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 several general and powerful calculation rules that can be directly implemented with help of higher-order pattern matching. The newly proposed warm-up transformation cannot only deal with programs that existing methods may fail, but also suitable for efficient implementation.

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 (a.k.a. *Deforestation*) [Wad90] is a technique of elimination of intermediate data structures. Descendent from it, Gill et al. [GLP93] proposes *shortcut fusion*, which is a single, local transformation rule, to ease the implementation [PTH01]. The precondition of shortcut fusion assumes the producer function to be expressed in terms of *build*.

If we could prepare all the functions in terms of *foldr* and *build*, shortcut fusion could remove intermediate data structures automatically. Functional programmer usually describes program as recursive definitions, since *foldr* form and *build* form are less natural than recursive definitions. Therefore, the problem is how to derive *foldr* form and *build* form from a recursive definition. There is an approach to automatically derive the *foldr* form [SF93, HIT96]. Therefore, the problem is settled down into the derivation of *build* form from a recursive function.

Warm fusion [LS95] accompanies shortcut fusion with a preprocess (hereafter we call it warm-up transformation). It transforms 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. There is another approach which derives the *build* form through type inference [Chi99, Chi00]. This approach is easier to implement and is able to transform more list producing functions into the *build* form than Launchbury and Sheard's approach [LS95]. But, the implementation of this approach is still complicated.

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

2 WARM-UP TRANSFORMATION

In this section, we formalize the warm-up transformation in the calculational form. 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.

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

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

$$build\ g = g\ (:)\ []$$

where function g builds up a list with the two data constructors of list. The result of the warm-up transformation of function *map* is

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

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

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

The warm-up transformation is followed by the well-known optimization technique, *shortcut fusion*, defined as follows[1].

Lemma 1 (Shortcut fusion [GLP93]).

$$foldr\ k\ z\ (build\ g) = g\ k\ z$$

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

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

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

$$sum\ (map\ f\ xs)$$

where *sum* is a function of computing summation of a given list:

$$sum \quad = \quad foldr\ (+)\ 0$$

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

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

While the function composition $sum \circ map\ f$ is inefficient due to the intermediate data structure passed from *map* to *sum*, the result function 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

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

However, this introduces an intermediate data structure passing from f to *foldr c n*. If we can fuse *foldr c n*, f, and *foldr* $(:)$ $[]$, we may optimize the code. This can be applied by the following lemma.

Lemma 2 (Promotion).

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

The promotion lemma gives the condition to fuse two functions. Function f is merged into function *foldr* and the result is also *foldr* form. If $(\odot)$ and z are $(:)$ and $[]$, this is merely the definition of the inductive recursive function on list. Then, if f is inductive recursive function and we can find $(\odot)$ and e satisfying the above conditions, the warm-up transformation problem is settled down into the problem to fuse

$$foldr\ c\ n \circ foldr\ (\oplus)\ e.$$

The promotion lemma is applicable to the composition. But, since the left function of the function composition is always function *foldr c n*, we can specialize the

theorem by using the fact. The theorem varies according to the form of the right *foldr*; we call that the function *foldr* is the first-order if it takes three arguments, and is the second-order if it takes the extra argument. We show the first-order case in the next subsection and show 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).

$$\frac{\begin{array}{rcl}\lambda c\ n.\ \mathit{foldr}\ c\ n\ e & = & e' \\ \lambda c\ n.\ \mathit{foldr}\ c\ n\ (x \oplus xs) & = & x \otimes \lambda c\ n.\ \mathit{foldr}\ c\ n\ xs\end{array}}{\lambda c\ n.\ \mathit{foldr}\ c\ n\ (\mathit{foldr}\ (\oplus)\ e\ xs) = \mathit{foldr}\ (\otimes)\ e'\ xs}$$

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

It is worth noting that in the second premise the two *foldr*s can 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).

$$\frac{\begin{array}{rcl}f\ [] & = & e \\ f\ (x : xs) & = & x \oplus f\ xs \\ \lambda c\ n.\ \mathit{foldr}\ c\ n\ e & = & e' \\ \lambda c\ n.\ \mathit{foldr}\ c\ n\ (x \oplus xs) & = & x \otimes \lambda c\ n.\ \mathit{foldr}\ c\ n\ xs\end{array}}{f = \lambda xs.\ \mathit{build}\ (\mathit{foldr}\ (\otimes)\ e'\ xs)}$$

Proof. See Appendix A. □

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 extra two arguments which abstract constructors. We show the application of this theorem to a function in the recursive definition.

Example 5 (reverse). Consider the following function to reverse a list:

$$\begin{array}{lcl}\mathit{reverse}\ [] & = & [] \\ \mathit{reverse}\ (x : xs) & = & \mathit{reverse}\ xs +\!\!+ [x] \\ \quad \textbf{where}\ xs +\!\!+ ys = \mathit{build}\ (\lambda c\ n.\ \mathit{foldr}\ c\ (\mathit{foldr}\ c\ n\ ys)\ xs) & & \end{array}$$

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

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

Remarkably, in the first argument of *foldr*, c' is passed to p without any computation. Thus, we can pass this operator directly to the argument of *foldr*, rather than through the accumulation parameter. This transformation can be as an instantiation of the promotion lemma.

Lemma 6 (Removing Accumulation Parameters).

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

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

The accumulation parameter c is removed and *foldr* in the right hand side takes the one smaller number of arguments than that in the left hand side.

Back to the example, the inner *foldr* can be simplified by Lemma 6. We obtain *build* form of function *reverse*:

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

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

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

2.2 Second-order Warm-up Transformation

In the previous subsection, we have seen function *reverse* has been warm-up transformed by Theorem 4. On the other hand, the following linear time reverse function can not be transformed.

$$\begin{array}{lcl} \mathit{lrev}\ \mathit{xs} & = & \mathit{lrev}'\ \mathit{xs}\ [] \\ \mathit{lrev}'\ []\ \mathit{ys} & = & \mathit{ys} \\ \mathit{lrev}'\ (x : \mathit{xs})\ \mathit{ys} & = & \mathit{lrev}'\ \mathit{xs}\ (x : \mathit{ys}) \end{array}$$

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

Hu et al. [HIT99] formulate systematic treatment of accumulations when function with accumulation is second-order catamorphisms, generalization of *foldr*. They give calculation theorems for treating accumulations. We adapt their accumulation promotion theorems to warm-up transformation as the following two lemmas in the sense that the both functions of the function composition is always described in terms of *foldr*, and we make the patterns in the theorem more suitable for higher-order matching.

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

Lemma 7 (Second-order Fusion of Two *foldr*'s).

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

This lemma is used to prove the following theorem.

Theorem 8 (Second-order Warm-up Transformation 1).

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

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

We show the example that Chitil's approach [Chi99] cannot derive *build* form.

Example 9. Consider computing reverse of the longest increasing prefix.

$$\begin{array}{lcl} \mathit{dec}\ xs & = & \mathit{dec}'\ xs\ (-\infty) \\ \mathit{dec}'\ []\ \_ & = & [] \\ \mathit{dec}'\ (x:xs)\ y & = & \textbf{if}\ x > y\ \textbf{then}\ \mathit{dec}'\ xs\ x +\!\!+ [x]\ \textbf{else}\ [] \end{array}$$

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

$$\begin{array}{lcl} e & = & \lambda\_.\ [] \\ x \oplus p & = & \lambda y.\ \textbf{if}\ x > y\ \textbf{then}\ p\ x +\!\!+ [x]\ \textbf{else}\ [] \end{array}$$

This is a second order *foldr* in the sense that it takes a list to yield a second-order function. Using these operators, we obtain

$$\begin{array}{lcl} e'\ c'\ n' & = & \lambda\_.\ n' \\ (x \otimes r)\ c'\ n' & = & \lambda y.\ \textbf{if}\ x > y\ \textbf{then}\ r\ c'\ (c'\ x\ n')\ x\ \textbf{else}\ n' \end{array}$$

By Theorem 8, we obtain

$$\begin{array}{l} \mathit{dec} = \lambda xs.\ \mathit{build}\ (\lambda c\ n.\ \mathit{foldr}\ (\lambda a\ r\ c\ n\ y.\ \textbf{if}\ a > y\ \textbf{then}\ r\ c\ (c\ a\ n)\ a\ \textbf{else}\ n) \\ \qquad\qquad (\lambda c\ n\ \_.\ n)\ xs\ c\ n\ (-\infty)) \end{array}$$

Lemma 6 simplifies the arguments of function *foldr*, and finally we obtain

$$\begin{array}{l} \mathit{dec} = \lambda xs.\ \mathit{build}\ (\lambda c\ n.\ \mathit{foldr}\ (\lambda a\ r\ n\ y.\ \textbf{if}\ a > y\ \textbf{then}\ r\ (c\ a\ n)\ a\ \textbf{else}\ n) \\ \qquad\qquad (\lambda n\ \_.\ n)\ xs\ n\ (-\infty)) \end{array}$$

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

As mentioned in the above, there is another instantiation of the promotion theorem into the second-order promotion lemma; here, we instantiate f as $(\circ \mathit{foldr}\ c\ n)$. We use the fusion lemma as 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 an accumulation parameter.

Lemma 10 (Pushing Computation into Accumulation Parameter).

$$\frac{\begin{array}{rcl} e\ r & = & e'\ (\mathit{foldr}\ c\ n\ r) \\ a \oplus (a_2 \circ \mathit{foldr}\ c'\ n') & = & (a \otimes a_2) \circ \mathit{foldr}\ (d_2\ c'\ n'\ a)\ (d_1\ c'\ n'\ a) \end{array}}{\begin{array}{l} \mathit{foldr}\ (\oplus)\ e\ xs = g \circ \mathit{foldr}\ c''\ n'' \\ \textbf{where}\ (g,(c'',n'')) = \mathit{foldr}\ (\oslash)\ (e',(c,n))\ xs \\ \qquad x \oslash (xs_1,(c',n')) = (x \otimes xs_1,(d_2\ c'\ n'\ x, d_1\ c'\ n'\ x)) \end{array}}$$

Proof. See Appendix B. □

This lemma is used to prove the following theorem.

Theorem 11 (Second-order Warm-up Transformation 2).

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

Proof. See Appendix C.

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

Example 12 (Partition). Consider a function partitioning a given list xs according to some predicate p.

$$\begin{array}{rcl} \mathit{part}\ p\ xs & = & \textbf{let}\ f\ []\ zs = zs \\ & & \qquad f\ (x:xs)\ zs = \textbf{if}\ p\ x\ \textbf{then}\ x : f\ xs\ zs \\ & & \qquad\qquad\qquad\qquad \textbf{else}\ f\ xs\ (zs +\!\!+ [x]) \\ & & \textbf{in}\ f\ xs\ [] \end{array}$$

For example, *part even* $[1,2,3,4,5,6]$ returns $[2,4,6,1,3,5]$. We can apply Theorem 11 to function f. By higher-order matching, we have

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

Then, we calculate

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

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

$$
\begin{array}{ll}
& \mathit{foldr}\ c\ n \circ e \\
= & \quad \{ \text{ Definition of } e \ \} \\
& \mathit{foldr}\ c\ n \circ (\lambda zs.\ zs) \\
= & \quad \{ \ \eta\text{-expansion and Definition of } (\circ) \ \} \\
& \mathit{foldr}\ c\ n
\end{array}
$$

and obtain

$$e' = \lambda zs.\ zs$$

For the last precondition, we calculate

$$
\begin{array}{ll}
& a \oplus (a_2 \circ \mathit{foldr}\ c\ n) \\
= & \quad \{ \text{ Definition of } (\oplus) \ \} \\
& \lambda zs.\ \textbf{if}\ p\ a\ \textbf{then}\ c\ a\ (a_2\ (\mathit{foldr}\ c\ n\ zs))\ \textbf{else}\ a_2\ (\mathit{foldr}\ c\ n\ (zs +\!\!+ [a])) \\
= & \quad \{ \text{ Definition of } (+\!\!+) \ \} \\
& \lambda zs.\ \textbf{if}\ p\ a\ \textbf{then}\ c\ a\ (a_2\ (\mathit{foldr}\ c\ n\ zs)) \\
& \qquad\qquad \textbf{else}\ a_2\ (\mathit{foldr}\ c\ n\ (\mathit{build}\ (\lambda c\ n.\ \mathit{foldr}\ c\ (\mathit{foldr}\ c\ n\ [a])\ zs))) \\
= & \quad \{ \text{ Shortcut Fusion } \} \\
& \lambda zs.\ \textbf{if}\ p\ a\ \textbf{then}\ c\ a\ (a_2\ (\mathit{foldr}\ c\ n\ zs))\ \textbf{else}\ a_2\ (\mathit{foldr}\ c\ (\mathit{foldr}\ c\ n\ [a])\ zs) \\
= & \quad \{ \text{ Definition of } \mathit{foldr} \ \} \\
& \lambda zs.\ \textbf{if}\ p\ a\ \textbf{then}\ c\ a\ (a_2\ (\mathit{foldr}\ c\ n\ zs))\ \textbf{else}\ a_2\ (\mathit{foldr}\ c\ (c\ a\ n)\ zs) \\
= & \quad \{ \star \} \\
& \lambda zs.\ (\textbf{if}\ p\ a\ \textbf{then}\ c\ a \circ a_2\ \textbf{else}\ a_2)\ (\mathit{foldr}\ c\ (\textbf{if}\ p\ a\ \textbf{then}\ n\ \textbf{else}\ c\ a\ n)\ zs)
\end{array}
$$

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

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

By Theorem 11, we finally obtain

$$\begin{aligned} part\ p\ xs = build\ (\lambda c\ n.\ \mathbf{let}\ (g,(c'',n'')) &= foldr\ (\oslash)\ (e',(c,n))\ xs \\ x \oslash (xs_1,(c',n')) &= (\mathbf{if}\ p\ a\ \mathbf{then}\ c'\ a \circ a_2\ \mathbf{else}\ a_2, \\ &\quad\ (c',\mathbf{if}\ p\ x\ \mathbf{then}\ n'\ \mathbf{else}\ c'\ x\ n')) \\ \mathbf{in}\ g\ n'') & \end{aligned}$$

At the every recursion of *foldr*, value c (c' or c'') is not changed. Therefore, we can hoist this. At the moment, we can not formalize such hoisting in the calculational form. Using this human insight and hand calculation, we obtain

$$\begin{aligned} part\ p\ xs = build\ (\lambda c\ n.\ \mathbf{let}\ (g,n'') &= foldr\ (\oslash)\ (e',n)\ xs \\ x \oslash (xs_1,n') &= (\mathbf{if}\ p\ a\ \mathbf{then}\ c\ a \circ a_2\ \mathbf{else}\ a_2, \\ &\quad\ \mathbf{if}\ p\ x\ \mathbf{then}\ n'\ \mathbf{else}\ c'\ x\ n') \\ \mathbf{in}\ g\ n'') & \end{aligned}$$

It is worth noting that this function traverses the input list *xs* only once.

3 CONCLUSION

We specify the warm-up transformation in calculation rules. We show that we can derive functions in the *build* form which the existing methods may fail to derive. These rules can be directly programmed as it is formalized owing to the expressive power of higher-order matching.

Warm-up transformation in calculation rules can be synthesized on the same formalism as fusion, tupling, accumulation, and parallelization described in calculation rules. We can write such transformations in the same way. We have implemented all the calculation rules in this paper by the Yicho system[2] [YHT05], and succeeded to test all the example except *part*. For function *part*, the splitting the **if** statement (the $\star$ calculation) and the last transformation are not yet specified by calculation rules.

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.

[2] Available at `http://www.ipl.t.u-tokyo.ac.jp/yicho`

[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 hylomorphisms 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.

[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.

[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. Submitted for publication. Available at `http://www.ipl.t.u-tokyo.ac.jp/yicho`.

A FIRST-ORDER WARM-UP TRANSFORMATION

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

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

The promotion lemma and the precondition transforms it into

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

By Lemma 3 and precondition, we finally obtain

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

□

B PUSHING COMPUTATION INTO ACCUMULATION PARAMETER

Proof. We prove by induction over xs. We omit the proof of base case. The calculation of inductive case is

$$
\begin{array}{l}
\text{lhs} \\
= \mathit{foldr}\ (\oplus)\ e\ (x : xs) \\
= x \oplus \mathit{foldr}\ (\oplus)\ e\ xs \\
= \quad \{\ \text{Induction Hypothesis}\ \} \\
\ \ x \oplus (g \circ \mathit{foldr}\ c''\ n'') \\
\ \ \textbf{where}\ (g,(c'',n'')) = \mathit{foldr}\ (\oslash)\ (e',(c,n))\ xs \\
= (x \otimes g) \circ \mathit{foldr}\ (d_2\ c''\ n''\ x)\ (d_1\ c''\ n''\ x) \\
\ \ \textbf{where}\ x \oslash (g,(c'',n'')) = x \oslash \mathit{foldr}\ (\oslash)\ (e',(c,n))\ xs \\
= (x \otimes g) \circ \mathit{foldr}\ (d_2\ c''\ n''\ x)\ (d_1\ c''\ n''\ x) \\
\ \ \textbf{where}\ (x \otimes g,(d_2\ c''\ n''\ x, d_1\ c''\ n''\ x)) = x \oslash \mathit{foldr}\ (\oslash)\ (e',(c,n))\ xs \\
= g \circ \mathit{foldr}\ c''\ n'' \\
\ \ \textbf{where}\ (g,(c'',n'')) = x \oslash \mathit{foldr}\ (\oslash)\ (e',(c,n))\ xs \\
= g \circ \mathit{foldr}\ c''\ n'' \\
\ \ \textbf{where}\ (g,(c'',n'')) = \mathit{foldr}\ (\oslash)\ (e',(c,n))\ (x : xs) \\
= \text{rhs}
\end{array}
$$

□

C SECOND-ORDER WARM-UP TRANSFORMATION 2

Proof. Since the function f is described as

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

we obtain

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

By the equation

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

and the higher-order promotion lemma gives

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

By the equations

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

and Lemma 10 gives

$$
\begin{array}{l}
\mathit{foldr}\ (\oplus)\ (\mathit{foldr}\ c\ n \circ e)\ xs = g \circ \mathit{foldr}\ c''\ n'' \\
\quad \textbf{where}\ (g,(c'',n'')) = \mathit{foldr}\ (\oslash)\ (e',(c,n))\ xs
\end{array} \tag{3}
$$

Equations (1-3) gives the conclusion of the theorem. □