Efficient Optimization of Iterative Queries

Leonidas Fegaras
Oregon Graduate Institute
A New Query Algebra Based on Folds

Folds

- can be defined for a large number of bulk data types;
- can capture most algebraic operators, such as \textit{map}, \textit{filter}, and \textit{join}.

Programming with folds supports:

- calculation-based program optimizations;
- deforestation and loop fusion;
- equational reasoning and theorem proving;
- program synthesis.

So \textbf{What is a Fold?}
**Algebraic Types**

A **fold** is the natural control structure for a freely constructed algebraic type (a sums-of-products type):

\[
T(\alpha_1, \ldots, \alpha_p) = \begin{cases} 
C_1(t_{1,1}, \ldots, t_{1,m_1}) \\
\vdots \\
C_n(t_{n,1}, \ldots, t_{n,m_n})
\end{cases}
\]

where \(\alpha_1, \ldots, \alpha_p\) are type variables;
the \(C_i\) are value constructors;
and \(t_{i,j}\) are either type variables
or instantiations of algebraic types
or the type \(T(\alpha_1, \ldots, \alpha_p)\) itself.
Examples of Algebraic Types

\[
\begin{align*}
\text{boolean} & \quad = \quad \text{False} \mid \text{True} \\
\text{prod}(\alpha, \beta) & \quad = \quad \text{Pair}(\alpha, \beta) \\
\text{list}(\alpha) & \quad = \quad \text{Nil} \mid \text{Cons}(\alpha, \text{list}(\alpha)) \\
\text{nat} & \quad = \quad \text{Zero} \mid \text{Succ}(\text{nat}) \\
\text{tree}(\alpha) & \quad = \quad \text{Tip}(\alpha) \mid \text{Node}(\text{tree}(\alpha), \text{tree}(\alpha)) \\
\text{bush}(\alpha) & \quad = \quad \text{Leaf}(\alpha) \mid \text{Branch}(\text{list}(\text{bush}(\alpha))) \\
\text{term}(\alpha) & \quad = \quad \text{Var}(\alpha) \mid \text{Lambda}(\alpha, \text{term}(\alpha)) \mid \text{Apply}(\text{term}(\alpha), \text{term}(\alpha)) \\
\text{person} & \quad = \quad \text{Make}\_\text{person}(\text{string, nat, string}) \\
\text{where} & \quad \text{string} \quad = \quad \text{list}(\text{nat})
\end{align*}
\]
The Fold Operator

• Lists:
  If \( g : \text{list}(\alpha) \rightarrow \beta \) can be expressed as:

  \[
  \begin{align*}
  g(\text{Nil}) &= f_n() \\
  g(\text{Cons}(a, l)) &= f_c(a, g(l))
  \end{align*}
  \]

  then \( g \) is a list fold: \( g = \text{fold}^{\text{list}}(f_n, f_c) \).

  \[
  \begin{align*}
  \text{fold}^{\text{list}}(f_n, f_c) \text{Nil} &= f_n() \\
  \text{fold}^{\text{list}}(f_n, f_c) \text{Cons}(a, l) &= f_c(a, \text{fold}^{\text{list}}(f_n, f_c) l)
  \end{align*}
  \]

• Natural numbers:

  \[
  \begin{align*}
  \text{fold}^{\text{nat}}(f_z, f_s) \text{Zero} &= f_z() \\
  \text{fold}^{\text{nat}}(f_z, f_s) \text{Succ}(i) &= f_s(\text{fold}^{\text{nat}}(f_z, f_s) i)
  \end{align*}
  \]

• Booleans:

  \[
  \begin{align*}
  \text{fold}^{\text{boolean}}(f_f, f_i) \text{False} &= f_f() \\
  \text{fold}^{\text{boolean}}(f_f, f_i) \text{True} &= f_i()
  \end{align*}
  \]

  Or \( \text{fold}^{\text{boolean}}(f_f, f_i) x \equiv \text{if } x \text{ then } f_i() \text{ else } f_f() \).

• Trees:

  \[
  \begin{align*}
  \text{fold}^{\text{tree}}(f_t, f_n) \text{Tip}(a) &= f_t(a) \\
  \text{fold}^{\text{tree}}(f_t, f_n) \text{Node}(l, r) &= f_n(\text{fold}^{\text{tree}}(f_t, f_n) l, \text{fold}^{\text{tree}}(f_t, f_n) r)
  \end{align*}
  \]
Example: List Append

\[
\text{append}(\text{Nil}, y) = y \\
\text{append}(\text{Cons}(a, l), y) = \text{Cons}(a, \text{append}(l, y))
\]

Append as a fold:

\[
\text{append}(x, y) = \text{fold}\text{\textsubscript{list}}(\lambda(). y, \lambda(a, r). \text{Cons}(a, r)) \ x
\]

Variable \( r \) is an *accumulative result variable*. 
Other Examples:

\[
\text{length}(x) = \text{fold}_{\text{list}}(\lambda().\text{Zero}, \lambda(a, r).\text{Succ}(r)) \; x
\]

\[
\text{reverse}(x) = \text{fold}_{\text{list}}(\lambda().\text{Nil}, \lambda(a, r).\text{append}(r, \text{Cons}(a, \text{Nil}))) \; x
\]

\[
x + y = \text{fold}_{\text{nat}}(\lambda().y, \lambda(r).\text{Succ}(r)) \; x
\]

\[
x \times y = \text{fold}_{\text{nat}}(\lambda().\text{Zero}, \lambda(r).y + r) \; x
\]

\[
\text{even}(x) = \text{fold}_{\text{nat}}(\lambda().\text{False}, \lambda(r).\text{not}(r)) \; x
\]

\[
x \land y = \text{fold}_{\text{boolean}}(\lambda().\text{False}, \lambda().y) \; x
\]

\[
\text{flat}(x) = \text{fold}_{\text{tree}}(\lambda(i).\text{Cons}(i, \text{Nil}), \lambda(l, r).\text{append}(l, r)) \; x
\]

\[
p.\text{address} = \text{fold}_{\text{person}}(\lambda(n, s, a).a) \; p
\]
The Fold Operator for any Algebraic Type $T$

For each constructor $C_i$ of $T$ we associate a functor $E_i^T$: e.g. Cons : $(\alpha \times \text{list}(\alpha)) \to \text{list}(\alpha)$

Then
\[
E_{\text{cons}}^{\text{list}(\alpha)}(S) = \alpha \times S
\]
\[
E_{\text{cons}}^{\text{list}(\alpha)}(g) = \text{id} \times g = \lambda(a, r). (a, g(r))
\]

The **fold** operator for any algebraic type $T$ is:
\[
\text{fold}^T(\bar{f}) \circ C_i = f_i \circ E_i^T(\text{fold}^T(\bar{f}))
\]
Promotion Theorems

- Lists:
  \[
  \begin{align*}
  \phi_n() &= g(f_n()) \\
  \phi_c(a, g(r)) &= g(f_c(a, r)) \\
  g(\text{fold}^{\text{list}}(f_n, f_c) x) &= \text{fold}^{\text{list}}(\phi_n, \phi_c) x
  \end{align*}
  \]

- Natural numbers:
  \[
  \begin{align*}
  \phi_z() &= g(f_z()) \\
  \phi_s(g(r)) &= g(f_s(r)) \\
  g(\text{fold}^{\text{nat}}(f_z, f_s) x) &= \text{fold}^{\text{nat}}(\phi_z, \phi_s) x
  \end{align*}
  \]

- Booleans:
  \[
  \begin{align*}
  \phi_f() &= g(f_f()) \\
  \phi_t() &= g(f_t()) \\
  g(\text{fold}^{\text{boolean}}(f_f, f_t) x) &= \text{fold}^{\text{boolean}}(\phi_f, \phi_t) x
  \end{align*}
  \]

Or equivalently:

\[
g(\text{if } x \text{ then } f_t() \text{ else } f_f()) = \text{if } x \text{ then } g(f_t()) \text{ else } g(f_f())
\]
The Promotion Theorem for any Algebraic Type $T$

\[ \forall i : \phi_i \circ E_i^T(g) = g \circ f_i \]
\[ g \circ \text{fold}^T(\overline{f}) = \text{fold}^T(\overline{\phi}) \]
Uniqueness Property

\[ \forall i : g \circ C_i = \phi_i \circ E_i^T(g) \iff g = \text{fold}^T(\phi) \]
Sets

Set constructors: Emptyset and Insert
Set selector: split

Set fold:

\[
fold^{set}(f_c, f_s) s = \begin{cases} 
    \text{if } s = \text{Emptyset} & \text{then } f_c() \\
    \text{else let } (a, r) = \text{split}(s) \text{ in } f_s(a, fold^{set}(f_c, f_s)(r)) 
\end{cases}
\]

For example:

\[
\text{union}(x, y) = fold^{set}(\lambda().y, \lambda(a, r).\text{Insert}(a, r)) \; x
\]

\[
\text{member}(e, x) = fold^{set}(\lambda().\text{False}, \lambda(a, r),(a = e) \; \text{or} \; r) \; x
\]

\[
\text{restrict}(s, f) = \\
fold^{set}(\lambda().\text{Emptyset}, \lambda(a, r).\text{if } f(a) \; \text{then } \text{Insert}(a, r) \; \text{else } r) \; s
\]

\[
\text{join}(x, y, M, J) = \\
fold^{set}(\lambda().\text{Emptyset}, \lambda(a, r).fold^{set}(\lambda().r, \lambda(b, s).\text{if } M(a, b) \; \text{then } \text{Insert}(J(a, b), s) \; \text{else } s) \; y) \; x
\]

For example,

\[
\text{join}(\text{employees, departments,} \\
\lambda(\text{emp,dept}).(\text{emp.dno}=\text{dept.dno} \; \text{and} \; \text{dept.name}=\text{"CSE"}), \\
\lambda(\text{emp,dept}).\text{emp})
\]
A function $f$ is **commutative-idempotent** if:

$$\forall m \forall n \forall s : f(m, f(n, s)) = f(n, f(m, s)) \quad (\text{commutativity})$$

$$\forall n \forall s : f(n, f(n, s)) = f(n, s) \quad (\text{idempotence})$$

If a function $f_s$ is commutative-idempotent then

$$\text{fold}^{set}(f_e, f_s)(\text{Insert}(a, s)) = f_{s}(a, \text{fold}^{set}(f_e, f_s)s)$$

**Promotion Theorem for Sets:**

$$\phi_e() = g(f_e())$$

$$\phi_s(a, g(r)) = g(f_s(a, r))$$

$$g(\text{fold}^{set}(f_e, f_s)x) = \text{fold}^{set}(\phi_e, \phi_s)x$$

and if $f_s$ is commutative-idempotent then so is $\phi_s$. 
The Term Language

A term $\tau$ in the language has one of the following forms:

- **variable**: $x$;
- **construction**: $C(\tau_1, \ldots, \tau_n)$;
- **fold**: $\text{fold}^T(f_1, \ldots, f_n) \tau$,
  where each $f_i$ has the form $\lambda(x_1, \ldots, x_m).\tau_i$.

**Safe Term**: A program in the term language is *safe* if it does not contain terms $\text{fold}^T(\overline{f}) \ t$, where $t$ contains a reference to an *accumulative result variable*.

E.g. this is *safe*:

$$\text{append}(x, y) = \text{fold}^\text{list}(\lambda() \cdot y, \lambda(a, r).\text{Cons}(a, r)) \ x$$

E.g. this is *NOT safe*:

$$\text{reverse}(x) = \text{fold}^\text{list}(\lambda() \cdot \text{Nil}, \lambda(a, r).\text{append}(r, \text{Cons}(a, \text{Nil}))) \ x$$

$$= \text{fold}^\text{list}(\lambda() \cdot \text{Nil},$$

$$\lambda(a, r).\text{fold}^\text{list}(\lambda() \cdot \text{Cons}(a, \text{Nil}),$$

$$\lambda(b, s).\text{Cons}(b, s)) \ r) \ x$$


**Canonical Terms**

A *canonical term* is a term in which

- all folds in the term are over variables;
- none of these variables are accumulative result variables.

**Theorem:** Any safe term can be transformed into a canonical form.
The Normalization Algorithm (Example)

We will improve \( \text{length}(\text{append}(x, y)) \), where:

\[
\text{length}(x) = \text{fold}^{\text{list}}(\lambda(). \text{Zero}, \lambda(a, r). \text{Succ}(r)) \ x
\]

\[
\text{append}(x, y) = \text{fold}^{\text{list}}(f_n, f_c) \ x \quad \text{where} \quad \begin{cases} f_n = \lambda(). y \\ f_c = \lambda(a, r). \text{Cons}(a, r) \end{cases}
\]

We need to find some

\[
\text{fold}^{\text{list}}(\phi_n, \phi_c) \ x = \text{length}(\text{append}(x, y))
\]

The \textit{list} promotion theorem is:

1) \( \phi_n() = \text{g}(f_n()) \)

2) \( \phi_c(a, g(r)) = \text{g}(f_c(a, r)) \)

\[
g(\text{fold}^{\text{list}}(f_n, f_c) \ x) = \text{fold}^{\text{list}}(\phi_n, \phi_c) \ x
\]

We apply this theorem with \( g = \text{length} \):

1) \( \phi_n() = \text{length}(f_n()) \)
   \[
   = \text{length}(y)
   \]

2) \( \phi_c(a, \text{length}(r)) = \text{length}(f_c(a, r)) \)
   \[
   = \text{length}(\text{Cons}(a, r))
   = \text{Succ}(\text{length}(r))
   \]

\[
\Rightarrow \phi_c(a, u) = \text{Succ}(u)
\]

Therefore:

\[
\text{length}(\text{append}(x, y)) = \text{fold}^{\text{list}}(\lambda(). \text{length}(y), \lambda(a, u). \text{Succ}(u)) \ x
\]
The Normalization Algorithm

The normalization algorithm is a calculation-based, \( \mathcal{O}(n \log n) \), algorithm that fuses piped folds:

\[
\text{fold}^S(g)(\text{fold}^T(f)x) \longrightarrow \text{fold}^T(\phi)x
\]

It facilitates the following:

- automation of the unfold-simplify-fold method;
- automation of the techniques of deforestation and loop fusion;
- generalization of many well-known algebraic optimizations
  e.g. \( \text{map}(f) \circ \text{map}(g) = \text{map}(f \circ g) \)
  and \( \text{restrict}(p_1) \circ \text{restrict}(p_2) = \text{restrict}(p_1 \land p_2) \);
- automation of a form of partial evaluation;
- inductive theorem proving;
- implementation of most stream-based pipelined techniques during optimization time (instead of during evaluation time).
The Fold Optimization Algorithm

Any program that computes $f$

Safe programs that compute $f$

Canonical programs that compute $f$

all $a$ in $x$: $P(a)$ and $Q(a)$

$\text{select}(P \land Q)(x)$

$\text{select}(P)\text{select}(Q)(x))$
The **fold optimization algorithm** is a search-based algorithm that enumerates all canonical programs that satisfy an equation: e.g. find $f$ in $g \circ f = \phi$ given $g$ and $\phi$.

- it performs program synthesis based on second-order pattern matching;
- most alternatives are derived from commutative operations (not from associative);
- the search tree is a long (sometimes infinite) but not a bushy tree (max branch factor is 3);
- it can be guided by heuristics and cost functions.
**Type Transformation**

Database implementation is the translation of *abstract functions* or queries that operate on abstract values into efficient *concrete algorithms* that manipulate storage structures.

The **type transformation model**:

- the designer specifies how abstract types are mapped into storage structures by providing *abstraction functions*;
- the translator uses this information to translate abstract functions into concrete programs.

\[
\begin{align*}
T_1 \times \cdots \times T_n & \xrightarrow{F} T \\
T_1 \times \cdots \times T_n & \xrightarrow{f} t \quad \text{abstract layer} \\
T_1 \times \cdots \times T_n & \xrightarrow{r_0} r_0 \\
T_1 \times \cdots \times T_n & \xrightarrow{r_1 \times \cdots \times r_n} r_1 \times \cdots \times r_n
\end{align*}
\]

The concrete function \( F \) is an *implementation* of the abstract function \( f \) iff:

\[
r_0 \circ F = f \circ (r_1 \times \cdots \times r_n)
\]

Type transformation supports a high degree of *data independence*: abstract values and their operations can be significantly independent of their implementations.
**Example: Translating Set Operations**

Abstract Operation:

\[
\text{union}(x, y) = \text{fold}^{set}(\lambda().y, \lambda(a, r).\text{Insert}(a, r)) x
\]

Abstraction function:

\[
R = \text{fold}^{list}(\lambda().\text{Emptyset}, \lambda(a, r).\text{Insert}(a, r))
\]

Any implementation UNION of union must satisfy:

\[
R \circ \text{UNION} = \text{union} \circ (R \times R)
\]

or equivalently:

\[
\forall x, y : R(\text{UNION}(x, y)) = \text{union}(R(x), R(y))
\]

E.g. one solution for UNION is list append.
Conclusion

- folds are uniformly defined over a wide spectrum of bulk data structures, which includes sets, lists, trees, tuples, numbers, and booleans;
- canonical programs are expressive enough to capture a large class of queries;
- the explicit iteration structure of fold programs supports calculation-based algebraic optimizations;
- there are very few canonical programs that compute a function, so search for the best is feasible in most cases;
- the fold optimization algorithm is a search-based algorithm that explores the reduced space of equivalent canonical forms.