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 *map*, *filter*, and *join*.

Programming with folds supports:

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

So What is a Fold?

Algebraic Types

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

$$\begin{array}{rcl} T(\alpha_1,\ldots,\alpha_p) & = & C_1(t_{1,1},\ldots,t_{1,m_1}) \\ & & \mid \; \cdots \\ & & \mid \; C_n(t_{n,1},\ldots,t_{n,m_n}) \end{array}$$

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

$$\mathrm{boolean} \quad = \quad \mathrm{False} \ \mid \ \mathrm{True}$$

$$\mathrm{prod}(\alpha, \beta) \quad = \quad \mathrm{Pair}(\alpha, \beta)$$

$$\mathrm{list}(\alpha) \quad = \quad \mathrm{Nil} \ \mid \ \mathrm{Cons}(\alpha, \mathrm{list}(\alpha))$$

$$\mathrm{nat} \quad = \quad \mathrm{Zero} \ \mid \ \mathrm{Succ}(\mathrm{nat})$$

$$\mathrm{tree}(\alpha) \quad = \quad \mathrm{Tip}(\alpha) \ \mid \ \mathrm{Node}(\mathrm{tree}(\alpha), \mathrm{tree}(\alpha))$$

$$\mathrm{bush}(\alpha) \quad = \quad \mathrm{Leaf}(\alpha) \ \mid \ \mathrm{Branch}(\mathrm{list}(\mathrm{bush}(\alpha)))$$

$$\mathrm{term}(\alpha) \quad = \quad \mathrm{Var}(\alpha) \ \mid \ \mathrm{Lambda}(\alpha, \mathrm{term}(\alpha)) \ \mid \ \mathrm{Apply}(\mathrm{term}(\alpha), \mathrm{term}(\alpha))$$

$$\begin{aligned} \mathrm{person} \quad = \quad & \mathrm{Make\_person}(\mathrm{string}, \mathrm{nat}, \mathrm{string}) \\ & \mathrm{where\ string} \ = \ \mathrm{list}(\mathrm{nat}) \end{aligned}$$

The Fold Operator

- **Lists:**
  If $g:\ \mathrm{list}(\alpha) \rightarrow \beta$ can be expressed as:

$$\begin{array}{rcl} g(\mathrm{Nil}) & = & f_n() \\ g(\mathrm{Cons}(a, l)) & = & f_c(a, g(l)) \end{array}$$

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

$$\begin{array}{rcl} \mathrm{fold}^{list}(f_n, f_c)\,\mathrm{Nil} & = & f_n() \\ \mathrm{fold}^{list}(f_n, f_c)\,\mathrm{Cons}(a, l) & = & f_c(a, \mathrm{fold}^{list}(f_n, f_c)\, l) \end{array}$$

- **Natural numbers:**

$$\begin{array}{rcl} \mathrm{fold}^{nat}(f_z, f_s)\,\mathrm{Zero} & = & f_z() \\ \mathrm{fold}^{nat}(f_z, f_s)\,\mathrm{Succ}(i) & = & f_s(\mathrm{fold}^{nat}(f_z, f_s)\, i) \end{array}$$

- **Booleans:**

$$\begin{array}{rcl} \mathrm{fold}^{boolean}(f_f, f_t)\,\mathrm{False} & = & f_f() \\ \mathrm{fold}^{boolean}(f_f, f_t)\,\mathrm{True} & = & f_t() \end{array}$$

  Or $\quad \mathrm{fold}^{boolean}(f_f, f_t)\, x \ \equiv\ \mathbf{if}\, x\, \mathbf{then}\, f_t()\, \mathbf{else}\, f_f().$

- **Trees:**

$$\begin{array}{rcl} \mathrm{fold}^{tree}(f_t, f_n)\,\mathrm{Tip}(a) & = & f_t(a) \\ \mathrm{fold}^{tree}(f_t, f_n)\,\mathrm{Node}(l, r) & = & f_n(\mathrm{fold}^{tree}(f_t, f_n)\, l, \mathrm{fold}^{tree}(f_t, f_n)\, r) \end{array}$$

Example: List Append

$$\begin{array}{lcl} \mathrm{append}(\mathrm{Nil}, y) & = & y \\ \mathrm{append}(\mathrm{Cons}(a, l), y) & = & \mathrm{Cons}(a, \mathrm{append}(l, y)) \end{array}$$

Append as a fold:

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

Variable r is an *accumulative result variable.*

Other Examples:

$$\begin{aligned}
\mathrm{length}(x) &= \mathrm{fold}^{list}(\lambda().\mathrm{Zero}, \lambda(a,r).\mathrm{Succ}(r))\ x \\
\mathrm{reverse}(x) &= \mathrm{fold}^{list}(\lambda().\mathrm{Nil}, \lambda(a,r).\mathrm{append}(r, \mathrm{Cons}(a, \mathrm{Nil})))\ x \\
x + y &= \mathrm{fold}^{nat}(\lambda().y, \lambda(r).\mathrm{Succ}(r))\ x \\
x \times y &= \mathrm{fold}^{nat}(\lambda().\mathrm{Zero}, \lambda(r).y + r)\ x \\
\mathrm{even}(x) &= \mathrm{fold}^{nat}(\lambda().\mathrm{False}, \lambda(r).\mathrm{not}(r))\ x \\
x \wedge y &= \mathrm{fold}^{boolean}(\lambda().\mathrm{False}, \lambda().y)\ x \\
\mathrm{flat}(x) &= \mathrm{fold}^{tree}(\lambda(i).\mathrm{Cons}(i, \mathrm{Nil}), \lambda(l,r).\mathrm{append}(l,r))\ x \\
p.\mathrm{address} &= \mathrm{fold}^{person}(\lambda(n,s,a).a)\ p
\end{aligned}$$

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 : $(\alpha \times \mathrm{list}(\alpha)) \rightarrow \mathrm{list}(\alpha)$
Then

$$E_{cons}^{list(\alpha)}(S) = \alpha \times S$$

$$E_{cons}^{list(\alpha)}(g) = \mathrm{id} \times g = \lambda(a, r).(a, g(r))$$

The **fold** operator for any algebraic type T is:

$$\mathrm{fold}^T(\overline{f}) \circ C_i = f_i \circ E_i^T(\mathrm{fold}^T(\overline{f}))$$

$$\begin{tikzcd}
T & E_i^T(T) \arrow[l, "C_i"'] \arrow[d, "E_i^T(\mathrm{fold}^T(\overline{f}))"] \\
\beta & E_i^T(\beta) \arrow[l, "f_i"]
\arrow[from=1-1, to=2-1, "\mathrm{fold}^T(\overline{f})"']
\end{tikzcd}$$

Promotion Theorems

- **Lists:**

$$\frac{\begin{array}{rcl}\phi_n() & = & g(f_n()) \\ \phi_c(a, g(r)) & = & g(f_c(a, r))\end{array}}{g(\mathrm{fold}^{list}(f_n, f_c)\, x) \;=\; \mathrm{fold}^{list}(\phi_n, \phi_c)\, x}$$

- **Natural numbers:**

$$\frac{\begin{array}{rcl}\phi_z() & = & g(f_z()) \\ \phi_s(g(r)) & = & g(f_s(r))\end{array}}{g(\mathrm{fold}^{nat}(f_z, f_s)\, x) \;=\; \mathrm{fold}^{nat}(\phi_z, \phi_s)\, x}$$

- **Booleans:**

$$\frac{\begin{array}{rcl}\phi_f() & = & g(f_f()) \\ \phi_t() & = & g(f_t())\end{array}}{g(\mathrm{fold}^{boolean}(f_f, f_t)\, x) \;=\; \mathrm{fold}^{boolean}(\phi_f, \phi_t)\, x}$$

  Or equivalently:

$$g(\mathbf{if}\; x \;\mathbf{then}\; f_t() \;\mathbf{else}\; f_f()) \;=\; \mathbf{if}\; x \;\mathbf{then}\; g(f_t()) \;\mathbf{else}\; g(f_f())$$

The Promotion Theorem for any Algebraic Type T

$$\frac{\forall i:\ \phi_i \circ E_i^T(g) \ = \ g \circ f_i}{g \circ \mathrm{fold}^T(\overline{f}) \ = \ \mathrm{fold}^T(\overline{\phi})}$$

$$\begin{tikzcd}
T & E_i^T(T) \arrow[l, "C_i"'] \arrow[d, "E_i^T(\mathrm{fold}^T(\overline{f}))"] \\
\beta \arrow[from=1-1, to=2-1, "\mathrm{fold}^T(\overline{f})"'] \arrow[d, "g"'] & E_i^T(\beta) \arrow[l, "f_i"] \arrow[d, "E_i^T(g)"] \\
\gamma & E_i^T(\gamma) \arrow[l, "\phi_i"]
\end{tikzcd}$$

Uniqueness Property

$$\forall i:\ \ g \circ C_i \ = \ \phi_i \circ E_i^T(g) \ \Longleftrightarrow \ g \ = \ \mathrm{fold}^T(\overline{\phi})$$

Sets

Set constructors: Emptyset and Insert
Set selector: split

Set fold:

$$\mathrm{fold}^{set}(f_e, f_s)\, s \;=\; \begin{cases} \textbf{if } s = \mathrm{Emptyset} \textbf{ then } f_e() \\ \textbf{else let } (a, r) = \mathrm{split}(s) \textbf{ in } f_s(a, \mathrm{fold}^{set}(f_e, f_s)\, r) \end{cases}$$

For example:

$$\begin{aligned} \textbf{union}(x, y) &= \mathrm{fold}^{set}(\lambda().y, \lambda(a, r).\mathrm{Insert}(a, r))\; x \\ \textbf{member}(e, x) &= \mathrm{fold}^{set}(\lambda().\mathrm{False}, \lambda(a, r).(a = e) \textbf{ or } r)\; x \end{aligned}$$

$$\begin{aligned} &\textbf{restrict}(s, f) \;= \\ &\quad \mathrm{fold}^{set}(\lambda().\mathrm{Emptyset}, \\ &\qquad\qquad \lambda(a, r).\textbf{if } f(a) \textbf{ then } \mathrm{Insert}(a, r) \textbf{ else } r)\; s \end{aligned}$$

$$\begin{aligned} &\textbf{join}(x, y, M, J) \;= \\ &\quad \mathrm{fold}^{set}(\lambda().\mathrm{Emptyset}, \\ &\qquad\qquad \lambda(a, r).\mathrm{fold}^{set}(\lambda().r, \lambda(b, s).\textbf{if } M(a, b) \textbf{ then } \mathrm{Insert}(J(a, b), s) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } s)\; y)\; x \end{aligned}$$

For example,

join(employees, departments,
λ(emp,dept).(emp.dno=dept.dno **and** dept.name="CSE"),
λ(emp,dept).emp)

A function f is **commutative-idempotent** if:

$$\begin{array}{lrcll} \forall m\,\forall n\,\forall s: & f(m, f(n,s)) & = & f(n, f(m,s)) & (\textbf{commutativity}) \\ \forall n\,\forall s: & f(n, f(n,s)) & = & f(n,s) & (\textbf{idempotence}) \end{array}$$

If a function f_s is commutative-idempotent then

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

Promotion Theorem for Sets:

$$\frac{\begin{array}{lcl} \phi_e() & = & g(f_e()) \\ \phi_s(a, g(r)) & = & g(f_s(a,r)) \end{array}}{g(\mathrm{fold}^{set}(f_e, f_s)\,x) \;=\; \mathrm{fold}^{set}(\phi_e, \phi_s)\,x}$$

and if f_s is commutative-idempotent then so is ϕ_s.

The Term Language

A term τ in the language has one of the following forms:

- **variable:** x;
- **construction:** $C(\tau_1, \ldots, \tau_n)$;
- **fold:** $\mathrm{fold}^T(f_1, \ldots, f_n)\,\tau$,
  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 $\mathrm{fold}^T(\overline{f})\,t$, where t contains a reference to an *accumulative result variable.*

e.g. this is *safe*:

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

e.g. this is *NOT safe*:

$$\begin{aligned}\mathrm{reverse}(x) \;&=\; \mathrm{fold}^{list}(\lambda().\mathrm{Nil}, \lambda(a, r).\mathrm{append}(r, \mathrm{Cons}(a, \mathrm{Nil})))\; x\\ &=\; \mathrm{fold}^{list}(\lambda().\mathrm{Nil},\\ &\qquad\qquad \lambda(a, r).\mathrm{fold}^{list}(\lambda().\mathrm{Cons}(a, \mathrm{Nil}),\\ &\qquad\qquad\qquad\qquad \lambda(b, s).\mathrm{Cons}(b, s))\; r)\; x\end{aligned}$$

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 length(append(x, y)), where:

$$\begin{array}{lcl} \text{length}(x) & = & \text{fold}^{list}(\lambda().\text{Zero}, \lambda(a,r).\text{Succ}(r))\, x \\ \text{append}(x,y) & = & \text{fold}^{list}(f_n, f_c)\, x \qquad \text{where} \left\{ \begin{array}{lcl} f_n & = & \lambda().y \\ f_c & = & \lambda(a,r).\text{Cons}(a,r) \end{array} \right. \end{array}$$

We need to find some

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

The *list* promotion theorem is:

$$\frac{\begin{array}{lcl} 1)\;\; \phi_n() & = & g(f_n()) \\ 2)\;\; \phi_c(a, g(r)) & = & g(f_c(a,r)) \end{array}}{g(\text{fold}^{list}(f_n, f_c)\, x) \;=\; \text{fold}^{list}(\phi_n, \phi_c)\, x}$$

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

$$\begin{array}{lcl} 1)\;\; \phi_n() & = & \text{length}(f_n()) \\ & = & \text{length}(y) \\ \\ 2)\;\; \phi_c(a, \text{length}(r)) & = & \text{length}(f_c(a,r)) \\ & = & \text{length}(\text{Cons}(a,r)) \\ & = & \text{Succ}(\text{length}(r)) \end{array}$$

$$\Rightarrow \phi_c(a,u) \;=\; \text{Succ}(u)$$

Therefore:

$$\text{length}(\text{append}(x,y)) \;=\; \text{fold}^{list}(\lambda().\text{length}(y), \lambda(a,u).\text{Succ}(u))\, x$$

The Normalization Algorithm

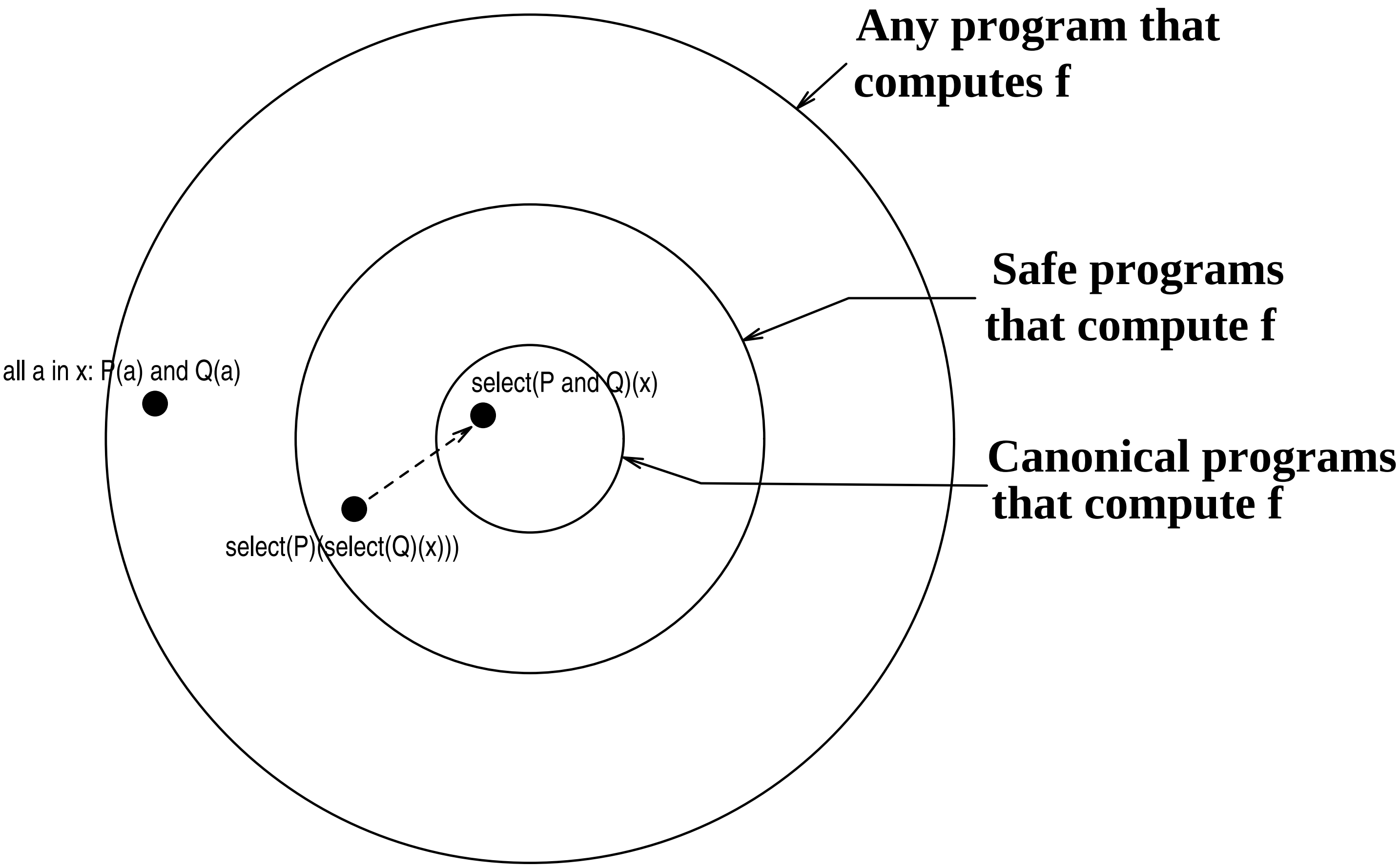
The normalization algorithm is a calculation-based, $\mathcal{O}(n \log n)$, algorithm that fuses piped folds:

$$\mathrm{fold}^S(\overline{g})\,(\mathrm{fold}^T(\overline{f})\,x) \;\longrightarrow\; \mathrm{fold}^T(\overline{\phi})\,x$$

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. $\mathrm{map}(f) \circ \mathrm{map}(g) \;=\; \mathrm{map}(f \circ g)$
  and $\mathrm{restrict}(p_1) \circ \mathrm{restrict}(p_2) \;=\; \mathrm{restrict}(p_1 \wedge p_2)$;
- 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



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 ϕ.

- 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.

$$\begin{tikzcd}
t_1 \times \cdots \times t_n \arrow[r, "f"] & t & \textbf{abstract layer} \\
T_1 \times \cdots \times T_n \arrow[r, "F"'] \arrow[u, "r_1 \times \cdots \times r_n"'] & T \arrow[u, "r_0"'] & \textbf{concrete layer}
\end{tikzcd}$$

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:

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

$$\begin{tikzcd}
\text{set * set} \arrow[r, "\text{union}"] & \text{set} \\
\text{list * list} \arrow[u, "R * R"'] \arrow[r, "\text{UNION}"'] & \text{list} \arrow[u, "R"']
\end{tikzcd}$$

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.