Semantics and Pragmatics of New Shortcut Fusion Rules*

Janis Voigtländer

Institut für Theoretische Informatik
Technische Universität Dresden
01062 Dresden, Germany

voigt@tcs.inf.tu-dresden.de

Abstract. We study various shortcut fusion rules for languages like Haskell. Following a careful semantic account of a recently proposed rule for circular program transformation, we propose a new rule that trades circularity for higher-orderedness, and thus attains better semantic properties. This also leads us to revisit the original foldr/build-rule, as well as its dual, and to develop variants that do not suffer from detrimental impacts of Haskell's mixed strict/nonstrict semantics. Throughout, we offer pragmatic insights about our new rules to investigate also their relative effectiveness, rather than just their semantic correctness.

1 Introduction

These are exciting times for enthusiasts of program transformations akin to shortcut fusion. After the seminal paper on foldr/build-fusion [4], a number of transformations derived from free theorems [15] have been developed over the years, transferring the technique to other types than lists [5, 11], or investigating new transformation schemes of similar flavour [2, 10, 12]. And recently there seems to occur another upsurge of successes in this direction. On the one hand, completely new ideas are developed, such as the circular fusion rule in [3]. On the other hand, existing techniques are revisited and further developed in a way that makes them more useful in practice [1]. And with the integration of call-pattern specialisation into GHC [8], an important building block for successful fusion (post-processing) is falling into place. With this paper we want to continue and nurture this trend, by advancing semantic and pragmatic aspects of existing and new transformations.

We take our start from the pfold/buildp-rule of [3]. It is of particular interest from a semantic viewpoint as it is, due to its use of a circular local binding, the first transformation in the shortcut fusion family that is usable exclusively from a lazy language. This raises questions as to how the rule interacts with the intricacies of Haskell's semantics surrounding $\bot$, fixpoint recursion, and selective strictness. The authors of [3] describe their reasoning as "fast and loose" in this respect. Here we investigate those issues, and prove total and partial correctness results for the circular pfold/buildp-rule in Haskell.

* In *FLOPS 2008, Proc.*, volume 4989 of LNCS, pages 163–179.

Guided by a folklore idea on replacing circularity by higher-orderedness, we then propose a new scheme for pfold/buildp-fusion that not only becomes usable again in a purely strict language, but also plays well with potentially mixed strict/nonstrict evaluation. In fact, we are able to prove total correctness of our new rule without any preconditions on the producer and consumer functions.

The latter is quite remarkable after the completely different experiences made in [6] for the classical foldr/build- and its dual destroy/unfoldr-rule. It leads us to revisit those veteran transformations and to look for potential "repairs" of their semantic deficiencies. And in fact we can transfer some insights and come up with new, and much better behaved, variants of foldr/build- and destroy/unfoldr-fusion.

Throughout, we stay in touch with pragmatic considerations by examining the impact of transformations on concrete examples. This allows us to investigate also the effectiveness, rather than just the correctness, of our new proposals. For example, we carefully weigh the circular and higher-order flavours of pfold/buildp-fusion against each other. And in some cases such pragmatic investigations actually lead to new rule variants.

We deliberately do not focus on a single program transformation. Instead, we report a laboratory-like experience in which working on one rule provides potential insights on another one as well, or indeed sparks a new idea that helps to resolve an issue existing for an at first sight somewhat unrelated fusion problem. This mode of operation has been very fruitful, and we would like to encourage others to push the boundaries of shortcut fusion as well.

2 Circular Shortcut Fusion

In [3] a fusion rule for circular program calculation is proposed. Even though it is originally given for arbitrary algebraic datatypes, we consider only the list case here. For other types the development and results would be similar.

The involved combinators are given as follows:

```
buildp :: (forall a. (b → a → a) → a → c → (a,z)) → c → ([b],z)
buildp g = g (:) []

pfold :: (b → a → z → a) → (z → a) → ([b],z) → a
pfold h1 h2 (bs,z) = foldr (λb a → h1 b a z) (h2 z) bs
```

The idea underlying buildp is that g describes the production of an abstract list (hence the list constructor arguments) over type b, and at the same time delivers an additional result of type z, both guided by an input parameter of type c. A typical application of that scheme is the definition of the following function:

```
splitWhen :: (b → Bool) → [b] → ([b],[b])
splitWhen p = buildp go
  where go con nil bs =
          case bs of
            [] → (nil,bs)
            b:bs' → if p b then (nil,bs)
                    else let (xs,ys) = go con nil bs' in (con b xs, ys)
```

The idea underlying pfold is that h_1 and h_2 describe the consumption of a list over type b by structural recursion (via the standard function foldr), but can take an additional parameter of type z into account while doing so. A typical application of that scheme is the definition of the following function:

```
pfilter :: (b → z → Bool) → ([b],z) → [b]
pfilter p = pfold (λb bs z → if p b z then b:bs else bs) (λ_ → [])
```

This variant of the classical filter-function uses a binary, rather than unary, predicate for selecting list elements, where the second argument of that predicate is fixed throughout and provided as additional input alongside the input list.

The rule from [3] now tells us, in general, to replace as follows:

$$\mathsf{pfold}\ h_1\ h_2\ (\mathsf{buildp}\ g\ c) \quad \rightsquigarrow \quad \mathbf{let}\ (a,z) = g\ (\lambda b\ a \to h_1\ b\ a\ z)\ (h_2\ z)\ c\ \mathbf{in}\ a\,.$$

Note the circularity in the right-hand side, preventing use of the rule in a strict functional language. To see the rule in action, consider the following definition:

```
repeatedAfter :: Eq b ⇒ (b → Bool) → [b] → [b]
repeatedAfter p bs = pfilter elem (splitWhen p bs)
```

It provides a very natural specification of the following task: from the initial part of an input list before a certain predicate holds for the first time, return those elements that are repeated afterwards. To benefit from the circular pfold/buildp-rule, we inline the definitions of pfilter and splitWhen, apply the rule, and afterwards perform some local optimisations as exemplified and described in more detail in [1]. The result is the following version:

```
repeatedAfter′ p bs =
  let
    (a,z) = go′ bs
    go′ bs = case bs of
               [] → ([],bs)
               b:bs′ → if p b then ([],bs) else
                                      let (xs,ys) = go′ bs′
                                      in (if elem b z then b:xs else xs, ys)
  in a
```

Note that even though z and go′ are mutually defined in terms of each other, there is no "true" circularity, as lazy evaluation can order the computation in an appropriate, terminating way.

However, the selective strictness feature of Haskell can ruin this approach in an unexpected way. As an (admittedly artificial) counterexample, consider $g = (\lambda\_\ nil\ c \to \mathsf{seq}\ nil\ (nil,c))$, $c = 42$, $h_2 = (+1)$, and arbitrary (but appropriately typed) h_1. Then $\mathsf{pfold}\ h_1\ h_2\ (\mathsf{buildp}\ g\ c)$ is 43, while the transformed expression $\mathbf{let}\ (a,z) = g\ (\lambda b\ a \to h_1\ b\ a\ z)\ (h_2\ z)\ c\ \mathbf{in}\ a$ does not terminate! To see why, take into account that by inlining g, c, and h_2, it is equivalent to the truly circular $\mathbf{let}\ (a,z) = \mathsf{seq}\ (z+1)\ (z+1,\ 42)\ \mathbf{in}\ a$.

For classical foldr/build-fusion we know from [6] that total correctness even in the presence of seq can be guaranteed by imposing certain restrictions on the arguments to foldr. Trying to transfer those insights to the present setting, we

come to investigate whether $h_1\ \bot\ \bot\ z \neq \bot$ and $h_2\ z \neq \bot$ (because the arguments to foldr in the definition of pfold $h_1\ h_2$ are $(\lambda b\ a \to h_1\ b\ a\ z)$ and $h_2\ z$). But this raises the question which z to consider here. It seems natural to consider the second element of the pair returned by buildp $g\ c$, as that is exactly what gets passed to pfold $h_1\ h_2$ before the circular fusion rule is applied. But tempting as this intuition is, it must be wrong! This is evidenced by the above counterexample, where buildp $g\ c = ([],42)$ and $h_2\ 42 = 43 \neq \bot$ (and h_1 could be chosen arbitrarily, in particular in a way such that $h_1\ \bot\ \bot\ 42 \neq \bot$ as well), and yet we found that applying the circular fusion rule was not semantics-preserving.

This motivates a more careful study of the latter's semantics than is currently available. To help us in this endeavour, we first establish an auxiliary lemma. By convention, a function f is strict if $f\ \bot = \bot$; total if $f\ x \neq \bot$ for every $x \neq \bot$.

Lemma 1. *Let* T_1, T_2, *and* T_3 *be types. Let* $c :: \mathsf{T}_2$ *and*

$$g :: \mathit{forall}\ a.\ (\mathsf{T}_1 \to a \to a) \to a \to \mathsf{T}_2 \to (a,\mathsf{T}_3)\,.$$

Then for every type T', $q :: \mathsf{T}_1 \to \mathsf{T}' \to \mathsf{T}'$, *and strict and total* $f :: [\mathsf{T}_1] \to \mathsf{T}'$,

$$\begin{aligned}(q \neq \bot \wedge \forall\, b :: \mathsf{T}_1.\ q\ b \neq \bot \wedge \forall\, bs :: [\mathsf{T}_1].\ f\ (b{:}bs) = q\ b\ (f\ bs)) \\ \Rightarrow g\ q\ (f\ [])\ c = \mathbf{case}\ \mathsf{buildp}\ g\ c\ \mathbf{of}\ (bs,z') \to (f\ bs, z')\,.\end{aligned} \tag{1}$$

The proof via a free theorem builds on the results from [6] and is given in the appendix. The most important pieces to note here are the preconditions relating to $\bot$ and the strictness and totality restrictions on f. These are exactly the kind of things that one needs to pay close attention to when trying to derive semantic statements that remain valid for Haskell even in the presence of general recursion and selective strictness. Note also that all Haskell types are pointed, so that, for example, the quantification over $b :: \mathsf{T}_1$ includes the case $b = \bot$.

Based on Lemma 1, we can now prove the following theorem which provides the desired preconditions for total correctness of circular pfold/buildp-fusion.

Theorem 1. *Let* T_1, T_2, T_3, *and* T_4 *be types. Let* $c :: \mathsf{T}_2$, $h_1 :: \mathsf{T}_1 \to \mathsf{T}_4 \to \mathsf{T}_3 \to \mathsf{T}_4$, $h_2 :: \mathsf{T}_3 \to \mathsf{T}_4$, *and*

$$g :: \mathit{forall}\ a.\ (\mathsf{T}_1 \to a \to a) \to a \to \mathsf{T}_2 \to (a,\mathsf{T}_3)\,.$$

If $h_1\ \bot\ \bot\ \bot \neq \bot$ *and* $h_2\ \bot \neq \bot$, *then*

$$\mathsf{pfold}\ h_1\ h_2\ (\mathsf{buildp}\ g\ c) \quad = \quad \mathbf{let}\ (a,z) = g\ (\lambda b\ a \to h_1\ b\ a\ z)\ (h_2\ z)\ c\ \mathbf{in}\ a\,.$$

The proof is given in the appendix. In it, the circular binding in the expression after fusion is described by an explicit use of fixpoint recursion. This helps to pin down why it is not enough to require $h_1\ \bot\ \bot\ z \neq \bot$ and $h_2\ z \neq \bot$ for the second element z of the pair returned by buildp $g\ c$: since fixpoint recursion conceptually starts from $\bot$ (as in $\mathsf{fix}\ f = \bigsqcup f^i\ \bot$; the actual definition used in the proof calculation is $\mathsf{fix}\ f = f\ (\mathsf{fix}\ f)$), the circular "hunting" for z in the program after fusion also starts out with $\bot$, which might then interfere with seq. This

precisely explains the counterexample we observed earlier, and why it was not sufficient there that $h_2\ 42 \neq \bot$.

For classical foldr/build-fusion we can avoid preconditions if settling for partial rather than total correctness [6]. Let us see whether the same is possible here. To that end, we need to look at "inequational" versions of the statements we have derived so far. Typically, to any "equational" free theorem correspond two inequational ones. For Lemma 1, one of the two variants is as follows.

Lemma 2. *Let* T_1, T_2, *and* T_3 *be types. Let* $c :: \mathsf{T}_2$ *and*

$$g :: \mathit{forall}\ a.\ (\mathsf{T}_1 \to a \to a) \to a \to \mathsf{T}_2 \to (a, \mathsf{T}_3)\ .$$

Then for every type T', $q :: \mathsf{T}_1 \to \mathsf{T}' \to \mathsf{T}'$, *and strict* $f :: [\mathsf{T}_1] \to \mathsf{T}'$,

$$\begin{array}{c}(\forall\ b :: \mathsf{T}_1,\ bs :: [\mathsf{T}_1].\ f\ (b{:}bs) \sqsupseteq q\ b\ (f\ bs)) \\ \Rightarrow g\ q\ (f\ [])\ c \sqsubseteq \mathbf{case}\ \mathsf{buildp}\ g\ c\ \mathbf{of}\ (bs, z') \to (f\ bs,\ z')\ .\end{array}$$

Note that the new lemma does not require f to be total. The price to pay for this is that the final statement only provides a semantic approximation. The reading of "$\sqsubseteq$" is that the right-hand side is at least as defined as the left-hand side.

As usual, there is also a second inequational variant. However, we have found that it does not lead to any insight beyond what we already know from the equational setting. That is why we only give Lemma 2 here. Based on it, we can prove (largely by mirroring the proof of Theorem 1) the following theorem which establishes partial correctness of circular pfold/buildp-fusion without preconditions.

Theorem 2. *Let* T_1, T_2, T_3, *and* T_4 *be types. Let* $c :: \mathsf{T}_2$, $h_1 :: \mathsf{T}_1 \to \mathsf{T}_4 \to \mathsf{T}_3 \to \mathsf{T}_4$, $h_2 :: \mathsf{T}_3 \to \mathsf{T}_4$, *and*

$$g :: \mathit{forall}\ a.\ (\mathsf{T}_1 \to a \to a) \to a \to \mathsf{T}_2 \to (a, \mathsf{T}_3)\ .$$

Then

$$\mathsf{pfold}\ h_1\ h_2\ (\mathsf{buildp}\ g\ c) \quad \sqsupseteq \quad \mathbf{let}\ (a, z) = g\ (\lambda b\ a \to h_1\ b\ a\ z)\ (h_2\ z)\ c\ \mathbf{in}\ a\ .$$

Note that the counterexample to total correctness given earlier fits into the picture here. There, we observed that 43 got transformed into $\bot$. This certainly agrees with the above statement.

Note also that the partial correctness result does by no means imply that the circular fusion rule decreases definedness *always* when any of the preconditions from Theorem 1 is violated. Indeed, the repeatedAfter-example from earlier in this section does not suffer from any introduction of failure, even though an investigation of the first argument to pfold in that fusion instance shows that the first precondition from Theorem 1 is not fulfilled, given that $(\mathbf{if}\ \mathsf{elem}\ \bot\ \bot\ \mathbf{then}\ \bot{:}\bot\ \mathbf{else}\ \bot) = \bot$. However, the problem is that for such consumers we may not *guarantee* total correctness. For example, one can easily come up with a producer whose fusion with pfilter elem does actually lead to a decrease in definedness, so that Theorem 2 is the best one can say. In particular, it does not make sense to try to prove a somehow "better" Theorem 1 that

makes do with less strong, and therefore more practical, preconditions. There is no circumventing the fact that there exist g of the given type that make the conditions $h_1\ \bot\ \bot\ \bot \neq \bot$ and $h_2\ \bot \neq \bot$ necessary in their full, combined pessimism. Inventing new rules, however, can make a difference.

3 Higher-Order Shortcut Fusion

It is an old idea to replace circular definitions, such as obtained from the elimination of multiple traversals, by higher-order ones. In the terminology of [7], this is achieved by import and export of information. Thus guided, we would like to develop a variant of pfold/buildp-fusion that is unaffected by selective (or, indeed, full) strict evaluation. In doing so, we clearly want to preserve the advantages of the circular fusion rule such as elimination of the intermediate list and effective handling of the additional result produced by buildp and used by pfold. We know from [13] that a transformation of circularity into higher-orderedness is not always possible. But for the setup we consider here, it turns out that there is a way to achieve it for every fusion instance.

Concretely, we propose to replace as follows:

$$\mathsf{pfold}\ h_1\ h_2\ (\mathsf{buildp}\ g\ c)$$
$$\rightsquigarrow$$
$$\mathbf{case}\ g\ (\lambda b\ k\ z \to h_1\ b\ (k\ z)\ z)\ (\lambda z \to h_2\ z)\ c\ \mathbf{of}\ (k,z) \to k\ z\ .$$

Note that there is no circularity in the right-hand side. Indeed, our new rule is applicable in a strict language just as well as in a lazy or mixed evaluation one. It is higher-order in the sense that it uses a function k where the circular rule used a value a.

To see the new rule in action, consider again the function definition for repeatedAfter. After inlining the definitions of pfilter and splitWhen, applying the higher-order fusion rule, and performing local optimisations as mentioned earlier, the result now is the following version:

```
repeatedAfter'' p bs =
  case
    let go' bs = case bs of
                   [] → (λz → [], bs)
                   b:bs' → if p b then (λz → [], bs)
                           else
                             let (xs,ys) = go' bs'
                             in (λz → if elem b z then b:(xs z) else xs z, ys)
    in go' bs
  of (k,z) → k z
```

Another interesting instance is the counterexample we used earlier to demonstrate the weaknesses of the circular fusion rule: $g = (\lambda\_\ nil\ c \to \mathsf{seq}\ nil\ (nil,c))$, $c = 42$, and $h_2 = (+1)$. For the higher-order fusion rule this poses no problems at all: after fusion, we still get 43 as result.

In order to see whether such positive outcome is obtained for every fusion instance, we investigate total correctness of the new rule. We can reuse Lemma 1 to this purpose. Indeed, based on it, we can prove the following theorem.

Theorem 3. *Let* T_1, T_2, T_3, *and* T_4 *be types. Let* $c :: \mathsf{T}_2$, $h_1 :: \mathsf{T}_1 \to \mathsf{T}_4 \to \mathsf{T}_3 \to \mathsf{T}_4$, $h_2 :: \mathsf{T}_3 \to \mathsf{T}_4$, *and*

$$g :: \mathit{forall}\ a.\ (\mathsf{T}_1 \to a \to a) \to a \to \mathsf{T}_2 \to (a, \mathsf{T}_3)\ .$$

Then

$$\mathsf{pfold}\ h_1\ h_2\ (\mathsf{buildp}\ g\ c) \;=\; \mathbf{case}\ g\ (\lambda b\ k\ z \to h_1\ b\ (k\ z)\ z)\ (\lambda z \to h_2\ z)\ c\ \mathbf{of}\ (k, z) \to k\ z\ .$$

We omit further proof details here. In fact, we will do so also for the remaining theorems in this paper. Suffice it to say that they can all be proved using the general goal-directed approach presented in [14].

Analysing *why* total correctness without preconditions holds in Theorem 3 leads to the realization that seq simply cannot do any harm in the presence of the "extra" lambda-abstractions that prevent g from encountering a $\bot$-value when combining its arguments, even though h_1 and/or h_2 might very well contain or produce such values. The need to preserve those protective lambda-abstractions also means that it is not safe to perform eta-reduction of $(\lambda z \to h_2\ z)$ to h_2. Indeed, eta-reduction is not valid in Haskell with seq and should not be performed by any compiler.

That the above theorem establishes total correctness unconditionally is a much more satisfying situation than with circular pfold/buildp-fusion. However, from a pragmatic, rather than semantic, viewpoint the picture is not quite as clear. Consider, for example, the following function definition:

```
greaterThanMinAfter :: Ord b ⇒ (b → Bool) → [b] → [b]
greaterThanMinAfter p bs = pfilter (λb bs′ → b > minimum bs′) (splitWhen p bs)
```

After inlining the definitions of pfilter and splitWhen, applying the higher-order fusion rule, and performing the usual local optimisations, we obtain from it a version greaterThanMinAfter′ whose function body differs from that of repeatedAfter″ seen earlier in this section only in that the third-last line looks as follows:

```
                                  in (λz → if b > minimum z then b:(xs z) else xs z, ys)
```

But precisely this line exposes an issue that we might want to improve on. Namely, we see that minimum z will be computed repeatedly for comparison against all elements of bs until p holds for the first time. There is no apparent way how to avoid this recomputation, even though it is actually the case for given arguments p and bs to greaterThanMinAfter′ that whenever program evaluation reaches this expression, the value of z is the same. But it would simply require too advanced a flow analysis from the compiler to automatically detect this. Similar observations regarding a loss of sharing for shortcut fusion rules were made in [10].

Closer analysis of the above issue reveals that it occurs whenever the consuming pfold would actually be better served with an image, under some function h, of the second element of the pair returned by the producing buildp, rather than with that second element itself. This insight leads us to establish the following slight variation of Theorem 3.

Theorem 4. *Let* T_1, T_2, T_3, T_3', *and* T_4 *be types. Let* $c :: \mathsf{T}_2$, $h :: \mathsf{T}_3' \to \mathsf{T}_3$, $h_1 :: \mathsf{T}_1 \to \mathsf{T}_4 \to \mathsf{T}_3 \to \mathsf{T}_4$, $h_2 :: \mathsf{T}_3 \to \mathsf{T}_4$, *and*

$$g :: \mathit{forall}\ a.\ (\mathsf{T}_1 \to a \to a) \to a \to \mathsf{T}_2 \to (a, \mathsf{T}_3')\,.$$

Then

$$\begin{array}{c}\mathbf{case}\ \mathsf{buildp}\ g\ c\ \mathbf{of}\ (bs,z') \to \mathsf{pfold}\ h_1\ h_2\ (bs,\ h\ z') \\ = \\ \mathbf{case}\ g\ (\lambda b\ k\ z \to h_1\ b\ (k\ z)\ z)\ (\lambda z \to h_2\ z)\ c\ \mathbf{of}\ (k,z) \to k\ (h\ z)\,. \end{array} \quad (2)$$

Note that this theorem again gives total correctness without any preconditions. Unfortunately, it is not as readily applicable for fusion as our earlier results, because the left-hand side is not a simple combination of a producer- and a consumer-combinator. This is easily remedied, though. We can define variants of the original combinators as follows:

```
buildp' :: (forall a. (b → a → a) → a → c → (a,z')) → c → (z' → z) → ([b],z)
buildp' g c h = case g (:) [] c of (bs,z') → (bs, h z')

pfold' :: (b → a → z → a) → (z → a) → (z' → z) → ([b],z') → a
pfold' h1 h2 h (bs,z') = let z = h z' in foldr (λb a → h1 b a z) (h2 z) bs
```

Then Theorem 4 tells us that we can semantics-preservingly replace either of pfold h_1 h_2 (buildp' g c h) and pfold' h_1 h_2 h (buildp g c) by (2) or, indeed, replace as follows:

$$\begin{array}{c}\mathsf{pfold}'\ h_1\ h_2\ h'\ (\mathsf{buildp}'\ g\ c\ h) \\ \rightsquigarrow \\ \mathbf{case}\ g\ (\lambda b\ k\ z \to h_1\ b\ (k\ z)\ z)\ (\lambda z \to h_2\ z)\ c\ \mathbf{of}\ (k,z) \to k\ (h'\ (h\ z))\,. \end{array}$$

The new combinators provide us with the means to define a variant of greaterThanMinAfter that after applying one of the just given semantics-preserving rules leads to a version greaterThanMinAfter'' which avoids repeated computation with minimum by having a function body that differs from the one of repeatedAfter'' only in that the three final lines look as follows:

```
                    in (λz → if b > z then b:(xs z) else xs z, ys)
  in go' bs
of (k,z) → k (minimum z)
```

Note that all the rules proposed in this section are also applicable in a purely strict language. There, however, they can increase (though never decrease) the definedness of a program. Consider, for example, the rule proposed last. If h_1 is a nonterminating expression, then the left-hand side pfold' h_1 h_2 h' (buildp' g c h) is nonterminating as well. But the corresponding right-hand side might very well be terminating, for example if g does not use its first argument.

4 Circular versus Higher-Order Fusion

Having observed that extra effort may be needed to prevent a certain loss of sharing when performing higher-order pfold/buildp-fusion, it should be interesting, by way of comparison, to see whether the same issue exists for the original, circular rule as well. So consider, again, the function definition for greaterThanMinAfter. After inlining the definitions of pfilter and splitWhen, applying the circular fusion rule leads to a version greaterThanMinAfter$'''$ whose function body differs from that of repeatedAfter$'$ seen in Section 2 only in that the next-to-last line looks as follows:

$$\textbf{in}\ (\textbf{if}\ b > \mathsf{minimum}\ z\ \textbf{then}\ b{:}xs\ \textbf{else}\ xs, ys)$$

In contrast to what we had with higher-order fusion, the z is now not locally lambda-bound. In fact, it is not local to the go$'$-function at all. This means that the full laziness transformation [9], to some extent implemented also in GHC, can effectively avoid recomputations of minimum z by floating that whole expression out. Then, a better degree of sharing is achieved than in higher-order fusion prior to introducing extra combinators and rules. However, recall that for circular fusion one cannot guarantee total correctness unrestrictedly. And indeed, the first argument to pfold in the greaterThanMinAfter-example does not satisfy the first precondition from Theorem 1. So the potential for better sharing here is bought by having to settle for only a partial correctness guarantee, or total correctness under conditions that cannot in general be checked automatically by a compiler.

In case full laziness is not implemented in the compiler, or does not "fire", good sharing can be recovered for circular fusion just as it was for higher-order fusion in the previous section, by giving appropriate rules for the generalised combinators pfold$'$ and buildp$'$. Note however, that in the absence of full laziness even the original program before any fusion might suffer from a lack of sharing. For example, an expression pfilter $(\lambda b\ bs' \to b > \mathsf{minimum}\ bs')\ (bs,z)$ is, by the definitions, the same as foldr $(\lambda b\ a \to \textbf{if}\ b > \mathsf{minimum}\ z\ \textbf{then}\ b{:}a\ \textbf{else}\ a)\ []\ bs$. Here only full laziness can prevent minimum z from being calculated repeatedly. This reinstates that pragmatics can be as important as semantics when designing and studying program transformations.

5 Variations of Classical Shortcut Fusion

That the higher-order version of pfold/buildp-fusion turned out to be totally correct without any preconditions is rather pleasing, and raises the question whether a similar "repair" is also possible for classical foldr/build-fusion from [4].

Recall that build is defined as follows:

$$\mathsf{build} :: (\textit{forall}\ a.\ (b \to a \to a) \to a \to a) \to [b]$$
$$\mathsf{build}\ g = g\ (:)\ []$$

and that classical shortcut fusion lets us replace as follows:

$$\mathsf{foldr}\ h_1\ h_2\ (\mathsf{build}\ g) \quad \rightsquigarrow \quad g\ h_1\ h_2\,.$$

As first observed in [12], this transformation is not totally correct in Haskell (while in [6] we have seen that it is partially correct, and totally correct under the preconditions that $h_1\ \bot\ \bot \neq \bot$ and $h_2 \neq \bot$). As mentioned earlier, the reason that total correctness without preconditions could be proved in Theorem 3 (and 4) is that seq could not do any harm there due to the omnipresent lambda-abstractions. This motivates to consider also, for example, the following rule:

$$\mathsf{foldr}\ h_1\ h_2\ (\mathsf{build}\ g) \quad \rightsquigarrow \quad g\ (\lambda b\ k\ z \to h_1\ b\ (k\ z))\ (\lambda z \to h_2)\ ()\ .$$

Note the use of $() :: ()$ as proxy value. Total correctness is established by the following theorem.

Theorem 5. *Let* T_1 *and* T_2 *be types. Let* $h_1 :: \mathsf{T}_1 \to \mathsf{T}_2 \to \mathsf{T}_2$, $h_2 :: \mathsf{T}_2$, *and*

$$g :: \mathit{forall}\ a.\ (\mathsf{T}_1 \to a \to a) \to a \to a\ .$$

Then

$$\mathsf{foldr}\ h_1\ h_2\ (\mathsf{build}\ g) \quad = \quad g\ (\lambda b\ k\ z \to h_1\ b\ (k\ z))\ (\lambda z \to h_2)\ ()\ .$$

The theorem provides a totally correct foldr/build-rule without preconditions. But what about the pragmatics of transformation? Consider the archetypical example for foldr/build-fusion:

```
upTo :: Int → [Int]
upTo n = build (go 1)
  where go i con nil = if i>n then nil else con i (go (i+1) con nil)

sum :: [Int] → Int
sum = foldr (+) 0

sumTo :: Int → Int
sumTo n = sum (upTo n)
```

Inlining the definitions of sum and upTo into that of sumTo and applying the rule suggested above leads to the following version:

```
sumTo' n = go' 1 () where go' i = if i>n then λz → 0 else λz → i+(go' (i+1) z)
```

It is apparent here that all the z will always be bound to the proxy value $()$. But there is little hope that the compiler is smart enough to transform this "plumbing" away. Similar issues appear for variations on the theme of employing higher-orderedness to make foldr/build-fusion unconditionally totally correct.

However, there is an alternative. As noted earlier, the essential role of the lambda-abstractions is to protect g from undesired encounters with $\bot$. But the same feat can be achieved without resorting to higher-orderedness. In fact, we propose a variant that is a kind of "defunctionalisation" of the above idea. To this end, we introduce the following datatype (purposefully *not* a **newtype**, in which case we would get $\mathsf{J}\ \bot = \bot$, contrary to what we want):

```
data J a = J {unJ :: a}
```

Then we propose to replace as follows:

$$\mathsf{foldr}\ h_1\ h_2\ (\mathsf{build}\ g) \quad \leadsto \quad \mathsf{unJ}\ (g\ (\lambda b\ a \to \mathsf{J}\ (h_1\ b\ (\mathsf{unJ}\ a)))\ (\mathsf{J}\ h_2))\,.$$

This is justified by the following theorem which establishes unconditional total correctness.

Theorem 6. *Let* T_1 *and* T_2 *be types. Let* $h_1 :: \mathsf{T}_1 \to \mathsf{T}_2 \to \mathsf{T}_2$, $h_2 :: \mathsf{T}_2$, *and*

$$g :: \mathit{forall}\ a.\ (\mathsf{T}_1 \to a \to a) \to a \to a\,.$$

Then

$$\mathsf{foldr}\ h_1\ h_2\ (\mathsf{build}\ g) \quad = \quad \mathsf{unJ}\ (g\ (\lambda b\ a \to \mathsf{J}\ (h_1\ b\ (\mathsf{unJ}\ a)))\ (\mathsf{J}\ h_2))\,.$$

If we use this new rule to transform the sumTo-example from above, we get the following version:

```
sumTo'' n = unJ (go' 1) where go' i = if i>n then J 0 else J (i+(unJ (go' (i+1))))
```

This again uses extra plumbing, now through the J type. However, there exists a very simple idea of eliminating that. In [8], specialisation of functions for constructor-call-patterns is proposed, with an existing implementation in GHC. The paper also discusses an extension to specialisation for function-call-patterns. Assume this were applied above by introducing a new local function unJ′ such that unJ′ *i* always corresponds to unJ (go′ *i*). This would give:

```
sumTo''' n = unJ' 1 where unJ' i = unJ (if i>n then J 0 else J (i+(unJ' (i+1))))
```

and finally, applying standard optimisations implemented in GHC:

```
sumTo'''' n = unJ' 1 where unJ' i = if i>n then 0 else i+(unJ' (i+1))
```

This is the version we ultimately want to see after foldr/build-fusion, and in fact do get to see after standard foldr/build-fusion à la [4] as well, but now it was obtained going only through perfectly safe transformations, rather than relying on the original fusion rule that may or may not be totally correct in a given situation. And the heuristics required for an implementation along the lines of [8] would be very simple: we can always try to specialise unJ for a function call occurring in its argument, assuming that J is a private datatype of the compiler so that unJ is introduced only during shortcut fusion as above.

At least as interesting as the example above is one where the standard foldr/build-rule actually breaks. Consider the following function:

```
lastEvenOrEmpty :: [INT] → [INT]
lastEvenOrEmpty bs =
  build (λcon nil → foldl' (λa b → if even b then con b nil else a) nil bs)
```

It uses the standard Haskell function foldl′ to return, in a singleton list, the last even element from an integer list provided as input. If no even element exists, the empty list is returned. To allow eventual fusion, the output list is abstracted over via build. Consider further the following function, relying on the above to return the last even element without packaging it in a list:

```
lastEven :: [Int] → Int
lastEven bs = head (lastEvenOrEmpty bs)
```

Here head is another standard Haskell function, defined via foldr as follows:[1]

```
head :: [b] → b
head = foldr (λb _ → b) (error "Prelude.head: empty list")
```

Applying the original foldr/build-rule leads to the following version of lastEven:

```
lastEven′ bs = foldl′ (λa b → if even b then b else a) (error "...") bs
```

Surprisingly, computing lastEven′ [1,2] leads to a runtime error, even though lastEven [1,2] = 2. Actually, an occurrence of seq inside the definition of foldl′:

```
foldl′ :: (a → b → a) → a → [b] → a
foldl′ f = go
  where
    go a bs = case bs of
                []   → a
                b:bs′ → let a′ = f a b in seq a′ (go a′ bs′)
```

has caused foldr/build-fusion to go wrong, in line with the observation that the rule is in general only partially correct.[2]

How, then, about our new rule that according to Theorem 6 is totally correct? Applying it to lastEven and afterwards inlining the definition of foldl′ leads to the following version:

```
lastEven′′ bs = unJ (go (J (error "Prelude.head: empty list")) bs)
  where
    go a bs = case bs of
                []   → a
                b:bs′ → let a′ = (if even b then J b else a) in seq a′ (go a′ bs′)
```

Here the J-constructors prevent unwarranted encounters of seq with ⊥, so that we still have, for example, lastEven′′ [1,2] = 2. Moreover, a clever compiler performing call-pattern specialisation of go for J as implemented in [8], as well as the proposed specialisation of unJ for a fixed function-call-pattern, should be able to transform the above into essentially the following definition without plumbing:

```
lastEven′′′ bs = unJ′ (error "Prelude.head: empty list") bs
  where
    unJ′ a bs = case bs of
                  []   → a
                  b:bs′ → let e = even b in seq e (unJ′ (if e then b else a) bs′)
```

This is safe and effective fusion in the presence of seq!

[1] Actually, head is defined by direct pattern matching in the standard Haskell prelude. However, GHC features a specialised head/build-rule whose effect is exactly the same as that of the general foldr/build-rule in combination with the formulation of head in terms of foldr as given here.

[2] Clearly, without that occurrence of seq, and thus with foldl instead of its strictified version foldl′, we would have obtained lastEven′ [1,2] = 2.

For comparison, if we had sticked with the higher-order approach, Theorem 5 would have led us from lastEven to the following version:

```
lastEven'''' bs = go (λz → error "Prelude.head: empty list") bs ()
  where
    go a bs =
      case bs of
        [] → a
        b:bs' → let a' = (if even b then λz → b else a) in seq a' (go a' bs')
```

This is just as safe as lastEven'' (and lastEven''') above, but seems to offer fewer possibilities for optimising the plumbing away.

6 Variation of the Dual of Classical Shortcut Fusion

Having dealt with classical foldr/build-fusion so successfully, we turn to its dual from [10], trying to tame the impact of seq on that transformation as well. Recall that the relevant combinators are defined as follows:

```
unfoldr :: (c → Maybe (b,c)) → c → [b]
unfoldr psi c = case psi c of
                  Nothing → []
                  Just (b,c') → b:(unfoldr psi c')

destroy :: (forall c. (c → Maybe (b,c)) → c → a) → [b] → a
destroy g = g (λbs → case bs of {[] → Nothing; b:bs' → Just (b,bs')})
```

The destroy/unfoldr-rule tells us to replace as follows:

$$\mathsf{destroy}\ g\ (\mathsf{unfoldr}\ psi\ c) \quad \rightsquigarrow \quad g\ psi\ c\ .$$

But in [6] we found that there are several semantic problems with this rule in Haskell. Even in the absence of seq it is no semantic equivalence, as the right-hand side might be more defined than the left-hand side. And in the presence of seq the even worse situation can occur that there is a decrease of definedness from left to right.

Based on the ideas from the previous section, we can provide a repair now. In fact, the following theorem holds, in the statement of which we use fmap from Haskell's Functor type class for brevity.

Theorem 7. *Let* T_1, T_2, *and* T_3 *be types. Let* $c :: \mathsf{T}_2$, $psi :: \mathsf{T}_2 \to \mathsf{Maybe}\ (\mathsf{T}_1,\mathsf{T}_2)$, *and*

$$g :: \mathit{forall}\ c.\ (c \to \mathsf{Maybe}\ (\mathsf{T}_1,c)) \to c \to \mathsf{T}_3\ .$$

Then

$$\mathsf{destroy}\ g\ (\mathsf{unfoldr}\ psi\ c) \quad \sqsubseteq \quad g\ (\mathsf{fmap}\ (\mathsf{fmap}\ \mathsf{J})\ .\ psi\ .\ \mathsf{unJ})\ (\mathsf{J}\ c)\ .$$

Note that the theorem has no extra preconditions. It thus recovers, for Haskell including seq, the situation that existed for the original destroy/unfoldr-rule in the absence of seq.

To evaluate the new destroy/unfoldr-rule's pragmatic worth, we again consider the sumTo-example, but this time with components expressed as follows:

```
upTo n = unfoldr psiUT 1 where psiUT i = if i>n then NOTHING else JUST (i, i+1)

sum = destroy g where g psi c = case psi c of
                                  NOTHING → 0
                                  JUST (b,c′) → b+(g psi c′)
```

Inlining these two definitions into that of sumTo and applying the rule suggested by Theorem 7 leads to the following version:

```
sumTo′ n = g′ (J 1)
  where g′ c = let i = unJ c in if i>n then 0 else i+(g′ (J (i+1)))
```

To optimise this further, we do not even need an extension of the technique from [8]. Instead, the version of call-pattern specialisation as currently implemented in GHC suffices to give the following version:

```
sumTo″ n = g″ 1 where g″ i = if i>n then 0 else i+(g″ (i+1))
```

Again, the "safety net" that was introduced during fusion to cope with potentially malicious uses of seq could be eliminated completely afterwards.

It would be interesting to see whether one can proceed similarly for stream fusion as recently proposed in [1]. This transformation is essentially derived from destroy/unfoldr-fusion and also suffers from interaction with seq. In particular, the authors of [1] remain rather vague on strictness issues as related to semantic correctness. A repair here would immediately benefit the safety of Haskell programs under the stream fusion optimisation scheme.

7 Discussion

During the course of this paper, we have studied various shortcut fusion rules. Our main contributions are a total correctness result for circular pfold/buildp-fusion under appropriate preconditions (Theorem 1), a corresponding partial correctness result without any preconditions (Theorem 2), an unconditionally totally correct higher-order pfold/buildp-fusion rule (Theorem 3), a variation of the latter scheme with improved sharing (Theorem 4), two unconditionally totally correct foldr/build-fusion rules (Theorems 5 and 6), an unconditionally safely approximating destroy/unfoldr-fusion rule (Theorem 7), as well as pragmatic insights about the relative effectiveness of our new rules.

To put the many rules in perspective, a few comparisons seem in place. First of all, the three flavours "parameter-passing fusion", "classical fusion", and "dual of classical fusion" address different kinds of consumer/producer-pairs, so there is no best choice among them. Below that level of distinction, however, some observations can be made.

One lesson we have learnt is that the variants pfold′ and buildp′ of pfold and buildp, respectively, are more robust with respect to sharing computation on the parameter being passed around. Independently of the correctness issue, the decision whether to prefer circular or higher-order fusion for this setting could

be largely influenced by the relative impact of these two strategies on efficiency. Preliminary measurements indicate that both are about on a par, but more systematic experimentation might provide new insights here.

For classical fusion and its dual, we have proposed two alternative strategies to improve semantic properties, in particular to prevent a decrease in program definedness via transformation. Of these, we clearly favour the approach via the datatype J over the approach via extra lambda-abstractions. The reason is that we then see better potential for automatic subsequent removal of the plumbing introduced to prevent undesirable encounters between seq and $\bot$. Even for the J-approach, the situation is not yet fully satisfactory. While for our destroy/unfoldr-example standard constructor-call-pattern specialisation suffices, the kind of post-processing required in the foldr/build-setting was more ad-hoc. Hopefully, a more general solution can be found here. Otherwise, it is unclear how the better behaviour in semantic regards will weigh up against pragmatic efficiency risks. Does correctness really trump performance?

The story of new fusion rules does not end here. One reviewer proposal was to apply the J-approach also to circular pfold/buildp-fusion. In fact, the rule

$$\begin{array}{c} \mathsf{pfold}\ h_1\ h_2\ (\mathsf{buildp}\ g\ c) \\ \rightsquigarrow \\ \mathbf{let}\ (a,z) = g\ (\lambda b\ a \to \mathtt{J}\ (h_1\ b\ (\mathsf{unJ}\ a)\ z))\ (\mathtt{J}\ (h_2\ z))\ c\ \mathbf{in}\ \mathsf{unJ}\ a \end{array}$$

can be shown unconditionally totally correct by a relatively straightforward adaptation of the proof in Appendix A.2, and a variant for pfold′ and buildp′ is possible as well. However, post-processing becomes even more of a problem here. Already for the repeatedAfter-example from Section 2 the above rule leads to a transformed program for which it is considerably more difficult to conceive of a successful plumbing-removal than for the examples seen in Sections 5 and 6.

Acknowledgements. I thank the anonymous reviewers for their detailed comments and suggestions, most of which I have tried to follow up on.

References

1. D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *International Conference on Functional Programming, Proceedings*, pages 315–326. ACM Press, 2007.
2. F. Domínguez and A. Pardo. Program fusion with paramorphisms. In *Mathematically Structured Functional Programming, Proceedings*, Electronic Workshops in Computing. British Computer Society, 2006.
3. J.P. Fernandes, A. Pardo, and J. Saraiva. A shortcut fusion rule for circular program calculation. In *Haskell Workshop, Proceedings*, pages 95–106. ACM Press, 2007.
4. A. Gill, J. Launchbury, and S.L. Peyton Jones. A short cut to deforestation. In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 223–232. ACM Press, 1993.

5. P. Johann. A generalization of short-cut fusion and its correctness proof. *Higher-Order and Symbolic Computation*, 15(4):273–300, 2002.
6. P. Johann and J. Voigtländer. The impact of *seq* on free theorems-based program transformations. *Fundamenta Informaticae*, 69(1–2):63–102, 2006.
7. A. Pettorossi and M. Proietti. Importing and exporting information in program development. In *Partial Evaluation and Mixed Computation, Proceedings*, pages 405–425. North-Holland, 1987.
8. S.L. Peyton Jones. Call-pattern specialisation for Haskell programs. In *International Conference on Functional Programming, Proceedings*, pages 327–337. ACM Press, 2007.
9. S.L. Peyton Jones and D. Lester. A modular fully-lazy lambda lifter in Haskell. *Software Practice and Experience*, 21(5):479–506, 1991.
10. J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *International Conference on Functional Programming, Proceedings*, pages 124–132. ACM Press, 2002.
11. A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 306–313. ACM Press, 1995.
12. J. Voigtländer. Concatenate, reverse and map vanish for free. In *International Conference on Functional Programming, Proceedings*, pages 14–25. ACM Press, 2002.
13. J. Voigtländer. Using circular programs to deforest in accumulating parameters. *Higher-Order and Symbolic Computation*, 17(1–2):129–163, 2004.
14. J. Voigtländer. Proving correctness via free theorems: The case of the destroy/build-rule. In *Partial Evaluation and Semantics-Based Program Manipulation, Proceedings*, pages 13–20. ACM Press, 2008.
15. P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359. ACM Press, 1989.

A Proofs Appendix

A.1 Proof of Lemma 1

Proof. The "equational" free theorem derived from the type of g is that for every choice of types T and T', strict and total $f :: \mathsf{T} \to \mathsf{T}'$, and arbitrary $p :: \mathsf{T}_1 \to \mathsf{T} \to \mathsf{T}$ and $q :: \mathsf{T}_1 \to \mathsf{T}' \to \mathsf{T}'$,

$$\begin{aligned}&((p \neq \bot \Leftrightarrow q \neq \bot)\\ &\quad \wedge \forall\, b :: \mathsf{T}_1.\ (p\ b \neq \bot \Leftrightarrow q\ b \neq \bot) \wedge \forall\, bs :: \mathsf{T}.\ f\ (p\ b\ bs) = q\ b\ (f\ bs))\\ &\Rightarrow \forall\, u :: \mathsf{T}, c :: \mathsf{T}_2.\\ &\qquad (g\ p\ u\ c, g\ q\ (f\ u)\ c) \in \{(\bot,\bot)\} \cup \{((bs,z),(bs',z')) \mid f\ bs = bs' \wedge z = z'\}\end{aligned}$$

We instantiate $\mathsf{T} = [\mathsf{T}_1]$, $p = (:)$, and $u = []$, observe that then $p \neq \bot$ and $p\ b \neq \bot$ for every $b :: \mathsf{T}_1$, and use the definition of buildp. This gives that if the precondition (1) holds, then for every $c :: \mathsf{T}_2$,

$$(\mathsf{buildp}\ g\ c, g\ q\ (f\ [])\ c) \in \{(\bot,\bot)\} \cup \{((bs,z),(bs',z')) \mid f\ bs = bs' \wedge z = z'\}.$$

The lemma follows easily from this. □

A.2 Proof of Theorem 1

We need the following auxiliary lemma.

Lemma 3. *Let* T_1, T_3, *and* T_4 *be types. Let* $h_1 :: \mathsf{T}_1 \to \mathsf{T}_4 \to \mathsf{T}_3 \to \mathsf{T}_4$, $h_2 :: \mathsf{T}_3 \to \mathsf{T}_4$, $bs :: [\mathsf{T}_1]$, *and* $z' :: \mathsf{T}_3$. *Then*

$$\mathsf{fst}\ (\mathsf{fix}\ (\lambda\ \tilde{}(\_,z) \to (\mathsf{foldr}\ (\lambda b\ a \to h_1\ b\ a\ z)\ (h_2\ z)\ bs, z')))$$

is equivalent to

$$\mathsf{foldr}\ (\lambda b\ a \to h_1\ b\ a\ z')\ (h_2\ z')\ bs\ .$$

Proof. Let $\mathsf{exp} = \mathsf{fix}\ (\lambda\ \tilde{}(\_,z) \to (\mathsf{foldr}\ (\lambda b\ a \to h_1\ b\ a\ z)\ (h_2\ z)\ bs, z'))$. It is easy to see that exp is equivalent to

$$(\lambda z \to (\mathsf{foldr}\ (\lambda b\ a \to h_1\ b\ a\ z)\ (h_2\ z)\ bs, z'))\ (\mathsf{snd}\ \mathsf{exp})\ .$$

This implies that $\mathsf{fst}\ \mathsf{exp}$ is equivalent to

$$\mathsf{foldr}\ (\lambda b\ a \to h_1\ b\ a\ (\mathsf{snd}\ \mathsf{exp}))\ (h_2\ (\mathsf{snd}\ \mathsf{exp}))\ bs\ ,$$

while $\mathsf{snd}\ \mathsf{exp}$ is equivalent to z'. The lemma follows easily from these facts. □

Now we can prove Theorem 1.

Proof. For every $z :: \mathsf{T}_3$, Lemma 1 with $\mathsf{T}' = \mathsf{T}_4$, $q = (\lambda b\ a \to h_1\ b\ a\ z)$, and $f = \mathsf{foldr}\ (\lambda b\ a \to h_1\ b\ a\ z)\ (h_2\ z)$ gives

$$g\ (\lambda b\ a \to h_1\ b\ a\ z)\ (h_2\ z)\ c$$
$$=$$
$$\mathbf{case}\ \mathsf{buildp}\ g\ c\ \mathbf{of}\ (bs,z') \to (\mathsf{foldr}\ (\lambda b\ a \to h_1\ b\ a\ z)\ (h_2\ z)\ bs, z')\ .$$

Note that the assumptions on h_1 and h_2 are equivalent to the requirement that the chosen f is strict and total for every $z :: \mathsf{T}_3$.

We express

$$\mathbf{let}\ (a,z) = g\ (\lambda b\ a \to h_1\ b\ a\ z)\ (h_2\ z)\ c\ \mathbf{in}\ a$$

via explicit fixpoint recursion as follows:

$$\mathsf{fst}\ (\mathsf{fix}\ (\lambda\ \tilde{}(\_,z) \to g\ (\lambda b\ a \to h_1\ b\ a\ z)\ (h_2\ z)\ c))\ .$$

By the above, this is equivalent to

$$\mathsf{fst}\ (\mathsf{fix}\ (\lambda\ \tilde{}(\_,z) \to \mathbf{case}\ \mathsf{buildp}\ g\ c\ \mathbf{of}\ (bs,z') \to (\mathsf{foldr}\ (\lambda b\ a \to h_1\ b\ a\ z)\ (h_2\ z)\ bs, z')))\ .$$

The subexpression $\mathsf{buildp}\ g\ c$ is either equivalent to $\bot$ or to (bs,z') for some fixed bs and z'. In both cases, the full expression is equivalent to

$$\mathbf{case}\ \mathsf{buildp}\ g\ c\ \mathbf{of}\ (bs,z') \to \mathsf{foldr}\ (\lambda b\ a \to h_1\ b\ a\ z')\ (h_2\ z')\ bs\ .$$

If $\mathsf{buildp}\ g\ c = \bot$, then this equivalence holds by $\mathsf{fst}\ (\mathsf{fix}\ (\lambda\_ \to \bot)) = \bot$. Otherwise, it follows from Lemma 3. By the definition of pfold we finally get equivalence to $\mathsf{pfold}\ h_1\ h_2\ (\mathsf{buildp}\ g\ c)$. □