprehensions” takes some getting used to.

• Optimizing aggregation is still a challenge even in SQL.

• But writing aggregation as a comprehension instead of a fold allows aggregation queries to participate in the powerful comprehension optimizations discussed next.
Constraints

• Constraints play a key role in large-scale data processing
  • Example: replace a full scan with a lookup
• But the naive approach says nothing about them
• This section: an elegant way to add constraints and to use them to optimize comprehensions
Example

MoviesBig = \( \text{for } (m1 \text{ in } \text{Movies}) \ (m2 \text{ in } \text{Movies}) \)

\[ \text{where } m1.\text{title} = m2.\text{title} \]

\[ \text{return } (m1.\text{director}, m2.\text{actor}) \]

MoviesSmall = \( \text{for } (m \text{ in } \text{Movies}) \)

\[ \text{return } (m.\text{director}, m.\text{actor}) \]

- This query returns a set of tuples \((d, a)\) where \(a\) acted in a movie directed by \(d\).
- These two queries are equivalent (in the set monad) exactly when the functional dependency \(\text{title} \rightarrow \text{director}\) holds.
Motivation

• We need to be able to express things like functional dependencies

• We need to be able to automatically re-write MoviesBig into MoviesSmall

• Some commercial SQL systems and information integration systems (e.g. Clio) do this
Embedded Dependencies

• Basic idea: constraints should have a very specific syntactic form

\[
\text{forall } (x \text{ in Movies}) (y \text{ in Movies}) \\
\text{where } x.\text{title} = y.\text{title} \\
\text{exists} \\
\text{where } x.\text{director} = y.\text{director}
\]
The Chase

• Given
  • A query Q1
  • A query Q2
  • An “acyclic” embedded dependency C
  • A monad algebra obeying additional equations

• The chase is a decision procedure for determining if Q1 is equivalent to Q2 when C holds
Tableaux Minimization

- We can use the chase to rewrite MoviesBig into MoviesSmall, a process called \textit{tableaux minimization}. This is \textbf{complete} for the set monad.

\begin{verbatim}
MoviesBig = for (m1 in Movies) (m2 in Movies)
  where m1.title = m2.title
  return (m1.director, m2.actor)

U = for (m1 in Movies) (m2 in Movies)
  where m1.title = m2.title
  and m1.director = m2.director
  return (m1.director, m2.actor)

MoviesSmall = for (m in Movies)
  return (m.director, m.actor)
\end{verbatim}
Constraints: Conclusion

• By adding constraints in this manner, we are able to reason about monad algebra comprehensions “modulo” constraints.

• This provides another way to minimize the number of bind operations in a query.
Verification Conditions

- In this development, we need verification 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.
  - Moreover, we allow users to write “assert” and “assume” statements about embedded dependencies.
  - A simple pass over the program emits Coq theorems, which must be proved by the user.
Conclusion

- An intermediate form based on folds and monads is a perennial idea
- Fell out of favor in the 90s, but returned as part of NoSQL
- In this talk we demonstrate four shortcomings in the naive approach, each of which has a solution discovered for other reasons in either the DB or PL communities.
- I am developing a “universal compiler” based on these principles for my Ph.D. thesis - stay tuned.