Improving Programs which Recurse over Multiple Inductive Structures

Leonidas Fegaras
Tim Sheard
Tong Zhou

Oregon Graduate Institute
Evolution of Control Structure

**Imperative Languages**

- goto
  - structured control
    - (e.g. while loops, if-then-else)

**Functional Languages**

- recursion
  - generic recursion schemes
    - (e.g. fold, map, reduce)
Goals

- describe a *generic recursion scheme* that captures patterns of recursion for inducting over multiple structures;
- provide a normalization algorithm that improves these programs;
- describe how ordinary recursive programs can be translated into these forms.
What is a Generic Reduction Scheme?

• Binary trees:
  \[
  \text{tree}(\alpha, \beta) = \begin{cases} 
    \text{Tip of } \alpha \\
    \text{Node of } \beta \times \text{tree}(\alpha, \beta) \times \text{tree}(\alpha, \beta)
  \end{cases}
  \]

  The tree reduction scheme is:
  \[
  F(\text{Tip}(a)) = f_1(a) \\
  F(\text{Node}(i, l, r)) = f_2(i, F(l), F(r))
  \]

  We give this scheme a name: \[ F = \text{red}^{\text{tree}}(f_1, f_2) \]

• Lists:
  \[
  \text{list}(\alpha) = \text{Nil} \mid \text{Cons of } \alpha \times \text{list}(\alpha)
  \]

  The list reduction scheme is:
  \[
  F(\text{Nil}) = f_1() \\
  F(\text{Cons}(a, s)) = f_2(a, F(s))
  \]

  where \[ F = \text{red}^{\text{list}}(f_1, f_2) \]

How do we abstract the patterns of recursions of any data type?
The Functor $E^T_c$

- **Binary trees:**
  \[
  F(\text{Node}(i, l, r)) = f_2(E^{\text{tree}}_{\text{Node}}(F)(i, l, r))
  \]
  where
  \[
  E^{\text{tree}}_{\text{Node}}(F)(i, l, r) = (i, F(l), F(r))
  \]
  The functor $E^{\text{tree}}_{\text{Node}}(F)$ pushes $F$ onto some of the constructor’s arguments.

- **Lists:**
  \[
  F(\text{Cons}(a, s)) = f_2(E^{\text{list}}_{\text{Cons}}(F)(a, s))
  \]
  where
  \[
  E^{\text{list}}_{\text{Cons}}(F)(a, s) = (a, F(s))
  \]

For most data types $T$ and for each value constructor $C$ of $T$ we can calculate the functor $E^T_c(F)$:
\[
E^T_c(F)(x_1, x_2, \ldots, x_n) = (??(x_1), ??(x_2), \ldots, ??(x_n))
\]
Unary Reduction

For each constructor $C$ of $T$:

$$\text{red}^T(\mathcal{F}) \circ C = f_c \circ E_c^T(\text{red}^T(\mathcal{F}))$$

e.g., for tree:

$$\text{red}^{\text{tree}}(f_1, f_2)(\text{Tip}(a)) = f_1(a)$$

$$\text{red}^{\text{tree}}(f_1, f_2)(\text{Node}(i, l, r)) = f_2(i, \text{red}^{\text{tree}}(f_1, f_2)l, \text{red}^{\text{tree}}(f_1, f_2)r)$$
Examples

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

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

\[
x + y = \text{red}^{\text{nat}}(\lambda().y, \lambda(r).\text{Succ}(r)) \ x
\]

\[
\text{if } x \text{ then } y \text{ else } z = \text{red}^{\text{boolean}}(\lambda().z, \lambda().y) \ x
\]

\[
\text{reflect_bush}(x) = \text{red}^{\text{bush}}(\lambda(a).\text{Leaf}(a),
\quad \text{\quad } \lambda(r).\text{Branch}(\text{reverse}(r))) \ x
\]
Unary Reductions can Capture Multiple Traversals

List structural equality

\[
\begin{align*}
\text{listeq}(\text{Nil}, \text{Nil}) &= \text{True} \\
\text{listeq}(\text{Nil}, \text{Cons}(b, s)) &= \text{False} \\
\text{listeq}(\text{Cons}(a, l), \text{Nil}) &= \text{False} \\
\text{listeq}(\text{Cons}(a, l), \text{Cons}(b, s)) &= (a = b) \land \text{listeq}(l, s)
\end{align*}
\]

can be expressed as a second-order unary reduction:

\[
\text{red}^{\text{list}}(\lambda().\lambda k.(k = \text{Nil}), \\
\lambda(a, r).\lambda k.\text{case } k \text{ of} \\
\quad \text{Nil} \Rightarrow \text{False} \\
\quad | \text{Cons}(b, s) \Rightarrow (a = b) \land r(s)) \\x y
\]

This is not symmetrical. 
Lack of symmetry causes problems when improving programs.
Binary Reduction Schemes

Simultaneous traversal of a list and a tree:

\[ F(\text{Nil}, \text{Tip}(b)) = f_1(b) \]
\[ F(\text{Nil}, \text{Node}(i, l, r)) = f_2(i, l, r) \]
\[ F(\text{Cons}(a, s), \text{Tip}(b)) = f_3(a, s, b) \]
\[ F(\text{Cons}(a, s), \text{Node}(i, l, r)) = f_4(a, i, F(s, l), F(s, r)) \]

As before, we want to find \( E_{\text{Cons} \times \text{Node}}^{\text{list} \times \text{tree}}(F) \) such that

\[ F \circ (\text{Cons} \times \text{Node}) = f_4 \circ E_{\text{Cons} \times \text{Node}}^{\text{list} \times \text{tree}}(F) \]

**Problem:**

Calculate \( E_{\text{Cons} \times \text{Node}}^{\text{list} \times \text{tree}} \) from \( E_{\text{Cons}}^{\text{list}} \) and \( E_{\text{Node}}^{\text{tree}} \)
**Generalization of \( E \)**

\( E_{c_1 \times c_2}^{T_1 \times T_2} \) is calculated by “nesting” \( E_{c_2}^{T_2} \) inside \( E_{c_1}^{T_1} \):

\[
E_{c_1 \times c_2}^{T_1 \times T_2}(f) = \lambda(x, y).E_{c_1}^{T_1}(\lambda x.E_{c_2}^{T_2}(\lambda w.f(z, w)) y) x
\]

e.g.

\[
E_{Cons \times Node}^{list \times tree}(f)(x, y) = E_{Cons}^{list}(\lambda z.E_{Node}^{tree}(\lambda w.f(z, w)) y) x
\]

where

\[
E_{Cons}^{list}(G)(a, s) = (a, [G(s)])
\]

\[
E_{Node}^{tree}(H)(i, l, r) = (i, [H(l), H(r)])
\]

If we set \( x = (a, s) \) and \( y = (i, l, r) \) we get:

\[
E_{Cons \times Node}^{list \times tree}(F)((a, s), (i, l, r)) = (a, (i, [F(s, l), F(s, r)]))
\]

Thus,

\[
F(Cons(a, s), Node(i, l, r)) = f_4(a, (i, F(s, l), F(s, r)))
\]
**Binary Reduction**

\[
\text{red}^{T_1 \times T_2}(\overline{f}) \circ (C_1 \times C_2) = f_{c_1 \times c_2} \circ E^{T_1 \times T_2}_{c_1 \times c_2}(\text{red}^{T_1 \times T_2}(\overline{f}))
\]

e.g., the binary reduction operator for \(\text{list} \times \text{list}\) is

\[
F = \text{red}^{\text{list} \times \text{list}}(f_{nn}, f_{nc}, f_{cn}, f_{cc})
\]
defined as:

- \(F(\text{Nil}, \text{Nil}) = f_{nn}((), ())\)
- \(F(\text{Nil}, \text{Cons}(b, s)) = f_{nc}((), (b, s))\)
- \(F(\text{Cons}(a, l), \text{Nil}) = f_{cn}((a, l), ())\)
- \(F(\text{Cons}(a, l), \text{Cons}(b, s)) = f_{cc}(a, (b, F(l, s)))\)

e.g.

\[
\text{listeq}(x, y) = \text{red}^{\text{list} \times \text{list}}(\lambda((), ()).\text{True}, \\
\lambda((), (b, s)).\text{False}, \\
\lambda((a, l), ()).\text{False}, \\
\lambda(a, (b, r)).r \land (a = b))(x, y)
\]

\[
\text{zip}(x, y) = \text{red}^{\text{list} \times \text{list}}(\lambda((), ()).\text{Nil}, \\
\lambda((), (b, s)).\text{Nil}, \\
\lambda((a, l), ()).\text{Nil}, \\
\lambda(a, (b, r)).\text{Cons}((a, b), r))(x, y)
\]
Program Improvement

Suppose that we want to improve \(\text{length}(\text{zip}(x, y))\), where

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

\[
\text{zip}(x, y) = \text{red}^{\text{list}\times\text{list}}(\lambda(()), (()).\text{Nil}, \\
\lambda(()), (b, s)).\text{Nil}, \\
\lambda((a, l), ()).\text{Nil}, \\
\lambda(a, (b, r)).\text{Cons}((a, b), r)) (x, y)
\]

length can be \textit{fused} with \text{zip} into a binary reduction:

\[
\text{length}(\text{zip}(x, y)) \rightarrow \text{red}^{\text{list}\times\text{list}}(\lambda().\text{Zero}, \\
\lambda(()), (b, s)).\text{Zero}, \\
\lambda((a, l), ()).\text{Zero}, \\
\lambda(a, (b, r)).\text{Succ}(r))(x, y)
\]

How do we achieve this loop-fusion automatically?
Promotion Theorem

\[ \begin{align*}
\phi_{nn}((\_), (\_)) &= g(f_{nn}((\_), (\_))) \\
\phi_{nc}((\_), (b, s)) &= g(f_{nc}((\_), (b, s))) \\
\phi_{cn}((a, l), ()) &= g(f_{cn}((a, l), ())) \\
\phi_{cc}(a, (b, g(r))) &= g(f_{cc}(a, (b, r)))
\end{align*} \]

\[
g(\text{red}^{\text{list} \times \text{list}}(f_{nn}, f_{nc}, f_{cn}, f_{cc}) (x, y)) = \text{red}^{\text{list} \times \text{list}}(\phi_{nn}, \phi_{nc}, \phi_{cn}, \phi_{cc}) (x, y)
\]
Promotion Theorems can be Expressed for any Type $T$

Promoting a unary function $g$ through a unary reduction:

$$\forall c : \phi_c \circ E^T_c(g) = g \circ f_c$$

$$g \circ \text{red}^T(f) = \text{red}^T(\phi)$$

Promoting a unary function $g$ through a binary reduction:

$$\forall c_1, \forall c_2 : \phi_{c_1 \times c_2} \circ E^T_{c_1}(E^T_{c_2}(g)) = g \circ f_{c_1 \times c_2}$$

$$g \circ \text{red}^{T_1 \times T_2}(f) = \text{red}^{T_1 \times T_2}(\phi)$$

A general form of promotion theorem is given in the paper.
Program Improvement

The promotion theorems can be used for loop fusion:

\[1) \quad \phi_{nn}((), ()) = g(f_{nn}((), ()))
\]
\[2) \quad \phi_{nc}((), (b, s)) = g(f_{nc}((), (b, s)))
\]
\[3) \quad \phi_{cn}((a, l), ()) = g(f_{cn}((a, l), ()))
\]
\[4) \quad \phi_{cc}(a, (b, g(r))) = g(f_{cc}(a, (b, r)))
\]

\[
g(\text{red}^{\text{list} \times \text{list}}(f_{nn}, f_{nc}, f_{cn}, f_{cc}) \ (x, y))
\]
\[
= \text{red}^{\text{list} \times \text{list}}(\phi_{nn}, \phi_{nc}, \phi_{cn}, \phi_{cc}) \ (x, y)
\]

How do we solve equation \#4?

- distribute \( g \) in \( g(f_{cc}(a, (b, r))) \) downwards over \( f_{cc} \) until it reaches \( r \). Then \( g(f_{cc}(a, (b, r))) \) is recast into a form \( \mathcal{H}(a, (b, g(r))) \) where every \( r \) has a \( g \) attached to it.

- generalize \( g(r) \) into a new variable \( s \) and set \( \phi_{cc}(a, (b, s)) = \mathcal{H}(a, (b, s)) \).

This loop fusion is not always possible.
How do we perform the promotion/generalization task automatically?

**Trick:**

- we put a stopping point $STOP[g]$ on each $r$ in the rhs of equation #4:

  $$\phi_{cc}(a, (b, r)) = g(f_{cc}(a, (b, STOP[g](r))))$$

- if we find a $g(STOP[g](r))$ during the transformation of $g(f_{cc}(a, (b, STOP[g](r))))$, then it is fused to $r$.

**When this trick works?**

- if all $STOP[g]$ are eliminated in $g(f_{cc}(a, (b, STOP[g](r))))$ after transformation, then $\text{red}^{list\times list}(\phi_{nn}, \phi_{nc}, \phi_{cn}, \phi_{cc})$ preserves the meaning of $g\text{red}^{list\times list}(f_{nn}, f_{nc}, f_{cn}, f_{cc})$;

- otherwise $g(\text{red}^{list}(f_n, f_c) x)$ should be left as is.

The same trick can be used for any promotion theorem.
Example of a Program Improvement

Improving \text{length}(\text{zip}(x, y)):

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

\[
\text{zip}(x, y) = \text{red}^{\text{list}\times\text{list}}(f_{nn}, f_{nc}, f_{cn}, f_{cc})(x, y)
\]
\[
= \text{red}^{\text{list}\times\text{list}}(\lambda((), ()).\text{Nil},
\lambda((), (b, s)).\text{Nil},
\lambda((a, l), ()).\text{Nil},
\lambda(a, (b, r)).\text{Cons}((a, b), r))(x, y)
\]

From the normalization algorithm:

\[
\text{length}(\text{zip}(x, y)) \rightarrow \text{red}^{\text{list}\times\text{list}}(\phi_{nn}, \phi_{nc}, \phi_{cn}, \phi_{cc})(x, y)
\]

where

1) \(\phi_{nn}((), ()) = \text{length}(f_{nn}((), ()))\)
2) \(\phi_{nc}((), (b, s)) = \text{length}(f_{nc}((), (b, s)))\)
3) \(\phi_{cn}((a, l), ()) = \text{length}(f_{cn}((a, l), ()))\)
4) \(\phi_{cc}(a, (b, r)) = \text{length}(f_{cc}(a, (b, \text{STOP}[\text{length}](r))))\)
1) \( \phi_{nn}() = \text{length} (\text{Nil}) = \text{Zero} \)

2) \( \phi_{nc}(() , (b,s)) = \text{length} (\text{Nil}) = \text{Zero} \)

3) \( \phi_{cn}((a,l), ()) = \text{length} (\text{Nil}) = \text{Zero} \)

4) \( \phi_{cc}(a, (b,r)) = \text{length} (\text{Cons}((a,b), \text{STOP} [\text{length}](r))) \\
\quad = \text{Succ} (\text{length} (\text{STOP} [\text{length}](r))) \\
\quad = \text{Succ}(r) \)

Therefore,

\[
\text{length} (\text{zip}(x,y)) \to \text{red}^{\text{list} \times \text{list}} (\lambda().\text{Zero} , \\
\quad \lambda(() , (b,s)).\text{Zero} , \\
\quad \lambda((a,l), ()).\text{Zero} , \\
\quad \lambda(a, (b,r)).\text{Succ}(r))(x,y)
\]
The normalization algorithm can be used for translating ordinary recursive programs (expressed in an ML-like language) into algebraic programs.

\[
\textbf{fun} \ \text{app}(\text{Nil}) \ y = y \\
\ | \ \text{app}(\text{Cons}(a, s)) \ y = \text{Cons}(a, \text{app}(s) \ y)
\]

is translated into

\[
\text{red}^{\text{list}}(\lambda().\lambda y. y, \lambda(a, s). \lambda y. \text{Cons}(a, \text{app}(\text{STOP} \text{[app]}(s)) \ y)) \\
= \text{red}^{\text{list}}(\lambda().\lambda y. y, \lambda(a, s). \lambda y. \text{Cons}(a, s(y))) \\
= \lambda y. \text{red}^{\text{list}}(\lambda(). y, \lambda(a, s). \text{Cons}(a, s))
\]