Applicative Shortcut Fusion

Germán Delbianco[1] Mauro Jaskelioff[2] Alberto Pardo[3]

[1]IMDEA Software Institute, Spain

[2]CIFASIS-CONICET/Universidad Nacional de Rosario, Argentina

[3]Instituto de Computación, Universidad de la República, Uruguay

Program Fusion

A program written in a compositional style:

$$\begin{aligned} \mathit{sumRecipDiffs} &:: [\mathit{Float}] \to [\mathit{Float}] \to \mathit{Maybe}\ \mathit{Float} \\ \mathit{sumRecipDiffs}\ \mathit{ys} &= \mathit{fmap}\ \mathit{sum} \circ \mathit{recipList} \circ \mathit{diffList}\ \mathit{ys} \end{aligned}$$

generates a series of intermediate data stuctures,

$$\begin{tikzcd} {[\mathit{Float}]} \arrow[r, bend left, "\mathit{diffList}\ \mathit{ys}"] \arrow[rrr, bend right, "\mathit{sumRecipDiffs}\ \mathit{ys}"'] & \color{red}{[\mathit{Float}]} \arrow[r, bend left, "\mathit{recipList}"] & \color{red}{\mathit{Maybe}\ [\mathit{Float}]} \arrow[r, bend left, "\mathit{fmap}\ \mathit{sum}"] & \mathit{Maybe}\ \mathit{Float} \end{tikzcd}$$

The goal of **fusion** is to obtain, whenever possible, an equivalent, monolithic definition without intermediate structures.

Shortcut Fusion

$$foldr\ f\ e \circ build\ g = g\ f\ e$$

where

$$\begin{aligned}
&foldr :: (a \to b \to b) \to b \to [a] \to b \\
&foldr\ f\ e\ [\,] \qquad\quad = e \\
&foldr\ f\ e\ (x : xs) = f\ x\ (foldr\ f\ e\ xs)
\end{aligned}$$

$$\begin{aligned}
&build \quad :: (\forall b.(a \to b \to b) \to b \to c \to b) \to c \to [a] \\
&build\ g = g\ (:)\ [\,]
\end{aligned}$$

Example: *factorial*

$$
\begin{aligned}
&\mathit{factorial} && :: \mathit{Int} \to \mathit{Int} \\
&\mathit{factorial}\ n && = \mathit{product}\ (\mathit{down}\ n)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{product} && :: [\mathit{Int}] \to \mathit{Int} \\
&\mathit{product}\ [\,] && = 1 \\
&\mathit{product}\ (a : as) && = a * \mathit{product}\ as
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{down} && :: \mathit{Int} \to [\mathit{Int}] \\
&\mathit{down}\ 0 && = [\,] \\
&\mathit{down}\ n && = n : \mathit{down}\ (n - 1)
\end{aligned}
$$

Example: *factorial*

$$product = foldr\ (*)\ 1$$

$$down = build\ gdown$$
where
$$gdown\ fc\ fn\ 0 = fn$$
$$gdown\ fc\ fn\ n = n\ `fc`\ gdown\ fc\ fn\ (n-1)$$

$$\begin{aligned} factorial &= product \circ down \\ &= foldr\ (*)\ 1 \circ build\ gdown \\ &= gdown\ (*)\ 1 \end{aligned}$$

Example: *factorial*

$$product = foldr\ (*)\ 1$$

$$down = build\ gdown$$
where
$$gdown\ fc\ fn\ 0 = fn$$
$$gdown\ fc\ fn\ n = n\ \text{`}fc\text{`}\ gdown\ fc\ fn\ (n-1)$$

$$\begin{aligned} factorial &= product \circ down \\ &= foldr\ (*)\ 1 \circ build\ gdown \\ &= gdown\ (*)\ 1 \end{aligned}$$

Extended Shortcut Fusion

$$fmap\ (foldr\ f\ e) \circ ebuild\ g = g\ f\ e$$

where

$$\begin{aligned} ebuild\ &:: Functor\ f \Rightarrow \\ &\quad (\forall b.(a \to b \to b) \to b \to c \to f\ b) \to c \to f\ [a] \\ ebuild\ g &= g\ (:)\ [\,] \end{aligned}$$

Functor f acts as a container of the generated list.

Monadic Shortcut Fusion

$$
\begin{aligned}
&mmap :: Monad\ m \Rightarrow (a \to b) \to (m\ a \to m\ b)\\
&mmap\ f\ \ m = \mathbf{do}\ \{a \leftarrow m; return\ (f\ \ a)\}
\end{aligned}
$$

$$
\begin{aligned}
mbuild &:: Monad\ m\\
&\quad \Rightarrow (\forall b.(a \to b \to b) \to b \to c \to m\ b)\\
&\quad \to c \to m\ [a]\\
mbuild\ g &= g\ (:)\ [\,]
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{do}\ \{as \leftarrow mbuild\ g\ \ c; return\ (fold\ f\ \ e\ as)\}\\
&\quad = g\ f\ \ e\ \ c
\end{aligned}
$$

Example: *lenLine*

$$lenLine = \mathbf{do}\ \{ cs \leftarrow getLine; return\ (length\ cs) \}$$

$$\begin{aligned}
&length && :: [a] \rightarrow Int \\
&length\ [\,] && = 0 \\
&length\ (x : xs) && = 1 + length\ xs
\end{aligned}$$

$$\begin{aligned}
&getLine :: IO\ String \\
&getLine = \mathbf{do}\ c \leftarrow getChar \\
&\qquad\qquad\quad \mathbf{if}\ c \equiv eol \\
&\qquad\qquad\qquad \mathbf{then}\ return\ [\,] \\
&\qquad\qquad\qquad \mathbf{else\ do}\ cs \leftarrow getLine \\
&\qquad\qquad\qquad\qquad\quad return\ (c : cs)
\end{aligned}$$

Example: *lenLine*

$$length = foldr\ (\lambda x\ y \rightarrow 1 + y)\ 0$$

$$
\begin{array}{l}
getLine = mbuild\ ggL \\
\quad \textbf{where}\ ggL\ fc\ fn = \textbf{do}\ c \leftarrow getChar \\
\qquad\qquad\qquad\qquad\qquad \textbf{if}\ c \equiv eol \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ return\ fn \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{else do}\ cs \leftarrow ggL\ fc\ fn \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad return\ (c\ `fc`\ cs)
\end{array}
$$

$$
\begin{array}{l}
lenLine = \textbf{do}\ c \leftarrow getChar \\
\qquad\qquad \textbf{if}\ c \equiv eol \\
\qquad\qquad\quad \textbf{then}\ return\ 0 \\
\qquad\qquad\quad \textbf{else do}\ n \leftarrow lenLine \\
\qquad\qquad\qquad\qquad return\ (1 + n)
\end{array}
$$

Example: *lenLine*

$$length = foldr\ (\lambda x\ y \rightarrow 1 + y)\ 0$$

$$
\begin{array}{l}
getLine = mbuild\ ggL \\
\quad \textbf{where}\ ggL\ fc\ fn = \textbf{do}\ c \leftarrow getChar \\
\qquad\qquad\qquad\qquad\qquad \textbf{if}\ c \equiv eol \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ return\ fn \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{else do}\ cs \leftarrow ggL\ fc\ fn \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad return\ (c\ `fc`\ cs)
\end{array}
$$

$$
\begin{array}{l}
lenLine = \textbf{do}\ c \leftarrow getChar \\
\qquad\qquad\quad \textbf{if}\ c \equiv eol \\
\qquad\qquad\qquad \textbf{then}\ return\ 0 \\
\qquad\qquad\qquad \textbf{else do}\ n \leftarrow lenLine \\
\qquad\qquad\qquad\qquad\quad return\ (1 + n)
\end{array}
$$

In this talk

- We present a shortcut fusion rules for data structures produced within *applicative computations*
- The rule illustrates, once more, the relevance and generality of **applicative traversals** for generating and consuming data structures in applicative contexts.
- We introduce two combinators, *ifold* and *ibuild*, which model uniform consumption and production schemes in the presence of applicative computations.

Applicative Functors

An **Applicative Functor** (or **idiom**) is a type constructor with two operations:

$$
\begin{array}{l}
\textbf{class } \mathit{Functor}\ f \Rightarrow \mathit{Applicative}\ f \textbf{ where} \\
\quad \mathit{pure} :: a \to f\ a \\
\quad (\circledast) :: f\ (a \to b) \to f\ a \to f\ b
\end{array}
$$

- *pure* lifts *pure values* into computations.
- $\circledast$ performs functional application, sequentializing effects

For example,

$$
\begin{array}{l}
\textbf{instance } \mathit{Applicative}\ \mathit{Maybe} \textbf{ where} \\
\quad \mathit{pure} = \mathit{Just} \\
\quad (\mathit{Just}\ f) \circledast (\mathit{Just}\ x) = \mathit{Just}\ (f\ x) \\
\quad \_ \circledast \_ = \mathit{Nothing}
\end{array}
$$

Applicative Functors

An **Applicative Functor** (or **idiom**) is a type constructor with two operations:

$$
\begin{array}{l}
\textbf{class } \mathit{Functor}\ f \Rightarrow \mathit{Applicative}\ f \textbf{ where} \\
\quad \mathit{pure} :: a \rightarrow f\ a \\
\quad (\circledast) :: f\ (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b
\end{array}
$$

- *pure* lifts *pure values* into computations.
- $\circledast$ performs functional application, sequentializing effects

For example,

$$
\begin{array}{l}
\textbf{instance } \mathit{Applicative}\ \mathit{Maybe} \textbf{ where} \\
\quad \mathit{pure} = \mathit{Just} \\
\quad (\mathit{Just}\ f) \circledast (\mathit{Just}\ x) = \mathit{Just}\ (f\ x) \\
\quad \_ \qquad\quad\ \circledast \_ \qquad\qquad = \mathit{Nothing}
\end{array}
$$

Traversable Functors

- **Traversable Functors** support **effectful traversals** with **applicative actions** of type $a \to f\ b$

$$
\begin{array}{l}
\mathbf{class}\ \mathit{Functor}\ t \Rightarrow \mathit{Traversable}\ t\ \mathbf{where} \\
\quad \mathit{traverse} :: \mathit{Applicative}\ f \Rightarrow (a \to f\ b) \to t\ a \to t\ (f\ b) \\
\quad \mathit{dist} \quad\quad :: \mathit{Applicative}\ f \Rightarrow t\ (f\ a) \to f\ (t\ a)
\end{array}
$$

Example (Lists)

$$
\begin{array}{ll}
\mathit{traverse} & :: \mathit{Applicative}\ f \Rightarrow (a \to f\ b) \to [a] \to f\ [b] \\
\mathit{traverse}\ \iota\ [\,] & = \mathit{pure}\ [\,] \\
\mathit{traverse}\ \iota\ (x : xs) & = \mathit{pure}\ (:) \circledast \iota\ x \circledast \mathit{traverse}\ \iota\ xs
\end{array}
$$

Traversable Functors

- **Traversable Functors** support **effectful traversals** with **applicative actions** of type $a \to f\ b$

$$
\begin{array}{l}
\textbf{class}\ \mathit{Functor}\ t \Rightarrow \mathit{Traversable}\ t\ \textbf{where} \\
\quad \mathit{traverse} :: \mathit{Applicative}\ f \Rightarrow (a \to f\ b) \to t\ a \to t\ (f\ b) \\
\quad \mathit{dist} \quad\quad :: \mathit{Applicative}\ f \Rightarrow t\ (f\ a) \to f\ (t\ a)
\end{array}
$$

Example (Lists)

$$
\begin{array}{ll}
\mathit{traverse} & :: \mathit{Applicative}\ f \Rightarrow (a \to f\ b) \to [a] \to f\ [b] \\
\mathit{traverse}\ \iota\ [\,] & = \mathit{pure}\ [\,] \\
\mathit{traverse}\ \iota\ (x : xs) & = \mathit{pure}\ (:) \circledast \iota\ x \circledast \mathit{traverse}\ \iota\ xs
\end{array}
$$

Applicative Fold

$$
\begin{aligned}
&\mathit{ifoldr} :: \mathit{Applicative}\ f \Rightarrow \\
&\qquad (b \to c \to c) \to c \to (a \to f\ b) \to [a] \to f\ c \\
&\mathit{ifoldr}\ f\ e\ \iota = \mathit{fmap}\ (\mathit{foldr}\ f\ e) \circ \mathit{traverse}\ \iota
\end{aligned}
$$

Fusing the parts,

$$
\begin{aligned}
&\mathit{ifoldr}\ f\ e\ \iota\ [\,] &&= \mathit{pure}\ e \\
&\mathit{ifoldr}\ f\ e\ \iota\ (x : xs) &&= \mathit{pure}\ f \circledast \iota\ x \circledast \mathit{ifoldr}\ f\ e\ \iota\ xs
\end{aligned}
$$

Observe that *ifoldr* is in turn a fold:

$$
\begin{aligned}
&\mathit{ifoldr}\ f\ e\ \iota = \mathit{foldr}\ \phi\ (\mathit{pure}\ e) \\
&\quad \textbf{where} \\
&\qquad \phi\ x\ y = \mathit{pure}\ f \circledast \iota\ x \circledast y
\end{aligned}
$$

Applicative Fold

$$
\begin{aligned}
&\mathit{ifoldr} :: \mathit{Applicative}\ f \Rightarrow \\
&\qquad (b \to c \to c) \to c \to (a \to f\ b) \to [a] \to f\ c \\
&\mathit{ifoldr}\ f\ e\ \iota = \mathit{fmap}\ (\mathit{foldr}\ f\ e) \circ \mathit{traverse}\ \iota
\end{aligned}
$$

Fusing the parts,

$$
\begin{aligned}
\mathit{ifoldr}\ f\ e\ \iota\ [\,] &= \mathit{pure}\ e \\
\mathit{ifoldr}\ f\ e\ \iota\ (x : xs) &= \mathit{pure}\ f \circledast \iota\ x \circledast \mathit{ifoldr}\ f\ e\ \iota\ xs
\end{aligned}
$$

Observe that *ifoldr* is in turn a fold:

$$
\begin{aligned}
&\mathit{ifoldr}\ f\ e\ \iota = \mathit{foldr}\ \phi\ (\mathit{pure}\ e) \\
&\quad \textbf{where} \\
&\qquad \phi\ x\ y = \mathit{pure}\ f \circledast \iota\ x \circledast y
\end{aligned}
$$

Applicative Fold

$$
\begin{aligned}
&\mathit{ifoldr} :: \mathit{Applicative}\ f \Rightarrow \\
&\qquad (b \to c \to c) \to c \to (a \to f\ b) \to [a] \to f\ c \\
&\mathit{ifoldr}\ f\ e\ \iota = \mathit{fmap}\ (\mathit{foldr}\ f\ e) \circ \mathit{traverse}\ \iota
\end{aligned}
$$

Fusing the parts,

$$
\begin{aligned}
\mathit{ifoldr}\ f\ e\ \iota\ [\,] &= \mathit{pure}\ e \\
\mathit{ifoldr}\ f\ e\ \iota\ (x : xs) &= \mathit{pure}\ f \circledast \iota\ x \circledast \mathit{ifoldr}\ f\ e\ \iota\ xs
\end{aligned}
$$

Observe that *ifoldr* is in turn a fold:

$$
\begin{aligned}
&\mathit{ifoldr}\ f\ e\ \iota = \mathit{foldr}\ \phi\ (\mathit{pure}\ e) \\
&\quad \textbf{where} \\
&\qquad \phi\ x\ y = \mathit{pure}\ f \circledast \iota\ x \circledast y
\end{aligned}
$$

Example: sum the reciprocals of a list

$$sumrecips :: [Float] \to Maybe\ Float$$
$$sumrecips = fmap\ (foldr\ (+)\ 0) \circ recipList$$

$$recipList :: [Float] \to Maybe\ [Float]$$
$$recipList = traverse\ recip$$

$$recip \quad :: Float \to Maybe\ Float$$
$$recip\ x = \mathbf{if}\ (x \not\equiv 0)\ \mathbf{then}\ pure\ (1\ /\ x)\ \mathbf{else}\ Nothing$$

Then, function *sumrecips* can be written as:

$$sumrecips = ifoldr\ (+)\ 0\ recip$$

Applicative Build

We define an applicative build as a standard *build* followed by *traverse*:

$$
\begin{aligned}
&\mathit{ibuild} :: \mathit{Applicative}\ f \Rightarrow \\
&\qquad (a \to f\ d) \to (\forall b.(a \to b \to b) \to b \to c \to b) \to c \to f\ [d] \\
&\mathit{ibuild}\ \iota\ g = \mathit{traverse}\ \iota \circ \mathit{build}\ g
\end{aligned}
$$

Since *traverse* is a fold,

$$
\begin{aligned}
&\mathit{traverse}\ \iota = \mathit{foldr}\ \psi\ (\mathit{pure}\ [\,]) \\
&\quad \mathbf{where} \\
&\qquad \psi\ x\ y = \mathit{pure}\ (:) \circledast \iota\ x \circledast y
\end{aligned}
$$

we get that:

$$
\mathit{ibuild}\ \iota\ g = g\ \psi\ (\mathit{pure}\ [\,])
$$

Applicative Build

We define an applicative build as a standard *build* followed by *traverse*:

$$
\begin{aligned}
&\mathit{ibuild} :: \mathit{Applicative}\ f \Rightarrow \\
&\qquad (a \to f\ d) \to (\forall b.(a \to b \to b) \to b \to c \to b) \to c \to f\ [d] \\
&\mathit{ibuild}\ \iota\ g = \mathit{traverse}\ \iota \circ \mathit{build}\ g
\end{aligned}
$$

Since *traverse* is a fold,

$$
\begin{aligned}
&\mathit{traverse}\ \iota = \mathit{foldr}\ \psi\ (\mathit{pure}\ [\,]) \\
&\quad \mathbf{where} \\
&\qquad \psi\ x\ y = \mathit{pure}\ (:) \circledast \iota\ x \circledast y
\end{aligned}
$$

we get that:

$$
\mathit{ibuild}\ \iota\ g = g\ \psi\ (\mathit{pure}\ [\,])
$$

It is interesting to see that the applicative build can be written as an extended build:

$$
\begin{aligned}
& \mathit{ibuild}\ \iota\ g = \mathit{ebuild}\ g' \\
& \quad \textbf{where} \\
& \qquad g' :: \mathit{Applicative}\ f \Rightarrow \\
& \qquad\qquad (\forall b.(d \to b \to b) \to b \to c \to b) \to c \to f\ [d] \\
& \qquad g'\ f\ e = g\ \phi\ (\mathit{pure}\ e) \\
& \qquad \phi :: a \to b \to b \\
& \qquad \phi\ x\ y = \mathit{pure}\ f \circledast \iota\ x \circledast y
\end{aligned}
$$

Example: reciprocals of difference of two lists

$$recipDiffList = recipList \circ diffList$$

where

$$\begin{array}{lll}
diffList & & :: [Float] \to [Float] \to [Float] \\
diffList\ ys & [\,] & = [\,] \\
diffList\ [\,] & (x : xs) & = [\,] \\
diffList\ (y : ys) & (x : xs) & = (y - x) : difflist\ ys\ xs
\end{array}$$

$$recipList = traverse\ recip$$

Since *diffList* can be written as a build,

$$
\begin{aligned}
&\mathit{diffList}\ \mathit{ys} = \mathit{build}\ (\mathit{gdiff}\ \mathit{ys})\\
&\mathit{gdiff}\ :: [\mathit{Float}] \to (\mathit{Float} \to b \to b) \to b \to [\mathit{Float}] \to b\\
&\mathit{gdiff}\ \mathit{ys}\qquad\quad c\ n\ [\,]\qquad\quad = n\\
&\mathit{gdiff}\ [\,]\qquad\quad\ \ c\ n\ (x : \mathit{xs}) = n\\
&\mathit{gdiff}\ (y : \mathit{ys})\ c\ n\ (x : \mathit{xs}) = (y - x)\ \text{`}c\text{`}\ (\mathit{gdiff}\ \mathit{ys}\ c\ n\ \mathit{xs})
\end{aligned}
$$

we have that:

$$\mathit{recipDiffList}\ \mathit{ys} = \mathit{ibuild}\ \mathit{recip}\ (\mathit{gdiff}\ \mathit{ys})$$

Applicative Shortcut Fusion

We can formulate a **shortcut fusion theorem** for intermediate structures with effects introduced by a traversal:

Theorem (Applicative Shortcut Fusion)

$$
\begin{aligned}
& \mathit{fmap}\ (\mathit{foldr}\ f\ \ e) \circ \mathit{traverse}\ \iota \circ \mathit{build}\ g \\
= \ & \\
& \mathit{ifoldr}\ f\ \ e\ \iota \circ \mathit{build}\ g \\
= \ & \\
& \mathit{fmap}\ (\mathit{foldr}\ f\ \ e) \circ \mathit{ibuild}\ \iota\ g \\
= \ & \\
& g\ \phi\ (\mathit{pure}\ e)
\end{aligned}
$$

where $\phi\ x\ y = \mathit{pure}\ f \circledast \iota\ x \circledast y$.

Example

Returning to our example,

$$sumRecipDiffs\ ys = fmap\ sum \circ recipList \circ diffList\ ys$$

if we apply our shortcut fusion theorem, then we obtain a monolithic definition for *sumRecipDiffs ys*:

$$sumRecipDiffs\ ys = gdiff\ \ ys\ (+)\ 0\ recip$$

Inlining,

$$\begin{array}{lll}
sumRecipDiffs & & :: [Float] \to [Float] \to Maybe\ Float \\
sumRecipDiffs\ ys & [\,] & = pure\ 0 \\
sumRecipDiffs\ [\,] & xs & = pure\ 0 \\
sumRecipDiffs\ (y : ys) & (x : xs) & = pure\ (+) \circledast recip\ (y - x) \\
 & & \qquad\qquad \circledast sumRecipDiffs\ ys\ xs
\end{array}$$

Summary

- We presented a **shortcut fusion rule** for applicative computations
- It shows the role of **applicative traversals** as the core of applicative computations over data structures.
- Associated with the rule we introduced the operators *ifold* and *ibuild*, which capture uniform ways of consuming and producing data structures in an applicative context.