# Improving Programs which Recurse over Multiple Inductive Structures

**Leonidas Fegaras**
**Tim Sheard**
**Tong Zhou**

**Oregon Graduate Institute**

# Evolution of Control Structure

| *Imperative Languages* | *Functional Languages* |
| --- | --- |
| goto | recursion |
| $\downarrow$ | $\downarrow$ |
| structured control | generic recursion schemes |
| (e.g. *while* loops, *if-then-else*) | (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:*

$$\begin{aligned} \mathrm{tree}(\alpha,\beta) \;=\; & \;\mathrm{Tip}\ \mathbf{of}\ \alpha \\ & \mid\ \mathrm{Node}\ \mathbf{of}\ \beta \times \mathrm{tree}(\alpha,\beta) \times \mathrm{tree}(\alpha,\beta) \end{aligned}$$

The *tree* reduction scheme is:

$$\begin{aligned} F(\mathrm{Tip}(a)) &= f_1(a) \\ F(\mathrm{Node}(i,l,r)) &= f_2(i, F(l), F(r)) \end{aligned}$$

We give this scheme a name: $\boxed{F \;=\; \mathrm{red}^{tree}(f_1, f_2)}$

- *Lists:*

$$\mathrm{list}(\alpha) \;=\; \mathrm{Nil} \mid \mathrm{Cons}\ \mathbf{of}\ \alpha \times \mathrm{list}(\alpha)$$

The *list* reduction scheme is:

$$\begin{aligned} F(\mathrm{Nil}) &= f_1() \\ F(\mathrm{Cons}(a,s)) &= f_2(a, F(s)) \end{aligned}$$

where $\boxed{F \;=\; \mathrm{red}^{list}(f_1, f_2)}$

How do we abstract the patterns of recursions of any data type?

# The Functor $E_c^T$

- *Binary trees:*

$$F(\text{Node}(i,l,r)) \;=\; f_2(E^{tree}_{Node}(F)(i,l,r))$$

where

$$E^{tree}_{Node}(F)\,(i,l,r) \;=\; (i, F(l), F(r))$$

The functor $E^{tree}_{Node}(F)$ pushes $F$ onto some of the constructor's arguments.

- *Lists:*

$$F(\text{Cons}(a,s)) \;=\; f_2(E^{list}_{Cons}(F)(a,s))$$

where

$$E^{list}_{Cons}(F)\,(a,s) \;=\; (a, F(s))$$

For most data types $T$ and for each value constructor $C$ of $T$ we can calculate the functor $E_c^T(F)$:

$$E_c^T(F)\,(x_1, x_2, \ldots, x_n) \;=\; (??(x_1), ??(x_2), \ldots, ??(x_n))$$

# Unary Reduction

For each constructor $C$ of $T$:

$$\boxed{\mathrm{red}^T(\overline{f}) \circ C = f_c \circ E_c^T(\mathrm{red}^T(\overline{f}))}$$

e.g., for *tree*:

$$\begin{array}{lcl}
\mathrm{red}^{tree}(f_1, f_2)\,(\mathrm{Tip}(a)) & = & f_1(a) \\
\mathrm{red}^{tree}(f_1, f_2)\,(\mathrm{Node}(i, l, r)) & = & f_2(i, \mathrm{red}^{tree}(f_1, f_2)\, l, \mathrm{red}^{tree}(f_1, f_2)\, r)
\end{array}$$

$$\begin{tikzcd}
\mathrm{tree}(\alpha, \beta) \arrow[d, "\mathrm{red}^{tree}(f_1, f_2)"'] & \beta \times \mathrm{tree}(\alpha, \beta) \times \mathrm{tree}(\alpha, \beta) \arrow[l, "\mathrm{Node}"'] \arrow[d, "E^{tree}_{Node}(\mathrm{red}^{tree}(f_1, f_2))"] \\
\gamma & \beta \times \gamma \times \gamma \arrow[l, "f_2"]
\end{tikzcd}$$

## Examples

$$
\begin{aligned}
\text{append}(x, y) &= \text{red}^{list}(\lambda().y, \lambda(a, r).\text{Cons}(a, r))\, x \\
\text{length}(x) &= \text{red}^{list}(\lambda().\text{Zero}, \lambda(a, r).\text{Succ}(r))\, x \\
x + y &= \text{red}^{nat}(\lambda().y, \lambda(r).\text{Succ}(r))\, x \\
\text{if}\, x \,\text{then}\, y \,\text{else}\, z &= \text{red}^{boolean}(\lambda().z, \lambda().y)\, x \\
\text{reflect\_bush}(x) &= \text{red}^{bush}(\lambda(a).\text{Leaf}(a), \\
&\qquad\qquad \lambda(r).\text{Branch}(\text{reverse}(r)))\, x
\end{aligned}
$$

## Unary Reductions can Capture Multiple Traversals

*List structural equality*

$$\begin{array}{lcl}
\text{listeq}(\text{Nil}, \text{Nil}) & = & \text{True} \\
\text{listeq}(\text{Nil}, \text{Cons}(b, s)) & = & \text{False} \\
\text{listeq}(\text{Cons}(a, l), \text{Nil}) & = & \text{False} \\
\text{listeq}(\text{Cons}(a, l), \text{Cons}(b, s)) & = & (a = b) \wedge \text{listeq}(l, s)
\end{array}$$

can be expressed as a second-order unary reduction:

$$\begin{array}{l}
\text{red}^{list}(\lambda().\lambda k.(k = \text{Nil}), \\
\qquad \lambda(a, r).\lambda k.\textbf{case}\, k\, \textbf{of} \\
\qquad\qquad\qquad \text{Nil} \Rightarrow \text{False} \\
\qquad\qquad\qquad \mid \text{Cons}(b, s) \Rightarrow (a = b) \wedge r(s))\ x\ y
\end{array}$$

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

## Binary Reduction Schemes

Simultaneous traversal of a *list* and a *tree*:

$$
\begin{aligned}
F(\mathrm{Nil}, \mathrm{Tip}(b)) &= f_1(b) \\
F(\mathrm{Nil}, \mathrm{Node}(i,l,r)) &= f_2(i,l,r) \\
F(\mathrm{Cons}(a,s), \mathrm{Tip}(b)) &= f_3(a,s,b) \\
F(\mathrm{Cons}(a,s), \mathrm{Node}(i,l,r)) &= f_4(a,i,F(s,l),F(s,r))
\end{aligned}
$$

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

$$F \circ (\mathrm{Cons} \times \mathrm{Node}) \;=\; f_4 \circ E^{list\times tree}_{Cons\times Node}(F)$$

*Problem:*

Calculate $E^{list\times tree}_{Cons\times Node}$ from $E^{list}_{Cons}$ and $E^{tree}_{Node}$

## Generalization of $E$

$E^{T_1 \times T_2}_{c_1 \times c_2}$ is calculated by "nesting" $E^{T_2}_{c_2}$ inside $E^{T_1}_{c_1}$:

$$\boxed{E^{T_1 \times T_2}_{c_1 \times c_2}(f) \;=\; \lambda(x,y).E^{T_1}_{c_1}(\lambda z.E^{T_2}_{c_2}(\lambda w.f(z,w))\,y)\,x}$$

e.g.

$$E^{list \times tree}_{Cons \times Node}(f)\,(x,y) \;=\; E^{list}_{Cons}(\lambda z.E^{tree}_{Node}(\lambda w.f(z,w))\,y)\,x$$

where

$$\begin{aligned} E^{list}_{Cons}(G)\,(a,s) &\;=\; (a, \boxed{G(s)}) \\ E^{tree}_{Node}(H)\,(i,l,r) &\;=\; (i, \boxed{H(l)}, \boxed{H(r)}) \end{aligned}$$

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

$$E^{list \times tree}_{Cons \times Node}(F)\,((a,s),(i,l,r)) \;=\; (a,(i,\boxed{F(s,l)},\boxed{F(s,r)}))$$

Thus,

$$F(\mathrm{Cons}(a,s), \mathrm{Node}(i,l,r)) \;=\; f_4(a,(i,F(s,l),F(s,r)))$$

# Binary Reduction

$$\boxed{\mathrm{red}^{T_1 \times T_2}(\overline{f}) \circ (C_1 \times C_2) \;=\; f_{c_1 \times c_2} \circ E_{c_1 \times c_2}^{T_1 \times T_2}(\mathrm{red}^{T_1 \times T_2}(\overline{f}))}$$

e.g., the binary reduction operator for $list \times list$ is

$$F \;=\; \mathrm{red}^{list \times list}(f_{nn}, f_{nc}, f_{cn}, f_{cc})$$

defined as:

$$\begin{array}{lcl}
F(\mathrm{Nil}, \mathrm{Nil}) & = & f_{nn}((), ()) \\
F(\mathrm{Nil}, \mathrm{Cons}(b, s)) & = & f_{nc}((), (b, s)) \\
F(\mathrm{Cons}(a, l), \mathrm{Nil}) & = & f_{cn}((a, l), ()) \\
F(\mathrm{Cons}(a, l), \mathrm{Cons}(b, s)) & = & f_{cc}(a, (b, F(l, s)))
\end{array}$$

e.g.

$$\begin{array}{lcl}
\mathrm{listeq}(x, y) & = & \mathrm{red}^{list \times list}(\lambda((), ()).\mathrm{True}, \\
& & \quad \lambda((), (b, s)).\mathrm{False}, \\
& & \quad \lambda((a, l), ()).\mathrm{False}, \\
& & \quad \lambda(a, (b, r)).r \wedge (a = b))\,(x, y)
\end{array}$$

$$\begin{array}{lcl}
\mathrm{zip}(x, y) & = & \mathrm{red}^{list \times list}(\lambda((), ()).\mathrm{Nil}, \\
& & \quad \lambda((), (b, s)).\mathrm{Nil}, \\
& & \quad \lambda((a, l), ()).\mathrm{Nil}, \\
& & \quad \lambda(a, (b, r)).\mathrm{Cons}((a, b), r))\,(x, y)
\end{array}$$

# Program Improvement

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

$$\begin{aligned}
\mathrm{length}(x) &= \mathrm{red}^{list}(\lambda().\mathrm{Zero}, \lambda(a, r).\mathrm{Succ}(r))\, x \\[1em]
\mathrm{zip}(x, y) &= \mathrm{red}^{list \times list}(\lambda((), ()).\mathrm{Nil}, \\
&\qquad\qquad \lambda((), (b, s)).\mathrm{Nil}, \\
&\qquad\qquad \lambda((a, l), ()).\mathrm{Nil}, \\
&\qquad\qquad \lambda(a, (b, r)).\mathrm{Cons}((a, b), r))\, (x, y)
\end{aligned}$$

length can be *fused* with zip into a binary reduction:

$$\begin{aligned}
\mathrm{length}(\mathrm{zip}(x, y)) \rightarrow \mathrm{red}^{list \times list}(&\lambda().\mathrm{Zero}, \\
&\lambda((), (b, s)).\mathrm{Zero}, \\
&\lambda((a, l), ()).\mathrm{Zero}, \\
&\lambda(a, (b, r)).\mathrm{Succ}(r))(x, y)
\end{aligned}$$

How do we achieve this loop-fusion automatically?

# Promotion Theorem

$$
\frac{\begin{array}{lcl}
\phi_{nn}((),()) & = & g(f_{nn}((),())) \\
\phi_{nc}((),(b,s)) & = & g(f_{nc}((),(b,s))) \\
\phi_{cn}((a,l),()) & = & g(f_{cn}((a,l),())) \\
\phi_{cc}(a,(b,g(r))) & = & g(f_{cc}(a,(b,r)))
\end{array}}{\begin{array}{l}
g(\mathrm{red}^{list \times list}(f_{nn}, f_{nc}, f_{cn}, f_{cc})\ (x,y)) \\
\qquad = \ \mathrm{red}^{list \times list}(\phi_{nn}, \phi_{nc}, \phi_{cn}, \phi_{cc})\ (x,y)
\end{array}}
$$

# Promotion Theorems can be Expressed for any Type $T$

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

$$\frac{\forall c:\ \phi_c \circ E_c^T(g) \ = \ g \circ f_c}{g \circ \mathrm{red}^T(\overline{f}) \ = \ \mathrm{red}^T(\overline{\phi})}$$

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

$$\frac{\forall c_1, \forall c_2:\ \phi_{c_1 \times c_2} \circ E_{c_1}^{T_1}(E_{c_2}^{T_2}(g)) \ = \ g \circ f_{c_1 \times c_2}}{g \circ \mathrm{red}^{T_1 \times T_2}(\overline{f}) \ = \ \mathrm{red}^{T_1 \times T_2}(\overline{\phi})}$$

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

# Program Improvement

The promotion theorems can be used for *loop fusion*:

$$\frac{\begin{array}{rlcl} 1) & \phi_{nn}((),()) & = & g(f_{nn}((),())) \\ 2) & \phi_{nc}((),(b,s)) & = & g(f_{nc}((),(b,s))) \\ 3) & \phi_{cn}((a,l),()) & = & g(f_{cn}((a,l),())) \\ 4) & \phi_{cc}(a,(b,g(r))) & = & g(f_{cc}(a,(b,r))) \end{array}}{\begin{array}{l} g(\mathrm{red}^{list\times list}(f_{nn},f_{nc},f_{cn},f_{cc})\ (x,y)) \\ \qquad = \ \mathrm{red}^{list\times list}(\phi_{nn},\phi_{nc},\phi_{cn},\phi_{cc})\ (x,y) \end{array}}$$

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 $\mathcal{STOP}[g]$ on each $r$ in the rhs of equation #4:

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

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

*When this trick works?*

- if **all** $\mathcal{STOP}[g]$ are eliminated in $g(f_{cc}(a,(b,\mathcal{STOP}[g](r))))$ after transformation, then $\mathrm{red}^{list\times list}(\phi_{nn},\phi_{nc},\phi_{cn},\phi_{cc})$ preserves the meaning of $g\circ\mathrm{red}^{list\times list}(f_{nn},f_{nc},f_{cn},f_{cc})$;
- otherwise $g(\mathrm{red}^{list}(f_n,f_c)\,x)$ should be left as is.

The same trick can be used for any promotion theorem.

# Example of a Program Improvement

Improving length(zip$(x, y)$):

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

$$\begin{aligned}
\text{zip}(x,y) \;&=\; \text{red}^{list\times list}(f_{nn}, f_{nc}, f_{cn}, f_{cc})\,(x,y)\\
&=\; \text{red}^{list\times list}(\lambda((),()).\text{Nil},\\
&\qquad\qquad\quad \lambda((),(b,s)).\text{Nil},\\
&\qquad\qquad\quad \lambda((a,l),()).\text{Nil},\\
&\qquad\qquad\quad \lambda(a,(b,r)).\text{Cons}((a,b),r))\,(x,y)
\end{aligned}$$

From the normalization algorithm:

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

where

$$\begin{aligned}
&1)\;\; \phi_{nn}((),()) &&=\; \text{length}(f_{nn}((),()))\\
&2)\;\; \phi_{nc}((),(b,s)) &&=\; \text{length}(f_{nc}((),(b,s)))\\
&3)\;\; \phi_{cn}((a,l),()) &&=\; \text{length}(f_{cn}((a,l),()))\\
&4)\;\; \phi_{cc}(a,(b,r)) &&=\; \text{length}(f_{cc}(a,(b,\mathcal{STOP}[\text{length}](r))))
\end{aligned}$$

$$
\begin{aligned}
&1)\ \phi_{nn}() &&= \text{length}(\text{Nil}) = \text{Zero}\\
&2)\ \phi_{nc}((), (b,s)) &&= \text{length}(\text{Nil}) = \text{Zero}\\
&3)\ \phi_{cn}((a,l), ()) &&= \text{length}(\text{Nil}) = \text{Zero}\\
&4)\ \phi_{cc}(a, (b,r)) &&= \text{length}(\text{Cons}((a,b), \mathcal{STOP}[\text{length}](r)))\\
& &&= \text{Succ}(\text{length}(\mathcal{STOP}[\text{length}](r)))\\
& &&= \text{Succ}(r)
\end{aligned}
$$

Therefore,

$$
\begin{aligned}
\text{length}(\text{zip}(x,y)) \rightarrow \text{red}^{list\times list}(&\lambda().\text{Zero},\\
&\lambda((), (b,s)).\text{Zero},\\
&\lambda((a,l), ()).\text{Zero},\\
&\lambda(a, (b,r)).\text{Succ}(r))(x,y)
\end{aligned}
$$

The normalization algorithm can be used for translating ordinary recursive programs (expressed in an ML-like language) into algebraic programs.

$$\begin{aligned} \textbf{fun}\ \ & \mathrm{app}(\mathrm{Nil})\, y && = \ y \\ & \mid \mathrm{app}(\mathrm{Cons}(a,s))\, y && = \ \mathrm{Cons}(a, \mathrm{app}(s)\, y) \end{aligned}$$

is translated into

$$\begin{aligned} & \mathrm{red}^{list}(\lambda().\lambda y.y, \lambda(a,s).\lambda y.\mathrm{Cons}(a, \mathrm{app}(\mathcal{STOP}[\mathrm{app}](s))\, y)) \\ = \ & \mathrm{red}^{list}(\lambda().\lambda y.y, \lambda(a,s).\lambda y.\mathrm{Cons}(a, s(y))) \\ = \ & \lambda y.\mathrm{red}^{list}(\lambda().y, \lambda(a,s).\mathrm{Cons}(a,s)) \end{aligned}$$