Science of Computer Programming 132 (2016) 77–95

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Multiple intermediate structure deforestation by shortcut fusion

Alberto Pardo [a,*], João Paulo Fernandes [b,c], João Saraiva [c]

[a] Instituto de Computación, Universidad de la República, Uruguay
[b] LISP/ ((rel)ease) Reliable and Secure Computation Group, Universidade da Beira Interior, Portugal
[c] HASLab/INESC TEC, Universidade do Minho, Portugal

ARTICLE INFO

Article history:
Received 18 April 2014
Received in revised form 19 July 2016
Accepted 19 July 2016
Available online 1 August 2016

Keywords:
Shortcut fusion
Circular programming
Deforestation
Functional programming

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 of type t.

Extended forms of shortcut fusion have also been proposed to cope with cases of function compositions in which the producer and the consumer communicate through some additional context information, besides the intermediate structure itself. 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, increasing the applicability scope 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].

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 $f_n \circ \cdots \circ f_0$, 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]. The fusion laws we present are surely not

* Corresponding author.
E-mail addresses: pardo@fing.edu.uy (A. Pardo), jpf@di.ubi.pt (J.P. Fernandes), jas@di.uminho.pt (J. Saraiva).

http://dx.doi.org/10.1016/j.scico.2016.07.004
0167-6423/

surprising, but, on the other hand, have the merit of dealing with nonstandard cases. Our fusion method, characterized by requiring certain conditions on the functions involved in the compositions, differs from that employed by *warm fusion* [7].

In this work our approach is essentially theoretical. We study the formalities associated with the formulation of the new laws, but we do not include performance results and benchmarks related with the application of the laws in practice.

We analyse two forms 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 it is possible to derive circular and higher-order programs by the application of fusion.

The fact that we are able of deriving circular programs from multiple function compositions also has strong connections with well-established research on Attribute Grammars (AG) [8]. Indeed, as Johnson [9] and Kuiper and Swierstra [10] originally showed, attribute grammars are naturally implemented in a lazy setting as circular programs. In the AG community, however, a program that relies on multiple function compositions (so, on multiple intermediate data structures) is usually derived from an AG (i.e., from a circular program) using advanced attribute scheduling algorithms [11,12] whose correctness is hard to establish in a formal way. By studying and proving the correctness of the opposite transformation, that is, how a circular program (i.e., an AG) can be derived from a program based on multiple function compositions, we hope to shed even more light into the relationship between attribute grammars/circular programs and their non-lazy implementations/equivalents.

Throughout the paper we will use Haskell notation, assuming a semantics in terms of pointed cpos (complete partial orders), but without the presence of the *seq* function [13].

The paper is structured as follows. In order to make the contents more accessible to a wider audience, we have decided to divide the paper into two parts. The first part, composed by Sections 2–4, develops the whole concepts and laws, but tailored to the specific case of the list type. In this part laws are presented without proofs. Section 2 reviews shortcut fusion, while Section 3 does so with the derivation of circular and higher-order programs. In Section 4 we discuss laws that pose the sufficient conditions for fusing multi-traversal programs over lists. The second part is formed by Sections 5–6 and is where the generic formulation of the concepts and laws developed is presented. By generic we mean valid for a wide class of datatypes. In Section 5 we present the theoretical concepts necessary for building the generic definitions. Section 6 discusses the laws necessary for fusing multi-traversal programs, but now over arbitrary data structures. A proof is presented for each of these laws. Finally, Section 7 concludes the paper.

2. Shortcut fusion on lists

Shortcut fusion [1] is a program transformation technique for the elimination of intermediate data structures generated in function compositions. In this section we focus on the elimination of lists as intermediate data structures. The formulation for arbitrary data structures is described later in Section 5.

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 corresponds 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. The formal definition of an algebra is given in Section 5.

For example, the function *filter*, which selects the elements of a list that satisfy a given predicate:

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

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

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

The producer, on the other hand, 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 *producer skeleton* as called by Chitil [15]) that abstracts the occurrences of the constructors of the intermediate data type. To guarantee correctness, the template should be a polymorphic function. In the case of lists, *build* is defined as follows:

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

where we denote by $in = ([\,], (:))$ the algebra composed by the list constructors. The polymorphic type of g ensures that the result value of type b can only be constructed by using the operations of the 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:

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

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

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

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).

$$fold\ (nil, cons) \circ build\ g = g\ (nil, cons)$$

This law is a direct consequence of a parametricity property (a "free theorem" [16]) associated with the polymorphic type of the template g of *build*. (See e.g. [1] for details).

Due to its rank-2 type, *build* cannot be defined in standard Haskell since Haskell type system is based on the Hindley–Milner type system [17], where type variables are only quantified at the outer level. It can be defined, however, in the Glasgow Haskell Compiler by compiling it with the language extension option `RankNTypes` which enables function definitions with higher-rank types.

Example 1. Let us consider the function that computes the proper factors of a non-negative number (i.e. the factors not including itself):

$$\begin{aligned} &factors \quad :: Int \to [\,Int\,] \\ &factors\ n = filter\ (`isFactorOf`\ n)\ (down\ (n\ `div`\ 2)) \\ &\quad \textbf{where}\ x\ `isFactorOf`\ n = n\ `mod`\ x == 0 \end{aligned}$$

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

$$\begin{aligned} &fd\ p\ 0 = [\,] \\ &fd\ p\ n = \textbf{if}\ p\ n\ \textbf{then}\ n : fd\ p\ (n-1)\ \textbf{else}\ fd\ p\ (n-1) \end{aligned}$$

2.1. Extended shortcut fusion

It is possible to formulate an extended form of shortcut fusion which captures the case where 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 [18,19], and for the derivation of (monadic) circular and higher-order programs [3]. It is based on an extended form of build:

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

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 $\star \to \star$ which comes equipped with a *map* function $map_N :: (a \to b) \to (N\ a \to N\ b)$ that preserves identities and compositions:

$$map_N\ id = id \qquad map_N\ (f \circ g) = map_N\ f \circ map_N\ g$$

Informally, the idea is that for a given a function f of type $a \to b$ and a structure t of type $N\ a$, $map_N\ f\ t$ returns a structure t' with the same shape as t and where each value v of type a contained in t has been replaced by the corresponding value $f\ v$ of type b.

A functor N is said to be *strictness-preserving* when function map_N preserves strict functions, i.e. if f is strict then so is $map_N\ f$.

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: $\textbf{type}\ N\ a = a$ and $map_N\ f = f$.

Based on this extension of build an extended shortcut fusion law can be formulated:

Law 2 (EXTENDED FOLD/BUILD). *For strictness-preserving N,*

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

Example 2. Let us consider the following composition, where *positivesLen* is such that, given a list of numbers, it returns a pair formed by a list containing the positive numbers and the length of that list. Given functions f and g, we denote by $(f \times g)$ the function such that $(f \times g)\ (x, y) = (f\ x, g\ y)$. By id we denote the identity function.

$$\begin{aligned}
&sumPosLen = (sum \times id) \circ positivesLen\\
&sum\ \ :: Num\ a \Rightarrow [\ a\ ] \to a\\
&sum\ [\ ]\ \ = 0\\
&sum\ (a : as) = a + sum\ as\\
&positivesLen\ \ :: Num\ a \Rightarrow [\ a\ ] \to ([\ a\ ], Int)\\
&positivesLen\ [\ ]\ \ = ([\ ], 0)\\
&positivesLen\ (x : xs) = \textbf{if}\ x > 0\ \textbf{then}\ (x : ys, 1 + l)\ \textbf{else}\ (ys, l)\\
&\quad \textbf{where}\ (ys, l) = positivesLen\ xs
\end{aligned}$$

In order to simplify the expression of *sumPosLen* we first observe that *positivesLen* can be written in terms of $build_N$ with functor $N\ a = (a, Int)$ and $map_N\ f = f \times id$.

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

On the other hand, *sum* is a fold: $sum = fold\ (0, (+))$. Therefore, we can write that $sumPosLen = map_N\ (fold\ (0, (+))) \circ build_N\ gPL$. By applying Law 2 we finally obtain that $sumPosLen = gPL\ (0, (+))$, which corresponds to the following recursive definition:

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

3. Derivation of circular and higher-order programs on lists

In this section we review shortcut fusion laws that make it possible to derive both circular and 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 that is passed between the functions. The derivation of programs of this kind can be done both for pure and monadic programs (see [3,20,21]). In this section we focus on the case in which the structure t is a list; the generic formulation for arbitrary data types is presented in Section 6.

Like for standard shortcut fusion, in this case it is required that both consumer and producer be expressible in terms of certain program schemes.

The consumer is required to be a structural recursive definition writeable as a *pfold*, a program scheme similar to *fold* which takes in addition a constant parameter for its computation. For lists, it corresponds to the following definition:

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

The producer, on the other hand, 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:

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

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

3.1. Derivation of circular programs

Circular programming was originally introduced as an efficient paradigm to avoid multiple traversals of data structures [2]. As the name suggests, circular programs are characterized by holding what appears to be a circular definition,

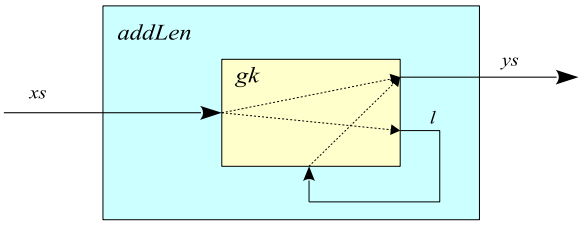

Fig. 1. Data dependencies in the circular definition of *addLen*.

where arguments of a function call depend on results of that same call. Although this seems to introduce non-termination, the fact is that lazy evaluation is able of determining the right evaluation order for these calls, if such an order exists.

Due to their nice properties, circular programs have been studied in varied contexts ranging from the implementation of efficient data structures [22] and traversal strategies [23] to the construction of Haskell compilers [24,25] and to express pretty printing algorithms [5] and type systems [26], for example.

Circular programs can be derived by the application of the following law. We frame circular arguments in order to improve the understanding of circular definitions.

Law 3 (PFOLD/BUILDP).

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

where $f\ \$\ x = f\ x$. It is interesting to see how the circularity arises as a consequence of this law: in the local definition (the **where** clause) function g computes two values, v and z, such that z (one of the resulting values) is used by the operations of the algebra $(knil, kcons)$ to compute v (the other result).

Example 3. Let us consider

$$
\begin{aligned}
&addLen = addL \circ positivesLen\\
&addL\ ([\,], l) \quad = [\,]\\
&addL\ (x : xs, l) = (x + l) : addL\ (xs, l)
\end{aligned}
$$

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

$$
\begin{aligned}
&addL = pfold\ (hnil, hcons)\\
&\quad \textbf{where}\ hnil\ l \quad = [\ ]\\
&\qquad\quad hcons\ x\ r\ l = (x + l) : r\\
&positivesLen = buildp\ gPL
\end{aligned}
$$

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

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

Observe how l, the length of the output list that is computed as the second component of the pair returned by *gk*, is used by *gk* itself for the computation of the first component of the resulting pair, which is a list containing the positive elements of the input list *xs*, each one incremented by precisely l. Fig. 1 shows the data dependencies within the circular definition of *addLen*.

Like standard shortcut fusion, Law 3 can also be extended. The extension 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.

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

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 [16] associated with the type of ϵ_N states that for every f,

$$f \circ \epsilon_N = \epsilon_N \circ map_N\ f \tag{1}$$

The extension of Law 3 is as follows.

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

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

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 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 as a higher-order fold. Given $\mathit{pfold}\ (\mathit{hnil}, \mathit{hcons}) :: ([\ a\ ], z) \to b$, with $\mathit{hnil} :: z \to b$ and $\mathit{hcons} :: a \to b \to z \to b$, we can write it as a fold of type $[\ a\ ] \to z \to b$:

$$
\begin{array}{l}
\mathit{pfold}\ (\mathit{hnil}, \mathit{hcons})\ (\mathit{xs}, z) = \mathit{fold}\ (\mathit{knil}, \mathit{kcons})\ \mathit{xs}\ z \\
\quad \textbf{where}\ \mathit{knil} = \mathit{hnil} \\
\qquad\quad \mathit{kcons}\ x\ r = \lambda z \to \mathit{hcons}\ x\ (r\ z)\ z
\end{array}
$$

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 [27].

Law 5 (H-O PFOLD/BUILDP).

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

Like in Law 3, $g\ (\mathit{knil}, \mathit{kcons})$ 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.

Example 4. Let us consider again the composition $\mathit{addLen} = \mathit{addL} \circ \mathit{positivesLen}$. By applying Law 5 we get the following definition:

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

The following is an extension of the previous law.

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

$$
\begin{array}{l}
\mathit{pfold}\ (\mathit{hnil}, \mathit{hcons}) \circ \epsilon_N \circ \mathit{buildp}_N\ g\ \$\ c = f\ z \\
\quad \textbf{where}\ (f, z) = \epsilon_N\ (g\ (\mathit{knil}, \mathit{kcons})\ c) \\
\qquad\quad \mathit{knil} = \mathit{hnil} \\
\qquad\quad \mathit{kcons}\ x\ r = \lambda z \to \mathit{hcons}\ x\ (r\ z)\ z
\end{array}
$$

4. Multiple intermediate lists deforestation

In this section we analyse how we can deal with a sequence of compositions $f_n \circ \cdots \circ f_0$, for $n >= 2$, where all the intermediate structures are lists. We start with the analysis of the standard case in which a single list is generated in each composition. We 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 more interesting situations where the intermediate lists are passed between functions as part of a pair. As we saw in Section 3, compositions of this kind give rise to circular and higher-order definitions.

4.1. Standard case

Let us suppose that in every composition $f_{i+1} \circ f_i$ only an intermediate data structure (a list) is produced. To derive a monolithic definition from the whole sequence $f_n \circ \cdots \circ f_0$ the involved functions need to satisfy certain conditions. Clearly, f_0 needs to be a producer and f_n a consumer. Functions $f_1, \ldots, f_{n-1}$ are more interesting since they all need to be both consumers and producers simultaneously in order to be possible to fuse each of them with their neighbor 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 proper factors:

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

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

By expanding the definition of *factors* we see that two intermediate lists are generated within *sumFactors*:

$$sumFactors\ n = sum \circ filter\ (`isFactorOf`\ n) \circ down\ \$\ n\ `div`\ 2$$

To eliminate those lists the expression to be fused is $sum \circ filter\ p \circ down$ where $p = (`isFactorOf`\ n)$. From Section 2 we know that *down* is a producer and *sum* is a consumer (a fold). Concerning *filter p*, it is a consumer, but it can also be seen as a producer even maintaining its formulation as a fold. This is possible by 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, from the body of a fold, or which is the same, from the operations of the algebra of a fold, the occurrences of the constructors of the data structure that is produced as result. In the case of *filter p*, we can write:

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

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

$$([\,], \lambda a\ r \rightarrow \textbf{if}\ p\ a\ \textbf{then}\ a : r\ \textbf{else}\ r)$$

of the fold for *filter p* 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*).*

$$\tau :: \forall\, b\ .\ (b, a \rightarrow b \rightarrow b) \rightarrow (b, a' \rightarrow b \rightarrow b)$$
$$\Rightarrow$$
$$fold\ (nil, cons) \circ fold\ (\tau\ in) = fold\ (\tau\ (nil, cons))$$

Returning to the composition $sum \circ filter\ p \circ down$, 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 p*, and then fusing the result with *down*. For fusing $sum \circ filter\ p$ 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 p* with *down* and then fusing the result with *sum*. Fusion of $filter\ p \circ down$ is performed by applying Law 1, obtaining $gd\ (\tau\ in)$ as result; this coincides with the function *fd p* shown in Example 1. If we now want to fuse *sum* with $gd\ (\tau\ in)$ 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 can be fused resulting in 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 the same type as in Law 7.

Law 8 *(*CHAIN LAW*).*

$$\tau :: \forall\, b\ .\ (b, a \rightarrow b \rightarrow b) \rightarrow (b, a' \rightarrow b \rightarrow b)$$
$$\Rightarrow$$
$$fold\ (\tau\ in) \circ build\ g = build\ (g \circ \tau)$$

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

$$sfd\ p\ 0 = 0$$
$$sfd\ p\ n = \textbf{if}\ p\ n\ \textbf{then}\ n + sfd\ p\ (n-1)\ \textbf{else}\ sfd\ p\ (n-1)$$

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

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

$$\begin{aligned} & \tau :: \forall\, b\, .\, (b, a \to b \to b) \to (b, a' \to b \to b) \\ \Rightarrow & \\ & map_N\ (fold\ (\tau\ in)) \circ build_N\ g = build_N\ (g \circ \tau) \end{aligned}$$

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 where 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,*

$$\begin{aligned} & \tau :: \forall\, b\, .\, (b, a \to b \to b) \to (W\ b, a' \to W\ b \to W\ b) \\ \Rightarrow & \\ & map_N\ (fold\ (\tau\ in)) \circ build_N\ g = build_{NW}\ (g \circ \tau) \end{aligned}$$

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 $f_n \circ \cdots \circ f_0$ is such that a pair (t_i, z_i) of a data structure t_i (a list) and a value z_i is generated in each composition. Like before, f_0 needs to be a producer, f_n a consumer, whereas $f_1, \ldots, f_{n-1}$ need to be simultaneously consumers and producers. Therefore, we assume a sequence of compositions of the form $pfold\ h_n \circ \cdots \circ pfold\ h_1 \circ buildp\ g$.

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

4.2.1. 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. In other words, 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 the case where 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 $\sigma\ k = (\sigma_1\ k, \sigma_2\ k)$.*

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

Example 5. Given a set of points in a plane, the following program returns the maximum distance between the points located above the average height and the highest point below it. We first compute the average height by means of function *avrgHeight*, which is a tail recursive definition with two accumulators. Then, with *takePoints*, we take the points that are above the average height and determine the highest point below the average. Finally, we compute the maximum distance between that point and the points selected.

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

In order to fuse this composition, first we need to express these functions in terms of the corresponding program schemes.

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

By observing that the algebra $(tnil, tcons)$ can be expressed as $\sigma\ in$, where σ is the transformer:

$$
\begin{array}{l}
\sigma\ (nil, cons) = (\lambda avH \to (nil, (0, 0)) \\
\qquad\qquad\ \ , \lambda(x, y)\ r\ avH \to \\
\qquad\qquad\qquad \textbf{let}\ (ps, hp) = r \\
\qquad\qquad\qquad \textbf{in if}\ y > avH\ \textbf{then}\ (cons\ (x, y)\ ps, hp) \\
\qquad\qquad\qquad\qquad \textbf{else}\ (ps, \textbf{if}\ y > snd\ hp\ \textbf{then}\ (x, y)\ \textbf{else}\ hp))
\end{array}
$$

our program corresponds to the following composition:

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

Now we are in conditions to apply fusion laws. The transformation from right to left then proceeds by first applying Law 11 and then Law 4. The circular program obtained contains two circular arguments, $\boxed{u}$ and $\boxed{z}$:

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

The program obtained has a local definition *gk*, which computes a nested pair containing the result d and the intermediate context values that are produced by the original composition. Those context values are now the circular arguments $\boxed{z}$ and $\boxed{u}$, which denote, respectively, the average height and the height of the highest point below the average height. Notice that the value of $\boxed{z}$ is taken into account to compute the result of the function but also the value of the other circular argument $\boxed{u}$.

4.2.2. Left-to-right transformation

When the transformation is from left to right 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 the fusion step is a consumer. If that is the case then it is guaranteed that we can successively apply fusion until the end. We then state sufficient conditions to establish when the composition of two pfolds is again a pfold. The following is an acid rain law inspired in fold-fold fusion (Law 7).

Law 12 *(*PFOLD–PFOLD FUSION*).*

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

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

$$(v, z_n),\ ((v, z_n), z_{n-1}),\ \ldots,\ ((\ldots((v, z_n), z_{n-1}), \ldots), z_i),$$

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

$$(v_1, z_1),\ ((v_2, z_2), z_1),\ \ldots,\ ((\ldots(v_i, z_i), \ldots), z_1).$$

Returning to Example 5, the transformation from left to right proceeds by simply applying Law 12 and then Law 3. The program obtained is of course the same as before.

4.3. Derivation of higher-order programs

Now we turn to the analysis of laws that make it possible the derivation of higher-order programs from a sequence of compositions $f_n \circ \cdots \circ f_0$. Like in the case of simple compositions, the derivation of a higher-order program from a sequence of compositions represents the independence from a language with lazy evaluation.

Like in Subsection 4.2, we assume that a pair (t_i, z_i) of a data structure and a value is generated in each composition. Again, f_0 needs to be a producer, f_n a consumer, whereas $f_1, \ldots, f_{n-1}$ need to be simultaneously consumers and producers.

During the transformation to a higher-order program we will deal once 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 higher-order program when we transform a sequence of compositions $f_n \circ \cdots \circ f_0$ from right to left and from left to right. Again one of the differences is the order in which the nested structure is generated.

4.3.1. Right-to-left transformation

For the transformation in this order we need to analyse again the conditions that make it possible that a consumer (pfold) composed with a producer (buildp) is again a producer (buildp). The situation is similar to that of 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:

$$\tau :: \forall\, b\ .\ (b, a \to b \to b) \to (z \to W\ b, a' \to (z \to W\ b) \to (z \to W\ b))$$

where $\tau\ k = (\tau_1\ k, \tau_2\ k)$ with

$$\tau_1\ k = \lambda z \to \sigma_1\ k\ z \quad \text{and} \quad \tau_2\ k\ x\ r = \lambda z \to \sigma_2\ k\ x\ (r\ z)\ z$$

Notice 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 $\sigma\ k = (\sigma_1\ k, \sigma_2\ k)$.*

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

Example 6. Recall the function *distance* presented in Example 5. The higher-order program derivation in right to left order applied to this function proceeds by first applying Law 13 and then Law 6. The higher-order program obtained is the following:

```
distance ps = f u
  where
    (f, u) = g z
    (g, z) = gk 0 0 ps
    gk h l [ ] = (λz → (λu → 0, (0, 0)), h / fromInteger l)
    gk h l ((x, y) : ps) =
      let (r, avH) = gk (y + h) (1 + l) ps
      in (λz →
        let (md, hp) = r z
        in if y > z
           then (λu → sqrt ((x − fst u)^2 + (y − snd u)^2) `max` (md u), hp)
           else (λu → md u, if y > snd hp then (x, y) else hp), avH)
```

In this case, function *gk* returns a pair formed by a function g, that takes an integer denoting the average height as parameter, and the average height z. The application of g to z then returns a new pair formed by a new function f, that takes an integer denoting the height of the highest point below the average height, and precisely that height, denoted by u. Finally, f is applied to u to compute the final result.

4.3.2. Left-to-right transformation

For the transformation in this other direction we proceed similarly as we did for standard 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*).*

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

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

5. Data type theory

The programs schemes and laws introduced so far are actually valid for a wide class of datatypes and not only for lists. In this section we introduce the theoretical underlying concepts that are necessary for the generic formulation of program schemes and laws.

We write $\pi_1 :: (a, b) \to a$ and $\pi_2 :: (a, b) \to b$ to denote the product projections. The split of two functions f and g is defined as $(f \vartriangle g)\ x = (f\ x, g\ x)$. Among other properties, it holds that:

$$f \circ \pi_1 = \pi_1 \circ (f \times g) \tag{2}$$

$$g \circ \pi_2 = \pi_2 \circ (f \times g) \tag{3}$$

$$f = (\pi_1 \circ f) \vartriangle (\pi_2 \circ f) \tag{4}$$

The structure of datatypes can be captured using the concept of a *functor*. Recall that a functor consists of a type constructor F together with a function $map_F :: (a \to b) \to (F\ a \to F\ b)$, which preserves identities and compositions.

Semantically, recursive datatypes are understood as least fixed points of functors: Given a datatype declaration it is possible to derive a functor F such that the datatype is the least solution to the equation $x \cong Fx$. We write μF to denote the type corresponding to the least solution. The isomorphism between μF and $F\ \mu F$ is provided by two strict functions $in_F :: F\ \mu F \to \mu F$ and $out_F :: \mu F \to F\ \mu F$, inverses of each other. Function in_F packs the constructors of the datatype while out_F the destructors (for more details see e.g. [28,29]).

Example 7. The structure of the list type $[\ a\ ]$ is captured by a functor L on two variables (what is usually called a bifunctor) due to the presence of the type parameter.

$$\begin{array}{l}\textbf{data}\ L\ a\ b = FNil \mid FCons\ a\ b\\ map_L :: (a \to c) \to (b \to d) \to L\ a\ b \to L\ c\ d\\ map_L\ f\ g\ FNil \quad\ = FNil\\ map_L\ f\ g\ (FCons\ a\ b) = FCons\ (f\ a)\ (g\ b)\end{array}$$

We will use $\mu(L\ a)$ and $[\ a\ ]$ interchangeably. Then,

$$\begin{array}{l}in_{La} :: L\ a\ [\ a\ ] \to [\ a\ ]\\ in_{La}\ FNil \quad\ = [\ ]\\ in_{La}\ (FCons\ a\ as) = a : as\\ out_{La} :: [\ a\ ] \to L\ a\ [\ a\ ]\\ out_{La}\ Nil \quad = FNil\\ out_{La}\ (a : as) = FCons\ a\ as\end{array}$$

5.1. Fold

Given a functor F that captures the structure of a datatype and a function $k :: F\ a \to a$ (called an *F-algebra*), *fold* [29] is defined as the least function such that:

$$\begin{array}{l}fold_F :: (F\ a \to a) \to \mu F \to a\\ fold_F\ k \circ in_F = k \circ F\ (fold_F\ k)\end{array}$$

If functor F is given by n constructors,

$$\textbf{data}\ F\ a = FC_1\ t_{1,1}\ \cdots\ t_{1,r_1}\ \mid \cdots \mid FC_n\ t_{n,1}\ \cdots\ t_{n,r_n}$$

then an algebra $k :: F\ a \to a$ has n components $(k_1, \ldots, k_n)$, each with type $k_i :: t_{i,1} \to \cdots \to t_{i,r_i} \to a$, such that $k\ (FC_i\ v_{i,1} \cdots v_{i,r_i}) = k_i\ v_{i,1} \cdots v_{i,r_i}$. For example, an algebra for the functor $L\ a$ is a function $k :: L\ a\ b \to b$ of the form:

$$\begin{aligned} k\ FNil &= nil \\ k\ (FCons\ a\ b) &= cons\ a\ b \end{aligned}$$

with components $nil :: b$ and $cons :: a \to b \to b$. This is the reason why in the specific instance of fold for the list type [a] (defined in Section 2) we wrote an algebra simply as a pair $(nil, cons)$. The same can be done with algebras associated to other datatypes.

Given a functor F, we can define a corresponding *build* operator that captures producer functions that generate structures of type μF.

$$\begin{aligned} build_F &:: (\forall\ a\ .\ (F\ a \to a) \to c \to a) \to c \to \mu F \\ build_F\ g &= g\ in_F \end{aligned}$$

Notice that the abstraction of the datatype's constructors is given in terms of an F-algebra. Associated with build we have the following shortcut fusion law [30].

Law 15 (FOLD/BUILD). *For strict k,*[1]

$$fold_F\ k \circ build_F\ g = g\ k$$

5.2. Fold with parameters

We are interested in consumer functions of the form $f :: (\mu F, z) \to a$, defined by structural recursion on μF and with context information of type z. Such functions can be defined as a higher-order fold of type $\mu F \to (z \to a)$. Another alternative is to avoid the use of higher-order and to define them directly in terms of a program scheme called *pfold* (a fold with parameters) [31]. The definition of pfold relies on the concept of *strength* of a functor F, a polymorphic function $st_F :: (F\ a, z) \to F\ (a, z)$ that distributes the value of type z to the variable positions of functor F (i.e. those of type a). To be a strength, st_F must satisfy certain coherence axioms (see [31,32] for details).

For instance, the strength corresponding to functor $L\ a$ is given by:

$$\begin{aligned} st_{La} &:: (L\ a\ b, z) \to L\ a\ (b, z) \\ st_{La}\ (FNil, z) &= FNil \\ st_{La}\ (FCons\ a\ b, z) &= FCons\ a\ (b, z) \end{aligned}$$

The strength plays an important role in the definition of pfold as it represents the distribution of the context information to the recursive calls.

Given a functor F and a function $h :: (F\ a, z) \to a$, *pfold* [31] is defined as the least function such that:

$$\begin{aligned} pfold_F &:: ((F\ a, z) \to a) \to (\mu F, z) \to a \\ pfold_F\ h \circ (in_F \times id) &= h \circ ((map_F\ (pfold_F\ h) \circ st_F) \vartriangle \pi_2) \end{aligned}$$

A function h is something similar to an algebra, but it also accepts the value of the parameters. In fact, if functor F is given by n constructors,

$$\textbf{data}\ F\ a = FC_1\ t_{1,1}\ \cdots\ t_{1,r_1}\ \mid \cdots \mid FC_n\ t_{n,1}\ \cdots\ t_{n,r_n}$$

then $h :: (F\ a, z) \to a$ has also n components $(h_1, \ldots, h_n)$, now each with type $h_i :: t_{i,1} \to \cdots \to t_{i,r_i} \to z \to a$. For example, for functor $L\ a$, $h :: (L\ a\ b, z) \to b$ is of the form:

$$\begin{aligned} h\ (FNil, z) &= hnil\ z \\ h\ (FCons\ a\ b, z) &= hcons\ a\ b\ z \end{aligned}$$

where $hnil :: z \to b$ and $hcons :: a \to b \to z \to b$ are the component functions. Like with algebras, in the specific instances of pfold we write the tuple of components instead of h. We did so in the instance for lists defined in Section 3.

The following equation shows one of the possible relationships between pfold and fold. For h with components $(h_1, \ldots, h_n)$, by fixing the value z we have that:

$$pfold_F\ h\ (t, z) = fold_F\ k\ t\ \textbf{where}\ k_i\ \bar{x} = h_i\ \bar{x}\ z \tag{5}$$

By $\bar{x}$ we denote a sequence of values $x_1 \cdots x_{r_i}$. Observe that k is an algebra with components $(k_1, \ldots, k_n)$.

[1] The strictness condition on k was not mentioned in the concrete instance of the law for lists (Law 1, shown in Section 2) because a function defined by pattern matching is strict. That is the case of the algebras considered in those instances.

5.3. Extended shortcut fusion

By extended shortcut fusion we mean the case where the intermediate data structure is generated as part of another structure. This extension has been a fundamental tool for the formulation of shortcut fusion laws for monadic programs [18,19], and for the derivation of (monadic) circular and higher-order programs [21,20]. It is based on an extended form of build:

$$build_{F,N} :: (\forall\, a\, .\, (F\, a \to a) \to c \to N\, a) \to c \to N\, \mu F$$
$$build_{F,N}\, g = g\, in_F$$

where N is a functor that represents the structure in which the produced data structure is contained. This is a natural extension of the standard build function as $build_F$ can be obtained from $build_{F,N}$ by considering the identity functor.

Law 16 (EXTENDED FOLD/BUILD). *For strict k and strictness-preserving N,*

$$map_N\ (fold_F\ k) \circ build_{F,N}\ g = g\ k$$

Proof. See e.g. [18]. □

6. Multiple intermediate structure deforestation, generically

In this section we present the generic formulation of the laws that give sufficient conditions for the derivation of monolithic programs from sequences of compositions. We start with the analysis of the standard case of a sequence of consumers given by folds followed by a build. Then we analyse the case of a sequence of pfolds followed by a buildp which gives rise to programs with multiple circularities or nested higher-order.

6.1. Standard case

As we saw in the list case, when the consumers are given by folds, i.e. when the sequence of compositions is of the form $fold\ k_n \circ \cdots fold\ k_1 \circ build\ g$, both transformations (left-to-right and right-to-left) are of the same degree of complexity.

The *left-to-right transformation* is based on the following standard *acid rain law* for fold which states conditions for fusing consumers. It is a generic version of Law 7 which shows the case for lists. The transformer τ has now a generic formulation, specifying a transformation from F-algebras to G-algebras, for arbitrary functors F and G.

Law 17 (FOLD-FOLD FUSION). *For strict k,*

$$\tau :: \forall\, a\, .\, (F\, a \to a) \to (G\, a \to a)$$
$$\Rightarrow$$
$$fold_F\ k \circ fold_G\ (\tau\ in_F) = fold_G\ (\tau\ k)$$

Proof. We use the fact that acid rain laws can be expressed in terms of shortcut fusion [30]. By defining $g\, k = fold_G\ (\tau\ k)$ it follows that $fold_G\ (\tau\ in_F) = build_F\ g$ and therefore by Law 15 we obtain the desired result. □

After fusing all consumers from left to right, the resulting fold is finally fused with the producer (the *build* function) using standard shortcut fusion.

The *right-to-left transformation* starts by fusing the producer with the first consumer. In order to keep going with the fusion process it is necessary that, at each intermediate step of the transformation, the result of fusing producer and consumer is again a producer. This condition is stated by the following law.

Law 18 (CHAIN LAW). *For strictness-preserving τ,*

$$\tau :: \forall\, a\, .\, (F\, a \to a) \to (G\, a \to a)$$
$$\Rightarrow$$
$$fold_G\ (\tau\ in_F) \circ build_G\ g = build_F\ (g \circ \tau)$$

It is then natural to state a chain law for the extension. The previous law is simply the case where N is the identity functor.

Law 19 (EXTENDED CHAIN LAW). *For strictness-preserving τ and N,*

$$\tau :: \forall\, a\, .\, (F\, a \to a) \to (G\, a \to a)$$
$$\Rightarrow$$
$$map_N\ (fold_G\ (\tau\ in_F)) \circ build_{G,N}\ g = build_{F,N}\ (g \circ \tau)$$

Proof. The algebra $\tau\ in_F$ is strict because in_F is a strict algebra and τ preserves strictness.

$$
\begin{aligned}
& N\ (fold_G\ (\tau\ in_F)) \circ build_{G,N}\ g \\
= & \quad \{\ \text{Law 16}\ \} \\
& g\ (\tau\ in_F) \\
= & \quad \{\ g \circ \tau :: \forall\, a\ .\ (F\ a \to a) \to c \to N\ a, \text{definition of extended } build\ \} \\
& build_{F,N}\ (g \circ \tau) \qquad \square
\end{aligned}
$$

We can apply the generalization a step further. The following law describes a situation where the transformer τ returns an algebra whose carrier is not exactly the carrier of the input algebra but the application of a function on it.

Law 20. *Let W be a functor. For strictness-preserving τ and N,*

$$
\begin{aligned}
& \tau :: \forall\, a\ .\ (F\ a \to a) \to (G\ (W\ a) \to (W\ a)) \\
\Rightarrow & \\
& map_N\ (fold_G\ (\tau\ in_F)) \circ build_{G,N}\ g = build_{F,NW}\ (g \circ \tau)
\end{aligned}
$$

The proof is similar to that of the previous law.

6.2. Derivation of programs with multiple circularities

The derivation of circular programs on arbitrary datatypes uses the generic version of *buildp*:

$$
\begin{aligned}
& buildp_F \quad :: (\forall\, a\ .\ (F\ 2a \to a) \to c \to (a, z)) \to c \to (\mu F, z) \\
& buildp_F\ g = g\ in_F
\end{aligned}
$$

Law 21 (PFOLD/BUILDP)*. For h with components $(h_1, \ldots, h_n)$,*

$$
\begin{aligned}
& pfold_F\ h \circ buildp_F\ g\ \$\ c = v \\
& \quad \textbf{where}\ (v, \boxed{z}) = g\ k\ c \\
& \qquad\ \ k_i\ \bar{x} = h_i\ \bar{x}\ \boxed{z}
\end{aligned}
$$

Proof. See [20]. $\square$

Like we saw in the case of lists, a sequence of compositions $pfold\ h_n \circ \cdots \circ pfold\ h_1 \circ buildp\ g$ gives rise to a program with multiple circularities whenever some conditions are satisfied. Before stating those conditions in the generic case, we present an extension of the previous law which will be used later. It uses an extension of buildp that captures the case where the pair is returned as part of another structure.

$$
\begin{aligned}
& buildp_{F,N} \quad :: (\forall\, a\ .\ (F\ a \to a) \to c \to N\ (a, z)) \to c \to N\ (\mu F, z) \\
& buildp_{F,N}\ g = g\ in_F
\end{aligned}
$$

As we did for lists, we assume that functor N has associated a polymorphic function $\epsilon_N :: N\ a \to a$.

Law 22. *Let (N, ϵ_N) be strictness-preserving. For h with components $(h_1, \ldots, h_n)$,*

$$
\begin{aligned}
& pfold_F\ h \circ \epsilon_N \circ buildp_{F,N}\ g\ \$\ c = v \\
& \quad \textbf{where}\ (v, \boxed{z}) = \epsilon_N \circ g\ k\ \$\ c \\
& \qquad\ \ k_i\ \bar{x} = h_i\ \bar{x}\ \boxed{z}
\end{aligned}
$$

Proof. This proof is a simply generalization of that of Law 21.

$$
\begin{aligned}
& pfold_F\ h \circ \epsilon_N \circ buildp_{F,N}\ g\ \$\ c \\
= & \quad \{\ \text{definition of } buildp\ \} \\
& pfold_F\ h \circ \epsilon_N \circ g\ in_F\ \$\ c \\
= & \quad \{\ (4)\ \} \\
& pfold_F\ h\ \$\ (\pi_1 \circ \epsilon_N \circ g\ in_F) \vartriangle (\pi_2 \circ \epsilon_N \circ g\ in_F)\ \$\ c \\
= & \quad \{\ (5),\ k_i\ \bar{x} = h_i\ \bar{x}\ z\ \} \\
& fold_F\ k \circ \pi_1 \circ \epsilon_N \circ g\ in_F\ \$\ c\ \textbf{where}\ z = \pi_2 \circ \epsilon_N \circ g\ in_F\ \$\ c \\
= & \quad \{\ (2)\ \} \\
& \pi_1 \circ (fold_F\ k \times id) \circ \epsilon_N \circ g\ in_F\ \$\ c\ \textbf{where}\ z = \pi_2 \circ \epsilon_N \circ g\ in_F\ \$\ c
\end{aligned}
$$

$$
\begin{aligned}
&= \quad \{\ (1)\ \} \\
&\quad \pi_1 \circ \epsilon_N \circ map_N\ (fold_F\ k \times id) \circ g\ in_F\ \$\ c\ \textbf{where}\ z = \pi_2 \circ \epsilon_N \circ g\ in_F\ \$\ c \\
&= \quad \{\ \text{Law 16 with functor } M\ a = N\ (a, z)\ \} \\
&\quad \pi_1 \circ \epsilon_N \circ g\ k\ \$\ c\ \textbf{where}\ z = \pi_2 \circ \epsilon_N \circ g\ in_F\ \$\ c \\
&= \quad \{\ (1) \text{ and } map_N\ \pi_2 \circ g\ in_F = map_N\ \pi_2 \circ g\ k\ \} \\
&\quad \pi_1 \circ \epsilon_N \circ g\ k\ \$\ c\ \textbf{where}\ z = \pi_2 \circ \epsilon_N \circ g\ k\ \$\ c \\
&= \quad \{\ (4)\ \} \\
&\quad v\ \textbf{where}\ (v, \boxed{z}) = \epsilon_N\ \$\ g\ k\ \$\ c \\
&\qquad\qquad k_i\ x = h_i\ \bar{x}\ \boxed{z} \qquad\qquad \square
\end{aligned}
$$

We start analyzing the *right-to-left transformation*. In order to proceed, it is necessary that, at each intermediate step of the transformation, the result of fusing a producer (buildp) with a consumer (pfold) is again a producer (buildp). This condition is established by the following laws.

Given a transformer $\sigma :: \forall\ a\ .\ (F\ a \to a) \to ((G\ a, z) \to a)$ such that, for each algebra k, $\sigma\ k$ has components $(\sigma_1\ k, \ldots, \sigma_n\ k)$, it is possible to derive an algebra transformer $\tau :: \forall\ a\ .\ (F\ a \to a) \to (G\ a \to a)$ from σ, such that $\tau\ k$ has components $(\tau_1\ k, \ldots, \tau_n\ k)$, where $\tau_i\ k\ \bar{x} = \sigma_i\ k\ \bar{x}\ z$ for a fixed z. In the same manner, an algebra transformer

$$\tau :: \forall\ a\ .\ (F\ a \to a) \to (G\ (W\ a) \to W\ a)$$

can be derived from

$$\sigma :: \forall\ a\ .\ (F\ a \to a) \to ((G\ (W\ a), z) \to W\ a)$$

Such a σ is used in the next law to represent the case where the consumer is also a producer. Observe that $pfold_G\ (\sigma\ in_F) :: (\mu G, z) \to W\ \mu F$.

Law 23. *Let (N, ϵ_N) be strictness-preserving and $M\ a = N\ (a, z)$. Let $\sigma\ k = (\sigma_1\ k, \ldots, \sigma_n\ k)$.*

$$
\begin{aligned}
&\quad \sigma :: \forall\ a\ .\ (F\ a \to a) \to ((G\ (W\ a), z) \to W\ a) \\
&\Rightarrow \\
&\quad pfold_G\ (\sigma\ in_F) \circ \epsilon_N \circ buildp_{G,N}\ g\ \$\ c = v \\
&\qquad \textbf{where}\ (v, \boxed{z}) = \epsilon_N \circ build_{F,MW}\ (g \circ \tau)\ \$\ c \\
&\qquad\qquad \tau_i\ k\ \bar{x} = \sigma_i\ k\ \bar{x}\ \boxed{z}
\end{aligned}
$$

Proof.

$$
\begin{aligned}
&\quad pfold\ (\sigma\ in_F)\ \$\ \epsilon_N\ \$\ buildp_{G,N}\ g\ \$\ c \\
&= \quad \{\ \text{Law 22}\ \} \\
&\quad v \quad \textbf{where}\ (v, \boxed{z}) = \epsilon_N\ \$\ g\ (\tau\ in_F)\ \$\ c \\
&\qquad\qquad \tau_i\ k\ \bar{x} = \sigma_i\ k\ \bar{x}\ \boxed{z} \\
&= \quad \{\ g \circ \tau :: \forall\ a\ .\ (F\ a \to a) \to c \to M\ (W\ a), \text{ definition of } build\ \} \\
&\quad v \quad \textbf{where}\ (v, \boxed{z}) = \epsilon_N\ \$\ build_{F,MW}\ (g \circ \tau)\ \$\ c \\
&\qquad\qquad \tau_i\ k\ \bar{x} = \sigma_i\ k\ \bar{x}\ \boxed{z} \qquad\qquad \square
\end{aligned}
$$

By taking $W\ a = (a, y)$ in the previous law and observing that, for that W, $build_{F,MW}\ g = buildp_{F,M}\ g$, we get the following chain rule that is the essential tool for the right-to-left derivation of programs with multiple circularities. In this case $pfold_G\ (\sigma\ in_F) :: (\mu G, z) \to (\mu F, y)$.

Law 24 (CHAIN RULE). *Let (N, ϵ_N) be strictness-preserving and $M\ a = N\ (a, z)$. Let $W\ a = (a, y)$, for some type y, and $\sigma\ k = (\sigma_1\ k, \ldots, \sigma_n\ k)$.*

$$
\begin{aligned}
&\quad \sigma :: \forall\ a\ .\ (F\ a \to a) \to ((G\ (W\ a), z) \to W\ a) \\
&\Rightarrow \\
&\quad pfold_G\ (\sigma\ in_F) \circ \epsilon_N \circ buildp_{G,N}\ g\ \$\ c = p \\
&\qquad \textbf{where}\ (p, \boxed{z}) = \epsilon_N \circ buildp_{F,M}\ (g \circ \tau)\ \$\ c \\
&\qquad\qquad \tau_i\ k\ \bar{x} = \sigma_i\ k\ \bar{x}\ \boxed{z}
\end{aligned}
$$

The *left-to-right transformation* is much easier and is supported by the following acid rain law.

Law 25 (PFOLD–PFOLD FUSION).

$$\begin{array}{l} \quad \sigma :: \forall\, a\,.\,(F\,a \to a) \to ((G\,a, z) \to a) \\ \Rightarrow \\ \quad pfold_F\ h \circ pfold_G\ (\sigma\ in_F)\ \$\ c = v \\ \quad\quad \textbf{where}\ (v, \boxed{z}) = pfold_G\ (\sigma\ k)\ c \\ \quad\quad\quad\quad k_i\ \bar{x} = h_i\ \bar{x}\ \boxed{z} \end{array}$$

Proof. Like fold-fold fusion, this law can also be formulated in terms of shortcut fusion. By defining $g\ k = pfold_G\ (\sigma\ k)$, we have that $pfold_G\ (\sigma\ in) = buildp_G\ g$. Then, by Law 21 the result follows. □

6.3. Derivation of higher-order programs

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 $h :: (F\,a, z) \to a$,

$$pfold_F\ h = apply \circ (fold_F\ \varphi_h \times id) \tag{6}$$

with the algebra $\varphi_h :: F\,(z \to a) \to (z \to a)$ given by

$$\varphi_h = curry\ (h \circ ((map_F\ apply \circ st_F) \vartriangle \pi_2))$$

where $apply :: (a \to b, a) \to b$ is function application $apply\ (f, x) = f\ x$. Therefore, $fold_F\ \varphi_h :: \mu F \to (z \to a)$ is the curried version of $pfold_F\ h$.

Using this relationship we can state the following law. As mentioned for lists, this law is a special case of a more general program transformation technique called *lambda abstraction* [27].

Law 26 (H-O PFOLD/BUILDP). *For left-strict h,*[2]

$$\begin{array}{l} pfold_F\ h \circ buildp_F\ g\ \$\ c = f\ z \\ \quad \textbf{where}\ (f, z) = g\ \varphi_h\ c \end{array}$$

As for circular programs, we can also formulate an extended version.

Law 27. *Let (N, ϵ_N) be strictness-preserving functor. For left-strict h,*

$$\begin{array}{l} pfold_F\ h \circ \epsilon_N \circ buildp_{F,N}\ g\ \$\ c = f\ z \\ \quad \textbf{where}\ (f, z) = \epsilon_N \circ g\ \varphi_h\ \$\ c \end{array}$$

The *right-to-left transformation* requires the chain law that we present below. Given a transformer

$$\sigma :: \forall\, a\,.\,(F\,a \to a) \to ((G\,(W\,a), z) \to W\,a)$$

with W an arbitrary functor, it is possible to derive an algebra transformer

$$\tau :: \forall\, a\,.\,(F\,a \to a) \to (G\,(z \to W\,a) \to (z \to W\,a))$$

from σ, given by $\tau\ k = \varphi_{\sigma k}$.

Law 28. *Let (N, ϵ_N) be strictness-preserving and $M\,a = N\,(z \to a)$. Let σ be strictness-preserving.*

$$\begin{array}{l} \quad \sigma :: \forall\, a\,.\,(F\,a \to a) \to ((G\,(W\,a), z) \to W\,a) \\ \Rightarrow \\ \quad pfold_G\ (\sigma\ in_F) \circ \epsilon_N \circ buildp_{G,N}\ g\ \$\ c = f\ z \\ \quad\quad\quad \textbf{where}\ (f, z) = \epsilon_N \circ build_{F,MW}\ (g \circ \tau)\ \$\ c \\ \quad\quad\quad\quad\quad \tau\ k = \varphi_{\sigma k} \end{array}$$

Finally, by considering $W\,a = (a, y)$ and using the fact that $build_{F,MW}\ g = buildp_{F,M}\ g$ we get the chain law.

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

Law 29 *(H-O CHAIN RULE). Let (N, ϵ_N) be strictness-preserving and $M\ a = N\ (z \to a)$. Let $W\ a = (a, y)$, for some type y, and σ strictness-preserving.*

$$\begin{aligned}
&\sigma :: \forall\, a\ .\ (F\ a \to a) \to ((G\ (W\ a), z) \to W\ a)\\
\Rightarrow\\
&\mathit{pfold}_G\ (\sigma\ \mathit{in}_F) \circ \epsilon_N \circ \mathit{buildp}_{G,N}\ g\ \$\ c = p\\
&\quad \textbf{where}\ (f, z) = \epsilon_N \circ \mathit{buildp}_{F,M}\ (g \circ \tau)\ \$\ c\\
&\qquad\quad \tau\ k = \varphi_{\sigma k}
\end{aligned}$$

The *left-to-right transformation* relies on the following acid rain law.

Law 30 *(H-O PFOLD–PFOLD FUSION). For left strict h,*

$$\begin{aligned}
&\sigma :: \forall\, a\ .\ (F\ a \to a) \to ((G\ a, z) \to a)\\
\Rightarrow\\
&\mathit{pfold}_F\ h \circ \mathit{pfold}_G\ (\sigma\ \mathit{in}_F)\ \$\ c = f\ z\\
&\quad \textbf{where}\ (f, z) = \mathit{pfold}_G\ (\sigma\ \varphi_h)\ c
\end{aligned}$$

7. Conclusions

In this paper, we presented a shortcut fusion approach to deforestation in programs consisting of an arbitrary number of function compositions. Indeed, while shortcut fusion was originally conceived for compositions of a single consumer function with a single producer function, we showed how multiple compositions can be fused together in a single function definition. We addressed cases of standard shortcut fusion [1] as well as extensions from which it is possible the derivation of circular and higher-order programs [3]. Our approach was calculational and established sufficient conditions for fusion to proceed. A relevant aspect of our work was the combination of shortcut fusion with the fusion approach based on the formulation of acid rain laws. This combination of fusion approaches led to the definition of the chain laws, which constitute one of the tools for the formulation of sufficient conditions for the fusion of multiple compositions.

We also analyzed the order in which the fusion process may be applied (left-to-right or right-to-left) and we showed that both cases are indistinct since they lead to exactly the same result (same code). The difference between them is in the complexity of their steps. Comparing the two, the left-to-right transformation is perhaps the simplest one as it only requires the successive application of acid rain laws (for fold or pfold) and a shortcut fusion law at the end. In contrast, the right-to-left transformation requires the successive application of chain laws, concluding with the application of a shortcut fusion law.

The derivation of circular programs is strongly associated with attribute grammar research [12,33], and we expect our work to clarify even further their similar nature. On the other hand, the derivation of higher-order programs is motivated by efficiency. Indeed, as the programs we considered here are of the same kind as the ones we have compared in an exploratory way in [34], we expect the derived higher-order programs to be more efficient when compared to their multiple traversal and circular counterparts. We believe, however, that comparing the performance characteristics (both in terms of runtime and memory efficiency) of the different types of programs we considered requires a non-trivial, systematic, complete and representative benchmark that on its own sets a very interesting path for future research.

In this work we only analyzed the case of multiple compositions of purely functional programs. The extension for multiple compositions of monadic programs (i.e. programs that produce effects modeled by monads) is left for future work. The starting point for this extension will be the shortcut fusion laws for single compositions of monadic programs [3,18].

Acknowledgements

We would like to thank the anonymous reviewers for their detailed and helpful comments. This work was partially 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.

References

[1] A. Gill, J. Launchbury, S. Peyton Jones, A short cut to deforestation, in: Functional Programming Languages and Computer Architecture, ACM, 1993.
[2] R. Bird, Using circular programs to eliminate multiple traversals of data, Acta Inform. 21 (1984) 239–250.
[3] A. Pardo, J.P. Fernandes, J. Saraiva, Shortcut fusion rules for the derivation of circular and higher-order programs, High.-Order Symb. Comput. 24 (1–2) (2011) 115–149.
[4] J. Voigtländer, Semantics and pragmatics of new shortcut fusion rules, in: Functional and Logic Programming, Springer-Verlag, 2008, pp. 163–179.
[5] D. Swierstra, O. Chitil, Linear, bounded, functional pretty-printing, J. Funct. Program. 19 (1) (2009) 1–16.
[6] Y. Onoue, Z. Hu, H. Iwasaki, M. Takeichi, A calculational fusion system HYLO, in: IFIP TC 2 Working Conference on Algorithmic Languages and Calculi, Chapman & Hall, 1997, pp. 76–106.

[7] J. Launchbury, T. Sheard, Warm fusion: deriving build-catas from recursive definitions, in: Functional Programming Languages and Computer Architecture, ACM, 1995.
[8] D.E. Knuth, Semantics of context-free languages, Math. Syst. Theory 2 (2) (1968) 127–145, Correction: Math. Syst. Theory 5 (1) (March 1971) 95–96.
[9] T. Johnsson, Attribute grammars as a functional programming paradigm, in: Functional Programming Languages and Computer Architecture, in: Lect. Notes Comput. Sci., vol. 274, Springer, 1987.
[10] M. Kuiper, D. Swierstra, Using attribute grammars to derive efficient functional programs, in: Computing Science in the Netherlands CSN'87, 1987.
[11] U. Kastens, Ordered attribute grammars, Acta Inform. 13 (1980) 229–256.
[12] J.P. Fernandes, J. Saraiva, Tools and Libraries to model and manipulate circular programs, in: Partial Evaluation and Program Manipulation, ACM, 2007.
[13] P. Johann, J. Voigtländer, Free theorems in the presence of seq, in: Symposium on Principles of Programming Languages, ACM, 2004, pp. 99–110.
[14] R. Bird, O. de Moor, Algebra of Programming, Prentice–Hall Int. Ser. Comput. Sci., vol. 100, Prentice–Hall, 1997.
[15] O. Chitil, Type-inference based deforestation of functional programs, Ph.D. thesis, RWTH Aachen, October 2000.
[16] P. Wadler, Theorems for free!, in: Functional Programming Languages and Computer Architecture, ACM, 1989.
[17] R. Milner, A theory of type polymorphism in programming, J. Comput. Syst. Sci. 17 (3) (1978) 348–375, http://dx.doi.org/10.1016/0022-0000(78)90014-4.
[18] C. Manzino, A. Pardo, Shortcut fusion of monadic programs, J. Univers. Comput. Sci. 14 (21) (2008) 3431–3446.
[19] N. Ghani, P. Johann, Short cut fusion for effects, in: TFP'08, in: Trends in Functional Programming, Intellect, vol. 9, 2009, pp. 113–128.
[20] J.P. Fernandes, A. Pardo, J. Saraiva, A shortcut fusion rule for circular program calculation, in: Haskell Workshop, 2007, pp. 95–106.
[21] A. Pardo, J.P. Fernandes, J. Saraiva, Shortcut fusion rules for the derivation of circular and higher-order monadic programs, in: Partial Evaluation and Program Manipulation, 2009, pp. 81–90.
[22] L. Allison, Circular programs and self-referential structures, Softw. Pract. Exp. 19 (2) (1989) 99–109.
[23] C. Okasaki, Breadth-first numbering: lessons from a small exercise in algorithm design, ACM SIGPLAN Not. 35 (9) (2000) 131–136.
[24] S. Marlow, S. Peyton Jones, The new GHC/Hugs Runtime System, 1999.
[25] R. Hinze, J. Jeuring, Generic Haskell: practice and theory, in: Summer School on Generic Programming, 2002.
[26] A. Dijkstra, D. Swierstra, Typing Haskell with an attribute grammar (Part I), Tech. rep. UU-CS-2004-037, Institute of Information and Computing Sciences, Utrecht University, 2004.
[27] A. Pettorossi, A. Skowron, The lambda abstraction strategy for program derivation, Fundam. Inform. 12 (4) (1989) 541–561.
[28] S. Abramsky, A. Jung, Domain theory, in: Handbook of Logic in Computer Science, Clarendon Press, 1994, pp. 1–168.
[29] J. Gibbons, Calculating functional programs, in: Algebraic and Coalgebraic Methods in the Mathematics of Program Construction, Springer, 2002, pp. 148–203.
[30] A. Takano, E. Meijer, Shortcut deforestation in calculational form, in: Functional Programming Languages and Computer Architecture, ACM, 1995, pp. 306–313.
[31] A. Pardo, Generic accumulations, in: IFIP WG2.1 Working Conference on Generic Programming, Dagstuhl, Germany, 2002.
[32] R. Cockett, D. Spencer, Strong categorical datatypes I, in: R. Seely (Ed.), International Meeting on Category Theory, 1991, in: Conf. Proc., Can. Math. Soc., vol. 13, 1991, pp. 141–169.
[33] J.P. Fernandes, J. Saraiva, D. Seidel, J. Voigtländer, Strictification of circular programs, in: Partial Evaluation and Program Manipulation, ACM, 2011.
[34] J.P. Fernandes, Design, implementation and calculation of circular programs, Ph.D. thesis, Department of Informatics, University of Minho, Portugal, 2009.