Generic Multiset Programming

Fritz Henglein

Department of Computer Science
University of Copenhagen
Email: henglein@diku.dk

SSGEP, Oxford, 2015-07-07

Example problem

Gather, aggregate and interpret bulk data.
Example: A conjunctive join query (in SQL notation)

```
SELECT depName, acctBalance
FROM   depositors, accounts
WHERE  depId = acctId
```

How to evaluate such a query?

Standard evaluation

Auxiliary definitions:

```
(f *** g) (x, y) = (f x, g y)
(p .==. q) (x, y) = (p x == q y)
prod s t = [ (x, y) | x <- s, y <- t ]
```

Query:

```
map (depName *** acctBalance)
          (filter (depId .==. acctId)
                    (depositors `prod` accounts))
```

+ Compositional, simple

−− $\Theta(n^2)$ time complexity (not scalable)

Dynamic symbolic computation

Query, with standard evaluation:

```
map (depName *** acctBalance)
        (filter (depId .==. acctId)
                (depositors ‘prod‘ accounts))
```

Query, with dynamic symbolic computation:

```
map (depName *** acctBalance)
        (filter ((depId, acctId) ‘Is‘ eqInt)
                 (depositors ‘prod‘ accounts)
```

Difference:

++ $\Theta(n)$ time complexity (scalable!)

Note: `map`, `filter`, `prod`, `***` have different types.

Lazy (symbolic) cross-products and unions

Add constructors for cross-product and union to **mulitset** datatype:

```
data MSet a where
     O       :: MSet a
     S       :: a -> MSet a
     U       :: MSet a -> MSet a -> MSet a
     X       :: MSet a -> MSet b -> MSet (a, b)

list s = ...
```

- `O`: Empty
- `S x`: Singleton
- `s1 ‘U‘ s2`: Union
- `s1 ‘X‘ s2`: **Cartesian product** (the new thing)

So what?

- `U`: Append lists[1].
  - Constant-time concatenation
  - Conversion to cons lists $\cong$ difference lists (efficient! coherent!)
  - Alternative: Allow pattern-matching on `U` (efficient! coherent?)
- `X`: Symbolic products
  - Constant-time Cartesian product
  - Conversion to append lists $\cong$ multiplying out (inefficient! coherent!)
  - **Alternative: Allow pattern-matching on `X` (efficient! coherent?)**

Idea: Exploit algebraic identities of Cartesian products for

- asymptotic performance improvements in *some* contexts
- at most constant-time overhead in *all* contexts

[1] Join lists, Boom lists, ropes, catenable lists

Example: Count (cardinality)

```
count :: MSet a -> Int
count O = 0
count (S x) = 1
count (s1 `U` s2) = count s1 + count s2
count (s1 `X` s2) = count s1 * count s2
```

- Pattern match on new constructors `X` and `U`
- Exploitation of algebraic properties (here: homomorphic property)
    - No multiplying out of cross-product!

Perform: Standard evaluation

- Generalized *projection*
- Functor action of `MSet` on **Set**-morphisms (`fmap`)

First try:

```
perform :: (a -> b) -> MSet a -> MSet b
perform f 0           = 0
perform f (S x)       = S (f x)
perform f (s ‘U‘ t)   = perform f s ‘U‘ perform f t
perform f s           = perform f (norm s)
```

where

`norm :: MSet a -> MSet a`

multiplies products out.

Perform: Looking for asymptotic speedups

For which `f`, `s`, `t`:

```
perform f (s ‘X‘ t) = ... (no norm (s ‘X‘ t)) ...?
```

Example:

```
perform fst (s ‘X‘ t) = times (count t) s
```

where

```
times 0 s = 0
times 1 s = s
times n s = s ‘U‘ times (n-1) s
```

Idea: Turn into evaluation rule. Need to pattern match on `fst`!

Performable functions (symbolic arrows)

```
data Func a b where
    Func     :: (a -> b) -> Func a b
    Id       :: Func a a
    (:***:)  :: Func a b -> Func c d ->
                Func (a, c) (b, d)
    Fst      :: Func (a, b) a
    Snd      :: Func (a, b) b

ext :: Func (a b) -> (a -> b)
ext (Func f) x = f x
ext Id x       = x
...
```

- `Func f`: Ordinary function as performable function
- `f :***: g`: Parallel composition of `f, g`
- `ext f`: Ordinary function represented by performable function

Perform: Definition

```
perform :: Func a b -> MSet a -> MSet b
perform f (s1 `U` s2)  = perform f s1 `U` perform f s2
perform (f1 :***: f2) (s1 `X` s2)  =
        perform f1 s1 `X` perform f2 s2
perform Fst (s1 `X` s2) = count s2 `times` s1
perform Snd (s1 `X` s2) = count s1 `times` s2
perform f s = perform f (norm s) -- default clause
...
```

- Clauses for `X` represent algebraic equalities that avoid multiplying out cross-product.
- Default clause corresponds to standard evaluation.
  - Catches all cases not caught by special matches.

Symbolic representation of scaling operator

Idea: Introduce lazy constructor for `times`.

```
data MSet a where
     O    :: MSet a
     S    :: a -> MSet a
     U    :: MSet a -> MSet a -> MSet a
     X    :: MSet a -> MSet b -> MSet (a, b)
     (:.) :: Integer -> MSet a -> MSet a
```

```
perform Fst (s1 `X` s2) = count s2 `:.` s1
perform Snd (s1 `X` s2) = count s1 `:.` s2
```

Plus additional clauses for `perform`, `select`, `count`, when applied to `(:.)`-constructor terms.

Reduction

- We also need to *aggregate* and interpret multisets; e.g. compute sum, maximum, minimum, product.
- *Reduction* = unique homomorphism from $(\mathrm{MSet}(S), \cup, \emptyset)$ to commutative monoid (S, f, n)

```
reduce :: ((a, a) -> a, a) -> MSet a -> a
reduce (f, n) O = n
reduce (f, n) (S x) = x
reduce (f, n) (s ‘U‘ t) = f (reduce f n s, reduce f n t)
reduce (f, n) (k ‘:.‘ s) = ...?
reduce (f, n) (s ‘X‘ t) = ...?
```

Problem: What to do about `X` and `(:.)`?

Useful algebraic properties for reduction

Notation:

$$
\begin{aligned}
S \mathbin{\widehat{\oplus}} T &= \mathrm{map} \oplus (S \times T) \qquad \text{for binary } \oplus \\
f(S) &= \mathrm{map}\, f(S) \qquad \text{if } f : U \to V, S \subseteq U \\
\Sigma &= \mathrm{reduce}(+, 0)
\end{aligned}
$$

Algebraic identities for certain functions mapped over cross-products:

$$
\begin{aligned}
\Sigma\,(S \mathbin{\widehat{+}} T) &= |T| \cdot \Sigma\, S + |S| \cdot \Sigma\, T \\
\Sigma\,(S \mathbin{\widehat{*}} T) &= \Sigma\, S * \Sigma\, T \\
\Sigma\,(S \mathbin{\widehat{+}} T)^2 &= |T| \cdot \Sigma\, S^2 + |S| \cdot \Sigma\, T^2 + 2 \cdot (\Sigma\, S) * (\Sigma\, T) \\
\Sigma\,(S \mathbin{\widehat{*}} T)^2 &= \Sigma\, S^2 * \Sigma\, T^2
\end{aligned}
$$

Reduction

- Add constructors for $+, *, ^2, \ldots$ to `Func a b`
- Add constructor `:$` for mapping symbolic arrows over Cartesian products

```
reduce :: (Func (a, a) a, a) -> MSet a -> a
reduce (f, n) O = n
reduce (f, n) (S x) = x
reduce (f, n) (s `U` t) =
       ext f (reduce f n s, reduce f n t)
reduce ((:+:), 0) ((:+:) :$ (s `X` t)) =
       count t * reduce (+, 0) s +
       count s * mreduce (+, 0) t
...  -- more algebraic simplifications
reduce (f, n) s = reduce (f, n) (norm s) -- default
```

Application: Finite probability distributions

Represent finite probability spaces ("distributions") with rational probabilities as multisets:

```
type Probability = Rational
type Dist a = MSet a
```

Probability of element x: $\frac{\#\text{ occurrences of } x \text{ in } s}{|s|}$

Probabilistic choice between two distributions:

```
choice :: Probability -> Dist a -> Dist a -> Dist a
choice p s t =
  let v = numerator p * count t
      w = (denominator p - numerator p) * count s
  in (v `:.` s) `U` (w `:.` t)
```

Computing mean and variance

```
msum = reduce ((:+:), 0)

mean p = msum p / count p

variance p =
  let n = count p                    -- sum X^0
      s = msum p                     -- sum X^1
      s2 = msum (perform Sq p)       -- sum X^2
  in (n * s2 - s^2) / n^2
```

+ Compositional, simple
+ Linear time for independent random variables (products of distributions)

Fuzzy sets

Idea: Extend admissible range of numbers to scale with; e.g.

```
data MSet a where
     0    :: MSet a
     S    :: a -> MSet a
     U    :: MSet a -> MSet a -> MSet a
     X    :: MSet a -> MSet b -> MSet (a, b)
     (:.) :: Float -> MSet a -> MSet a
```

Allow instead of `Float`

- Booleans: *sets*;
- nonnegative integers: *multisets*;
- integers: *hybrid sets*;
- reals in $[0 \ldots 1]$: *fuzzy sets*;
- reals in $[0 \ldots \infty]$: *fuzzy multisets*;
- all reals: *fuzzy hybrid sets*

Wait a minute: "Hybrid sets"? "Fuzzy hybrid sets?"

Summary: Dynamic symbolic computation

Method for adding symbolic processing step by step to base implementation:

1. Identify (asymptotically) expensive operation
2. Introduce symbolic data constructor for its result
3. *Exploit algebraic properties during evaluation*
   - Not just lazy evaluation
4. This may lead to new needs/opportunities for applying dynamic symbolic computation: Repeat!

Relation to query optimization

Implementation performs classical algebraic query optimizations, including

- filter promotion (performing selections early)
- join introduction (replacing product followed by selection by join)
- join composition (combining join conditions to avoid intermediate multiplying out)

Observe:

- Done at run-time
- No static preprocessing
- Data-dependent optimization possible.
- Deforestatation of intermediate materialized data structures not necessary due to lazy evaluation.

Staged symbolic computation

1. Static symbolic computation
   - *All* operations treated as constructors ("abstract syntax tree")
   - Rewriting on open terms (unknown/parametric input)
   - Rewriting by interpretation
2. Standard evaluation
   - *Few* operations treated as constructors (only value constructors)
   - Rewriting on ground terms only
   - Compiled evaluation ("normalization by evaluation")

$+$: Staging: Symbolic operations executed only once

$-$: Narrowing or no narrowing for free variables? (Lots of rewrite rules)

$-$: Standard evaluation steps implemented twice

$-$: Interpreted symbolic computation

$-$: Compositionality?

. . . and dynamic symbolic computation

1. Symbolic and standard computation steps intermixed
   - *Some* operations treated as constructors (driven by asymptotic performance)
   - Ground terms only
   - Compiled symbolic computation and evaluation

– : Unstaged: Symbolic operations incur (constant-time) run-time overhead

– : Ground terms only: No need for narrowing (Few rewrite rules)

– : Standard evaluation steps implemented only once

– : Compiled symbolic computation

– : Compositionality!

Compositionality: Functional abstraction

```
module AccountManagement where
  accts = ...
  deps = ...
  countFilter :: Pred (Account, Depositor) -> Int
  countFilter pred =
       count (select pred (accts ‘X‘ deps))
```

```
module Run where
  res = ( countFilter ((acctId, depId) ‘Is‘ eqInt32),
          countFilter TT )
```

Related work

- Henglein, *Optimizing relational algebra operations using generic partitioning discriminators and lazy products*, PEPM 2010
- Henglein, Larsen, *Generic multiset programming for language-integrated querying*, WGP 2010
- Henglein, Larsen, *Generic Multiset Programming with Discrimination-based Joins and Symbolic Cartesian Products*, HOSC 2010
- Henglein, *Dynamic Symbolic Computation for Domain-Specific Language Implementation*, LOPSTR 2011 (also XLDI 2012)
- Olteanu, Závodný, *Factorised Representations of Query Results: Size Bounds and Readability*, ICDT 2012, journal version to appear in TODS; see *Factorised Databases*, http://www.cs.ox.ac.uk/projects/FDB/

Future work ("Homework")

- Conjectures:
  - Subsumes all static algebraic relational algebra optimizations.
  - Is subsumed by SQL-query optimization for SPJ-queries.
  - Properly improves upon SQL-query optimization (for some systems) for *nested* SQL-queries
- Predictable performance: Compositional performance analysis by abstract interpretation?
- Robust performance: Performance closed under which local transformations?
- Willard-Goyal-Paige query optimization for complex join queries on more than 2 multisets
- High-performance implementation for querying distributed data sources
- Scalable data-parallel algorithms and implementations; key problem: join (discriminination)

Where are we?

1. Yesterday: Generic discrimination
2. Today: Generic multiset programming
3. Thursday: Fuzzing the counts and going negative

Thank you!