Multiple Intermediate Structure Deforestation by Shortcut Fusion*

Alberto Pardo[1], João Paulo Fernandes[2,3], and João Saraiva[2]

[1] Instituto de Computación, Universidad de la República, Uruguay
pardo@fing.edu.uy
[2] HASLab / INESC TEC, Universidade do Minho, Portugal
{jpaulo,jas}@di.uminho.pt
[3] Reliable and Secure Computation Group ((Rel)ease), Universidade da Beira Interior, Portugal

Abstract. Shortcut fusion is a well-known optimization technique for functional programs. Its aim is to transform multi-pass algorithms into single pass ones, achieving deforestation of the intermediate structures that multi-pass algorithms need to construct. Shortcut fusion has already been extended in several ways. It can be applied to monadic programs, maintaining the global effects, and also to obtain circular and higher-order programs. The techniques proposed so far, however, only consider programs defined as the composition of a single producer with a single consumer. In this paper, we analyse shortcut fusion laws to deal with programs consisting of an arbitrary number of function compositions.

1 Introduction

Shortcut fusion [1] is an important optimization technique for functional programs. It was proposed as a technique for the elimination of intermediate data structures generated in function compositions $fc \circ fp$, where a producer $fp :: a \to t$ builds a data structure t, which is then traversed by a consumer $fc :: t \to b$ to produce a result. When some conditions are satisfied, we may transform these programs into equivalent ones that do not construct the intermediate structure t.

Extended forms of shortcut fusion have also been proposed to eliminate intermediate structures in function compositions in which the producer and the consumer share some additional context information. These extensions transform compositions $fc \circ fp$, where $fp :: a \to (t, z)$ and $fc :: (t, z) \to b$, into circular [2,3] and higher-order [3,4] equivalent programs, and have increased the applicability of shortcut fusion. Nevertheless, they consider programs consisting of the composition between two functions only. As a consequence, it is not possible to (straightforwardly) apply such techniques to programs that rely on multiple traversal strategies, like compilers and advanced pretty-printing algorithms [5].

* This work is funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within projects FCOMP-01-0124-FEDER-020532 and FCOMP-01-0124-FEDER-022701.

A. Rauber Du Bois and P. Trinder (Eds.): SBLP 2013, LNCS 8129, pp. 120–134, 2013.

The main contribution of this paper is to present generalized forms of shortcut fusion which apply to an arbitrary number of function compositions of the form $f_n \circ \cdots \circ f_1$, for $n > 2$. We establish sufficient conditions on each f_i that guarantee that consecutive fusion steps are applicable when following both a left-to-right and a right-to-left strategy. By means of what we call *chain laws*, we show how to obtain the intermediate fused definitions in such a way that further fusion steps apply. The formulation of the chain laws is the result of combining two fusion approaches: that of shortcut fusion and the one used in the formulation of fusion laws known as *acid rain* [6]. Our fusion method, characterised by requiring certain conditions on the functions involved in the compositions, differs from that employed by *warm fusion* [7].

We analyse two cases of multi-traversal programs: a) the standard case where only a data structure is passed between producer and consumer, and b) programs where in each composition, besides a data structure, some additional information is passed. Case b) is particularly interesting because it is the case where circular and higher-order programs are derived by applying fusion. The type of circular programs we derive are the natural representation of Attribute Grammars (AG) in a lazy setting [8,9]. In the AG community, however, a multi-pass program is usually derived from an AG (i.e., from a circular program) [10,11]. In this paper we study and prove correctness of the opposite transformation, that is, we study how a circular program (i.e., an AG) is derived from a multi-pass one.

Throughout the paper we will use Haskell notation, assuming a cpo semantics in terms of pointed cpos, but without the presence of the *seq* function [12]. For the sake of presentation, we will focus on definitions, laws, and examples over the list datatype only. A generic formulation of all concepts and laws developed in the paper, as well as their proofs, valid for a wide range of datatypes, is presented in an extended version of the paper.

The paper is organized as follows. In Section 2 we review shortcut fusion while in Section 3 we do so with the laws that serve for the derivation of circular and higher-order programs. In Section 4 we analyse the problem of fusing multi-traversal programs and present laws that give conditions for the derivation of circular and higher-order programs in those cases. Section 5 concludes the paper.

2 Shortcut Fusion

Shortcut fusion [1] is a program transformation technique for the elimination of intermediate data structures generated in function compositions. This technique is a consequence of parametricity properties, known as "free theorems" [13], associated with polymorphic functions. For its application, shortcut fusion requires the consumer to process all the elements of the intermediate data structure in a uniform way. This condition is established by requiring that the consumer is expressible as a *fold* [14], a program scheme that captures function definitions by structural recursion. For example, for lists, *fold* is defined as:

$$
\begin{aligned}
&\mathit{fold} && :: (b, a \to b \to b) \to [\,a\,] \to b \\
&\mathit{fold}\ (\mathit{nil}, \mathit{cons})\ [\,] && = \mathit{nil} \\
&\mathit{fold}\ (\mathit{nil}, \mathit{cons})\ (a : \mathit{as}) && = \mathit{cons}\ a\ (\mathit{fold}\ (\mathit{nil}, \mathit{cons})\ \mathit{as})
\end{aligned}
$$

This function is equivalent to the well-known *foldr* function [14]. It traverses the list and replaces [] by the constant *nil* and the occurrences of (:) by function *cons*. The pair $(nil, cons)$ is called an algebra. We denote by $in_L = ([\,], (:))$ the algebra composed by the list constructors. For example, the function that selects the elements of a list that satisfy a given predicate:

$$\begin{aligned} &\mathit{filter} && :: (a \to \mathit{Bool}) \to [a] \to [a] \\ &\mathit{filter}\ \mathit{pr}\ [\,] && = [\,] \\ &\mathit{filter}\ \mathit{pr}\ (a : \mathit{as}) && = \mathbf{if}\ \mathit{pr}\ a\ \mathbf{then}\ a : \mathit{filter}\ \mathit{pr}\ \mathit{as}\ \mathbf{else}\ \mathit{filter}\ \mathit{pr}\ \mathit{as} \end{aligned}$$

can be written in terms of *fold* as follows:

$$\begin{aligned} \mathit{filter}\ \mathit{pr} = \mathit{fold}\ (\mathit{fnil}, \mathit{fcons})\ \mathbf{where}\ \mathit{fnil} &= [\,] \\ \mathit{fcons}\ a\ r &= \mathbf{if}\ \mathit{pr}\ a\ \mathbf{then}\ a : r\ \mathbf{else}\ r \end{aligned}$$

On the other hand, the producer must be a function such that the computation that builds the output data structure consists of the repeated application of the data type constructors. To meet this condition the producer is required to be expressible in terms of a function, called *build* [1], which carries a "template" (a polymorphic function) that abstracts the occurrences of the constructors of the intermediate data type. In the case of lists:

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

The polymorphic type of g ensures that it can only construct its result of type b by using the operations of its argument algebra of type $(b, a \to b \to b)$. As a result, *build* returns a list that is uniformly constructed by the repeated application of the list constructors [] and (:). For example, the function that constructs the list of numbers between n and 1:

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

can be written in terms of *build* as follows:

$$\begin{aligned} \mathit{down} = \mathit{build}\ \mathit{gd}\ \mathbf{where}\ \mathit{gd}\ (\mathit{nil}, \mathit{cons})\ 0 &= \mathit{nil} \\ \mathit{gd}\ (\mathit{nil}, \mathit{cons})\ n &= \mathit{cons}\ n\ (\mathit{gd}\ (\mathit{nil}, \mathit{cons})\ (n-1)) \end{aligned}$$

The essential idea behind shortcut fusion is then to replace, in the producer, the occurrences of the intermediate datatype's constructors (abstracted in the "template" of the build) by appropriate values/functions specified within the consumer (the algebra carried by the fold). As a result, one obtains a definition that computes the same as the original composition but without building the intermediate data structure. This transformation is usually referred to as the *fold/build* law.

Law 1 (FOLD/BUILD FOR LISTS) $\mathit{fold}\ (\mathit{nil}, \mathit{cons}) \circ \mathit{build}\ g = g\ (\mathit{nil}, \mathit{cons})$

To see an example of application of this law, let us consider the function that computes the factors of a number:

$$\begin{aligned} &\mathit{factors} && :: \mathit{Int} \to [\mathit{Int}] \\ &\mathit{factors}\ n && = \mathit{filter}\ (\text{`}\mathit{isFactorOf}\text{`}\ n)\ (\mathit{down}\ (n\ \text{`}\mathit{div}\text{`}\ 2)) \end{aligned}$$

$\quad$ **where** x `` `isFactorOf` `` $n = n$ `` `mod` `` $x == 0$

Since *filter* is a fold and *down* a build, we can apply the law to eliminate the intermediate list. If we define $fd\ pr = filter\ pr \circ down$, then $factors\ n = fd$ (`` `isFactorOf` `` n) (n `` `div` `` 2) and by Law 1 we obtain $fd\ pr = gd\ (fnil, fcons)$. Inlining we get the following recursive definition for $fd\ pr$:

$fd\ pr\ 0 = [\,]$

$fd\ pr\ n =$ **if** $pr\ n$ **then** $n : fd\ pr\ (n-1)$ **else** $fd\ pr\ (n-1)$

It is possible to formulate an extended form of shortcut fusion which captures the case when the intermediate data structure is generated as part of another structure. This extension has been fundamental for the formulation of shortcut fusion laws for monadic programs [15,16], and for the derivation of (monadic) circular and higher-order programs [3]. It is based on an extended form of build:

$build_N :: (\forall\ b\ .\ (b, a \to b \to b) \to c \to N\ b) \to c \to N\ [a]$

$build_N\ g = g\ in_L$

where N represents a data structure in which the produced list is contained. Technically, N is a *functor*, i.e. a type constructor N of kind $\star \to \star$ equipped with a function $map_N :: (a \to b) \to (N\ a \to N\ b)$, which preserves identities and compositions: $map_N\ id = id$ and $map_N\ (f \circ g) = map_N\ f \circ map_N\ g$.

The above is a natural extension of the standard build function. In fact, *build* can be obtained from $build_N$ by considering the identity functor corresponding to the identity type constructor: **type** $N\ a = a$ and $map_N\ f = f$.

Based on this extended form of build an extended shortcut fusion law can be formulated (see [15] for a proof):

Law 2 (EXTENDED FOLD/BUILD) *For strictness-preserving* [1] N,

$$map_N\ (fold\ (nil, cons)) \circ build_N\ g = g\ (nil, cons)$$

To see an example consider the following composition, where *filterLen* is the function that, given a list of numbers, returns a pair formed by a list containing the positive numbers together with the length of the output list.

$sumFilLen = (sum \times id) \circ filterLen$

$sum\ ::\ Num\ a \Rightarrow [a] \to a$

$sum\ [\,] = 0$

$sum\ (a : as) = a + sum\ as$

$filterLen\ ::\ Num\ a \Rightarrow [a] \to ([a], Int)$

$filterLen\ [\,] = ([\,], 0)$

$filterLen\ (x : xs) =$ **if** $x > 0$ **then** $(x : ys, 1 + l)$ **else** (ys, l)

$\quad$ **where** $(ys, l) = filterLen\ xs$

where $(f \times g)$ is defined as $(f \times g)\ (x, y) = (f\ x, g\ y)$; id is the identity function. To simplify the expression of *sumFilLen* we first observe that *filterLen* can be written as a $build_N$ with functor $N\ a = (a, Int)$ and $map_N\ f = f \times id$.

[1] By strictness-preserving we mean that map_N preserves strict functions, i.e. if f is strict, then so is $map_N\ f$.

$$
\begin{aligned}
&\mathit{filterLen} = \mathit{build}_N\ \mathit{gfL}\\
&\quad \textbf{where}\\
&\qquad \mathit{gfL}\ (\mathit{nil}, \mathit{cons})\ [\,] \qquad = (\mathit{nil}, 0)\\
&\qquad \mathit{gfL}\ (\mathit{nil}, \mathit{cons})\ (x : xs) = \textbf{if}\ x > 0\ \textbf{then}\ (\mathit{cons}\ x\ ys, 1 + l)\ \textbf{else}\ (ys, l)\\
&\qquad\quad \textbf{where}\ (ys, l) = \mathit{gfL}\ (\mathit{nil}, \mathit{cons})\ xs
\end{aligned}
$$

It is easy to see that *sum* is a fold: $\mathit{sum} = \mathit{fold}\ (0, (+))$. Then, by applying Law 2 we can deduce: $\mathit{sumFilLen} = (\mathit{fold}\ (0, (+)) \times \mathit{id}) \circ \mathit{build}_N\ \mathit{gfL} = \mathit{gfL}\ (0, (+)))$, which corresponds to the following recursive definition:

$$
\begin{aligned}
&\mathit{sumFilLen}\ [\,] \qquad = (0, 0)\\
&\mathit{sumFilLen}\ (x : xs) = \textbf{if}\ x > 0\ \textbf{then}\ (x + s, 1 + l)\ \textbf{else}\ (s, l)\\
&\quad \textbf{where}\ (s, l) = \mathit{sumFilLen}\ xs
\end{aligned}
$$

3 Circular and Higher-Order Programs

In this section we review the laws that make it possible to derive circular as well as higher-order programs from function compositions that communicate through an intermediate pair (t, z), where t is a data structure and z some additional information. The derivation can be done both for pure and monadic programs (see [3]), and like for shortcut fusion, it requires both consumer and producer to be expressible in terms of certain program schemes. The consumer is required to be a structural recursive definition that can be written as a *pfold*, a program scheme which is similar to *fold* but that additionally takes a constant parameter for its computation. For lists, it corresponds to the following definition:

$$
\begin{aligned}
&\mathit{pfold} :: (z \to b, a \to b \to z \to b) \to ([\,a\,], z) \to b\\
&\mathit{pfold}\ (\mathit{hnil}, \mathit{hcons})\ ([\,], z) \qquad = \mathit{hnil}\ z\\
&\mathit{pfold}\ (\mathit{hnil}, \mathit{hcons})\ (a : as, z) = \mathit{hcons}\ a\ (\mathit{pfold}\ (\mathit{hnil}, \mathit{hcons})\ (as, z))\ z
\end{aligned}
$$

The producer is required to be expressible in terms of a kind of build function, called *buildp*, that returns a pair formed by a data structure and a value instead of simply a data structure. For lists:

$$
\begin{aligned}
&\mathit{buildp} \quad :: (\forall\ b\ .\ (b, a \to b \to b) \to c \to (b, z)) \to c \to ([\,a\,], z)\\
&\mathit{buildp}\ g = g\ \mathit{in}_L
\end{aligned}
$$

Note that *buildp* corresponds to build_N with functor $N\ a = (a, z)$ for some z.

3.1 Derivation of Circular Programs

The derivation of purely functional circular programs is stated by the following law (see [3] for further details and a proof). To improve the understanding of circular definitions we frame their circular arguments.

Law 3 (PFOLD/BUILDP)

$$
\begin{aligned}
&\mathit{pfold}\ (\mathit{hnil}, \mathit{hcons})\ (\mathit{buildp}\ g\ c) = v\\
&\quad \textbf{where}\ (v, \boxed{z}) = g\ (\mathit{knil}, \mathit{kcons})\ c\\
&\qquad\quad \mathit{knil} = \mathit{hnil}\ \boxed{z}\\
&\qquad\quad \mathit{kcons}\ x\ r = \mathit{hcons}\ x\ r\ \boxed{z}
\end{aligned}
$$

To see an example, let us consider $addLen = addL \circ filterLen$ where:

$$addL\ ([\,], l) = [\,]$$
$$addL\ (x : xs, l) = (x + l) : addL\ (xs, l)$$

First, we express *addL* and *filterLen* in terms of pfold and buildp, respectively:

$$addL = pfold\ (hnil, hcons)\ \textbf{where}\ hnil\ l = [\,]$$
$$hcons\ x\ r\ l = (x + l) : r$$
$$filterLen = buildp\ gfL$$

where *gfL* is the same function presented in Section 2. Then, by applying Law 3 we derive the following circular definition:

$$addLen\ xs = ys$$
$$\textbf{where}\ (ys, \boxed{l}) = gk\ xs$$
$$gk\ [\,] = ([\,], 0)$$
$$gk\ (x : xs) = \textbf{if}\ x > 0\ \textbf{then}\ ((x + \boxed{l}) : ys, 1 + n)\ \textbf{else}\ (ys, n)$$
$$\textbf{where}\ (ys, n) = gk\ xs$$

Law 3 can be generalized similarly to extended shortcut fusion. The generalization works on an extended form of *buildp* and represents the case where the intermediate pair is produced within another structure given by a functor N.

$$buildp_N :: (\forall\ b\ .\ (b, a \to b \to b) \to c \to N\ (b, z)) \to c \to N\ ([a], z)$$
$$buildp_N\ g = g\ in_L$$

Observe that $buildp_N = build_M$ for $M\ a = N\ (a, z)$.

For the formulation of the new law it is necessary to assume that functor N possesses an associated polymorphic function $\epsilon_N :: N\ a \to a$ that projects a value of type a from a structure of type $N\ a$. A free theorem [13] associated with the type of ϵ_N states that $f \circ \epsilon_N = \epsilon_N \circ map_N\ f$, for every f.

The desired generalization of Law 3 is as follows. Let $f\ \$\ x = f\ x$.

Law 4 (PFOLD/BUILDPN) *Let* (N, ϵ_N) *be a strictness-preserving.*

$$pfold\ (hnil, hcons) \circ \epsilon_N \circ buildp_N\ g\ \$\ c = v$$
$$\textbf{where}\ (v, \boxed{z}) = \epsilon_N\ (g\ (knil, kcons)\ c)$$
$$knil = hnil\ \boxed{z}$$
$$kcons\ x\ r = hcons\ x\ r\ \boxed{z}$$

3.2 Derivation of Higher-Order Programs

Starting from the same kind of compositions used to derive circular programs it is possible to derive, by alternative laws, higher-order programs [3]. Higher-order programs are sometimes preferred over circular ones as they are not restricted to a lazy evaluation setting and their running performance is often better than that of their circular equivalents.

The transformation into higher-order programs is based on the fact that every pfold can be expressed in terms of a higher-order fold. For example, given

$pfold\ (hnil, hcons) :: ([a], z) \to b$, with $hnil :: z \to b$ and $hcons :: a \to b \to z \to b$, we can write it as a fold of type $[a] \to z \to b$:

$$pfold\ (hnil, hcons)\ (xs, z) = fold\ (knil, kcons)\ xs\ z$$

where $knil = \lambda z \to hnil\ z :: z \to b$ and $kcons\ x\ r = \lambda z \to hcons\ x\ (r\ z)\ z :: a \to (z \to b) \to (z \to b)$. With this relationship at hand we can state the following law, which is the instance to our context of a more general program transformation technique called lambda abstraction [17].

Law 5 (H-O PFOLD/BUILDP) *For left-strict hcons,*[2]

$$\begin{aligned} &pfold\ (hnil, hcons)\ (buildp\ g\ c) = f\ z \\ &\quad \textbf{where}\ (f, z) = g\ (knil, kcons)\ c \\ &\qquad\quad knil = \lambda z \to hnil\ z \\ &\qquad\quad kcons\ x\ r = \lambda z \to hcons\ x\ (r\ z)\ z \end{aligned}$$

Like in Law 3, $g\ (knil, kcons)$ returns a pair, but now composed by a function of type $z \to b$ and a value of type z. The final result then corresponds to the application of the function to that value.

To see an example of the use of this law, let us consider again the composition $addLen = addL \circ filterLen$. By applying Law 5 we get the following definition:

$$\begin{aligned} &addLen\ xs = f\ l \\ &\quad \textbf{where}\ (f, l) = gk\ xs \\ &\qquad gk\ [\,] \quad\ = (\lambda l \to [\,], 0) \\ &\qquad gk\ (x : xs) = \textbf{if}\ x > 0\ \textbf{then}\ (\lambda l \to (x + l) : f'\ l, 1 + l')\ \textbf{else}\ (f', l') \\ &\qquad\quad \textbf{where}\ (f', l') = gk\ xs \end{aligned}$$

The following is a generalization of the previous law.

Law 6 (H-O PFOLD/BUILDPN) *Let* (N, ϵ_N) *be a strictness-preserving functor.*

$$\begin{aligned} &pfold\ (hnil, hcons) \circ \epsilon_N \circ buildp_N\ g\ \$\ c = f\ z \\ &\quad \textbf{where}\ (f, z) = \epsilon_N\ (g\ (knil, kcons)\ c) \\ &\qquad\quad knil = \lambda z \to hnil\ z \\ &\qquad\quad kcons\ x\ r = \lambda z \to hcons\ x\ (r\ z)\ z \end{aligned}$$

4 Multiple Intermediate Structure Deforestation

In this section we analyse how can we deal with a sequence of compositions $f_n \circ \cdots \circ f_1$, for $n > 2$. We start with the analysis of the standard case in which a single data structure is generated in each composition. Our aim is to look at the conditions the functions f_i need to satisfy in order to be possible to derive a monolithic definition from such a composition. We then turn to the analysis of situations in which the intermediate data structures are passed between functions inside a pair. As we saw in Section 3, compositions of this kind give rise to circular and higher-order definitions.

[2] By left-strict we mean strict on the first argument, that is, $hcons\ (\bot, z) = \bot$.

4.1 Standard Case

Let us suppose that in every composition $f_{i+1} \circ f_i$ only an intermediate data structure is passed between the functions. To derive a monolithic definition from the whole sequence $f_n \circ \cdots \circ f_1$ the involved functions need to satisfy certain conditions. Clearly, f_1 needs to be a producer while f_n a consumer. Functions $f_2, \ldots, f_{n-1}$ are more interesting since they all need to be both consumers and producers in order to be possible to fuse them with their neighbour functions.

Suppose, for example, that we want to test whether a number is perfect. A number is said to be *perfect* when it is equal to the sum of its factors:

$$perfect\ n = sumFactors\ n == n$$

$$sumFactors\ n = sum\ (factors\ n)$$

Notice that two intermediate lists are generated by *sumFactors*: one by *factors* and the other in the composition of *sum* with *factors*. If we want to eliminate those data structures the essential expression to fuse is $sum \circ filter\ pr \circ down$. As shown in Section 2, *down* is a producer. On the other hand, *sum* is a consumer as it can be written as a fold. Concerning *filter pr*, it is a consumer, but it can also be a producer maintaining its formulation as a fold, appealing to the notion of an *algebra transformer*, traditionally used in the context of fusion laws known as *acid rain* [6]. Similar to the "template" of a build, a transformer makes it possible to abstract the occurrences of the constructors of the data structure produced as result from the body of a fold, or which is the same, from the operations of the algebra carried by a fold. In the case of *filter pr*, we can write:

$$filter\ pr = fold\ (\tau\ in_L)$$
$$\textbf{where}\ \tau\ (nil, cons) = (nil, \lambda a\ r \rightarrow \textbf{if}\ pr\ a\ \textbf{then}\ cons\ a\ r\ \textbf{else}\ r)$$

The algebra transformer $\tau :: \forall\ b\ .\ (b, a \rightarrow b \rightarrow b) \rightarrow (b, a \rightarrow b \rightarrow b)$ simply abstracts the list constructors from the algebra $([\,], \lambda a\ r \rightarrow \textbf{if}\ pr\ a\ \textbf{then}\ a : r\ \textbf{else}\ r)$ of the fold for *filter pr* by replacing its occurrences by the components of an arbitrary algebra $(nil, cons)$. As mentioned above, transformers are useful in the context of acid rain laws because they permit to specify producers given by folds. The following is an acid rain law with a transformer between list algebras.

Law 7 (FOLD-FOLD FUSION FOR LISTS)

$$\begin{array}{l} \tau :: \forall\ b\ .\ (b, a \rightarrow b \rightarrow b) \rightarrow (b, a' \rightarrow b \rightarrow b) \\ \Rightarrow \\ fold\ (nil, cons) \circ fold\ (\tau\ in_L) = fold\ (\tau\ (nil, cons)) \end{array}$$

Acid rain laws can be expressed in terms of shortcut fusion [18]. For example, Law 7 can be seen as a particular case of Law 1. In fact, by defining $gfold\ k = fold\ (\tau\ k)$ it follows that $fold\ (\tau\ in_L) = build\ gfold$.

Returning to the composition $sum \circ filter\ pr \circ down$, there are various ways in which fusion can proceed in this case. One way is to proceed from left-to-right by first fusing *sum* with *filter pr*, and then fusing the result with *down*. For fusing $sum \circ filter\ pr$ we can apply Law 7, obtaining as result $fold\ (\tau\ (0, (+)))$. Fusing this fold with *down* by Law 1 we obtain $gd\ (\tau\ (0, (+)))$ as final result.

An equivalent alternative is to proceed from right-to-left by first fusing *filter pr* with *down* and then fusing the result with *sum*. Fusion of *filter pr* $\circ$ *down* is performed by applying Law 1, obtaining $gd\ (\tau\ in_L)$ as result; this coincides with the function *fd pr* shown in Section 2. If we now want to fuse *sum* with $gd\ (\tau\ in_L)$ then we first need to rewrite this function as a build. It is in such a situation that a new law, that we call *chain law*, comes into play. It states conditions under which the composition of a consumer with a producer, such that the consumer happens to be also a producer, can be fused resulting in a function that can be directly expressed as a producer. The key idea of this law is the appropriate combination of the fusion approaches represented by shortcut fusion and acid rain. We present the case of the chain law for an algebra transformer with same type as in Law 7.

Law 8 (CHAIN LAW FOR LISTS)

$$\begin{array}{l} \quad \tau :: \forall\ b\ .\ (b, a \to b \to b) \to (b, a' \to b \to b) \\ \Rightarrow \\ \quad fold\ (\tau\ in_L) \circ build\ g = build\ (g \circ \tau) \end{array}$$

Applying this law we have that *filter pr* $\circ$ *down* $=$ *build* $(gd \circ \tau)$, which can be directly fused with *sum*, obtaining $gd\ (\tau\ (0, (+)))$ as before. To see its recursive definition, let us define $sfd\ pr = gd\ (\tau\ (0, (+)))$. Inlining,

$sfd\ pr\ 0 = 0$
$sfd\ pr\ n =$ **if** $pr\ n$ **then** $n + sfd\ pr\ (n-1)$ **else** $sfd\ pr\ (n-1)$

It is then natural to state a chain law associated with the extension of build.

Law 9 (EXTENDED CHAIN LAW) *For strictness-preserving* N*,*

$$\begin{array}{l} \quad \tau :: \forall\ b\ .\ (b, a \to b \to b) \to (b, a' \to b \to b) \\ \Rightarrow \\ \quad map_N\ (fold\ (\tau\ in_L)) \circ build_N\ g = build_N\ (g \circ \tau) \end{array}$$

The next law describes a more general situation where the transformer τ returns an algebra whose carrier is the result of applying a functor W to the carrier of the input algebra. Law 9 is then the special case when W is the identity functor. By NW we denote the composition of functors N and W, that is, $NW\ a = N\ (W\ a)$ and $map_{NW}\ f = map_N\ (map_W\ f)$.

Law 10 *Let* W *be a functor. For strictness-preserving* N*,*

$$\begin{array}{l} \quad \tau :: \forall\ b\ .\ (b, a \to b \to b) \to (W\ b, a' \to W\ b \to W\ b) \\ \Rightarrow \\ \quad map_N\ (fold\ (\tau\ in_L)) \circ build_N\ g = build_{NW}\ (g \circ \tau) \end{array}$$

4.2 Derivation of Programs with Multiple Circularities

We now analyse laws that make it possible the derivation of programs with multiple circularities. We consider that the sequence of compositions $f_n \circ \cdots \circ f_1$

is such that a pair (t_i, z_i) of a data structure t_i and a value z_i is generated in each composition. Like before, f_1 needs to be a producer, f_n a consumer, whereas $f_2, \ldots, f_{n-1}$ need to be simultaneously consumers and producers. Therefore, in this case the sequence of compositions is of the form $pfold \circ \cdots \circ pfold \circ buildp$.

Like in the standard case, we want to analyse the transformation in both directions: from left-to-right and right-to-left. We will see that in this case there are significant differences between the transformation in each direction, not in the result, but in the complexity of the laws that need to be applied.

Right-to-Left Transformation. Following a similar approach to the one used for the standard case, when the transformation proceeds from right to left it is necessary to state sufficient conditions that permit us to establish when the composition of a pfold with a buildp is again a buildp. Interestingly, the resulting definition would be not only a producer (a buildp) that can be fused with the next pfold in the sequence, but by Law 3 it would be also a circular program that internally computes a pair (v, z) formed by the result of the program (v) and the circular argument (z). Therefore, by successively fusing the compositions in the sequence from right to left we finally obtain a program with multiple circular arguments, one for each fused composition. During this process, we incrementally introduce a new circular argument at every fusion step without affecting the circular arguments previously introduced.

At the i-th step, the calculated circular program internally computes a nested product of the form $((\ldots(v_i, z_i), \ldots), z_1)$, where v_i is the value returned by that program and $z_1, \ldots, z_i$ are the circular arguments introduced so far. As a consequence of this, at each step it is necessary to employ an extended shortcut fusion law because the pair (t_i, z_i) to be consumed by the next pfold is generated within the structure formed by the nested product. Thus, we will be handling extensions with functors of the form $N\ a = ((\ldots(a, z_j), \ldots), z_1)$.

Therefore, to deal with this process appropriately we need to state a chain law in the sense of Law 9 but now associated with the composition of a pfold with an extended buildp. Given a transformer $\sigma :: \forall\ b\ .\ (b, a \to b \to b) \to (z \to W\ b, a' \to W\ b \to z \to W\ b)$, where W is a functor and, for each algebra k, $\sigma\ k = (\sigma_1\ k, \sigma_2\ k)$, it is possible to derive an algebra transformer: $\tau :: \forall\ b\ .\ (b, a \to b \to b) \to (W\ b, a' \to W\ b \to W\ b)$ such that $\tau\ k = (\tau_1\ k, \tau_2\ k)$ with $\tau_1\ k = \sigma_1\ k\ z$ and $\tau_2\ k\ x\ r = \sigma_2\ k\ x\ r\ z$, for a fixed z. Such a σ is used in the next law to represent a case when the consumer, given by a pfold, is also a producer. In fact, observe that the pfold in the law has type $([a'], z) \to ([a], y)$.

Law 11 (CHAIN RULE) *Let (N, ϵ_N) be a strictness-preserving functor, and $M\ a = N\ (a, z)$. Let $W\ a = (a, y)$, for some type y. Let $\sigma\ k = (\sigma_1\ k, \sigma_2\ k)$.*

$$
\begin{array}{l}
\quad \sigma :: \forall\ b\ .\ (b, a \to b \to b) \to (z \to W\ b, a' \to W\ b \to z \to W\ b) \\
\Rightarrow \\
\quad pfold\ (\sigma\ in_L) \circ \epsilon_N \circ buildp_N\ g\ \$\ c = p \\
\qquad \textbf{where}\ (p, \boxed{z}) = \epsilon_N\ (buildp_M\ (g \circ \tau)\ c) \\
\qquad\qquad \tau\ k = (\tau_1\ k, \tau_2\ k) \\
\qquad\qquad \tau_1\ k = \sigma_1\ k\ \boxed{z} \\
\qquad\qquad \tau_2\ k\ x\ r = \sigma_2\ k\ x\ r\ \boxed{z}
\end{array}
$$

Example 1. Consider the following program that given a set of points in a plane returns the maximum distance between the points located above the average height and the highest point below the average height. We assume that the height of all points is non-negative.

$$
\begin{aligned}
&\textbf{type } \mathit{Point} \quad = (\mathit{Float}, \mathit{Float})\\
&\textbf{type } \mathit{Height} \quad = \mathit{Float}\\
&\textbf{type } \mathit{Distance} = \mathit{Float}\\
&\mathit{distance} = \mathit{maxDistance} \circ \mathit{takePoints} \circ \mathit{avrgHeight}\ 0\ 0\\
&\mathit{avrgHeight} :: \mathit{Height} \to \mathit{Integer} \to [\mathit{Point}] \to ([\mathit{Point}], \mathit{Height})\\
&\mathit{avrgHeight}\ h\ l\ [\,] = ([\,], h\ /\ \mathit{fromInteger}\ l)\\
&\mathit{avrgHeight}\ h\ l\ ((x,y) : ps) = \textbf{let } (ps', avH) = \mathit{avrgHeight}\ (y+h)\ (1+l)\ ps\\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in } ((x,y) : ps', avH)\\
&\mathit{takePoints} :: ([\mathit{Point}], \mathit{Height}) \to ([\mathit{Point}], \mathit{Point})\\
&\mathit{takePoints}\ ([\,], avH) = ([\,], (0,0))\\
&\mathit{takePoints}\ ((x,y) : ps, avH) = \textbf{let } (ps', hp) = \mathit{takePoints}\ (ps, avH)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in if } y > avH \textbf{ then } ((x,y) : ps', hp)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } (ps', \textbf{if } y > \mathit{snd}\ hp \textbf{ then } (x,y) \textbf{ else } hp)\\
&\mathit{maxDistance} :: ([\mathit{Point}], \mathit{Point}) \to \mathit{Distance}\\
&\mathit{maxDistance}\ ([\,], hp) = 0\\
&\mathit{maxDistance}\ ((x,y) : ps, hp@(hx,hy))\\
&\quad = \mathit{sqrt}\ ((x-hx)^2 + (y-hy)^2)\ `\mathit{max}`\ \mathit{maxDistance}\ (ps, hp)
\end{aligned}
$$

To apply the rules, first we need to express these functions in terms of the corresponding program schemes.

$$
\begin{aligned}
&\mathit{avrgHeight} = \mathit{buildp}\ \mathit{gavrgH}\\
&\quad \textbf{where } \mathit{gavrgH}\ (\mathit{nil}, \mathit{cons})\ h\ l\ [\,] = (\mathit{nil}, h\ /\ \mathit{fromInteger}\ l)\\
&\qquad\quad \mathit{gavrgH}\ (\mathit{nil}, \mathit{cons})\ h\ l\ ((x,y) : ps)\\
&\qquad\qquad = \textbf{let } (ps', avH) = \mathit{gavrgH}\ (\mathit{nil}, \mathit{cons})\ (y+h)\ (1+l)\ ps\\
&\qquad\qquad\quad \textbf{in } (\mathit{cons}\ (x,y)\ ps', avH)\\
&\mathit{takePoints} = \mathit{pfold}\ (\mathit{tnil}, \mathit{tcons})\\
&\quad \textbf{where } \mathit{tnil}\ avH \qquad\qquad = ([\,], (0,0))\\
&\qquad\quad \mathit{tcons}\ (x,y)\ r\ avH = \textbf{let } (ps, hp) = r\\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{in if } y > avH \textbf{ then } ((x,y) : ps, hp)\\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } (ps, \textbf{if } y > \mathit{snd}\ hp \textbf{ then } (x,y) \textbf{ else } hp)\\
&\mathit{maxDistance} = \mathit{pfold}\ (\mathit{hnil}, \mathit{hcons})\\
&\quad \textbf{where } \mathit{hnil}\ hp = 0\\
&\qquad\quad \mathit{hcons}\ (x,y)\ r\ hp@(hx,hy) = \mathit{sqrt}\ ((x-hx)^2 + (y-hy)^2)\ `\mathit{max}`\ r
\end{aligned}
$$

Since $(\mathit{tnil}, \mathit{tcons})$ can be expressed as $\sigma\ \mathit{in}_L$, where σ is a transformer:

$$
\begin{aligned}
&\sigma\ (\mathit{nil}, \mathit{cons}) = (\lambda avH \to (\mathit{nil}, (0,0))\\
&\qquad\qquad\qquad , \lambda(x,y)\ r\ avH \to\\
&\qquad\qquad\qquad\quad \textbf{let } (ps, hp) = r
\end{aligned}
$$

$$\begin{array}{l}\quad\quad\quad\textbf{in if}\ y > avH\ \textbf{then}\ (cons\ (x,y)\ ps, hp)\\ \quad\quad\quad\quad\textbf{else}\ (ps, \textbf{if}\ y > snd\ hp\ \textbf{then}\ (x,y)\ \textbf{else}\ hp))\end{array}$$

our program corresponds to the following composition:

$$distance = pfold\ (hnil, hcons) \circ pfold\ (\sigma\ in_L) \circ buildp\ gavrgH\ 0\ 0$$

The transformation from right to left essentially proceeds by first applying Law 11 and then Law 4. The program that is finally obtained is the following:

$$\begin{array}{l}distance\ inp = v\\ \quad \textbf{where}\ (v, \boxed{u}) = w\\ \quad\quad (w, \boxed{z}) = gk\ 0\ 0\ inp\\ \quad\quad gk\ h\ l\ [\,] = ((0,(0,0)), h\ /\ fromInteger\ l)\\ \quad\quad gk\ h\ l\ ((x,y):ps) =\\ \quad\quad\quad \textbf{let}\ (ps', avH) = gk\ (y+h)\ (1+l)\ ps\\ \quad\quad\quad \textbf{in}\ (\textbf{let}\ (qs, hp) = ps'\\ \quad\quad\quad\quad \textbf{in if}\ y > \boxed{z}\\ \quad\quad\quad\quad\quad \textbf{then}\ (sqrt\ ((x - fst\ \boxed{u})^2 + (y - snd\ \boxed{u})^2)\ `max`\ qs, hp)\\ \quad\quad\quad\quad\quad \textbf{else}\ (qs, \textbf{if}\ y > snd\ hp\ \textbf{then}\ (x,y)\ \textbf{else}\ hp), avH)\end{array}$$

Left-to-right Transformation. When the transformation is in left to right order we worry about the opposite situation. Except for the last step, at each intermediate stage of the transformation process we are interested in that the definition that results from a fusion step is a consumer. If that is the case then it is guaranteed that we can successively apply fusion until cover all function compositions. It is then necessary to state sufficient conditions to establish when the composition of two pfolds is again a pfold.

The following acid rain law is inspired in fold-fold fusion (Law 7).

Law 12 (PFOLD-PFOLD FUSION)

$$\begin{array}{l}\quad\quad \sigma :: \forall\ b\ .\ (b, a \to b \to b) \to (z \to b, a' \to b \to z \to b)\\ \Rightarrow\\ \quad\quad pfold\ (hnil, hcons)\ (pfold\ (\sigma\ in_L)\ c) = v\\ \quad\quad\quad \textbf{where}\ (v, \boxed{z}) = pfold\ (\sigma\ (knil, kcons))\ c\\ \quad\quad\quad\quad\quad knil = hnil\ \boxed{z}\\ \quad\quad\quad\quad\quad kcons\ x\ r = hcons\ x\ r\ \boxed{z}\end{array}$$

Like fold-fold fusion, this law can also be formulated in terms of shortcut fusion. By defining $gpfold\ k = pfold\ (\sigma\ k)$, it follows that $pfold\ (\sigma\ in_L) = buildp\ gpfold$. Then, by Law 3 we obtain the same result.

Observe that, unlike the right to left transformation, now we do not need to worry about any data structure (a nested pair) inside which fusion is performed. A nested pair is in fact created, but on the result side of the consumers that are successively calculated. It is interesting to see how the nested pair that appears in the final circular program is incrementally generated in each transformation. In the left-to-right transformation the nested pair is generated from inside to outside, i.e. the pair generated in each fusion step contains the previous existing

pair, whereas in the right-to-left transformation the nested pair is generated from outside to inside.

Returning to the example of function *distance*, the transformation from left to right essentially proceeds by simply applying Law 12 and then Law 3. The program that is finally obtained is of course the same.

4.3 Derivation of Higher-Order Programs

During the transformation to a higher-order program we will deal again with a nested structure. Instead of a nested pair we will incrementally construct a structure of type $(z_1 \to (z_2 \to (\cdots \to (z_i \to a, z_i)\cdots), z_2), z_1)$ where the zs are the types of the context parameters that are passed in the successive compositions. So, a structure of this type is a pair (p_1, z_1) composed by a function p_1, which returns a pair (p_2, z_2) such that p_2 is a function that returns again a pair, and so on. Associated to each of these structures we can define a functor $N\ a = (z_1 \to (z_2 \to (\cdots \to (z_i \to a, z_i)\cdots), z_2), z_1)$ whose projection function $\epsilon_N :: N\ a \to a$ is given by iterated function application: $\epsilon_N(p_1, z_1) = p_i\ z_i$ where $(p_j, z_j) = p_{j-1}\ z_{j-1}$, $j = 2, i$.

Like for circular programs, we will see differences in the process of derivation of a a higher-order program when we transform a sequence of compositions $f_n \circ \cdots \circ f_1$ from right to left and left to right. Again one of the differences is the order in which the nested structure is generated.

Right-to-Left Transformation. For the transformation in this direction we need to consider again the situation in which the consumer (a pfold) composed with a producer (a buildp) is again a buildp, The situation is similar to the one faced with Law 11 with the only difference that now we are in the context of a higher-order program derivation. Given a transformer $\sigma :: \forall\ b\ .\ (b, a \to b \to b) \to (z \to W\ b, a' \to W\ b \to z \to W\ b)$, where W is a functor and, for each algebra k, $\sigma\ k = (\sigma_1\ k, \sigma_2\ k)$, it is possible to derive an algebra transformer: $\tau :: \forall\ b\ .\ (b, a \to b \to b) \to (z \to W\ b, a' \to (z \to W\ b) \to (z \to W\ b))$ such that $\tau\ k = (\tau_1\ k, \tau_2\ k)$ with $\tau_1\ k = \lambda z \to \sigma_1\ k\ z$ and $\tau_2\ k\ x\ r = \lambda z \to \sigma_2\ k\ x\ (r\ z)\ z$. Observe that the pfold in the next law has type $([a'], z) \to ([a], y)$.

Law 13 (H-O CHAIN RULE) *Let (N, ϵ_N) be a strictness-preserving functor and $M\ a = N\ (a, z)$. Let $W\ a = (a, y)$, for some type y. Let $\sigma\ k = (\sigma_1\ k, \sigma_2\ k)$.*

$$
\begin{array}{l}
\quad \sigma :: \forall\ b\ .\ (b, a \to b \to b) \to (z \to W\ b, a' \to W\ b \to z \to W\ b) \\
\Rightarrow \\
\quad \mathit{pfold}\ (\sigma\ \mathit{in}_L) \circ \epsilon_N \circ \mathit{buildp}_N\ g\ \$\ c = f\ z \\
\qquad \textbf{where}\ (f, z) = \epsilon_N\ (\mathit{buildp}_M\ (g \circ \tau)\ c) \\
\qquad\qquad \tau\ k = (\tau_1\ k, \tau_2\ k) \\
\qquad\qquad \tau_1\ k = \lambda z \to \sigma_1\ k\ z \\
\qquad\qquad \tau_2\ k\ x\ r = \lambda z \to \sigma_2\ k\ x\ (r\ z)\ z
\end{array}
$$

The higher-order program derivation in right to left order applied to *distance* essentially proceeds by first applying Law 13 and then Law 6. As result we obtain the following higher-order program:

$$
\begin{array}{l}
\mathit{distance}\ \mathit{inp} = f\ u \\
\quad \textbf{where}\ (f, u) = g\ z \\
\qquad\quad (g, z) = \mathit{gk}\ 0\ 0\ \mathit{inp} \\
\qquad\quad \mathit{gk}\ h\ l\ [\,] = (\lambda z \to (\lambda u \to 0, (0, 0)), h\ /\ \mathit{fromInteger}\ l) \\
\qquad\quad \mathit{gk}\ h\ l\ ((x, y) : ps) = \\
\qquad\quad \textbf{let}\ (ps', \mathit{avH}) = \mathit{gk}\ (y + h)\ (1 + l)\ ps \\
\qquad\quad \textbf{in}\ (\lambda z \to \\
\qquad\qquad \textbf{let}\ (qs, hp) = ps'\ z \\
\qquad\qquad \textbf{in if}\ y > z \\
\qquad\qquad\quad \textbf{then}\ (\lambda u \to \mathit{sqrt}\ ((x - \mathit{fst}\, u)^2 + (y - \mathit{snd}\, u)^2)\ \text{`}\mathit{max}\text{`}\ (qs\ u), hp) \\
\qquad\qquad\quad \textbf{else}\ (\lambda u \to qs\ u, \textbf{if}\ y > \mathit{snd}\ hp\ \textbf{then}\ (x, y)\ \textbf{else}\ hp), \mathit{avH})
\end{array}
$$

Left-to-Right Transformation. For the transformation in this other direction we proceed similarly as we did for circular programs. The same considerations hold in this case. The calculation of the successive consumers from left to right is performed using the following acid rain law:

Law 14 (H-O PFOLD-PFOLD FUSION)

$$
\begin{array}{l}
\quad \sigma :: \forall\ b\ .\ (b, a \to b \to b) \to (z \to b, a' \to b \to z \to b) \\
\Rightarrow \\
\quad \mathit{pfold}\ (\mathit{hnil}, \mathit{hcons})\ (\mathit{pfold}\ (\sigma\ \mathit{in}_L)\ c) = f\ z \\
\quad\ \ \textbf{where}\ (f, z) = \mathit{pfold}\ (\sigma\ (\mathit{knil}, \mathit{kcons}))\ c \\
\qquad\qquad\quad \mathit{knil} = \lambda z \to \mathit{hnil}\ z \\
\qquad\qquad\quad \mathit{kcons}\ x\ r = \lambda z \to \mathit{hcons}\ x\ (r\ z)\ z
\end{array}
$$

This law can also be formulated in terms of shortcut fusion. By defining $\mathit{gpfold}\ k = \mathit{pfold}\ (\sigma\ k)$, we have that $\mathit{pfold}\ (\sigma\ \mathit{in}_L) = \mathit{buildp}\ \mathit{gpfold}$. Then, by Law 5 we obtain the same result.

Concerning the example of function *distance*, the higher-order program derivation from left to right essentially proceeds by applying Law 14 and then Law 5.

5 Conclusions

In this paper, we have presented an approach, based on shortcut fusion, to achieve deforestation in an arbitrary number of function compositions. Our work generalizes standard shortcut fusion [1], *circular* shortcut fusion [3] and *higher-order* shortcut fusion [3]. The derivation of circular programs is strongly associated with attribute grammar research [10,11], and we expect our work to clarify even further their similar nature. The derivation of higher-order programs is motivated by efficiency. Indeed, as the programs we consider here are of the same kind as the ones we have benchmarked in [19], we expect the derived higher-order programs to be significantly more efficient when compared to their multiple traversal and circular counterparts.

Our approach is calculational and establishes sufficient conditions for fusion to proceed. For now, we have not focused on implementation details, that we

are considering to present in an extended version of this paper as well as further demonstrational examples that have not been included here due to space limitations.

References

1. Gill, A., Launchbury, J., Peyton Jones, S.: A short cut to deforestation. In: Functional Programming Languages and Computer Architecture. ACM (1993)
2. Bird, R.: Using circular programs to eliminate multiple traversals of data. Acta Informatica 21, 239–250 (1984)
3. Pardo, A., Fernandes, J.P., Saraiva, J.: Shortcut fusion rules for the derivation of circular and higher-order programs. Higher-Order and Symbolic Computation 24(1-2), 115–149 (2011)
4. Voigtländer, J.: Semantics and pragmatics of new shortcut fusion rules. In: Garrigue, J., Hermenegildo, M.V. (eds.) FLOPS 2008. LNCS, vol. 4989, pp. 163–179. Springer, Heidelberg (2008)
5. Swierstra, D., Chitil, O.: Linear, bounded, functional pretty-printing. Journal of Functional Programming 19(1), 1–16 (2009)
6. Onoue, Y., Hu, Z., Iwasaki, H., Takeichi, M.: A Calculational Fusion System HYLO. In: IFIP TC 2 Working Conference on Algorithmic Languages and Calculi, pp. 76–106. Chapman & Hall (1997)
7. Launchbury, J., Sheard, T.: Warm fusion: Deriving build-catas from recursive definitions. In: Funct. Prog. Lang. and Computer Architecture. ACM (1995)
8. Johnsson, T.: Attribute grammars as a functional programming paradigm. In: Kahn, G. (ed.) FPCA 1987. LNCS, vol. 274, pp. 154–173. Springer, Heidelberg (1987)
9. de Moor, O., Backhouse, K., Swierstra, S.D.: First-class attribute grammars. Informatica (Slovenia) 24(3) (2000)
10. Fernandes, J.P., Saraiva, J.: Tools and Libraries to Model and Manipulate Circular Programs. In: Workshop on Partial Eval. and Program Manipulation. ACM (2007)
11. Fernandes, J.P., Saraiva, J., Seidel, D., Voigtländer, J.: Strictification of circular programs. In: Workshop on Partial Eval. and Program Manipulation. ACM (2011)
12. Johann, P., Voigtländer, J.: Free theorems in the presence of seq. In: Symposium on Principles of Programming Languages, pp. 99–110. ACM (2004)
13. Wadler, P.: Theorems for free! In: Functional Programming Languages and Computer Architecture. ACM (1989)
14. Bird, R., de Moor, O.: Algebra of Programming. Prentice-Hall Inernational Series in Computer Science, vol. 100. Prentice-Hall (1997)
15. Manzino, C., Pardo, A.: Shortcut Fusion of Monadic Programs. Journal of Universal Computer Science 14(21), 3431–3446 (2008)
16. Ghani, N., Johann, P.: Short cut fusion for effects. In: TFP 2008. Trends in Functional Programming, vol. 9, pp. 113–128. Intellect (2009)
17. Pettorossi, A., Skowron, A.: The lambda abstraction strategy for program derivation. Fundamenta Informaticae 12(4), 541–561 (1989)
18. Takano, A., Meijer, E.: Shortcut deforestation in calculational form. In: Functional Programming Languages and Computer Architecture, pp. 306–313. ACM (1995)
19. Fernandes, J.P.: Desing, Implementation and Calculation of Circular Programs. PhD thesis, Dept. of Informatics, Univ. of Minho, Portugal (2009)