

Calculation Rules for Warming-up in Fusion Transformation

Tetsuo Yokoyama, Zhenjiang Hu, and Masato Takeichi

Department of Mathematical Informatics,
Graduate School of Information Science and Technology,
The University of Tokyo
{tetsuo.yokoyama,hu,takeichi}@mist.i.u-tokyo.ac.jp

Abstract

Warm-up transformation is an important preprocess for shortcut fusion. In this paper, we formalize the warm-up transformation by proposing a set of general and powerful calculation rules that can be directly implemented with higher-order pattern matching. The newly formalized warm-up transformation can deal with programs that existing methods may fail, and have been efficiently implemented with the Yicho calculation system. One important advantage of our calculational approach is its compatibility with other calculations, such as fusion, tupling, accumulation, and parallelization. Therefore, warm-up transformation in this form can coexist well with many other calculations in the same system.

Keywords. Warm Fusion, Program Transformation, Calculation

1 INTRODUCTION

Constructing programs from components, i.e., modules, is an important technique in designing large software. Dividing problems into small modules realizes localization of problems. Consequently, programs are easy to maintain. If each module is maintained as a versatile component, reusability is improved. The problem is, however, in its inefficiency due to unnecessary intermediate data structures passed between modules. Improvement of performance can be achieved if these intermediate data structures are removed by automatic analysis and automatic transformation.

Fusion also known as *Deforestation* is a technique for elimination of intermediate data structures [Wad90]. Descendent from it, *shortcut fusion*, a single, local transformation rule was proposed by Gill et al. [GLP93], to ease the implementation [PTH01]. The precondition of shortcut fusion is that the producer function should be expressed in terms of *build*.

If we could predefine all the functions in terms of *foldr* and *build*, shortcut fusion could remove intermediate data structures automatically. Functional programmers, however, usually describe program by general recursive definitions, instead of *foldr* or *build*. Therefore, the problem is how to derive *foldr* or *build* from a recursive definition. There is an approach to automatically deriving *foldr* [SF93,

HIT96], so the problem is settled down into the derivation of *build* from a recursive function.

Warm fusion [LS95] accompanies shortcut fusion with a preprocess (hereafter we call it warm-up transformation). It transforms a class of recursive functions into functions in terms of *build* automatically. The technique has been implemented with the Stratego language [JV00], which is based on term rewriting framework. The implementation of warm fusion, however, appears to be complex. Another approach to deriving *build* is through type inference [Chi99, Chi00]. This approach is easier to implement and is able to transform a wider class of list producing functions into the *build* form than Launchbury and Sheard’s approach [LS95]. But, the implementation is still complicated.

In this paper, we formalize the warm-up transformation by proposing several general and powerful calculation rules [THT98] that can be directly implemented with help of higher-order pattern matching. The newly formalized warm-up transformation cannot only deal with programs that existing methods may fail, but also suitable for efficient implementation.

The rest of this paper is structured as follows. In Section 2, we present warm-up transformation problem and formalize warm-up transformation in calculational form. Depending on the order of functions to be transformed, the calculation rules are different. We show first-order one in Section 2.1 and second-order one in Section 2.2. We discuss future works and conclude our paper in Section 3.

2 WARM-UP TRANSFORMATION

In this section, we formalize the warm-up transformation in the calculational form following the idea in [THT98]. Throughout the paper, we use Haskell notation [Bir98].

The warm-up transformation is to transform recursive programs into constructor-abstraction form (e.g. *build*). Consider, for example, the familiar *map* function, which applies a function to each element of a list.

$$\begin{aligned} \text{map } f \ [] &= [] \\ \text{map } f \ (x : xs) &= f\ x : \text{map } f\ xs \end{aligned}$$

This function produces a list with two constructors (*:*) and *[]*. To abstract these constructors, we introduce function *build*:

$$\text{build } g = g \ (\cdot) \ [].$$

The result of the warm-up transformation for function *map* is

$$\text{map } f = \lambda xs. \text{build } (\lambda c\ n. \text{foldr } (c \circ f) \ n \ xs).$$

Here, function *foldr* (\oplus) *e* is a useful Haskell function, which essentially replaces constructors (*:*) and *[]* in a given list with (\oplus) and *e* respectively.

$$\begin{aligned} \text{foldr } (\oplus) \ e \ [] &= e \\ \text{foldr } (\oplus) \ e \ (x : xs) &= x \oplus \text{foldr } (\oplus) \ e \ xs \end{aligned}$$

The warm-up transformation is followed by the well-known optimization technique, *shortcut fusion*, defined as follows¹.

Lemma 1 (Shortcut fusion [GLP93]).

$$\text{foldr } k \ z \ (\text{build } g) = g \ k \ z$$

Instead of producing a list by passing $(:)$ and $[]$ to g and then replacing list constructors $(:)$ and $[]$ with k and z respectively, we do not produce any intermediate list but directly pass k and z to function g .

To see how the shortcut deforestation rule works, consider the following program:

$$\text{sum } (\text{map } f \ xs)$$

where sum is a function to sum up all elements of a given list:

$$\text{sum} = \text{foldr } (+) \ 0.$$

In GHC [GHC], at the compile time, the shortcut fusion rule is applied in the following way by term rewriting with the simple traversing strategy.

$$\begin{aligned} & \text{sum } (\text{map } f \ xs) \\ = & \ \{ \text{Inline } \text{sum} \ \text{and } \ \text{map} \} \\ & \text{foldr } (+) \ 0 \ (\text{build } (\lambda c \ n. \ \text{foldr } (c \circ f) \ n \ xs)) \\ = & \ \{ \text{Shortcut Fusion} \} \\ & \text{foldr } ((+) \circ f) \ 0 \ xs \end{aligned}$$

While the initial program $\text{sum} \circ \text{map } f$ is inefficient due to the intermediate data structure passed from map to sum , the result does not have any intermediate data structure.

It may be surprising that any recursion can be trivially transformed into the *build* form. Given any function f , both producing and consuming lists, can be rewritten as

$$\lambda xs. \ \text{build } (\lambda c \ n. \ \text{foldr } c \ n \ (f \ (\text{foldr } (:) \ [] \ xs)))$$

However, this introduces an intermediate data structure passing from f to $\text{foldr } c \ n$. If we can fuse $\text{foldr } c \ n$, f , and $\text{foldr } (:) \ []$, we may obtain more efficient code. Though the general fusion problem is very difficult, if the right hand side of the composition is described in terms of *foldr*, they satisfy a number of calculational properties, one of which is the promotion lemma.

Lemma 2 (Promotion).

$$\frac{\begin{array}{l} f \ z \ = \ e \\ f \ (x \odot \ xs) \ = \ x \oplus \ f \ xs \end{array}}{f \circ \text{foldr } (\odot) \ z \ = \ \text{foldr } (\oplus) \ e}$$

¹Strictly speaking, we need certain type restriction on g .

Now return to the fusion of

$$\text{foldr } c \ n \ (f \ (\text{foldr } (\odot) \ [] \ xs)).$$

Function f is merged into function foldr and the result is also foldr form. If (\odot) and z are (\odot) and $[],$ this is merely the definition of the inductive recursive function on list. Then, if f is inductive recursive function and there are (\odot) and e satisfying the above conditions, the warm-up transformation problem is settled down into the problem to fuse

$$\text{foldr } c \ n \circ \text{foldr } (\oplus) \ e.$$

The promotion lemma becomes applicable to the composition. Since the left function of the function composition is always function $\text{foldr } c \ n,$ we can specialize the theorem by using the fact. The theorem varies according to the form of the right $\text{foldr};$ we call that the function foldr is the first-order if it takes three arguments, and is the second-order if it takes extra arguments. We show the first-order case in the next subsection and the second-order case in the following subsection.

2.1 First-order Warm-up Transformation

When the right foldr takes three arguments, we can use the first-order warm-up transformation theorem. Before stating and proving the theorem, we prepare the calculation lemma for fusing two foldr 's.

Lemma 3 (First-order Promotion of Two foldr 's).

$$\begin{array}{l} \lambda c \ n. \text{foldr } c \ n \ e \quad = \quad e' \\ \lambda c \ n. \text{foldr } c \ n \ (x \oplus xs) \quad = \quad x \otimes \lambda c' \ n'. \text{foldr } c' \ n' \ xs \\ \hline \lambda c \ n. \text{foldr } c \ n \ (\text{foldr } (\oplus) \ e \ xs) \quad = \quad \text{foldr } (\otimes) \ e' \ xs \end{array}$$

Proof. Instantiate f in Lemma 2 into $\lambda xs \ c \ n. \text{foldr } c \ n \ xs.$ □

Though the expressive power of this lemma is less than Promotion Theorem, this lemma is more suitable for implementation. It is worth noting that in the second precondition the two foldr s take different arguments. A first-order function which takes a list and returns a list can be transformed into the *build* form by the following theorem.

Theorem 4 (First-order Warm-up Transformation).

$$\begin{array}{l} f \ [] \quad = \quad e \\ f \ (x : xs) \quad = \quad x \oplus f \ xs \\ \lambda c \ n. \text{foldr } c \ n \ e \quad = \quad e' \\ \lambda c \ n. \text{foldr } c \ n \ (x \oplus xs) \quad = \quad x \otimes \lambda c \ n. \text{foldr } c \ n \ xs \\ \hline f \ = \ \lambda xs. \text{build } (\text{foldr } (\otimes) \ e' \ xs) \end{array}$$

Proof. As seen before, any function f which takes a list and produces a list can be trivially transformed into

$$\lambda xs. \text{build } (\lambda c n. \text{foldr } c n (f (\text{foldr } (:) [] xs))).$$

By the promotion lemma, it is transformed into

$$\lambda xs. \text{build } (\lambda c n. \text{foldr } c n (\text{foldr } (\oplus) e xs)).$$

By Lemma 3 and the precondition, we finally obtain

$$f = \lambda xs. \text{build } (\lambda c n. \text{foldr } (\otimes) e' xs c n).$$

□

The input function f is transformed into *build* form; the new operators (\otimes) and e' are produced by the preconditions. Function *foldr* in the argument of *build* is higher-order and takes two extra arguments which abstract constructors.

Since the result of Theorem 4 is always a higher-order *foldr*, it makes inefficient closures when consuming the input. What is worse, sequential shortcut transformation such as

$$\text{foldr } \dots \circ \text{build } \dots \circ \text{build } \dots$$

will stop halfway; even if given *foldr* is first-order, the result of shortcut fusion becomes second-order and this *foldr* can not be fused with the following *build*. We will show the solution to this point in the following example.

Example 5 (map). Consider *map f* function. We attempt to apply Theorem 4. By the definition of *map f*, we obtain

$$\begin{aligned} e &= [] \\ (\oplus) &= ((:) \circ f) \end{aligned}$$

Using these operators, we calculate

$$\begin{aligned} &\text{foldr } c' n' e \\ &= \{ \text{Definition of } e \} \\ &\text{foldr } c' n' [] \\ &= \{ \text{Definition of } \text{foldr} \} \\ &n' \end{aligned}$$

$$\begin{aligned} &\text{foldr } c' n' (x \oplus xs) \\ &= \{ \text{Definition of } \oplus \} \\ &\text{foldr } c' n' (f x : xs) \\ &= \{ \text{Definition of } \text{foldr} \} \\ &c' (f x) (\text{foldr } c' n' xs) \end{aligned}$$

Then, we try to match $e' c' n'$ with n' . By first-order matching, it fails. By second-order matching, it returns $\{e' \mapsto \lambda c' n'. n'\}$. In the following, we use matching for higher-order matching. For inductive case, matching

$$(x \otimes \lambda c' n'. foldr c' n' xs) c' n'$$

with

$$c' (f x) (foldr c' n' xs)$$

returns

$$\{(\otimes) \mapsto \lambda x p c' n'. c' (f x) (p c' n')\}.$$

By Theorem 4 with these conditions, we obtain

$$map f = \lambda xs. build (foldr (\lambda x p c' n'. c' (f x) (p c' n'))) (\lambda c' n'. n') xs).$$

Remarkably, in the argument of *foldr*, c' and n' is passed to p without any change. Thus, we can pass these operators directly to the argument of *foldr*, rather than through the accumulation parameter. This transformation can be considered as an instantiation of the promotion lemma.

Lemma 6 (Removing Accumulation Parameters).

$$\frac{\begin{array}{l} e c = e' \\ (a \oplus x) c = a \otimes x c \end{array}}{foldr (\oplus) e xs c = foldr (\otimes) e' xs}$$

Proof. Instantiate f into $\lambda f. f c$ in the promotion lemma. □

The accumulation parameter c is removed and *foldr* in the right hand side takes one less arguments than that in the left hand side. It is noted that this transformation is mechanical; the user only needs to notice that the accumulation parameters do not change during computation, and just applies this lemma.

Go back to the derivation of *build* form of *map*. With this lemma, the inner *foldr* can be simplified as follows.

$$\begin{aligned} & foldr (\lambda x p c' n'. c' (f x) (p c' n')) (\lambda c' n'. n') xs \\ &= \{ \eta\text{-expansion} \} \\ & \lambda c. foldr (\lambda x p c' n'. c' (f x) (p c' n')) (\lambda c' n'. n') xs c \\ &= \{ \text{Lemma 6} \} \\ & \lambda c. foldr (\lambda x p n'. c (f x) (p n')) id xs \\ &= \{ \eta\text{-expansion} \} \\ & \lambda c n. foldr (\lambda x p n'. c (f x) (p n')) id xs n \\ &= \{ \text{Lemma 6} \} \\ & \lambda c n. foldr (c \circ f) n xs \end{aligned}$$

Finally, we obtain build-form of function *map f*

$$map f = \lambda xs. build (\lambda c n. foldr (c \circ f) n xs)$$

Since the result of the transformation is a first-order *foldr*, shortcut fusion can be sequentially applied. Note that transformation that removes accumulation parameters is not always succeeded, as we will see in the following example.

Example 7 (reverse). Consider the following reverse function:

$$\begin{aligned} \text{reverse } [] &= [] \\ \text{reverse } (x : xs) &= \text{reverse } xs ++ [x] \\ \textbf{where } xs ++ ys &= \text{build } (\lambda c n. \text{foldr } c (\text{foldr } c n ys) xs). \end{aligned}$$

For simplicity, we assume $(++)$ is already in the *build* form. By Theorem 4 with these conditions, we obtain

$$\text{reverse} = \lambda xs. \text{build } (\text{foldr } (\lambda x p c' n'. p c' (c' x n')) (\lambda c' n'. n') xs).$$

Here, in the first argument of *foldr*, c' is passed to p without any change. We may want to pass this operator directly to the argument of *foldr*, and we obtain *build* form of function *reverse* by Lemma 6:

$$\text{reverse} = \lambda xs. \text{build } (\lambda c. \text{foldr } (\lambda x p r. p (c x r)) \text{id } xs).$$

However, operator c changes each time it is passed to p , and should not be removed.

It is worth noting that this result can not be obtained by applying the promotion lemma to

$$\text{foldr } c n \circ \text{foldr } (\lambda x r. r ++ [x]) [],$$

but we have to derive it from

$$\lambda c n. \text{foldr } c n (\text{foldr } (\lambda x r. r ++ [x]) [] xs).$$

Though it is difficult to instantiate f in Promotion Theorem into $\lambda xs c n. \text{foldr } c n xs$ automatically, using Theorem 4 programmer does not have to notice this.

2.2 Second-order Warm-up Transformation

In the previous subsection, we have seen the success of warm-up transformation for reverse function by Theorem 4. But, Theorem 4 will fail for the following linear time reverse function:

$$\begin{aligned} \text{lrev } xs &= \text{lrev}' xs [] \\ \text{lrev}' [] ys &= ys \\ \text{lrev}' (x : xs) ys &= \text{lrev}' xs (x : ys). \end{aligned}$$

This is because *lrev'* has an accumulation parameter.

Hu et al. [HIT99] formulate a systematic treatment of accumulations when function with accumulation is second-order *foldr*. They give calculation theorems for manipulating accumulations. We adapt their accumulation theorems to warm-up transformation as the following two lemmas, where the both functions in the composition are described in terms of *foldr*.

There are two ways to make the promotion lemma second-order, instantiating f as $(\text{foldr } c n \circ)$ and $(\circ \text{foldr } c n)$. First, we show the former.

Lemma 8 (Second-order Fusion of Two *foldr*'s).

$$\frac{\begin{array}{l} \text{foldr } c' \ n' \circ e = e' \ c' \ n' \\ \text{foldr } c' \ n' \circ (a \oplus r) = (a \otimes (\lambda c'' \ n''. \text{foldr } c'' \ n'' \circ r)) \ c' \ n' \end{array}}{\text{foldr } c \ n \circ \text{foldr } (\oplus) \ e \ xs = \text{foldr } (\otimes) \ e' \ xs \ c \ n}$$

The proof is similar to that for Lemma 3. Instantiate f in Lemma 2 into $\lambda r \ c \ n. \text{foldr } c \ n \circ r$. This lemma is useful to prove the following theorem.

Theorem 9 (Second-order Warm-up Transformation 1).

$$\frac{\begin{array}{l} f \ [] = e \\ f \ (x : xs) = x \oplus f \ xs \\ \text{foldr } c' \ n' \circ e = e' \ c' \ n' \\ \text{foldr } c' \ n' \circ (a \oplus r) = (a \otimes (\lambda c'' \ n''. \text{foldr } c'' \ n'' \circ r)) \ c' \ n' \end{array}}{f = \lambda xs \ ys. \text{build } (\lambda c \ n. \text{foldr } (\otimes) \ e' \ xs \ c \ n \ ys)}$$

The proof is similar to that for Theorem 4. As in Theorem 4, the input function f is transformed into *build* form. In this case, the input function takes an extra argument for accumulation, though.

We show the example that Chitil's approach [Chi99] cannot derive *build* form, while Theorem 9 can.

Example 10. Consider computing reverse of the longest increasing prefix.

$$\begin{array}{l} \text{dec } xs = \text{dec}' \ xs \ (-\infty) \\ \text{dec}' \ [] \ _ = [] \\ \text{dec}' \ (x : xs) \ y = \mathbf{if} \ x > y \ \mathbf{then} \ \text{dec}' \ xs \ x \ ++ \ [x] \ \mathbf{else} \ [] \end{array}$$

For example, $\text{dec} \ [1, 3, 5, 4, 2]$ returns $[5, 3, 1]$. Function dec' can be described as a function $\text{foldr} (\oplus) e$ over a list where

$$\begin{array}{l} e = \lambda_. \ [] \\ x \oplus p = \lambda y. \ \mathbf{if} \ x > y \ \mathbf{then} \ p \ x \ ++ \ [x] \ \mathbf{else} \ [] \end{array}$$

This is a second order *foldr* because it takes a list to yield a second-order function.

With the above e and (\oplus) , we can calculate as follows.

$$\begin{aligned}
& \text{foldr } c' \ n' \circ e \\
&= \{ \text{Definition of } e \} \\
& \text{foldr } c' \ n' \circ \lambda_{..} \ [] \\
&= \lambda_{..} . \text{foldr } c' \ n' \ [] \\
&= \{ \text{Definition of foldr} \} \\
&= \lambda_{..} . n' \\
\\
& \text{foldr } c' \ n' \circ (a \oplus r) \\
&= \{ \text{Definition of } (\oplus) \} \\
& \text{foldr } c' \ n' \circ \lambda y. \text{if } x > y \text{ then } r \ x \ ++ \ [x] \ \text{else } [] \\
&= \lambda y. \text{foldr } c' \ n' \ (\text{if } x > y \text{ then } r \ x \ ++ \ [x] \ \text{else } []) \\
&= \{ \text{Distribute foldr } c' \ n' \ \text{over if statement, foldr } c' \ n' \ \text{is strict} \} \\
&= \lambda y. \text{if } x > y \text{ then } \text{foldr } c' \ n' \ (r \ x \ ++ \ [x]) \ \text{else } \text{foldr } c' \ n' \ [] \\
&= \{ \text{Definition of foldr} \} \\
& \lambda y. \text{if } x > y \text{ then } \text{foldr } c' \ n' \ (r \ x \ ++ \ [x]) \ \text{else } n' \\
&= \{ \text{Definition of } (++) \} \\
& \lambda y. \text{if } x > y \text{ then } \text{foldr } c' \ n' \ (\text{build } (\lambda c \ n. \text{foldr } c \ (\text{foldr } c \ n \ [x]) \ (r \ x))) \ \text{else } n' \\
&= \{ \text{Shortcut Fusion} \} \\
& \lambda y. \text{if } x > y \text{ then } \text{foldr } c' \ (\text{foldr } c' \ n' \ [x]) \ (r \ x) \ \text{else } n' \\
&= \{ \text{Definition of foldr} \} \\
& \lambda y. \text{if } x > y \text{ then } \text{foldr } c' \ (c' \ x \ n') \ (r \ x) \ \text{else } n'
\end{aligned}$$

We obtain

$$\begin{aligned}
e' \ c' \ n' &= \lambda_{..} . n' \\
(x \otimes r) \ c' \ n' &= \lambda y. \text{if } x > y \text{ then } r \ c' \ (c' \ x \ n') \ x \ \text{else } n'
\end{aligned}$$

By Theorem 9, we obtain

$$\begin{aligned}
\text{dec} &= \lambda xs. \text{build } (\lambda c \ n. \text{foldr } (\lambda a \ r \ c \ n \ y. \text{if } a > y \text{ then } r \ c \ (c \ a \ n) \ a \ \text{else } n) \\
& \quad (\lambda c \ n \ .. \ n) \ xs \ c \ n \ (-\infty))
\end{aligned}$$

Lemma 6 can be used to simplify the arguments of function foldr , and leading to

$$\begin{aligned}
\text{dec} &= \lambda xs. \text{build } (\lambda c \ n. \text{foldr } (\lambda a \ r \ n \ y. \text{if } a > y \text{ then } r \ (c \ a \ n) \ a \ \text{else } n) \\
& \quad (\lambda n \ .. \ n) \ xs \ n \ (-\infty)).
\end{aligned}$$

Since at each recursion the accumulation parameter n is changed, we cannot apply Lemma 6 anymore.

As mentioned above, there is another instantiation of the promotion theorem into the second-order promotion lemma; here, we instantiate f as $(\circ\text{foldr } c \ n)$. We use the fusion lemma for the reverse direction in the sense that a single foldr is decomposed into two functions g and foldr . The aim of this lemma is to sweep out all the constructor abstraction variables n and c into accumulation parameters.

Lemma 11 (Pushing Computation into Accumulation Parameter).

$$\begin{array}{l}
e r = e' (\text{foldr } c \ n \ r) \\
a \oplus (a_2 \circ \text{foldr } c' \ n') = (a \otimes a_2) \circ \text{foldr } (d_2 \ c' \ n' \ a) \ (d_1 \ c' \ n' \ a) \\
\hline
\text{foldr } (\oplus) \ e \ xs = g \circ \text{foldr } c'' \ n'' \\
\text{where } (g, (c'', n'')) = \text{foldr } (\odot) \ (e', (c, n)) \ xs \\
x \odot (xs_1, (c', n')) = (x \otimes xs_1, (d_2 \ c' \ n' \ x, d_1 \ c' \ n' \ x))
\end{array}$$

Proof. As seen before, any function f which takes a list and produces a list can be trivially transformed into

$$\lambda xs. \text{build } (\lambda c \ n. \text{foldr } c \ n \ (f \ (\text{foldr } (:) \ [] \ xs)))$$

The promotion lemma and the precondition transforms it into

$$\lambda xs. \text{build } (\lambda c \ n. \text{foldr } c \ n \ (\text{foldr } (\oplus) \ e \ xs))$$

By Lemma 3 and the precondition, we obtain

$$f = \lambda xs. \text{build } (\lambda c \ n. \text{foldr } (\otimes) \ e' \ xs \ c \ n).$$

□

This lemma is used to prove the following theorem.

Theorem 12 (Second-order Warm-up Transformation 2).

$$\begin{array}{l}
f [] = e \\
f (x : xs) = x \odot f \ xs \\
\text{foldr } c \ n \circ (a \odot r) = a \oplus (\text{foldr } c \ n \circ r) \\
\text{foldr } c \ n \circ e = e' \circ \text{foldr } c \ n \\
a \oplus (a_2 \circ \text{foldr } c \ n) = (a \otimes a_2) \circ \text{foldr } (d_2 \ c \ n \ a) \ (d_1 \ c \ n \ a) \\
\hline
f = \lambda xs \ r. \text{build } (\lambda c \ n. \text{let } (g, (c'', n'')) = \text{foldr } (\odot) \ (e', (c, n)) \ xs \\
x \odot (xs_1, (c', n')) = (x \otimes xs_1, (d_2 \ c' \ n' \ x, d_1 \ c' \ n' \ x)) \\
\text{in } g \ (\text{foldr } c'' \ n'' \ r))
\end{array}$$

Proof. Since the function f is described as

$$\begin{array}{l}
f [] = e \\
f (x : xs) = x \odot f \ xs
\end{array}$$

we obtain

$$f = \text{foldr } (\odot) \ e \tag{1}$$

By the equation

$$\text{foldr } c \ n \circ (a \odot r) = a \oplus (\text{foldr } c \ n \circ r)$$

and the higher-order promotion lemma gives

$$\text{foldr } c \ n \circ \text{foldr } (\odot) \ e \ xs = \text{foldr } (\oplus) \ (\text{foldr } c \ n \circ e) \ xs \tag{2}$$

By the equations

$$\begin{aligned} e &= e' \circ \text{foldr } c \ n \\ a \oplus (a_2 \circ \text{foldr } c \ n) &= (a \otimes a_2) \circ \text{foldr } (d_2 \ c \ n \ a) \ (d_1 \ c \ n \ a) \end{aligned}$$

and Lemma 11 gives

$$\begin{aligned} \text{foldr } (\oplus) \ (\text{foldr } c \ n \circ e) \ xs &= g \circ \text{foldr } c'' \ n'' \\ \textbf{where } (g, (c'', n'')) &= \text{foldr } (\odot) \ (e', (c, n)) \ xs \end{aligned} \quad (3)$$

Equations (1-3) give the conclusion of the theorem. \square

This theorem abstracts constructors with accumulation parameters. To show it, we borrow the following example from [Voi02]. Warm fusion rewriting rule in [LS95] without higher-order variables and some trick, and type inference base warm-up transformation [Chi99] are not applicable to the function.

Example 13 (Partition). Consider the following function to partition a list xs according to predicate p .

$$\begin{aligned} \text{part } p \ xs &= \textbf{let } f \ [] \ zs = zs \\ &\quad f \ (x : xs) \ zs = \textbf{if } p \ x \ \textbf{then } x : f \ xs \ zs \\ &\quad \quad \quad \textbf{else } f \ xs \ (zs ++ [x]) \\ &\textbf{in } f \ xs \ [] \end{aligned}$$

For example, $\text{part even } [1, 2, 3, 4, 5, 6]$ returns $[2, 4, 6, 1, 3, 5]$. We can apply Theorem 12 to function f . By higher-order matching, we have

$$\begin{aligned} e \ zs &= zs \\ (x \odot r) \ zs &= \textbf{if } p \ x \ \textbf{then } x : r \ zs \ \textbf{else } r \ (zs ++ [x]) \end{aligned}$$

Then, we calculate

$$\begin{aligned} &\text{foldr } c \ n \circ (x \odot r) \\ &= \{ \text{Definition of } (\odot) \} \\ &\quad \text{foldr } c \ n \circ (\lambda zs. \textbf{if } p \ x \ \textbf{then } x : r \ zs \ \textbf{else } r \ (zs ++ [x])) \\ &= \{ \eta\text{-expansion and Definition of } (\odot) \} \\ &\quad \lambda zs. \text{foldr } c \ n \ (\textbf{if } p \ x \ \textbf{then } x : r \ zs \ \textbf{else } r \ (zs ++ [x])) \\ &= \{ \text{Distribute } \text{foldr } c \ n \ \text{over if statement} \} \\ &\quad \lambda zs. \textbf{if } p \ x \ \textbf{then } \text{foldr } c \ n \ (x : r \ zs) \ \textbf{else } \text{foldr } c \ n \ (r \ (zs ++ [x])) \\ &= \{ \text{Definition of } \text{foldr} \} \\ &\quad \lambda zs. \textbf{if } p \ x \ \textbf{then } c \ x \ (\text{foldr } c \ n \ (r \ zs)) \ \textbf{else } \text{foldr } c \ n \ (r \ (zs ++ [x])) \end{aligned}$$

Matching the result with $a \oplus (\text{foldr } c \ n \circ r)$ gives

$$x \oplus r = \lambda zs. \textbf{if } p \ x \ \textbf{then } c \ x \ (r \ zs) \ \textbf{else } r \ (zs ++ [x])$$

For the fourth precondition, we calculate

$$\begin{aligned}
& \text{foldr } c \ n \circ e \\
&= \{ \text{Definition of } e \} \\
& \text{foldr } c \ n \circ (\lambda z s. z s) \\
&= \{ \eta\text{-expansion and Definition of } (\circ) \} \\
& \text{foldr } c \ n
\end{aligned}$$

and obtain

$$e' = \lambda z s. z s$$

For the last precondition, we calculate

$$\begin{aligned}
& a \oplus (a_2 \circ \text{foldr } c \ n) \\
&= \{ \text{Definition of } (\oplus) \} \\
& \lambda z s. \mathbf{if } p \ a \ \mathbf{then } c \ a \ (a_2 \ (\text{foldr } c \ n \ z s)) \ \mathbf{else } a_2 \ (\text{foldr } c \ n \ (z s \ ++ \ [a])) \\
&= \{ \text{Definition of } (\++) \} \\
& \lambda z s. \mathbf{if } p \ a \ \mathbf{then } c \ a \ (a_2 \ (\text{foldr } c \ n \ z s)) \\
& \quad \mathbf{else } a_2 \ (\text{foldr } c \ n \ (\text{build } (\lambda c \ n. \text{foldr } c \ (\text{foldr } c \ n \ [a]) \ z s))) \\
&= \{ \text{Shortcut Fusion} \} \\
& \lambda z s. \mathbf{if } p \ a \ \mathbf{then } c \ a \ (a_2 \ (\text{foldr } c \ n \ z s)) \ \mathbf{else } a_2 \ (\text{foldr } c \ (\text{foldr } c \ n \ [a]) \ z s) \\
&= \{ \text{Definition of } \text{foldr} \} \\
& \lambda z s. \mathbf{if } p \ a \ \mathbf{then } c \ a \ (a_2 \ (\text{foldr } c \ n \ z s)) \ \mathbf{else } a_2 \ (\text{foldr } c \ (c \ a \ n) \ z s) \\
&= \{ \star \} \\
& \lambda z s. (\mathbf{if } p \ a \ \mathbf{then } c \ a \circ a_2 \ \mathbf{else } a_2) \ (\text{foldr } c \ (\mathbf{if } p \ a \ \mathbf{then } n \ \mathbf{else } c \ a \ n) \ z s)
\end{aligned}$$

Matching the result with $(a \otimes a_2) \circ \text{foldr } (d_2 \ c \ n \ a) \ (d_1 \ c \ n \ a)$ returns

$$\begin{aligned}
a \otimes a_2 &= \mathbf{if } p \ a \ \mathbf{then } c \ a \circ a_2 \ \mathbf{else } a_2 \\
d_1 \ c \ n \ a &= \mathbf{if } p \ a \ \mathbf{then } n \ \mathbf{else } c \ a \ n \\
d_2 \ c \ n \ a &= c
\end{aligned}$$

By Theorem 12, we finally obtain

$$\begin{aligned}
\text{part } p \ xs &= \text{build } (\lambda c \ n. \mathbf{let } (g, (c'', n'')) = \text{foldr } (\otimes) \ (\lambda z s. z s, (c, n)) \ xs \\
& \quad x \otimes (xs_1, (c', n')) = (\mathbf{if } p \ x \ \mathbf{then } c' \ x \circ xs_1 \ \mathbf{else } xs_1, \\
& \quad \quad \quad (c', \mathbf{if } p \ x \ \mathbf{then } n' \ \mathbf{else } c' \ x \ n')) \\
& \quad \mathbf{in } g \ n'')
\end{aligned}$$

At each recursive step of *foldr*, value c (c' or c'') is not changed. Therefore, we can hoist this. At the moment, we are not clear how to formalize such hoisting concisely as a calculational rule. Using this human insight and hand calculation, we obtain

$$\begin{aligned}
\text{part } p \ xs &= \text{build } (\lambda c \ n. \mathbf{let } (g, n'') = \text{foldr } (\otimes) \ (\lambda z s. z s, n) \ xs \\
& \quad x \otimes (xs_1, n') = (\mathbf{if } p \ x \ \mathbf{then } c \ x \circ xs_1 \ \mathbf{else } xs_1, \\
& \quad \quad \quad \mathbf{if } p \ x \ \mathbf{then } n' \ \mathbf{else } c \ x \ n') \\
& \quad \mathbf{in } g \ n'')
\end{aligned}$$

Here, the function *part* traverses the input list xs only once.

3 CONCLUSION

In this paper, we have proposed a set of calculation rules for warm-up transformation, which can deal with a wide class of functions some of which the existing methods may fail to derive. These calculation rules are concise and easy to implement; they can be directly programmed as it is formalized. In fact, we have implemented all the calculation rules with the Yicho system² [YHT05] and succeeded in testing all examples except *part*. An important advantage of our calculational approach is its compatibility with other calculations, such as fusion and tupling. Therefore, warm-up transformation in this form can coexist well with many other calculations in the same system. As a future work, we are going to apply our approach for fusing larger and more practical examples.

REFERENCES

- [Bir98] Richard Bird. *Introduction to Functional Programming using Haskell (second edition)*. Prentice Hall, 1998.
- [Chi99] Olaf Chitil. Type inference builds a short cut to deforestation. In *ACM SIGPLAN International Conference on Functional Programming*, volume 34, pages 249–260, Paris, France, September 1999. ACM Press.
- [Chi00] Olaf Chitil. Type-inference based short cut deforestation (nearly) without inlining. In Chris Clack and Pieter Koopman, editors, *Proceedings of 11th International Workshop on Implementation of Functional Languages (1999)*, number 1868 in LNCS, pages 19–36, Netherlands, unknown 2000. Springer.
- [GHC] The glasgow haskell compiler. <http://www.haskell.org/ghc>.
- [GLP93] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Proceedings of the 6th International Conference on Functional Programming Languages and Computer Architecture (FPCA'93)*, pages 223–232, Copenhagen, Denmark, June 1993. ACM Press.
- [HIT96] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Deriving structural hylo-morphisms from recursive definitions. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pages 73–82, 1996.
- [HIT99] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Calculating accumulations. *New Generation Computing*, 17(2), 1999.
- [JV00] Patricia Johann and Eelco Visser. Warm fusion in stratego: A case study in the generation of program transformation systems. Technical Report Technical Report UU-CS-2000-43, Institute of Information and Computing Sciences, Utrecht University, 2000.
- [LS95] John Launchbury and Tim Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *Conference on Functional Programming Languages and Computer Architecture*, pages 314–323, La Jolla, CA, USA, June 1995. ACM.

²Available at <http://www.ip1.t.u-tokyo.ac.jp/yicho>

- [PTH01] Simon L. Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell Workshop*, 2001.
- [SF93] T. Sheard and L. Fegaras. A fold for all seasons. In *Conference on Functional Programming Languages and Computer Architecture*, pages 233–242, 1993.
- [THT98] Akihiko Takano, Zhenjiang Hu, and Masato Takeichi. Program transformation in calculational form. *ACM Comput. Surv.*, 30(3es):7, 1998.
- [Voi02] Janis Voigtländer. Concatenate, reverse and map vanish for free. In *ICFP*, pages 14–25, 2002.
- [Wad90] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.
- [YHT05] Tetsuo Yokoyama, Zhenjiang Hu, and Masato Takeichi. Design and implementation of deterministic higher-order patterns, 2005. Draft. Available at <http://www.ipl.t.u-tokyo.ac.jp/yicho>.