Algebraic Run-Time Optimization
for Multiset Programming
(Dynamic Symbolic Computation)

Fritz Henglein

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

XLDI 2012 invited talk, Copenhagen, 2012-09-09
Example problem

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

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

How to evaluate such a query?
Standard evaluation

Auxiliary definitions:

\[(f *** g) (x, y) = (f x, g y)\]
\[(p .==. q) (x, y) = (p x == q y)\]
\[\text{prod } s \ t = \{ (x, y) \mid x \leftarrow s, y \leftarrow t \}\]

Query:

\[\text{map (depName *** acctBalance)}\]
\[\quad \text{(filter (depId .==. acctId)}\]
\[\quad \text{(depositors ‘prod‘ accounts))}\]

+ 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:

++ Θ(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 `multiset` datatype:

```haskell
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?

- \( \mathbb{U} \): Append lists\(^1\).
  - Constant-time concatenation
  - Conversion to cons lists \( \cong \) difference lists (efficient! coherent!)
  - Alternative: Allow pattern-matching on \( \mathbb{U} \) (efficient! coherent?)

- \( \mathbb{X} \): Symbolic products
  - Constant-time Cartesian product
  - Conversion to append lists \( \cong \) multiplying out (inefficient! coherent!)
  - Alternative: Allow pattern-matching on \( \mathbb{X} \) (efficient! coherent?)

Idea: Exploit algebraic identities of Cartesian products for
- asymptotic performance improvements in some contexts
- constant-time overhead in all contexts

\(^1\)Join lists, Boom lists, ropes, catenable lists
Example: Count (cardinality)

\[
\text{count} :: \text{MSet} \ a \rightarrow \text{Int}
\]
\[
\text{count} \ 0 = 0
\]
\[
\text{count} \ (S \ x) = 1
\]
\[
\text{count} \ (s1 \ 'U' \ s2) = \text{count} \ s1 + \text{count} \ s2
\]
\[
\text{count} \ (s1 \ 'X' \ s2) = \text{count} \ s1 \times \text{count} \ s2
\]

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

```
perform :: (a -> b) -> MSet a -> MSet b
perform f O = O
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 \( f, s, t \):

\[
\text{perform } f (s 'X' t) = \ldots \text{(no norm } (s 'X' t)) \ldots ?
\]

Example:

\[
\text{perform } fst (s 'X' t) = \text{times } (\text{count } t) s
\]

where

\[
\begin{align*}
\text{times } 0 s &= 0 \\
\text{times } 1 s &= s \\
\text{times } n s &= s 'U' \text{times } (n-1) s
\end{align*}
\]

Idea: Turn into evaluation rule. Need to pattern match on \( fst \)!
Performable functions (symbolic arrows)

```haskell
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 \((\text{Bag}(S), \cup, \emptyset)\) to commutative monoid \((S, f, n)\)

```haskell
reduce :: ((a, a) -> a, a) -> Bag 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:

\[ S \oplus T = \text{map} \oplus (S \times T) \quad \text{for binary } \oplus \]
\[ f(S) = \text{map} f(S) \quad \text{if } f : U \to V, S \subseteq U \]
\[ \Sigma = \text{reduce}(+, 0) \]

Algebraic identities for certain functions mapped over cross-products:

\[ \Sigma (S \oplus T) = |T| \cdot \Sigma S + |S| \cdot \Sigma T \]
\[ \Sigma (S \ast T) = \Sigma S \ast \Sigma T \]
\[ \Sigma (S \oplus T)^2 = |T| \cdot \Sigma S^2 + |S| \cdot \Sigma T^2 + 2 \cdot (\Sigma S) \ast (\Sigma T) \]
\[ \Sigma (S \ast T)^2 = \Sigma S^2 \ast \Sigma T^2 \]
\[ \Pi (S \ast T) = (\Pi S)^{|T|} \ast (\Pi T)^{|S|} \]
Reduction

- Add constructors for +, *, \( ^2 \), ... to \( \text{Func} \ a \ b \)
- Add constructor \( \texttt{\$} \) for mapping symbolic arrows over Cartesian products

\[
\text{reduce} :: (\text{Func} (a, a) a, a) \rightarrow \text{Bag} a \rightarrow a
\]
\[
\text{reduce} (f, n) 0 = n
\]
\[
\text{reduce} (f, n) (\text{S} \ x) = x
\]
\[
\text{reduce} (f, n) (\text{U} \ t) =
\]
\[
\quad \text{ext} f (\text{reduce} f n s, \text{reduce} f n t)
\]
\[
\text{reduce} ((\text{+:+:}), 0) ((\text{+:+:}) \texttt{\$} (s \ 'X' \ t)) =
\]
\[
\quad \text{count} t * \text{reduce} (+, 0) s +
\]
\[
\quad \text{count} s * \text{mreduce} (+, 0) t
\]
... -- more algebraic simplifications
\[
\text{reduce} (f, n) s = \text{reduce} (f, n) (\text{norm} s) -- \text{default}
\]
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
dchoice 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

\[
\text{msum} = \text{reduce } (\text{:+:}, 0)
\]

\[
\text{mean } p = \frac{\text{msum } p}{\text{count } p}
\]

\[
\text{variance } p = \\
\text{let } n = \text{count } p \quad \text{-- sum } X^0 \\
\text{s} = \text{msum } p \quad \text{-- sum } X^1 \\
\text{s2} = \text{msum (perform Sq } p) \quad \text{-- sum } X^2 \\
\text{in } \left( n \times \text{s2} - \text{s}^2 \right) / n^2
\]

- 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

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

In:
Future work

- Conjectures: Subsumes all static algebraic relational algebra optimizations; properly improves upon SQL-query optimization
- 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)
Perspectives for XLDI

- Methodology for cross-model DSL design and agile implementation
  - algebraic properties
  - for symbolic computation improving asymptotic performance
  - added step by step to canonical, "obviously correct" implementation
- Alternative to embedding external DSL as abstract syntax
End of talk

Thank you!