Higher-Order Symb Comput (2011) 24:115–149
DOI 10.1007/s10990-011-9076-x

Shortcut fusion rules for the derivation of circular and higher-order programs

Alberto Pardo · João Paulo Fernandes · João Saraiva

Published online: 22 July 2011

Abstract Functional programs often combine separate parts using intermediate data structures for communicating results. Programs so defined are modular, easier to understand and maintain, but suffer from inefficiencies due to the generation of those gluing data structures. To eliminate such redundant data structures, some program transformation techniques have been proposed. One such technique is shortcut fusion, and has been studied in the context of both pure and monadic functional programs.

In this paper, we study several shortcut fusion extensions, so that, alternatively, circular or higher-order programs are derived. These extensions are also provided for effect-free programs and monadic ones. Our work results in a set of generic calculation rules, that are widely applicable, and whose correctness is formally established.

Keywords Shortcut fusion · Circular and higher-order programming · Monadic computations

Research conducted under the SSaaPP research project, PTDC/EIA-CCO/108613/2008.
J.P. Fernandes was supported by Fundação para a Ciência e Tecnologia, Grant No. SFRH/BPD/46987/2008.

A. Pardo (✉)
Instituto de Computación, Universidad de la República, Montevideo, Uruguay
e-mail: pardo@fing.edu.uy

J.P. Fernandes
Departamento de Eng. Informática, Faculdade de Engenharia, Universidade do Porto, Porto, Portugal
e-mail: jpaulo@fe.up.pt

J.P. Fernandes · J. Saraiva
HASLab/CCTC, Departamento de Informática, Universidade do Minho, Braga, Portugal

J.P. Fernandes
e-mail: jpaulo@di.uminho.pt

J. Saraiva
e-mail: jas@di.uminho.pt

1 Introduction

In a purely functional setting, programs are often constructed as a set of simple and clear components, which are *glued* together by using intermediate data structures. Compilers are a typical example of programs designed in this way: a function, the *parser*, constructs a syntax tree that is later decorated by another function.

There are many advantages in structuring our programs in this way. Considered in isolation, each function, or traversal, may be relatively simple. Consequently, programs so defined are easier to write, to understand and are potentially more reusable. By separating distinct phases, it becomes possible to focus on a single task, rather than attempting to do many things at the same time. Furthermore, there are algorithms that rely on a multiple traversal strategy because *context information* must first be collected before it can be used. In other words, information needs to flow from one traversal to the next one. This type of information is usually conveyed using intermediate data structures. The construction, traversal and destruction of a (potentially) large number of such data structures, however, may degrade the efficiency of the complete program.

An alternative way to write multiple traversal programs in a lazy functional setting, avoiding the definition of intermediate data structures, is to use circular programming. Circular programs were first proposed by Bird [3] as an elegant technique to eliminate multiple traversals of data structures. As the name suggests, these programs hold what appears to be a circular definition of the form $(\ldots, x, \ldots) = f \ldots x \ldots$, where the argument x in the call to f is also a result of that same call.

In order to motivate the use of circular programs, Bird introduces the following problem, widely known as *repmin*: consider that we want to transform a binary leaf tree into a second tree, identical in shape to the original one, but with all the leaf values replaced by the minimal one. In order to implement *repmin*, we start by defining a representation for binary leaf trees; we use the following HASKELL data-type definition:

$$\textbf{data}\ \mathit{Tree} = \mathit{Leaf}\ \mathit{Int} \mid \mathit{Fork}\ \mathit{Tree}\ \mathit{Tree}$$

We may now consider different implementations for solving *repmin*. In a strict, purely functional setting, for example, solving this problem requires a two traversal strategy. First, we need to traverse the input tree in order to compute its minimum value:

$$\begin{aligned}
&\mathit{tmin} :: \mathit{Tree} \to \mathit{Int}\\
&\mathit{tmin}\ (\mathit{Leaf}\ n) \ = n\\
&\mathit{tmin}\ (\mathit{Fork}\ l\ r) = \mathit{min}\ (\mathit{tmin}\ l)\ (\mathit{tmin}\ r)
\end{aligned}$$

Having traversed the input tree to compute its minimum value, we need to traverse that tree again. We need to replace all its leaf values by the minimum value:

$$\begin{aligned}
&\mathit{replace} :: (\mathit{Tree}, \mathit{Int}) \to \mathit{Tree}\\
&\mathit{replace}\ (\mathit{Leaf}\ _, m) \ = \mathit{Leaf}\ m\\
&\mathit{replace}\ (\mathit{Fork}\ l\ r, m) = \mathit{Fork}\ (\mathit{replace}\ (l, m))\ (\mathit{replace}\ (r, m))
\end{aligned}$$

Having implemented functions *replace* and *tmin*, we now only need, in order to solve *repmin*, to combine them appropriately:

$$\begin{aligned}
&\mathit{transform} :: \mathit{Tree} \to \mathit{Tree}\\
&\mathit{transform}\ t = \mathit{replace}\ (t, \mathit{tmin}\ t)
\end{aligned}$$

As we have noticed, this solution to *repmin* traverses any input tree twice. The original tree serves as the intermediate data structure that glues the two traversals together. However, a two traversal strategy is not essential to solve the *repmin* problem. An alternative solution

can, on a single traversal, compute the minimum leaf value and, at the same time, replace all values by that minimum value. Bird [3] showed how the single traversal program, presented next, may be obtained by transforming the original program, using well known techniques such as tupling, fold-unfold and local recursion.

$$
\begin{aligned}
&\mathit{repmin}\ (\mathit{Leaf}\ n, m) = (\mathit{Leaf}\ m, n)\\
&\mathit{repmin}\ (\mathit{Fork}\ l\ r, m) = (\mathit{Fork}\ t_1\ t_2, \mathit{min}\ m_1\ m_2)\\
&\quad \textbf{where}\ (t_1, m_1) = \mathit{repmin}\ (l, m)\\
&\qquad\quad (t_2, m_2) = \mathit{repmin}\ (r, m)\\
&\\
&\mathit{transform}\ t = nt\\
&\quad \textbf{where}\ (nt, \boxed{\mathrm{m}}) = \mathit{repmin}\ (t, \boxed{\mathrm{m}})
\end{aligned}
$$

Notice the circularity in this program: m is both an argument and a result of the *repmin* call in the *transform* function.[1] Although this definition seems to induce both a cycle and non-termination of this program, the fact is that, in a *lazy* setting, the *lazy* evaluation machinery is able to determine, at runtime, the right order to evaluate it.

Bird's work showed the power of circular programming, not only as an optimization technique to eliminate multiple traversals of data, but also as a powerful, elegant and concise technique to express multiple traversal algorithms. Indeed, using this style of programming, it is not necessary to define and schedule the different functions, since a single function has to be defined. Neither it is necessary to define intermediate gluing structures to convey values between traversals because there is a single traversal function only.

Due to their nice properties, circular programs have been used in varied contexts: in the construction of HASKELL compilers [19, 28], aspect-oriented compilers [9], to express pretty printing algorithms [37] and type systems [10], to implement breadth-first traversal strategies [23, 30] and as an optimization technique in the deforestation of accumulating parameters [40]. They have also been studied in the context of partial evaluation [26] and continuations [8]. Circular programs are also strongly related to attribute grammars [32]. Indeed, as [22] and [25] originally showed, circular programs are the natural representation of attribute grammars in a lazy setting.

Deriving circular programs, however, is not the only way to eliminate multiple traversals of data structures. In particular, the original non-circular *repmin* solution may be transformed, by the application of a well-known technique called lambda-abstraction [35], into a higher-order program. As a result, we obtain[2]:

$$
\begin{aligned}
&\mathit{repmin}\ (\mathit{Leaf}\ n) = (\lambda m \rightarrow \mathit{Leaf}\ m, n)\\
&\mathit{repmin}\ (\mathit{Fork}\ l\ r) = (\lambda m \rightarrow \mathit{Fork}\ (t_1\ m)\ (t_2\ m), \mathit{min}\ m_1\ m_2)\\
&\quad \textbf{where}\ (t_1, m_1) = \mathit{repmin}\ l\\
&\qquad\quad (t_2, m_2) = \mathit{repmin}\ r\\
&\\
&\mathit{transform}\ t = f\ m\\
&\quad \textbf{where}\ (f, m) = \mathit{repmin}\ t
\end{aligned}
$$

Regarding this new version of *repmin*, we may notice that it is a higher-order program, since f, the first component of the result produced by the call *repmin* t, is a function. Later,

[1] In order to make it easier for the reader to identify circular definitions, we frame the occurrences of variables that induce them (m in this case).

[2] In the program, we use two anonymous functions that are defined using the symbol λ. Defining $\lambda m \rightarrow \mathit{Leaf}\ m$, for example, is equivalent to defining $g\ m = \mathit{Leaf}\ m$.

f is applied to m, the second component of the result produced by that same call, therefore producing the desired tree result. Thus, this version does not perform multiple traversals. Furthermore, it does not use any intermediate data structure: instead it constructs an intermediate tree of function calls. This higher-order *repmin* solution is extensionally equivalent to both the *repmin* solutions presented so far: the original solution and the circular solution derived from it.

This paper presents calculational techniques to eliminate intermediate data structures that occur in program compositions. In particular, we introduce new program calculation techniques to transform strict multiple traversal programs into circular ones. Our methods are presented in the style of shortcut fusion, under generic calculation rules, that can be instantiated for a wide class of programs. Furthermore, the rules that we present are studied both in the context of pure and monadic/effectful programming. Post-calculation optimizations, that trade circularities for higher-order definitions, are also exploited and presented.

The main contributions of the paper are:

- the definition of generic program calculation rules, in the style of shortcut deforestation, to obtain circular (Sect. 2.3) and higher-order programs (Sect. 5.1);
- the extension of these calculation rules to monadic programming (Sect. 3.3 for circular programs and Sect. 5.2.3 for higher-order ones);
- the formal proof that all the rules are correct (in the corresponding sections);
- a brief study on how we can increase the sharing of computations in the programs we manipulate and, as a consequence, in the programs we calculate from them (Sect. 2.3).

2 Calculation of circular programs

Circular programs provide a very appropriate formalism to model multiple traversal algorithms in the form of elegant and concise single traversal solutions.

However, circular programs are also known to be difficult to write and to understand. Besides, even for advanced functional programmers, it is sometimes difficult to write a circular program that terminates. Bird proposes to derive such programs from their correct and natural strict solution. His approach is an elegant application of the fold-unfold transformation method coupled with tupling and local recursion. Bird's approach, however, has a severe drawback since it preserves partial correctness only. The derived circular programs are not guaranteed to terminate. Furthermore, as an optimization technique, Bird's method focuses on eliminating multiple traversals over the same data structure. Nevertheless, one often encounters, instead of programs that traverse the same data structure twice, programs that construct an intermediate data structure different from the input one. Indeed, programs are often defined as the composition of two functions: the first traverses the input data and produces an intermediate data structure whose type (possibly) differs from the type of the input data, which is traversed by the second function to produce the final results.

Several attempts have successfully been made to combine such compositions of two functions into a single one, eliminating the use of intermediate data structures, usually called *deforestation* [18, 29, 31, 42]. In those situations, circular programs have also been advocated suitable for deforesting intermediate data structures in compositions of two functions with accumulating parameters [40].

On the other hand, when the second traversal requires some additional information computed from the input in order to produce its outcome, none of the methods discussed in the previous paragraph produce satisfactory results. In fact, as a side-effect of eliminating the

intermediate structure, these methods maintain residual traversals of the input structure. This is due to the fact that deforestation methods focus on eliminating the intermediate data structure, without taking into account the computation of the additional information necessary for the second traversal.

Our motivation for the work presented in this section is then to transform programs such as $prog = cons \circ prod$, where $prod :: a \to (t, z)$ and $cons :: (t, z) \to b$, into programs that construct no intermediate data-structure of type t and that traverse the input structure of type a only once. In other words, we want to perform deforestation on those programs and, subsequently, to eliminate the multiple traversals that deforestation introduces. These goals are achieved by transforming *prog* into a circular program. We allow the first traversal, *prod*, to produce completely general intermediate data structures of type t and context informations of type z. The second traversal, *cons* then uses the context information so that, by consuming the data structure of type t, it is able to compute the desired results.

The method we propose is based on a variant of the well-known *fold/build* rule [17, 18]. The standard *fold/build* rule does not apply to the kind of programs we wish to calculate as they need to convey context information computed in one traversal into the following one. The new rule we introduce, called *pfold/buildp*, was designed to support contextual information to be passed between the first and the second traversals and also the use of completely general intermediate data structures. Like *fold/build*, our rule is cheap and practical to implement in a compiler.

The *pfold/buildp* rule states that program compositions such as the one defined in *prog* naturally induce circular programs. These circular programs compute the same results as the original programs, but they do this by performing a single traversal over their input structure. Furthermore, and since a single traversal is performed, the intermediate data structures lose their purpose. In fact, they are deforested by our rule.

In this section, we also present the formal proof that the *pfold/buildp* rule is correct, and that it introduces no *real* circularity, i.e., that the circular programs it derives preserve the same termination properties as the original programs. Recall that Bird's approach preserves partial correctness only: the circular programs it derives are not guaranteed to terminate, even when the original programs do.

We start by showing the intuition of our method to the specific case of *repmin*.

2.1 Calculating the circular *repmin*

Recall the straightforward solution to *repmin* that we presented in the previous section:

$$
\begin{aligned}
&transform :: Tree \to Tree\\
&transform\ t = replace\ (t, tmin\ t)\\
\\
&tmin :: Tree \to Int\\
&tmin\ (Leaf\ n)\ \ = n\\
&tmin\ (Fork\ l\ r) = min\ (tmin\ l)\ (tmin\ r)\\
\\
&replace :: (Tree, Int) \to Tree\\
&replace\ (Leaf\ \_, m)\ \ = Leaf\ m\\
&replace\ (Fork\ l\ r, m) = Fork\ (replace\ (l, m))\ (replace\ (r, m))
\end{aligned}
$$

The calculational method that we propose in this section is, in particular, suitable for calculating a circular program equivalent to *transform*. Our calculational method, however, is used to calculate circular programs from programs that consist of the composition $f \circ g$ of a producer g and a consumer f, where $g :: a \to (t, z)$ and $f :: (t, z) \to b$. So, in order to be

able to apply our method to *transform*, we need to slightly change its form. In *transform*, the consumer (function *replace*) fits the desired structure; however, no explicit producer occurs, since the input tree is copied as an argument to function *replace*. We then define[3]:

$$
\begin{aligned}
&\mathit{transform} :: \mathit{Tree} \to \mathit{Tree}\\
&\mathit{transform}\ t = \mathit{replace}\ (\mathit{tmint}\ t)\\
&\\
&\mathit{tmint} :: \mathit{Tree} \to (\mathit{Tree}, \mathit{Int})\\
&\mathit{tmint}\ (\mathit{Leaf}\ n)\ \ = (\mathit{Leaf}\ n, n)\\
&\mathit{tmint}\ (\mathit{Fork}\ l\ r) = (\mathit{Fork}\ l'\ r', \mathit{min}\ n_1\ n_2)\\
&\qquad \textbf{where}\ (l',\ n_1) = \mathit{tmint}\ l\\
&\qquad\qquad\ \ (r', n_2) = \mathit{tmint}\ r
\end{aligned}
$$

A leaf tree (that is equal to the input one) is now the intermediate data structure that acts with the purpose of gluing the two functions.

Although the original *transform* solution needs to be slightly modified, so that it is possible to apply our method to it, we still use it as a motivational example since it is very intuitive and since *repmin* is, by far, the most well-known example for circular programming. Later in this paper we will present a realistic example (in Sect. 3.2) which shows that, in general, we need intermediate data to hold new information that is computed in one traversal and used in the following one. This fact forces the definition of new data structures in order to glue the different traversals together. Therefore, our method directly applies to such examples.

Now we want to obtain a new version of *transform* that avoids the generation of the intermediate tree produced in the composition of *replace* and *tmint*. The method we propose proceeds in two steps.

First we observe that we can rewrite the definition of *transform* as follows[4]:

$$
\begin{aligned}
\mathit{transform}\ t &= \mathit{replace}\ (\mathit{tmint}\ t)\\
&= \mathit{replace}\ (\pi_1\ (\mathit{tmint}\ t), \pi_2\ (\mathit{tmint}\ t))\\
&= \mathit{replace}' \circ \pi_1 \circ \mathit{tmint}\ \$\ t\\
&\quad\ \textbf{where}\ \mathit{replace}'\ x = \mathit{replace}\ (x, m)\\
&\qquad\qquad\ \ m = \pi_2\ (\mathit{tmint}\ t)\\
&= \pi_1 \circ (\mathit{replace}' \times \mathit{id}) \circ \mathit{tmint}\ \$\ t\\
&\quad\ \textbf{where}\ \mathit{replace}'\ x = \mathit{replace}\ (x, m)\\
&\qquad\qquad\ \ m = \pi_2\ (\mathit{tmint}\ t)
\end{aligned}
$$

where π_1 and π_2 are the projections $\pi_1\ (x, y) = x$ and $\pi_2\ (x, y) = y$, $(f \times g)\ (x, y) = (f\ x, g\ y)$, and *id* the identity function. Therefore, we can redefine *transform* as:

$$
\begin{aligned}
&\mathit{transform}\ t = \mathit{nt}\\
&\quad \textbf{where}\ (\mathit{nt}, \_) = \mathit{repmin}\ t\\
&\qquad\quad \mathit{repmin}\ t = (\mathit{replace}' \times \mathit{id}) \circ \mathit{tmint}\ \$\ t\\
&\qquad\quad \mathit{replace}'\ x = \mathit{replace}\ (x, m)\\
&\qquad\quad m = \pi_2\ (\mathit{tmint}\ t)
\end{aligned}
$$

We can now synthesize a recursive definition for *repmin* using, for example, the fold-unfold method, obtaining:

[3] In function *tmint*, we could just copy the input tree in the output. However, our method relies in the explicit construction of an intermediate data structure, and so in this case we need to define a deep copy of the input tree.

[4] We use the right associative application, $f\ \$\ x = f\ x$, to avoid the use of parenthesis.

$$
\begin{aligned}
&\mathit{transform}\ t = nt\\
&\quad \textbf{where}\ (nt, \_) = \mathit{repmin}\ t\\
&\qquad\quad m = \pi_2\ (\mathit{tmint}\ t)\\
&\qquad\quad \mathit{repmin}\ (\mathit{Leaf}\ n) \ = (\mathit{Leaf}\ m, n)\\
&\qquad\quad \mathit{repmin}\ (\mathit{Fork}\ l\ r) = \textbf{let}\ (l', n_1) \ = \mathit{repmin}\ l\\
&\qquad\qquad\qquad\qquad\qquad\quad (r', n_2) = \mathit{repmin}\ r\\
&\qquad\qquad\qquad\qquad\quad \textbf{in}\ (\mathit{Fork}\ l'\ r', \mathit{min}\ n_1\ n_2)
\end{aligned}
$$

In our method this synthesis will be obtained by the application of a particular shortcut fusion law. The resulting program avoids the generation of the intermediate tree, but maintains the residual computation of the minimum of the input tree, as that value is strictly necessary for computing the final tree. Therefore, this step did eliminate the intermediate tree but introduced multiple traversals over t.

The second step of our method is then the elimination of the multiple traversals. Similar to Bird, we will try to obtain a single traversal function by introducing a circular definition. In order to do so, we first observe that the computation of the minimum is the same in *tmint* and *repmin*, in other words,

$$\pi_2 \circ \mathit{tmint} = \pi_2 \circ \mathit{repmin} \tag{1}$$

This may seem a particular observation for this specific case but it is a property that holds in general for all transformed programs of this kind. In fact, later on we will see that *tmint* and *repmin* are both instances of a single polymorphic function and actually this equality is a consequence of a free theorem [41] about that function. Using this equality we may substitute *tmint* by *repmin* in the new version of *transform*, finally obtaining:

$$
\begin{aligned}
&\mathit{transform}\ t = nt\\
&\quad \textbf{where}\ (nt, \boxed{\mathrm{m}}) = \mathit{repmin}\ t\\
&\qquad\quad \mathit{repmin}\ (\mathit{Leaf}\ n) \ = (\mathit{Leaf}\ \boxed{\mathrm{m}}, n)\\
&\qquad\quad \mathit{repmin}\ (\mathit{Fork}\ l\ r) = \textbf{let}\ (l', n_1) \ = \mathit{repmin}\ l\\
&\qquad\qquad\qquad\qquad\qquad\quad (r', n_2) = \mathit{repmin}\ r\\
&\qquad\qquad\qquad\qquad\quad \textbf{in}\ (\mathit{Fork}\ l'\ r', \mathit{min}\ n_1\ n_2)
\end{aligned}
$$

This new definition not only unifies the computation of the final tree and the minimum in *repmin*, but it also introduces a circularity on m. The introduction of the circularity is a direct consequence of this unification. As expected, the resulting circular program traverses the input tree only once. Furthermore, it does not construct the intermediate leaf-tree, which has been eliminated during the transformation process.

The introduction of the circularity is safe in our context. Unlike Bird, our introduction of the circularity is always made in such a way that it is possible to safely schedule the computations. For instance, in our example, the essential property that makes this possible is the equality (1), which is a consequence of the fact that in both *tmint* and *repmin* the computation of the minimum does not depend on the computation of the corresponding tree. The fact that this property is not specific to this particular example, but is an instance of a general one, is what makes it possible to generalize the application of our method to a wide class of programs.

In this section, we have shown an instance of our method for obtaining a circular lazy program from an initial solution that makes no essential use of laziness. In the next sections we formalize our method using a calculational approach. Furthermore, we present the formal proof that guarantees its correctness.

2.2 Data type theory

Our method for the derivation of circular programs can be actually applied to a wide class of compositions that are expressed in terms of certain program schemes. This allows us to give a generic formulation of the transformation rule in the sense that it is parametric in the structure of the intermediate data type involved in the composition to be transformed.

Throughout this paper we shall assume we are working in the context of a lazy functional language with a *cpo* semantics, in which types are interpreted as pointed cpos (complete partial orders with a least element $\bot$) and functions are interpreted as continuous functions between pointed cpos. However, our semantics differs from that of HASKELL in that we do not consider lifted products and function spaces. In Sect. 4, we analyze the implications that the semantics of HASKELL has on our rule for calculating circular programs. As usual, a function f is said to be *strict* if it preserves the least element, i.e. $f \bot = \bot$.

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

$$map_F\ id = id \tag{2}$$

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

A standard example of a functor is that formed by the list type constructor and the well-known *map* function.

In the assumed semantics pairs are interpreted as the Cartesian product of the corresponding cpos. Associated with the product we can define a split function:

$$\begin{aligned} &(\vartriangle) \quad :: (c \to a) \to (c \to b) \to c \to (a, b) \\ &f \vartriangle g\ c = (f\ c, g\ c) \end{aligned}$$

Among other properties, product operations satisfy the following ones:

$$f \circ \pi_1 = \pi_1 \circ (f \times g) \tag{4}$$

$$g \circ \pi_2 = \pi_2 \circ (f \times g) \tag{5}$$

$$f = (\pi_1 \circ f) \vartriangle (\pi_2 \circ f) \tag{6}$$

Product can be generalized to an arbitrary number of components in the obvious way.

Semantically, recursive datatypes are understood as least fixed points of functors: given the declaration of a datatype τ it is possible to derive a functor F such that the semantic interpretation of the datatype corresponds to the least solution to the equation $x \cong Fx$. We write μF to denote the type corresponding to the least solution; we will use μF and τ interchangeably. 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; see, e.g., [1, 16] for more details.

For example, for the type of leaf trees we can derive a functor T given by:

$$\begin{aligned} &\textbf{data}\ T\ a = FLeaf\ Int \mid FFork\ a\ a \\ &\\ &map_T \qquad\qquad :: (a \to b) \to (T\ a \to T\ b) \\ &map_T\ f\ (FLeaf\ n) \quad = FLeaf\ n \\ &map_T\ f\ (FFork\ a\ a') = FFork\ (f\ a)\ (f\ a') \end{aligned}$$

Then,

$$
\begin{array}{ll}
in_T :: T\ Tree \to Tree & out_T :: Tree \to T\ Tree \\
in_T\ (FLeaf\ n) = Leaf\ n & out_T\ (Leaf\ n) = FLeaf\ n \\
in_T\ (FFork\ l\ r) = Fork\ l\ r & out_T\ (Fork\ l\ r) = FFork\ l\ r
\end{array}
$$

In the case of polymorphic lists, the structure is captured by a bifunctor (a functor on two variables) as it is necessary to reflect the presence of the type parameter:

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

The interpretation of $[a]$ then corresponds to $mu\ (L\ a)$. Thus,

$$
\begin{array}{ll}
in_{L_a} :: L\ a\ [a] \to [a] & out_{L_a} :: [a] \to L\ a\ [a] \\
in_{L_a}\ FNil = [\,] & out_{L_a}\ Nil = FNil \\
in_{L_a}\ (FCons\ a\ as) = a : as & out_{L_a}\ (a : as) = FCons\ a\ as
\end{array}
$$

2.2.1 Fold

Fold [4, 16] is a pattern of recursion that captures function definitions by structural recursion. The best known example of fold is its definition for lists, which corresponds to the *foldr* operator.

Given a functor F and a function $k :: F\ a \to a$, *fold* (also called *catamorphism*), denoted by $fold_F\ k :: \mu F \to a$, is defined as the least function f that satisfies the equation $f \circ in_F = k \circ map_F\ f$. Because out_F is the inverse of in_F, this is the same as:

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

A function $k :: F\ a \to a$ is called an *F-algebra*. The functor F plays the role of the signature of the algebra, as it encodes the information about the operations of the algebra. The type a is called the carrier of the algebra. An *F-homomorphism* between two algebras $k :: F\ a \to a$ and $k' :: F\ b \to b$ is a function $f :: a \to b$ between the carriers that commutes with the operations. This is specified by the condition $f \circ k = k' \circ map_F\ f$. Notice that $fold_F\ k$ is nothing more than a homomorphism between the algebras in_F and k.

When a 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 T is a function $k :: T\ a \to a$ of the form

$$
\begin{array}{l}
k\ (FLeaf\ n) = leaf\ n \\
k\ (FFork\ a\ a') = fork\ a\ a'
\end{array}
$$

with components $leaf :: Int \to a$ and $fork :: a \to a \to a$.

When showing specific instances of fold for concrete datatypes, we will denote the algebras involved by simply writing their tuple of components. For example, fold for leaf trees is given by:

$$
\begin{array}{l}
fold_{Tree} :: (Int \to a, a \to a \to a) \to Tree \to a \\
fold_{Tree}\ (leaf, fork) = f_T \\
\quad \textbf{where}\ f_T\ (Leaf\ n) = leaf\ n \\
\qquad\quad f_T\ (Fork\ l\ r) = fork\ (f_T\ l)\ (f_T\ r)
\end{array}
$$

We can then express, for example, function *tmin* in terms of a fold as follows:

$$tmin = fold_{Tree}\ (id, min)$$

Fold enjoys many algebraic laws that are useful for program transformation [16]. A well-known example is *shortcut fusion* [17, 18, 38] (also known as the *fold/build* rule), which is an instance of a free theorem [41]. The idea of shortcut fusion is that the producer must generate the intermediate data structure using uniquely the constructors of the data-type and not whatever values that are passed to it as arguments. To meet this condition, the producer is required to be expressible in terms of a so-called *build* function, which is a function that carries a *template* that exhibits the occurrences of the constructors of the intermediate data-type.

Law 1 (Fold/build rule) *For k strict*,

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

where

$$build_F :: (\forall\ a\ .\ (F\ a \to a) \to c \to a) \to c \to \mu F$$
$$build_F\ g = g\ in_F$$

This definition of $build_F$ is slightly different from the standard one [18], in the sense that it has an additional parameter of type c. This is because we want to formulate our laws at the functional level. The instance of Law 1 for leaf trees is the following:

$$fold_{Tree}\ (leaf, fork) \circ build_{Tree}\ g = g\ (leaf, fork) \tag{7}$$

where

$$build_{Tree} :: (\forall\ a\ .\ (Int \to a, a \to a \to a) \to c \to a) \to c \to Tree$$
$$build_{Tree}\ g = g\ (Leaf, Fork)$$

Observe that in the instance of the law the strictness condition on the algebra of the fold disappears. The reason is that a function defined by pattern matching is strict; in fact, recall that in the instances of fold (and build) we show only the components of the algebras, but they are actually defined by case analysis on the functor constructors. The same will happen in the instances for other concrete datatypes.

To illustrate the use of this law, consider the following program that computes the minimum value of a tree obtained by selecting internal nodes of the input tree according to the value of a boolean argument.

$$tmp = tmin \circ pick$$

$$pick :: (Tree, Bool) \to Tree$$
$$pick\ (Leaf\ n, b) = Leaf\ n$$
$$pick\ (Fork\ l\ r, b) = \mathbf{if}\ b\ \mathbf{then}\ Fork\ (pick\ (l, False))\ (pick\ (r, False))\ \mathbf{else}\ pick\ (l, True)$$

To apply the law we first need to express *pick* in terms of $build_{Tree}$:

$$pick = build_{Tree}\ g$$
$$\mathbf{where}\ g\ (leaf, fork)\ (Leaf\ n, b) = leaf\ n$$
$$g\ (leaf, fork)\ (Fork\ l\ r, b)$$
$$= \mathbf{if}\ b\ \mathbf{then}\ fork\ (g\ (leaf, fork)\ (l, False))\ (g\ (leaf, fork)\ (r, False))$$
$$\mathbf{else}\ g\ (leaf, fork)\ (l, True)$$

Then, by (7) we have that $tmp = g\ (id, min)$. Inlining,

$$tmp\ (Leaf\ n, b) \ = n$$
$$tmp\ (Fork\ l\ r, b) = \textbf{if}\ b\ \textbf{then}\ min\ (tmp\ (l, False))\ (tmp\ (r, False))\ \textbf{else}\ tmp\ (l, True)$$

As expected, the resulting function does not construct the intermediate tree.

Due to space limitation we focus in this paper on the laws that dictate the transformations and we leave out pragmatical aspects, such as the mechanisms of how the laws are actually applied in a real compiler, performance analysis, or the methods that make it possible to automatically rewrite the involved functions in terms of *fold* and *build*. For further details on these issues, the interested reader may consult [12].

In the same line of reasoning, we can state another fusion law for a slightly different producer function.

Law 2 (Fold/buildp rule) *For k strict*,

$$(fold_F\ k \times id) \circ buildp_F\ g = g\ k$$

where

$$buildp_F :: (\forall\ a\ .\ (F\ a \to a) \to c \to (a, z)) \to c \to (\mu F, z)$$
$$buildp_F\ g = g\ in_F$$

Proof From the polymorphic type of g we can deduce the following free theorem: for f strict,

$$f \circ \phi = \psi \circ map_F\ f \ \Rightarrow\ (f \times id) \circ g\ \phi = g\ \psi$$

By taking $f = fold_F\ k$, $\phi = in_F$, $\psi = k$ we obtain that $(fold_F\ k \times id) \circ g\ in_F = g\ k$, since the equation on the left-hand side of the implication becomes true by definition of fold. The requirement that f is strict is satisfied by the fact that every fold with a strict algebra is strict, and by hypothesis k is strict. Finally, by definition of $buildp_F$ the desired result follows. □

Finally, the following property is an immediate consequence of Law 2.

Law 3 *For any strict k and* $g :: \forall\ a\ .\ (F\ a \to a) \to c \to (a, z)$,

$$\pi_2 \circ g\ in_F = \pi_2 \circ g\ k$$

Proof

$$\pi_2 \circ g\ in_F = \{\ (5)\ \} = \pi_2 \circ (fold_F\ k \times id) \circ g\ in_F = \{\ \text{Law 2}\ \} = \pi_2 \circ g\ k$$ □

This property states that the construction of the second component of the pair returned by g is independent of the particular algebra that g carries; it only depends on the input value of type c. This is a consequence of the polymorphic type of g and the fact that the second component of its result is of a fixed type z.

2.2.2 *Fold with parameters*

Some recursive functions use context information in the form of constant parameters for their computation. The aim of this section is to present a program scheme for the definition of structurally recursive functions of the form $f :: (\mu F, z) \to a$, where the type z represents the context information. Our interest in these functions is based on the fact that our method will assume that consumers are functions of this kind.

Functions of this form can be defined in different ways. One alternative consists of fixing the value of the parameter and performing recursion on the other, which permits to write the function as a fold, $f\ (t, z) = fold_F\ k\ t$, such that the context information contained in z may eventually be used in the algebra k. This is the case of, for example, the function *replace*:

$$replace\ (t, m) = fold_{Tree}\ (\lambda n \to Leaf\ m, Fork)\ t$$

Another alternative is the use of currying, defining a function of type $\mu F \to (z \to a)$ as a higher-order fold. For instance, in the case of *replace* it holds that:

$$curry\ replace = fold_{Tree}\ (\lambda n \to Leaf, \lambda f\ f' \to \lambda z \to Fork\ (f\ z)\ (f'\ z))$$

This is an alternative we will study in detail in Sect. 5.1.

A third alternative is to define the function $f :: (\mu F, z) \to a$ in terms of a program scheme, called *pfold* [33, 34], which, unlike fold, is able to manipulate constant and recursive arguments simultaneously. The definition of pfold relies on the concept of *strength* of a functor F, which is a polymorphic function $st_F :: (F\ a, z) \to F\ (a, z)$ that satisfies certain coherence axioms (see [6, 33] for further details). The strength distributes the value of type z to the variable positions (positions of type a) of the functor. For instance, the strength $st_T :: (T\ a, z) \to T\ (a, z)$ corresponding to functor T is given by:

$$st_T\ (FLeaf\ n, z) = FLeaf\ n \qquad st_T\ (FFork\ a\ a', z) = FFork\ (a, z)\ (a', z)$$

In the definition of pfold the strength of the underlying functor plays an important role as it represents the distribution of the context information contained in the constant parameters to the recursive calls.

Given a functor F and a function $h :: (F\ a, z) \to a$, *pfold*, denoted by $pfold_F\ h :: (\mu F, z) \to a$, is defined as the least function f that satisfies the following equation:

$$f \circ (in_F \times id) = h \circ ((map_F\ f \circ st_F) \vartriangle \pi_2)$$

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

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

then $h :: (F\ a, z) \to a$ has also n components $(h_1, \ldots, h_n)$, where now each has type $h_i :: t_{i,1} \to \cdots \to t_{i,r_i} \to z \to a$. For example, for functor T, $h :: (T\ a, z) \to a$ is of the form: $h\ (FLeaf\ n, z) = hleaf\ n\ z$ and $h\ (FFork\ a\ a', z) = hfork\ a\ a'\ z$ where $hleaf :: z \to a$ and $hfork :: a \to a \to z \to a$ are the component functions. As with algebras, in the specific instances of pfold we will write the tuple of components instead of h.

For example, in the case of leaf trees the definition of pfold is as follows:

$$\begin{aligned}
&pfold_{Tree} :: (Int \to z \to a, a \to a \to z \to a) \to (Tree, z) \to a\\
&pfold_{Tree}\ (hleaf, hfork) = p_T\\
&\quad \mathbf{where}\ p_T\ (Leaf\ n, z)\ \ = hleaf\ n\ z\\
&\qquad\qquad p_T\ (Fork\ l\ r, z) = hfork\ (p_T\ (l, z))\ (p_T\ (r, z))\ z
\end{aligned}$$

We can then write *replace* in terms of a pfold:

$$replace = pfold_{Tree}\ (\lambda n\ m \to Leaf\ m, \lambda l\ r\ m \to Fork\ l\ r)$$

The following equation shows one of the possible relationships between pfold and fold. For h with components $(h_1, \ldots, h_n)$,

$$pfold_F\ h\ (t, z) = fold_F\ k\ t\ \mathbf{where}\ k_i\ \bar{x} = h_i\ \bar{x}\ z \tag{8}$$

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

Like fold, pfold satisfies a set of algebraic laws. We do not show any of them here as they are not necessary for the calculational work presented in this paper. The interested reader may consult [33, 34].

2.3 The pfold/buildp rule

In this section, we present a generic formulation and proof of correctness of a transformation rule for calculating circular programs. The rule takes a composition $cons \circ prod$ of a producer $prod :: a \to (t, z)$ followed by a consumer $cons :: (t, z) \to b$, and returns an equivalent deforested circular program that performs a single traversal over the input value. The reduction of this expression into an equivalent one without intermediate data structures is performed in two steps. Firstly, we apply standard deforestation techniques in order to eliminate the intermediate data structure of type t. The program obtained is deforested, but in general contains multiple traversals over the input as a consequence of residual computations of the other intermediate values (e.g. the computation of the minimum in the case of *repmin*). Therefore, as a second step, we perform the elimination of the multiple traversals by the introduction of a circular definition.

The rule makes some natural assumptions about *cons* and *prod*: t is a recursive data type μF, the consumer *cons* is defined by structural recursion on t, i.e. as a pfold, and the intermediate value of type z is taken as a constant parameter by *cons*. In addition, it is required that *prod* is a "good producer", in the sense that it is possible to express it in terms of the *buildp* function corresponding to the intermediate type t. In summary, the rule is applied to compositions of the form: $pfold\ h \circ buildp\ g$.

Law 4 (Pfold/buildp rule) *For h with components $(h_1, \ldots, h_n)$,*

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

Proof The proof will show in detail the two steps of our method. The first step corresponds to the application of deforestation, which is represented by Law 2. For that we need first to express the pfold as a fold.

$$\begin{aligned} &pfold_F\ h\ (buildp_F\ g\ c) \\ =\quad &\{\ \text{definition of } buildp \text{ and } (6)\ \} \\ &pfold_F\ h \circ ((\pi_1 \circ g\ in_F) \vartriangle (\pi_2 \circ g\ in_F))\ \$\ c \\ =\quad &\{\ (8)\ \} \\ &fold_F\ k \circ \pi_1 \circ g\ in_F\ \$\ c \\ &\quad \textbf{where}\ z = \pi_2 \circ g\ in_F\ \$\ c \\ &\qquad\ k_i\ \bar{x} = h_i\ \bar{x}\ z \\ =\quad &\{\ (4)\ \} \\ &\pi_1 \circ (fold_F\ k \times id) \circ g\ in_F\ \$\ c \\ &\quad \textbf{where}\ z = \pi_2 \circ g\ in_F\ \$\ c \\ &\qquad\ k_i\ \bar{x} = h_i\ \bar{x}\ z \\ =\quad &\{\ \text{Law } 2\ \} \\ &\pi_1 \circ g\ k\ \$\ c \\ &\quad \textbf{where}\ z = \pi_2 \circ g\ in_F\ \$\ c \\ &\qquad\ k_i\ \bar{x} = h_i\ \bar{x}\ z \end{aligned}$$

Law 2 was applicable because by construction the algebra k is strict.

Once we have reached this point we observe that the resulting program is deforested, but it contains two traversals on c. The elimination of the multiple traversals is then performed by introducing a circular definition. The essential property that guarantees the safe introduction of the circularity is Law 3, which states that the computation of the second component of type z is independent of the particular algebra that is passed to g. This is a consequence of the polymorphic type of g. Therefore, we can replace in_F by another algebra and we will continue producing the same value z. In particular, we can take k as this other algebra, and in that way we are introducing the circularity. It is this property that ensures that no terminating program is turned into a nonterminating one.

$$
\begin{aligned}
&\pi_1 \circ g\ k\ \$\ c \\
&\quad \textbf{where}\ z = \pi_2 \circ g\ in_F\ \$\ c \\
&\qquad\quad k_i\ \bar{x} = h_i\ \bar{x}\ z \\
=\quad &\{\ \text{Law 3}\ \} \\
&\pi_1 \circ g\ k\ \$\ c \\
&\quad \textbf{where}\ \boxed{z} = \pi_2 \circ g\ k\ \$\ c \\
&\qquad\quad k_i\ \bar{x} = h_i\ \bar{x}\ \boxed{z} \\
=\quad &\{\ (6)\ \} \\
&v\ \textbf{where}\ (v, \boxed{z}) = g\ k\ c \\
&\qquad\quad k_i\ \bar{x} = h_i\ \bar{x}\ \boxed{z}
\end{aligned}
$$

□

Now, let us see the application of the pfold/buildp rule in the case of the *repmin* problem. Recall the definition, presented on p. 120, that we want to transform:

$$
\begin{aligned}
&transform :: Tree \to Tree \\
&transform\ t = replace\ (tmint\ t)
\end{aligned}
$$

To apply the rule, we have to express *replace* and *tmint* in terms of pfold and buildp, respectively:

$$
replace = pfold_{Tree}\ (\lambda n\ m \to Leaf\ m, \lambda l\ r\ m \to Fork\ l\ r)
$$

$$
\begin{aligned}
&buildp_{Tree} :: (\forall\ a\ .\ (Int \to a, a \to a \to a) \to c \to (a, z)) \to c \to (Tree, z) \\
&buildp_{Tree}\ g = g\ (Leaf, Fork)
\end{aligned}
$$

$$
\begin{aligned}
&tmint = buildp_{Tree}\ gtm \\
&\quad \textbf{where}\ gtm\ (leaf, fork)\ (Leaf\ n) \ = (leaf\ n, n) \\
&\qquad\quad gtm\ (leaf, fork)\ (Fork\ l\ r) = \textbf{let}\ (l', n_1) = gtm\ (leaf, fork)\ l \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad (r', n_2) = gtm\ (leaf, fork)\ r \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ (fork\ l'\ r', min\ n_1\ n_2)
\end{aligned}
$$

By applying Law 4 we get:

$$
\begin{aligned}
&transform\ t = nt\ \textbf{where}\ (nt, \boxed{\mathrm{m}}) = gtm\ (kleaf, kfork)\ t \\
&\qquad\qquad\qquad\qquad\quad kleaf\ \_ \ = Leaf\ \boxed{\mathrm{m}} \\
&\qquad\qquad\qquad\qquad\quad kfork\ l\ r = Fork\ l\ r
\end{aligned}
$$

Inlining the above definition, we obtain the one we showed previously in Sect. 2.1:

$$
\begin{aligned}
&transform\ t = nt \\
&\quad \textbf{where}\ (nt, \boxed{\mathrm{m}}) = repmin\ t \\
&\qquad\quad repmin\ (Leaf\ n) \ = (Leaf\ \boxed{\mathrm{m}}, n)
\end{aligned}
$$

$$\begin{aligned} repmin\ (Fork\ l\ r) = \textbf{let}\ (l', n_1) &= repmin\ l \\ (r', n_2) &= repmin\ r \\ \textbf{in}\ (Fork\ l'\ r', min\ n_1\ n_2) \end{aligned}$$

The context parameter of type z produced in compositions $pfold\ h \circ buildp\ g$ is worth a final remark. Indeed, depending on where the computations on that parameter are located, as part of the *pfold* function or in the *buildp*, the performance of these functions may be affected. This also occurs in the circular and in the higher-order programs that we derive from such compositions. If computations on the context parameter are placed in the pfold, then the parameter may need to be recomputed several times as a result of a loss of sharing. A similar observation was made by Voigtländer [39].

In order to help us in restoring the sharing of computations, we will introduce calculational rules for pfold and buildp that assist us in moving computations on the context parameter. We show the propagation of the inefficiency in the case of a circular program derivation, but the same analysis can be done both for the corresponding higher-order programs that can be derived from the same compositions (Sect. 5.1), and for the case of monadic programs.

As an example, let us consider a slightly different formulation of *repmin*.

$$transform' = mapMin \circ copyAndLeaves$$

$$\begin{aligned} &mapMin :: (Tree, [Int]) \to Tree \\ &mapMin\ (Leaf\ \_, ns) = Leaf\ (minimum\ ns) \\ &mapMin\ (Fork\ l\ r, ns) = Fork\ (mapMin\ (l, ns))\ (mapMin\ (r, ns)) \end{aligned}$$

$$\begin{aligned} &copyAndLeaves :: Tree \to (Tree, [Int]) \\ &copyAndLeaves\ (Leaf\ n) = (Leaf\ n, [n]) \\ &copyAndLeaves\ (Fork\ l\ r) = (Fork\ l'\ r', ns_1 +\!\!+ ns_2) \\ &\quad \textbf{where}\ (l', ns_1) = copyAndLeaves\ l \\ &\qquad\quad (r', ns_2) = copyAndLeaves\ r \end{aligned}$$

This definition differs from the one shown earlier in that the minimum is now computed as part of the pfold (and not within the buildp as before) and this is done each time it is going to be placed in a leaf. This means that the minimum is recomputed as many times as the number of leaves in the tree. Another difference is that now a list with the values contained in the leaves of the input tree is generated as a context parameter of the composition.

As before, we can derive a circular program from $transform'$. For that we need to express *mapMin* and *copyAndLeaves* in terms of pfold and buildp, respectively:

$$\begin{aligned} &mapMin = pfold_{Tree}\ (hleafmm, hforkmm) \\ &\quad \textbf{where}\ hleafmm\ \_ns = Leaf\ (minimum\ ns) \\ &\qquad\quad hforkmm\ l\ r\ ns = Fork\ l\ r \end{aligned}$$

$$\begin{aligned} &copyAndLeaves = buildp_{Tree}\ gcal \\ &\quad \textbf{where}\ gcal\ (leaf, fork)\ (Leaf\ n) = (leaf\ n, [n]) \\ &\qquad\quad gcal\ (leaf, fork)\ (Fork\ l\ r) = \textbf{let}\ (l', ns_1) = gcal\ (leaf, fork)\ l \\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad (r', ns_2) = gcal\ (leaf, fork)\ r \\ &\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ (fork\ l'\ r', ns_1 +\!\!+ ns_2) \end{aligned}$$

By applying Law 4, we get:

$$\begin{aligned} &tranform'\ t = nt \\ &\quad \textbf{where}\ (nt, \boxed{ns}) = repmin'\ t \\ &\qquad\quad repmin'\ (Leaf\ n) = (Leaf\ (minimum\ \boxed{ns}), [n]) \\ &\qquad\quad repmin'\ (Fork\ l\ r) = \textbf{let}\ (l', ns_1) = repmin'\ l \end{aligned}$$

$$(r', ns_2) = repmin'\ r$$
$$\textbf{in}\ (Fork\ l'\ r', ns_1 +\!\!+ ns_2)$$

Now the circular argument is the list of leaves *ns*. Observe that the recomputation of the minimum is maintained in the circular program; it could be isolated by introducing a local argument $ms = minimum\ ns$, but this would not prevent the generation of the intermediate list *ns*.

It is possible to transform the definition of *transform*′ to the original definition of *transform* where the computation of the minimum is performed as part of the producer *tmint* and therefore is shared by all leaves of the output tree. The transformation is based on the following properties, which show how to move computations on the context parameter: Let $f :: z \to z'$,

$$pfold_F\ h \circ (id \times f) = pfold_F\ (h \circ (id \times f)) \tag{9}$$

$$(id \times f) \circ buildp_F\ g = buildp_F\ ((id \times f)\ \bullet\ g) \tag{10}$$

where $(f\ \bullet\ g)\ x = f \circ g\ x$. Read from right to left, (9) [33] states that computations on the context parameter can be moved outside a pfold. On the contrary, (10) states how to move computations on the context parameter inside a buildp.

Returning to our example, we can calculate the following:

$$\begin{aligned}
&mapMin \circ copyAndLeaves \\
=\quad &\{\ (9)\ \} \\
&replace \circ (id \times minimum) \circ copyAndLeaves \\
=\quad &\{\ (10)\ \} \\
&replace \circ buildp_{Tree}\ ((id \times minimum)\ \bullet\ gcal) \\
=\quad &\{\ \text{fusion}\ \} \\
&replace \circ tmint
\end{aligned}$$

which proves the equivalence between our two definitions of *repmin* (and, by transitivity, the equivalence of the circular programs). The last step of the calculation means to fuse $(id \times minimum)$ with *gcal* obtaining as result the definition of *gtm*; for this we can apply standard fold fusion [4] as in fact *gcal* is a fold on trees. To apply fusion this property is required: $minimum\ (xs +\!\!+ ys) = min\ (minimum\ xs)\ (minimum\ ys)$.

In the next section we study the calculation of circular programs, as with Law 4, but in the context of monadic programs.

3 Calculation of monadic circular programs

In the previous section, we have shown how circular programs can be used to achieve intermediate data structure deforestation in compositions $cons \circ prod$, where $prod :: a \to (t, z)$ and $cons :: (t, z) \to b$. The rule we introduced to this end is generic in the sense that it can be applied to a wide range of programs and datatypes. However, it does not handle programs with effects, that is, programs that, for example, rely on a global state or perform I/O operations. Thus, the rule has a limited applicability scope since several programs, like compilers, pretty-printers or parsers do rely on global effects.

Our motivation for the work presented in this section is to extend the derivation of circular programs to the context of monadic programming. Our approach follows the studies

by Ghani and Johann [15], Manzino and Pardo [27] that propose monadic extensions to standard shortcut fusion. Our goal is to achieve fusion of monadic programs, maintaining the global effects. We study two cases: the case where the producer function is monadic and the consumer is given by a pure function, and the case where both functions are monadic. For both cases, fusion is achieved by transforming the original program into a circular one. We do not consider the case where the producer is given by a pure function (and the consumer is given by a monadic one) since it can already be fused using the pfold/buildp rule presented before.

3.1 Bit string transformation

To illustrate our technique to derive monadic circular programs we first consider an example based on a simple bit string conversion that has applications in cryptography [2]. Suppose we want to transform a sequence of bits into a new one, of the same length, by applying the *exclusive or* between each bit and the binary sum (sum modulo 2) of the sequence. We consider that the input sequence is given as a string of bits, which will be parsed into a list and then transformed. It is in the parsing phase that computational effects come into play, as we use a monadic parser.

Suppose we are given the string `"101110110001"`. To transform this string of bits, we start by parsing it, computing as result a list of bits $[1,0,1,1,1,0,1,1,0,0,0,1]$, and its binary sum (1 in this case). Having the list and the binary sum, the original sequence is transformed into this one $[0,1,0,0,0,1,0,0,1,1,1,0]$ after applying the exclusive or of each bit with 1 (the binary sum).

The parser for bit strings will be constructed as a monadic parser, which is a function that relies on a monad. A monad is usually presented as a triple $(M, return, \gg\!=)$ consisting of a type constructor M and two polymorphic functions, $return :: a \to M\ a$ and $(\gg\!=) :: M\ a \to (a \to M\ b) \to M\ b$, that obey the following laws:

$$m \gg\!= return = m \qquad return\ x \gg\!= f = f\ x \qquad (m \gg\!= f) \gg\!= g = m \gg\!= (\lambda x \to f\ x \gg\!= g)$$

In HASKELL, we can capture the definition of a monad as a type class.

$$\begin{aligned}&\textbf{class}\ Monad\ m\ \textbf{where}\\ &\quad return :: a \to m\ a\\ &\quad (\gg\!=) :: m\ a \to (a \to m\ b) \to m\ b\end{aligned}$$

With the aim at improving readability of monadic programs, HASKELL provides a special syntax, called *do notation*, which is defined by the following translation rules:

$$\textbf{do}\ \{x \leftarrow m; m'\} = m \gg\!= \lambda x \to \textbf{do}\ \{m'\} \qquad\qquad \textbf{do}\ \{m\} = m$$

Associated with every monad it is possible to define a map function, which together with the type constructor m satisfies the conditions to be a functor.

$$\begin{aligned}&mmap \quad :: Monad\ m \Rightarrow (a \to b) \to (m\ a \to m\ b)\\ &mmap\ f\ m = \textbf{do}\ \{a \leftarrow m; return\ (f\ a)\}\end{aligned}$$

To implement the parser we will adopt the usual definition of parser monad (see [20] for more details):

$$\begin{aligned}&\textbf{newtype}\ Parser\ a = P\ (String \to [(a, String)])\\ &\\ &\textbf{instance}\ Monad\ Parser\ \textbf{where}\\ &\quad return\ a\ = P\ (\lambda cs \to [(a, cs)])\\ &\quad p \gg\!= f = P\ (\lambda cs \to concat\ [parse\ (f\ a)\ cs' \mid (a, cs') \leftarrow parse\ p\ cs])\\ &\\ &parse :: Parser\ a \to String \to [(a, String)]\end{aligned}$$

```
parse (P p) = p

(⟨|⟩) :: Parser a → Parser a → Parser a
(P p) ⟨|⟩ (P q) = P (λcs → case p cs ++ q cs of [ ]      → [ ]
                                                (x : xs) → [x])

pzero :: Parser a
pzero = P (λcs → [ ])

item :: Parser Char
item = P (λcs → case cs of [ ]      → [ ]
                           (c : cs) → [(c, cs)])
```

Alternatives are represented by a deterministic choice operator ($\langle|\rangle$), which returns at most one result. The parser *pzero* is a parser that always fails. The *item* parser returns the first character in the input string.

We can use these simple parser combinators to define parsers for bits and bit strings. The binary sum is calculated as the exclusive or of the bits of the parsed sequence. We write $\oplus$ to denote exclusive or over the type *Bit*.

```
data Bit = Zero | One

bit :: Parser Bit
bit = do c ← item
         case c of '0' → return Zero
                   '1' → return One
                   _   → pzero

bitstring :: Parser ([Bit], Bit)
bitstring = (do b       ← bit
                (bs, s) ← bitstring
                return (b : bs, b ⊕ s)) ⟨|⟩ return ([ ], Zero)
```

Now, we implement the transformation function:

```
transform :: ([Bit], Bit) → [Bit]
transform ([ ], _)     = [ ]
transform (b : bs, s) = (b ⊕ s) : transform (bs, s)
```

In summary, our bit string transformer consists of:

```
shift :: Parser [Bit]
shift = do {(bs, s) ← bitstring; return (transform (bs, s))}
```

Regarding the above solution, we notice that function *bitstring* constructs an intermediate list of bits, that is later consumed by function *transform* in order to produce the desired result. The construction of this list may result in inefficiency of the program and therefore we would like to eliminate it. Following a similar strategy to the one used for purely functional programs, such elimination is achieved by applying a shortcut fusion law to *shift*, with the difference that now the fusion law deals with monadic functions. Below, we present the specific instance of the fusion law for lists (the type of the intermediate data structure in the example), and in Sect. 3.3 we show that it is an instance of a generic law that can be formulated for several datatypes.

Like in standard shortcut fusion, our law assumes that the producer and the consumer (*bitstring* and *transform* in our case) are expressed in terms of certain program schemes. The consumer must be given by structural recursion in terms of a *pfold*,

```
pfold_List :: (z → b, a → b → z → b) → ([a], z) → b
pfold_List (hnil, hcons) = p_L
```

$$\begin{aligned}&\textbf{where } p_L\ ([\,], z) &&= hnil\ z\\ &\qquad p_L\ (a : as, z) &&= hcons\ a\ (p_L\ (as, z))\ z\end{aligned}$$

and the producer as a *buildp* function. The difference with the purely functional case is that now we consider producers that generate the intermediate values as the result of a monadic computation. For lists, this is expressed by a function called $mbuildp_{List}$:

$$\begin{aligned}&mbuildp_{List} &&:: Monad\ m \Rightarrow (\forall\, b\ .\ (b, a \to b \to b) \to m\ (b, z)) \to m\ ([a], z)\\ &mbuildp_{List}\ g &&= g\ ([\,], (:))\end{aligned}$$

Having stated the forms required to the producer and the consumer it is now possible to formulate the law and to use it to calculate a circular program equivalent to *shift*.

Law 5 (Pfold/mbuildp for lists) *Let m be a recursive monad.*

$$\begin{aligned}&\textbf{do } \{(xs, z) \leftarrow mbuildp_{List}\ g;\ return\ (pfold_{List}\ (hnil, hcons)\ (xs, z))\}\\ &=\\ &\quad \textbf{do rec } (v, \boxed{z}) \leftarrow \textbf{let } knil &&= hnil\ \boxed{z}\\ &\qquad\qquad\qquad\qquad\qquad\quad kcons\ (x, y) &&= hcons\ ((x, y), \boxed{z})\\ &\qquad\qquad\qquad\quad \textbf{in } g\ (knil, kcons)\\ &\qquad return\ v\end{aligned}$$

This law transforms a monadic composition, where the producer is an effectful function but the consumer may not necessarily be, into a single monadic function with a circular argument z. Indeed, z is a value computed by $g\ (knil, kcons)$ but in turn used by *knil* and *kcons*. An interesting feature of this law is the fact that the introduction of the circularity needs the use of a recursive binding within a monadic computation, and therefore requires the monad to be *recursive* [11]. A recursive **do** is supported by HASKELL for those monads that are declared an instance of the *MonadFix* class.

$$\begin{aligned}&\textbf{class } Monad\ m \Rightarrow MonadFix\ m \textbf{ where}\\ &\quad mfix :: (a \to m\ a) \to m\ a\end{aligned}$$

The monadic fixed-point operator *mfix* needs to obey the following semantic laws[5]:

$$\begin{aligned}f\ \bot = \bot \Leftrightarrow mfix\ f = \bot\\ mfix\ (return \circ h)\ = return\ (fix\ h)\\ mfix\ (\lambda x \to q \gg\!= \lambda y \to f\ x\ y) = q \gg\!= (\lambda y \to mfix\ (\lambda x \to f\ x\ y)),\\ \text{if } x \text{ does not appear free in } q\end{aligned} \tag{11}$$

The parsing monad presented earlier can be declared an instance of the *MonadFix* class, for example, as follows:

$$\begin{aligned}&\textbf{instance } MonadFix\ Parser \textbf{ where}\\ &\quad mfix\ f = P\ (\lambda cs \to mfix_L\ (\tilde{\lambda}(x, y) \to parse\ (f\ x)\ cs))\\ &\qquad \textbf{where } mfix_L\ f = \textbf{case } fix\ (f \circ head) \textbf{ of } [\,] &&\to [\,]\\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad (x : \_) &&\to x : mfix_L\ (tail \circ f)\end{aligned}$$

To see Law 5 in action, we write *transform* and *bitstring* in terms of $pfold_{List}$ and $mbuildp_{List}$, respectively:

[5] *fix* is the usual fixed-point operator for pure functions $fix :: (a \to a) \to a, fix\ f = f\ (fix\ f)$.

$$
\begin{array}{l}
transform = pfold_{List}\ (hnil, hcons) \\
\quad \textbf{where}\ hnil\ \_ \qquad = [\,] \\
\qquad\qquad hcons\ b\ r\ s = (b \oplus s) : r
\end{array}
$$

$$
\begin{array}{l}
bitstring = mbuildp_{List}\ g \\
\quad \textbf{where}\ g\ (nil, cons) = (\textbf{do}\ b \qquad\quad \leftarrow bit \\
\qquad\qquad\qquad\qquad\qquad\quad (bs, s) \leftarrow g\ (nil, cons) \\
\qquad\qquad\qquad\qquad\qquad\quad return\ (cons\ b\ bs, b \oplus s))\ \langle|\rangle\ return\ (nil, Zero)
\end{array}
$$

Then, by applying the law we obtain:

$$shift = \textbf{do}\ \{\textbf{rec}\ (bs, \boxed{s}) \leftarrow g\ ([\,], \lambda b\ r \rightarrow (b \oplus \boxed{s}) : r);\ return\ bs\}$$

Inlining, we get the following circular monadic program:

$$
\begin{array}{l}
shift = \textbf{do}\ \textbf{rec}\ (bs, \boxed{s}) \leftarrow \textbf{let}\ gk = (\textbf{do}\ b \leftarrow bit \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (bs', s') \leftarrow gk \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad return\ ((b \oplus \boxed{s}) : bs', b \oplus s')\ \langle|\rangle\ return\ ([\,], Zero) \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ gk \\
\qquad\qquad return\ bs
\end{array}
$$

The above program avoids the construction of the intermediate list of bits, by introducing a circular definition. Indeed, we may notice that s (the modulo 2 sum of the input sequence of bits) is used (in $b \oplus s$) in the function call to gk. However, s is also a result of that same call, hence the circularity.

In this section, we have calculated a circular program from a composition of an effectful producer and a consumer given by a pure function. In the next section, we study the fusion of monadic programs where both the consumer and the producer functions are effectful.

3.2 Algol 68 scope rules

In this section, we consider the application of our calculational methods to a real example: the Algol 68 scope rules that are used, for example, in the Eli system [24] to define a generic component for the name analysis task of a compiler.

We wish to construct a program to deal with the scope rules of a block structured language, the Algol 68. In this language a definition of an identifier x is visible in the smallest enclosing block, with the exception of local blocks that also contain a definition of x. In this case, the definition of x in the local scope hides the definition in the global one. In a block an identifier may be declared at most once. We shall analyze these rules via the *Block* language, which consists of programs of the following form:

$$
\begin{array}{l}
[\,\textbf{use}\ y;\ \textbf{decl}\ x; \\
\quad [\,\textbf{decl}\ y;\ \textbf{use}\ y;\ \textbf{use}\ w;\ ] \\
\quad \textbf{decl}\ x;\ \textbf{decl}\ y;\ ]
\end{array}
$$

In HASKELL we may define the following data-types to represent *Block* programs.

$$\textbf{type}\ Prog = [It] \qquad \textbf{data}\ It = Use\ Var \mid Decl\ Var \mid Block\ Prog \qquad \textbf{type}\ Var = String$$

Such programs describe the basic block-structure found in many languages, with the peculiarity however that declarations of identifiers may also occur after their first use (but in the same level or in an outer one).

According to the rules of the language the above program contains two errors: at the outer level, variable x has been declared twice and the use of w, at the inner level, has no binding occurrence at all. Our goal is to process this kind of programs and compute a list containing the identifiers that do not obey the rules of the language. In order to make it easier to detect them, we

require that the list of errors follows the sequential structure of the input program. Thus, for the example above we would compute the list $[w, x]$.

We also want to show messages describing the encountered errors. So, for the example sentence provided, the following messages must be displayed:

Duplicate: decl x
Missing: decl w

The error messages must be displayed in a certain order: errors resulting from duplicate declarations first and then errors that result from missing declarations. Furthermore, the errors occurring in nested blocks are displayed only after the errors occurring in the outer ones.

Because we allow a *use-before-declare* discipline, a conventional implementation of the required analysis leads to a program which traverses the abstract syntax tree twice: one for accumulating the declarations of identifiers, i.e., the environment, and another one for checking the uses of identifiers, according to the environment. The uniqueness of names can be detected in the first traversal, while errors resulting from missing declarations can only be computed in the second one. In such an implementation, a "*gluing*" data structure has to be constructed: to pass the detected errors from the first to the second traversal in order to compute the list of errors in the desired order; and to pass between the two traversals of a block the names of the variables declared in it, i.e., its environment **type** $Env = [(Var, Int)]$, in order to compute the missing declarations.

Observe that the environment computed for a block is the global environment for its nested blocks. Thus, only during the second traversal of a block (i.e., after collecting all its declarations) the program actually begins the traversals of its nested blocks; as a consequence the computations related to the first and second traversals are intermingled. Furthermore, the information on the nested blocks (the instructions they define and the blocks' level) has to be explicitly passed from the first to the second traversal. This is also achieved constructing an intermediate structure. In order to pass the necessary information around, we define:

$$\textbf{type}\ Prog_2 = [It_2] \qquad \textbf{data}\ It_2 = Block_2\ Int\ Prog \mid Dupl_2\ Var \mid Use_2\ Var$$

Errors resulting from duplicate declarations are passed from the first to the second traversal using constructor $Dupl_2$. The level of a nested block and the instructions defined in it are passed to the second traversal using constructor $Block_2$.

According to the defined strategy, our semantic analysis consists of:

$$\begin{aligned}
&semantics :: Prog \to IO\ [Var]\\
&semantics\ p = \textbf{do}\ \{(p', env) \leftarrow duplicate\ 0\ [\,]\ p;\ missing\ (p', env)\}
\end{aligned}$$

The function *duplicate* detects duplicate variable declarations by collecting all the declarations occurring in a block. It is a monadic function since it needs to output error messages[6]:

$$\begin{aligned}
&duplicate :: Int \to Env \to Prog \to IO\ (Prog_2, Env)\\
&duplicate\ lev\ ds\ [\,] = return\ ([\,], ds)\\
&\\
&duplicate\ lev\ ds\ (Use\ var : its) = \textbf{do}\ (its_2, ds') \leftarrow duplicate\ lev\ ds\ its\\
&\qquad\qquad\qquad\qquad\qquad\qquad return\ (Use_2\ var : its_2, ds')\\
&duplicate\ lev\ ds\ (Decl\ var : its)\\
&\quad = \textbf{if}\ ((var, lev) \in ds)\ \textbf{then do}\ put\ (\texttt{"Duplicate: decl "} +\!\!+ var)\\
&\qquad\qquad\qquad\qquad\qquad\qquad (its_2, ds') \leftarrow duplicate\ lev\ ((var, lev) : ds)\ its\\
&\qquad\qquad\qquad\qquad\qquad\qquad return\ (Dupl_2\ var : its_2, ds')\\
&\qquad\qquad\qquad\qquad\textbf{else}\ duplicate\ lev\ ((var, lev) : ds)\ its\\
&\\
&duplicate\ lev\ ds\ (Block\ nested : its) = \textbf{do}\ (its_2, ds') \leftarrow duplicate\ lev\ ds\ its\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad return\ ((Block_2\ (lev + 1)\ nested) : its_2, ds')
\end{aligned}$$

[6] We abbreviate *putStrLn* as *put*.

Function *duplicate* computes a data structure that is later traversed in order to detect variables that are used without being declared. This detection is performed by function *missing*, which is monadic as it also outputs error messages:

$$
\begin{aligned}
&missing :: (Prog_2, Env) \to IO\,[Var]\\
&missing\,([\,], \_) = return\,[\,]\\
&\\
&missing\,(Use_2\ var : its_2, env)\\
&\quad = \mathbf{if}\ (var \in map\,\pi_1\ env)\ \mathbf{then}\ missing\,(its_2, env)\\
&\qquad\qquad\qquad\qquad \mathbf{else\ do}\ put\,(\texttt{"Missing: decl "} +\!\!+ var)\\
&\qquad\qquad\qquad\qquad\quad errs \leftarrow missing\,(its_2, env)\\
&\qquad\qquad\qquad\qquad\quad return\,(var : errs)\\
&\\
&missing\,(Dupl_2\ var : its_2, env) = \mathbf{do}\ errs \leftarrow missing\,(its_2, env)\\
&\qquad\qquad\qquad\qquad\qquad\qquad return\,(var : errs)\\
&\\
&missing\,((Block_2\ lev\ its) : its_2, env)\quad = \mathbf{do}\ errs_2 \leftarrow missing\,(its_2, env)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad errs_1 \leftarrow \mathbf{do}\ (p_2, env_2) \leftarrow duplicate\ lev\ env\ its\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad missing\,(p_2, env_2)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad return\,(errs_1 +\!\!+ errs_2)
\end{aligned}
$$

Now, we want to eliminate the intermediate data structure that glues *duplicate* and *missing*. Since the consumer is also monadic, Law 5 does not directly apply: the result of the law is a function that returns a monadic computation which in turn yields a monadic computation (and not a value) as result, that is, something of type $m\,(m\,a)$, for some a. To obtain a value as final result, it is simply necessary to run the computation. This gives the following shortcut fusion law, which is able to fuse two effectful functions.

Law 6 (Effectful pfold/mbuildp for lists) *Let m be a recursive monad.*

$$
\begin{aligned}
&\mathbf{do}\ \{(xs, z) \leftarrow mbuildp_{List}\ g\ c; pfold_{List}\ (hnil, hcons)\ (xs, z)\}\\
&=\\
&\quad \mathbf{do}\ \{\mathbf{rec}\ (m, \boxed{z}) \leftarrow \mathbf{let}\ knil = hnil\ \boxed{z}\\
&\qquad\qquad\qquad\qquad\qquad\quad kcons\ x\ y = hcons\ x\ y\ \boxed{z}\\
&\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \ g\,(knil, kcons)\ c; m\}
\end{aligned}
$$

Observe that, in this case, $hnil :: z \to m\,b$ and $hcons :: a \to m\,b \to z \to m\,b$, for some monad m, and therefore, $pfold_{List}\ (hnil, hcons) :: ([a], z) \to m\,b$. Also, notice that,

$$
\begin{aligned}
&mbuildp_{List} :: Monad\ m \Rightarrow (\forall\, b\,.\,(b, a \to b \to b) \to c \to m\,(b, z)) \to c \to m\,([a], z)\\
&mbuildp_{List}\ g = g\,([\,], (:))
\end{aligned}
$$

that is, $mbuildp_{List}\ g$ is now a function of type $c \to m\,([a], z)$. It is in this way that $mbuildp_{List}\ g$ will be considered in Sect. 3.3 when we will define the generic formulation of the laws. However, in Sect. 3.1 it was defined as a value of type $m\,([a], z)$ because that form is more appropriate for writing monadic parsers.

For this example we do not need to provide the instance of the *MonadFix* class for the *IO* monad as it is automatically provided by GHC, the compiler we are using. Now, if we write *missing* and *duplicate* in terms of $pfold_{List}$ and $mbuildp_{List}$, respectively, we can apply Law 6 to *semantics*. We obtain:

$$
\begin{aligned}
&semantics\ p\\
&\quad = \mathbf{do}\ \{\mathbf{rec}\ (m_{errs}, \boxed{\mathrm{env}})\\
&\qquad\qquad \leftarrow \mathbf{let}\ gk\ lev\ ds\ env\ [\,] = return\,(return\,[\,], ds)\\
&\qquad\qquad\qquad\qquad gk\ lev\ ds\ env\ (Use\ var : its)
\end{aligned}
$$

$$
\begin{array}{l}
\quad = \textbf{do}\ (its_2, ds') \leftarrow gk\ lev\ ds\ env\ its \\
\qquad return\ (\textbf{if}\ (var \in map\ \pi_1\ env) \\
\qquad\qquad \textbf{then}\ its_2 \\
\qquad\qquad \textbf{else do}\ put\ (\texttt{"Missing: decl "} +\!\!+ var) \\
\qquad\qquad\qquad errs \leftarrow its_2 \\
\qquad\qquad\qquad return\ (var : errs), ds') \\
gk\ lev\ ds\ env\ (Decl\ var : its) \\
\quad = \textbf{if}\ ((var, lev) \in ds) \\
\qquad \textbf{then do}\ put\ (\texttt{"Duplicate: decl "} +\!\!+ var) \\
\qquad\qquad (its_2, ds') \leftarrow gk\ lev\ ((var, lev) : ds)\ env\ its \\
\qquad\qquad return\ (\textbf{do}\ \{errs \leftarrow its_2; return\ (var : errs)\}, ds') \\
\qquad \textbf{else}\ gk\ lev\ ((var, lev) : ds)\ env\ its \\
\\
gk\ lev\ ds\ env\ (Block\ nested : its) \\
\quad = \textbf{do}\ (its_2, ds') \leftarrow gk\ lev\ ds\ env\ its \\
\qquad return \\
\qquad\quad (\textbf{do}\ errs_2 \leftarrow its_2 \\
\qquad\qquad errs_1 \leftarrow \textbf{do rec}\ (m_{errs_1}, \boxed{\text{env}_2}) \leftarrow gk\ lev\ ds \\
\qquad\qquad\qquad\qquad\qquad \boxed{\text{env}_2}\ nested\ m_{errs_1} \\
\qquad\qquad return\ (errs_1 +\!\!+ errs_2), ds') \\
\textbf{in}\ gk\ 0\ [\,]\ \boxed{\text{env}}\ p; m_{errs}\}
\end{array}
$$

The calculated program does not construct any intermediate data, while maintaining the original computations and side-effects.

3.3 The generic construction

In this section, we show that Laws 5 and 6, presented in the previous sections, are instances of generic definitions valid for a wide class of data types and monads.

3.3.1 Extended shortcut fusion

Shortcut fusion laws for monadic programs can be obtained as a special case of an extended form of shortcut fusion that captures the case when the intermediate data structure is generated as part of another structure given by a functor [15, 27]. This extension is based on an extended form of build: Given a functor F (signature of a datatype) and another functor N (representing the container structure), we can define

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

This is a natural extension of the standard *build*. In fact, $build_F$ can be obtained from $build_{F,N}$ by considering the identity functor $N\ a = a$. Moreover, $buildp_F$ is also a particular case obtained by considering the functor $N\ a = (a, z)$ and $map_N\ f = f \times id$.

Law 7 (Extended fold/build) *For strict k and strictness preserving N,*[7]

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

[7] The strictness-preserving assumption on the functor means that map_N preserves strict functions, *i.e.*, if f is strict, then so is $map_N\ f$.

See [15, 27] for a proof.

Similarly, we can also consider an extension for *buildp*:

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

with the following shortcut fusion law:

Law 8 (Extended fold/buildp) *For strict k and strictness-preserving N,*

$$map_N\;(fold_F\;k \times id) \circ buildp_{F,N}\;g = g\;k$$

Proof By considering $N'\;a = N\;(a,z)$, we have that $build_{F,N'}\;g = buildp_{F,N}\;g$ and $map_{N'}\;f = map_N\;(f \times id)$. Then, the left-hand side of the equation can be rewritten as: $map_{N'}\;(fold_F\;k) \circ build_{F,N'}\;g$. Finally, we apply Law 7. □

3.3.2 Monadic shortcut fusion

We are interested in studying Law 8 for the case when the functor N is the composition of a monad m with a product: For some type z,

$$N\;a = m\;(a,z) \qquad \text{and} \qquad map_N\;f = mmap\;(f \times id)$$

where recall that *mmap* is the map function for the monad m. The producer then corresponds to a monadic version of buildp:

$$\begin{aligned} &mbuildp_F \quad :: Monad\;m \Rightarrow (\forall\, a\,.\,(F\,a \to a) \to c \to m\,(a,z)) \to c \to m\,(\mu F, z)\\ &mbuildp_F\;g = g\;in_F \end{aligned}$$

A monadic shortcut fusion law can be directly obtained as an instance of Law 8. We unfold the definition of *mmap* so that we get a formulation in terms of **do**-notation:

Law 9 (Fold/mbuildp) *For strict k and strictness preserving mmap,*

$$\mathbf{do}\,\{(t,z) \leftarrow mbuildp_F\;g\;c;\; return\;(fold_F\;k\;t,z)\} = g\;k\;c$$

Using this law we can state a first monadic extension of the pfold/buildp rule (Law 4). Law 5, that was used to calculate a circular version of the bit string transformer, is the instance for lists of such monadic extension. For the introduction of a circular definition we need the assumption that the monad is *recursive* [11] as we require the use of a circular binding within a monadic computation. In HASKELL terms, this can be expressed using the recursive **do**-notation provided that the monad is an instance of the *MonadFix* class. We will use the following relationship:

$$\mathbf{do}\,\{\mathbf{rec}\;x \leftarrow e;\; e'\} = \mathbf{do}\,\{x \leftarrow mfix\;(\lambda x \to e);\; e'\} \tag{12}$$

Law 10 (Pfold/mbuildp) *Let m be a recursive monad with strictness-preserving mmap. For h with components $(h_1, \ldots, h_n)$ and strict,*

$$\begin{aligned} &\mathbf{do}\,\{(t,z) \leftarrow mbuildp_F\;g\;c;\; return\;(pfold_F\;h\;(t,z))\}\\ =\;&\\ &\mathbf{do}\,\{\mathbf{rec}\;(v, \boxed{z}) \leftarrow \mathbf{let}\;k_i\;\bar{x} = h_i\;\bar{x}\;\boxed{z}\;\mathbf{in}\;g\;k\;c;\; return\;v\} \end{aligned}$$

Proof

$$\begin{aligned} &\mathbf{do}\,\{(t,z) \leftarrow mbuildp_F\;g\;c;\; return\;(pfold_F\;h\;(t,z))\}\\ =\quad &\{\;(8)\;\} \end{aligned}$$

$$
\begin{array}{ll}
 & \mathbf{do}\ \{(t,z) \leftarrow mbuildp_F\ g\ c; \mathbf{let}\ k_i\ \bar{x} = h_i\ \bar{x}\ z\ \mathbf{in}\ return\ (fold_F\ k\ t)\} \\
= & \quad \{\ \text{definition of } \pi_1\ \} \\
 & \mathbf{do}\ \{(t,z) \leftarrow mbuildp_F\ g\ c; \mathbf{let}\ k_i\ \bar{x} = h_i\ \bar{x}\ z\ \mathbf{in}\ return\ (\pi_1\ (fold_F\ k\ t, z))\} \\
= & \\
 & \mathbf{do}\ (t,z) \leftarrow mbuildp_F\ g\ c \\
 & \quad (v,z') \leftarrow \mathbf{let}\ k_i\ \bar{x} = h_i\ \bar{x}\ z\ \mathbf{in}\ return\ (fold_F\ k\ t, z) \\
 & \quad return\ v \\
= & \quad \{\ \text{Lemma 11}\ \} \\
 & \mathbf{do\ rec}\ (t,z) \leftarrow mbuildp_F\ g\ c \\
 & \quad\quad (v,\boxed{z'}) \leftarrow \mathbf{let}\ k_i\ \bar{x} = h_i\ \bar{x}\ \boxed{z'}\ \mathbf{in}\ return\ (fold_F\ k\ t, z) \\
 & \quad return\ v \\
= & \quad \{\ \text{Law 9}\ \} \\
 & \mathbf{do}\ \{\mathbf{rec}\ (v,\boxed{z}) \leftarrow \mathbf{let}\ k_i\ \bar{x} = h_i\ \bar{x}\ \boxed{z}\ \mathbf{in}\ g\ k\ c; return\ v\}
\end{array}
$$

□

The circularity introduced in Law 10 is safe and therefore computations can be ordered under lazy evaluation. In order to prove this, we first need to prove the following property.

$$\forall\, z\, f\ .\ fix\ (\lambda(v,z') \rightarrow (f\ z', z)) = (f\ z, z) \tag{13}$$

Proof We prove this property by fixed-point induction with admissible predicate: $P\ (x,y) \equiv (x,y) \neq \bot \Rightarrow (x,y) = (f\ z, z)$.

Let us define $\phi\ (x,y) = (f\ y, z)$. The proof needs to consider two cases:

Base case: $P\ (\bot)$ is trivially true since the antecedent of P fails.

Inductive step: Assume that $P\ (x,y)$ holds. The inductive hypothesis is, therefore, that $(x,y) = (f\ z, z)$. We will prove that $P\ (\phi\ (x,y))$ holds.

$$\phi\ (x,y) = \{\ \text{def. } \phi\ \} = (f\ y, z) = \{\ \text{inductive hyp.}\ \} = (f\ z, z)$$

□

Now, we can state and prove the following law, that guarantees the safe introduction of circular definitions in Law 10.

Lemma 11 (Monadic Local Recursion) *Let m be a recursive monad with strictness-preserving mmap. For h with components $(h_1, \ldots, h_n)$ and strict,*

$$
\begin{array}{ll}
 & \mathbf{do}\ (v,z') \leftarrow \mathbf{let}\ k_i\ \bar{x} = h_i\ \bar{x}\ z\ \mathbf{in}\ return\ (fold_F\ k\ t, z) \\
 & \quad return\ v \\
= & \\
 & \mathbf{do\ rec}\ (v,\boxed{z'}) \leftarrow \mathbf{let}\ k_i\ \bar{x} = h_i\ \bar{x}\ \boxed{z'}\ \mathbf{in}\ return\ (fold_F\ k\ t, z) \\
 & \quad return\ v
\end{array}
$$

Proof

$$
\begin{array}{ll}
 & \mathbf{do\ rec}\ (v,\boxed{z'}) \leftarrow \mathbf{let}\ k_i\ \bar{x} = h_i\ \bar{x}\ \boxed{z'}\ \mathbf{in}\ return\ (fold_F\ k\ t, z) \\
 & \quad return\ v \\
= & \quad \{\ (12)\ \}
\end{array}
$$

$$
\begin{array}{l}
\quad \textbf{do}\ (v, z') \leftarrow \mathit{mfix}\ (\lambda(v, z') \rightarrow \textbf{let}\ k_i\ \bar{x} = h_i\ \bar{x}\ z'\ \textbf{in}\ \mathit{return}\ (\mathit{fold}_F\ k\ t, z)) \\
\qquad\quad \mathit{return}\ v \\
= \qquad \{\ (11)\ \} \\
\quad \textbf{do}\ (v, z') \leftarrow \mathit{return}\ (\mathit{fix}\ (\lambda(v, z') \rightarrow \textbf{let}\ k_i\ \bar{x} = h_i\ \bar{x}\ z'\ \textbf{in}\ (\mathit{fold}_F\ k\ t, z))) \\
\qquad\quad \mathit{return}\ v \\
= \qquad \{\ (13)\ \} \\
\quad \textbf{do}\ (v, z') \leftarrow \mathit{return}\ (\textbf{let}\ k_i\ \bar{x} = h_i\ \bar{x}\ z\ \textbf{in}\ (\mathit{fold}_F\ k\ t, z)) \\
\qquad\quad \mathit{return}\ v \\
= \\
\quad \textbf{do}\ (v, z') \leftarrow \textbf{let}\ k_i\ \bar{x} = h_i\ \bar{x}\ z\ \textbf{in}\ \mathit{return}\ (\mathit{fold}_F\ k\ t, z) \\
\qquad\quad \mathit{return}\ v
\end{array}
$$

□

Laws 9 and 10 handle monadic producers and purely functional consumers. When the consumer is also an effectful function, it is possible to state two other fusion laws. The formulation of these laws follows the approach presented by Chitil [5] and Ghani and Johann [15].

Law 12 (Effectful fold/mbuildp) *For strict* $k :: F\ (m\ a) \rightarrow m\ a$ *and strictness preserving mmap*,

$$
\begin{array}{l}
\quad \textbf{do}\ \{(t, z) \leftarrow \mathit{mbuildp}_F\ g\ c; v \leftarrow \mathit{fold}_F\ k\ t; \mathit{return}\ (v, z)\} \\
= \\
\quad \textbf{do}\ \{(m, z) \leftarrow g\ k\ c; v \leftarrow m; \mathit{return}\ (v, z)\}
\end{array}
$$

Proof

$$
\begin{array}{l}
\quad \textbf{do}\ \{(t, z) \leftarrow \mathit{mbuildp}_F\ g\ c; v \leftarrow \mathit{fold}_F\ k\ t; \mathit{return}\ (v, z)\} \\
= \textbf{do}\ (t, z) \quad \leftarrow \mathit{mbuildp}_F\ g\ c \\
\qquad\quad (m, \_) \leftarrow \mathit{return}\ (\mathit{fold}_F\ k\ t, z) \\
\qquad\quad v \qquad \leftarrow m \\
\qquad\quad \mathit{return}\ (v, z) \\
= \textbf{do}\ (m, z) \leftarrow \textbf{do}\ \{(t, z) \leftarrow \mathit{mbuildp}_F\ g\ c; \mathit{return}\ (\mathit{fold}_F\ k\ t, z)\} \\
\qquad\quad v \qquad \leftarrow m \\
\qquad\quad \mathit{return}\ (v, z) \\
= \textbf{do}\ \{(m, z) \leftarrow g\ k\ c; v \leftarrow m; \mathit{return}\ (v, z)\}
\end{array}
$$

□

Using this law we can now state a shortcut fusion law for the derivation of monadic circular programs in those cases where both the producer and consumer are effectful functions. Again, like in Law 10, the monad is required to be recursive because of the introduction of a recursive binding within the monadic computation.

Law 13 (Effectful pfold/mbuildp) *Let* m *be a recursive monad with strictness-preserving mmap. For* $h :: (F\ (m\ a), z) \rightarrow m\ a$ *with components* $(h_1, \ldots, h_n)$ *and strict*,

$$
\begin{array}{l}
\quad \textbf{do}\ \{(t, z) \leftarrow \mathit{mbuildp}_F\ g\ c; \mathit{pfold}_F\ h\ (t, z)\} \\
= \\
\quad \textbf{do}\ \{\textbf{rec}\ (m, \boxed{z}) \leftarrow \textbf{let}\ k_i\ \bar{x} = h_i\ \bar{x}\ \boxed{z}\ \textbf{in}\ g\ k\ c; m\}
\end{array}
$$

Proof

$$
\begin{array}{l}
\quad \textbf{do}\ \{(t, z) \leftarrow \mathit{mbuildp}_F\ g\ c; \mathit{pfold}_F\ h\ (t, z)\} \\
=
\end{array}
$$

$$\begin{aligned}
&\textbf{do}\ \{(t,z) \leftarrow \mathit{mbuildp}_F\ g\ c; m \leftarrow \mathit{return}\ (\mathit{pfold}_F\ h\ (t,z)); m\}\\
=\quad &\{\ \text{Law 10}\ \}\\
&\textbf{do}\ \{\textbf{rec}\ (m, \boxed{z}) \leftarrow \textbf{let}\ k_i\ \bar{x} = h_i\ \bar{x}\ \boxed{z}\ \textbf{in}\ g\ k\ c; m\}
\end{aligned}$$

□

Law 6 is the specific instance of Law 13 for the case where the intermediate data structure is a list.

4 Termination analysis and semantic considerations

In this section, we address the termination properties of the circular programs calculated with Laws 4, 10 and 13. We also analyze in detail the implications that the semantics of HASKELL has on Law 4. The same type of analysis is not needed for the calculation of higher-order programs, since the corresponding rules (to be presented in Sect. 5) are already valid under HASKELL's semantics and no termination issues arise in that context.

An important feature of our transformations is that both in the case of pure and monadic programs the circular programs we derive hold the same termination properties of the original ones. In both cases the safe introduction of the circularity is based on the fact that the components of a pair are obtained by independent computations, which enables us to tie the knot and make the computation of one of the components depend on the value computed in the other one.

In the case of pure programs it is Law 3, which comes as a consequence of the polymorphic type of $g :: \forall\, a\ .\ (F\ a \to a) \to c \to (a, z)$, that ensures the safe introduction of a circular definition. This law states that the computed value of type z is always the same whatever the particular algebra that is applied to g. As a consequence of that in Law 4 we can safely tie the knot by substituting the algebra in_F by an algebra k whose operations depend on the value of type z that is computed by g itself. In this case the introduction of the circularity was on the producer side in combination with the operations of the consumer.

In the monadic case (Laws 10 and 13), however, we must proceed a bit differently as it is not possible to reproduce the same reasoning on the producer side due to the presence of effects. The introduction of the circularity is then performed on the consumer side as part of Lemma 11 with (13) as the essential property that ensures that it is safe to tie the knot. In fact, given an expression of the form $\textbf{let}\ (v, z') = (f\ z, z)\ \textbf{in}\ v$, for some z, (13) states that it is correct to introduce a circularity and turn this expression into this other one $\textbf{let}\ (v, \boxed{z'}) = (f\ \boxed{z'}, z)\ \textbf{in}\ v$. We could have applied this same property on the consumer side for the derivation of pure programs, but in that case we preferred to rely on the nice property given by g's polymorphic type.

In the terminology of Danielsson et al. [7], Law 4 is "morally correct" *only* in HASKELL. In fact, the proof of the rule required two applications of *surjective pairing* (6) involving function g. However, (6) is not valid in HASKELL: though it holds for defined values, it fails when the result of function g is undefined, because $\bot$ is different from $(\bot, \bot)$ as a consequence of lifted products. Therefore, (6) is morally correct *only* and, in the same sense, so is our rule. In [14], we pointed out that due to the presence of *seq* additional pre-conditions should be defined in our rule in order to guarantee its correctness in HASKELL [21].

Following our work, Voigtländer [39] performed a rigorous study of various shortcut fusion rules for languages like HASKELL. In particular, Voigtländer presents semantic and pragmatic considerations about Law 4. As a first result, it is possible to prove its total correctness by adding the pre-condition: for every $1 \leqslant i \leqslant n$,

$$h_i \underbrace{\bot \cdots \bot\ \bot}_{r_i+1} \neq \bot$$

The law that results from the addition of this pre-condition is, however, *pessimistic*. By this we mean that, even if the newly added pre-condition is violated, it does not necessarily imply that the law gets broken. In fact, Voigtländer [39] presents an (admittedly artificial) example where the pre-condition is violated, causing no harm in the calculated equivalent program. However, this version of the law is the most general one can present for calculating circular programs in terms of total correctness.

In the line of the semantic analysis presented for Law 4 we intend in the future to perform a rigorous study of the laws we developed for monadic circular programs in the context of HASKELL semantics.

5 Calculation of higher-order programs

In the previous sections we have studied calculational techniques for the derivation of circular programs from compositions $prog = cons \circ prod$, where $prod :: a \to (t, z)$ and $cons :: (t, z) \to b$, in the context of both pure and monadic programs.

An alternative solution is to transform programs such as *prog* into higher-order programs using a well-known program transformation technique called lambda-abstraction [35]. In our particular context, the idea of the transformation is to derive a new function $prog' :: a \to (z \to b, z)$, which returns a function and the same value of type z that would be generated by *prod*, such that $prog\ a = f\ z$ **where** $(f, z) = prog'\ a$. Based on this idea, Voigtländer [39] introduced a shortcut fusion rule for the derivation of pure, higher-order programs from compositions like *prog* when lists are the intermediate structure. In this paper, we extend this result in two ways. First, we present a generic formulation of the shortcut fusion rule for the derivation of pure, higher-order programs that can be applied to a wide range of datatypes as intermediate data structures. Second, we extend the generic rule to the context of monadic programming, obtaining shortcut fusion rules for the derivation of monadic higher-order programs. The rules we present in this section consider three cases: the case where the producer and the consumer are both pure functions, the case where the producer function is monadic and the consumer is given by a pure function, and the case where both functions are monadic.

Obtaining higher-order programs is interesting since their execution is not restricted to a lazy evaluation setting as it happens with the execution of the circular ones. Furthermore, experimental benchmarks presented in [12] suggest that the performance of the higher-order programs derived from programs like *prog* is significantly better than the performance of their circular or original equivalents, both in terms of run time and memory consumption. Even by considering that the programs in such experiments are not covered by our transformations, the fact that they are of the same kind as ours (circular, higher-order and programs in compositional style) induces us to expect similar results here.

5.1 The higher-order pfold/buildp rule

Starting from a composition of a *pfold* and a *buildp*, we present an alternative transformation which exploits the fact that every *pfold* can be expressed in terms of a higher-order fold: For $h :: (F\ a, z) \to a$,

$$pfold_F\ h = apply \circ (fold_F\ \varphi_h \times id) \tag{14}$$

where $fold_F\ \varphi_h :: \mu F \to (z \to a)$, the curried version of $pfold_F\ h$, has algebra $\varphi_h :: F\ (z \to a) \to (z \to a)$ given by

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

and $apply :: (a \to b, a) \to b$ is defined by $apply\ (f, x) = f\ x$.

With this relationship at hand we can state the following shortcut fusion law, which is the instance to our context of a more general program transformation technique called lambda abstraction [35]. The specific case of this law when lists are the intermediate data structure was recently introduced by Voigtländer [39].

Law 14 (Higher-order pfold/buildp) *For left-strict h,*[8]

$$pfold_F\ h \circ buildp_F\ g = apply \circ g\ \varphi_h$$

Proof

$$\begin{aligned}
& pfold_F\ h \circ buildp_F\ g \\
= & \quad \{\ (14)\ \} \\
& apply \circ (fold_F\ \varphi_h \times id) \circ buildp_F\ g \\
= & \quad \{\ \text{Law } 2\ \} \\
& apply \circ g\ \varphi_h
\end{aligned}$$

□

Like in the derivation of circular programs, $g\ \varphi_h$ returns a pair, but now composed of a function of type $z \to a$ and an object of type z. The final result then corresponds to the application of the function to the object:

$$pfold_F\ h\ (buildp_F\ g\ c) = \mathbf{let}\ (f, z) = g\ \varphi_h\ c\ \mathbf{in}\ f\ z$$

To see an example of the application of Law 14, consider again the straightforward solution to the *repmin* problem:

$$transform\ t = replace\ (tmint\ t)$$

The expression of *replace* and *tmint* in terms of *pfold* and *buildp*, respectively, was given at the end of Sect. 2.3. In order to apply the law, we need the expression of the higher-order fold that corresponds to the curried version of *replace*:

$$\begin{aligned}
& replace_{ho} :: Tree \to (Int \to Tree) \\
& replace_{ho} = fold_{Tree}\ (\varphi_{hleaf}, \varphi_{hfork}) \\
& \quad \mathbf{where}\ \varphi_{hleaf}\ _ = \lambda z \to Leaf\ z \\
& \qquad\qquad \varphi_{hfork}\ l\ r = \lambda z \to Fork\ (l\ z)\ (r\ z)
\end{aligned}$$

Then, by direct application of Law 14 to *transform*, we obtain:

$$transform = apply \circ gtm\ (\varphi_{hleaf}, \varphi_{hfork})$$

Inlining the above definition, we obtain the higher-order solution to *repmin* that we had already presented in p. 117:

$$\begin{aligned}
& transform\ t = nt\ m \\
& \quad \mathbf{where}\ (nt, m) = repmin\ t \\
& \qquad repmin\ (Leaf\ n) = (\lambda z \to Leaf\ z, n) \\
& \qquad repmin\ (Fork\ l\ r) = \mathbf{let}\ (l', n_1) = repmin\ l \\
& \qquad\qquad\qquad\qquad\qquad\quad (r', n_2) = repmin\ r \\
& \qquad\qquad\qquad\qquad\quad \mathbf{in}\ (\lambda z \to Fork\ (l'\ z)\ (r'\ z), min\ n_1\ n_2)
\end{aligned}$$

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

5.2 Calculation of monadic higher-order programs

In the same way we did for circular program derivation, we now wish to extend the derivation of higher-order programs to monadic programs. We start by reviewing the two monadic examples we presented in Sect. 3: the bit string transformer (Sect. 3.1) and the semantic analyzer for Algol (Sect. 3.2). Finally, we present the formal definition of the constructions that give rise to the specific laws we present in those examples.

5.2.1 Bit string transformation

Recall that in this example we wanted to transform a sequence of bits into a new one, of the same length, by applying the exclusive or between each bit and the binary sum (sum modulo 2) of the sequence. We considered that the input sequence was given as a string of bits, which was parsed into a list and then transformed. Recall also that, for the parsing phase, we used a monadic parser. In Sect. 3.1, we defined the following solution to this problem:

$$shift = \mathbf{do}\ \{(bs, s) \leftarrow bitstring;\ return\ (transform\ (bs, s))\}$$

This solution constructs an intermediate list of bits, which can be eliminated by using a specific fusion rule that transform this (monadic) composition into a higher-order program. The transformation closely follows the ideas introduced in Sect. 5.1, except that, now, we deal with monadic functions. The fusion rule we present below is specific for lists and uses the version of $mbuildp_{List}$ presented on p. 133.

Law 15 (Higher-order pfold/mbuildp for lists)

$$\begin{array}{l}\quad \mathbf{do}\ \{(t, z) \leftarrow mbuildp_{List}\ g;\ return\ (pfold_{List}\ (hnil, hcons)\ (t, z))\} \\ = \\ \quad \mathbf{do}\ \{(f, z) \leftarrow g\ (\varphi_{hnil}, \varphi_{hcons});\ return\ (f\ z)\}\end{array}$$

The pair $(\varphi_{hnil}, \varphi_{hcons})$ are the components of the algebra of the higher-order fold that corresponds to the curried version of $pfold_{List}\ (hnil, hcons)$.

To apply the law to *shift*, we need to express *transform* as a higher-order fold:

$$\begin{array}{l} transform_{ho} :: [Bit] \rightarrow (Bit \rightarrow [Bit]) \\ transform_{ho} = fold_L\ (\varphi_{hnil}, \varphi_{hcons}) \\ \quad \mathbf{where}\ \varphi_{hnil} \quad\ = \lambda\_ \rightarrow [\,] \\ \qquad\quad \varphi_{hcons}\ b\ r = \lambda s \rightarrow (b \oplus s) : r\ s \end{array}$$

Then, by Law 15 we obtain the following higher-order monadic program:

$$shift = \mathbf{do}\ \{(f, s) \leftarrow g\ (\varphi_{hnil}, \varphi_{hcons});\ return\ (f\ s)\}$$

Inlining the above definition, we obtain

$$\begin{array}{l} shift = \mathbf{do}\ \{(f, s) \leftarrow g_\varphi;\ return\ (f\ s)\} \\ \quad \mathbf{where}\ g_\varphi = (\mathbf{do}\ b \quad\ \leftarrow bit \\ \qquad\qquad\qquad\quad (f, s) \leftarrow g_\varphi \\ \qquad\qquad\qquad\quad return\ (\lambda s' \rightarrow (b \oplus s') : f\ s', b \oplus s))\ \langle|\rangle\ return\ (\lambda\_ \rightarrow [\,], Zero) \end{array}$$

The *shift* program that we have just derived is a higher-order program in the sense that f, one of the results produced by function g_φ, is itself a function. Once applied to s, function f produces the desired transformed list of bits.

5.2.2 Algol 68 scope rules

In this section, we calculate a higher-order monadic program equivalent to the semantic analyzer for the Algol 68 scope rules that we presented in Sect. 3.2. There, we defined:

$$semantics\ p = \mathbf{do}\ \{\,(p', env) \leftarrow duplicate\ 0\ [\,]\ p;\ missing\ (p', env)\,\}$$

In our calculation, we use a rule that is an instance for lists of a more general law to be introduced in Sect. 5.2.3. The rule is very similar to Law 15, used to calculate a higher-order version of *shift*, but with the difference that now, in *semantics*, both the consumer and the producer are monadic functions. For this new rule we use the version of $mbuildp_{List}$ presented on p. 136.

Law 16 (Effectful higher-order pfold/mbuildp for lists)

$$\begin{aligned}&\mathbf{do}\ \{\,(t, z) \leftarrow mbuildp_{List}\ g\ c;\ pfold_{List}\ (hnil, hcons)\ (t, z)\,\}\\ =\\ &\mathbf{do}\ \{\,(f, z) \leftarrow g\ (\varphi_{hnil}, \varphi_{hcons})\ c; f\ z\}\end{aligned}$$

Again, the pair $(\varphi_{hnil}, \varphi_{hcons})$ is the algebra of the higher-order fold corresponding to $pfold_{List}\ (hnil, hcons)$.

In order to apply Law 16 to *semantics*, we need to express *missing* in terms of a higher-order fold, since its algebra $(\varphi_{hnil}, \varphi_{hcons})$ is necessary to apply the law.

$$\begin{aligned}&missing_{ho} = fold_L\ (\varphi_{hnil}, \varphi_{hcons})\\ &\quad\mathbf{where}\ \varphi_{hnil} = \lambda\_ \rightarrow return\ [\,]\\ &\qquad\varphi_{hcons}\ (Use_2\ var)\ ferrs\\ &\qquad\quad = \lambda env \rightarrow \mathbf{if}\ (var \in map\ \pi_1\ env)\ \mathbf{then}\ ferrs\ env\\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else\ do}\ put\ (\texttt{"Missing: decl "} +\!\!+ var)\\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad errs \leftarrow ferrs\ env\\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad return\ (var : errs)\\ &\qquad\varphi_{hcons}\ (Dupl_2\ var)\ ferrs\\ &\qquad\quad = \lambda env \rightarrow \mathbf{do}\ errs \leftarrow ferrs\ env\\ &\qquad\qquad\qquad\quad return\ (var : errs)\\ &\qquad\varphi_{hcons}\ (Block_2\ lev\ its)\ ferrs\\ &\qquad\quad = \lambda env \rightarrow \mathbf{do}\ errs_2 \leftarrow ferrs\ env\\ &\qquad\qquad\qquad\quad errs_1 \leftarrow \mathbf{do}\ (p_2, env_2) \leftarrow duplicate'\ lev\ env\ its\\ &\qquad\qquad\qquad\qquad\qquad\quad missing'\ (p_2, env_2)\\ &\qquad\qquad\qquad\quad return\ (errs_1 +\!\!+ errs_2)\end{aligned}$$

Then, by Law 16, we obtain:

$$\begin{aligned}&semantics\ p = \mathbf{do}\ \{(ferrs, env) \leftarrow g_\varphi\ 0\ [\,]\ p; ferrs\ env\}\\ &\quad\mathbf{where}\\ &\qquad g_\varphi\ lev\ ds\ [\,] = return\ (\lambda\_ \rightarrow return\ [\,], ds)\\ &\qquad g_\varphi\ lev\ ds\ (Use\ var : its)\\ &\qquad\quad = \mathbf{do}\ (ferrs, ds') \leftarrow g_\varphi\ lev\ ds\ its\\ &\qquad\qquad\quad return\ (\lambda env \rightarrow \mathbf{if}\ (var \in map\ \pi_1\ env)\\ &\qquad\qquad\qquad\qquad\qquad\quad \mathbf{then}\ ferrs\ env\\ &\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else\ do}\ put\ (\texttt{"Missing: decl "} +\!\!+ var)\\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad errs \leftarrow ferrs\ env\\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad return\ (var : errs), ds')\\ &\qquad g_\varphi\ lev\ ds\ (Decl\ var : its)\\ &\qquad\quad = \mathbf{if}\ ((var, lev) \in ds)\ \mathbf{then\ do}\ put\ (\texttt{"Duplicate: decl "} +\!\!+ var)\\ &\qquad\qquad\qquad\qquad\qquad\quad (ferrs, ds') \leftarrow g_\varphi\ lev\ ((var, lev) : ds)\ its\\ &\qquad\qquad\qquad\qquad\qquad\quad return\ (\lambda env \rightarrow \mathbf{do}\ errs \leftarrow ferrs\ env\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad return\ (var : errs), ds')\end{aligned}$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ g_\varphi\ lev\ ((var, lev) : ds)\ its \\
g_\varphi\ lev\ ds\ (Block\ nested : its) \\
\quad = \textbf{do}\ (ferrs_2, ds') \leftarrow g_\varphi\ lev\ ds\ its \\
\qquad\quad return\ (\lambda env \rightarrow \textbf{do}\ errs_2 \leftarrow ferrs_2\ env \\
\qquad\qquad\qquad\qquad\qquad\qquad errs_1 \leftarrow \textbf{do}\ (ferrs_1, env_1) \leftarrow g_\varphi\ (lev + 1)\ env\ nested \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad ferrs_1\ env_1 \\
\qquad\qquad\qquad\qquad\qquad\qquad return\ (errs_1 +\!\!+ errs_2), ds')
\end{array}
$$

5.2.3 Calculating monadic higher-order programs, generically

Now, we present a generic formulation of the laws showed in the examples just presented. Those generic laws may be considered as an extension to monadic programs of the law presented in Sect. 5.1.

First, we show the generalization of Law 14 in the sense of extended shortcut fusion, that is, for an arbitrary functor N.

Law 17 *For left-strict h and strictness-preserving N,*

$$map_N\ (pfold_F\ h) \circ buildp_{F,N}\ g = map_N\ apply \circ g\ \varphi_h$$

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

Proof

$$
\begin{array}{ll}
& map_N\ (pfold\ h) \circ buildp_{F,N}\ g \\
= & \qquad \{\ (14), (3)\ \} \\
& map_N\ apply \circ map_N\ (fold\ \varphi_h \times id) \circ buildp_{F,N}\ g \\
= & \qquad \{\ \text{Law } 8\ \} \\
& map_N\ apply \circ g\ \varphi_h
\end{array}
$$

□

Let us consider the particular case when N is the functor of a monad. Unlike the transformation to circular programs, now we do not need to require the monad to be recursive. In the first place we state the case when the *pfold* is a pure function. This is the case of the bitstring transformer presented in Sect. 5.2.1.

Law 18 (Higher-order pfold/mbuildp) *For left-strict h and strictness-preserving mmap,*

$$
\begin{array}{l}
\quad \textbf{do}\ \{\ (t, z) \leftarrow mbuildp_F\ g\ c;\ return\ (pfold\ h\ (t, z))\ \} \\
= \\
\quad \textbf{do}\ \{\ (f, z) \leftarrow g\ \varphi_h\ c;\ return\ (f\ z)\ \}
\end{array}
$$

Notice that $mbuildp_F\ g$ is a function of type $mbuildp_F\ g :: c \rightarrow m\ (\mu F, z)$, whereas in Sect. 5.2.1 it was defined as a value of type $m\ (\mu F, z)$, for μF the type of lists, because the latter form was more appropriate for writing monadic parsers.

Finally, we present a generic fusion rule that is able to deal with programs where the consumer is also an effectful function. One example of such a program is the analyzer of the Algol 68 scope rules that was presented in Sect. 5.2.2.

Law 19 (Effectful higher-order pfold/mbuildp) *For left-strict $h :: (F\ (m\ a), z) \rightarrow m\ a$ and strictness-preserving mmap,*

$$
\begin{array}{l}
\quad \textbf{do}\ \{\ (t, z) \leftarrow mbuildp_F\ g\ c;\ pfold\ h\ (t, z)\ \} \\
= \\
\quad \textbf{do}\ \{\ (f, z) \leftarrow g\ \varphi_h\ c; f\ z\}
\end{array}
$$

Proof

$$
\begin{aligned}
&\mathbf{do}\ \{(t,z) \leftarrow mbuildp_F\ g\ c;\ pfold\ h\ (t,z)\}\\
=&\\
&\mathbf{do}\ (t,z) \leftarrow mbuildp_F\ g\ c\\
&\quad\ \ m \quad\ \leftarrow return\ (pfold\ h\ (t,z))\\
&\quad\ \ m\\
=&\qquad \{\ \text{Law 18}\ \}\\
&\mathbf{do}\ m \leftarrow \mathbf{do}\ \{(f,z) \leftarrow g\ \varphi_h\ c;\ return\ (f\ z)\}\\
&\quad\ \ m\\
=&\\
&\mathbf{do}\ \{(f,z) \leftarrow g\ \varphi_h\ c; f\ z\}
\end{aligned}
$$

□

6 Conclusions

In this paper we have presented shortcut fusion rules for the derivation of circular and higher-order programs. The rules apply to compositions between a producer and a consumer, such that the producer computes some context information and constructs an intermediate data structure that is later traversed by the consumer. As a result of applying our method, such programs are transformed into equivalent ones that construct no intermediate data structures.

The rules were first introduced in the context of pure programs and were later studied for monadic programs as well; they have been given a generic formulation so that they can be instantiated for a wide class of algebraic data types and monads. We have formally proved that all the rules are correct and we have shown that they can be widely applicable through several examples of practical interest.

The emphasis of the paper is on the laws that dictate the transformations. For more pragmatical aspects, we refer the reader to [12].

Various aspects of the ideas presented in this paper deserve further elaboration. The examples we presented here consist of compositions of a single producer and consumer functions. We would like, however, to be able to achieve the same fusion goals for programs consisting in an arbitrary number of function compositions. Indeed, we are now studying how to generalize our work in order to optimize programs of the form $f_n \circ \cdots \circ f_1$ such that in each composition a data structure t_i and a value z_i are produced. Circular programs and attribute grammars (AGs) are closely related [36]. We would like to express in a calculational form the AG-based circular program transformations presented in [13], so that their correctness can be proved. Indeed, although these techniques are largely used by the AG community, their correctness remains to be formally proved!

Acknowledgements We wish to thank the anonymous reviewers for detailed and insightful comments on earlier versions of this paper.

References

1. Abramsky, S., Jung, A.: Domain theory. In: Handbook of Logic in Computer Science, pp. 1–168. Clarendon, Oxford (1994)
2. Baier, H., Kugler, D., Margraf, M.: Elliptic curve cryptography based on ISO 15946. Technical report, Federal Office for Information Security (2007)

3. Bird, R.: Using circular programs to eliminate multiple traversals of data. Acta Inform. **21**, 239–250 (1984)
4. Bird, R., de Moor, O.: Algebra of Programming. Prentice-Hall International Series in Computer Science, vol. 100. Prentice Hall, New York (1997)
5. Chitil, O.: Type-inference based deforestation of functional programs. Ph.D. thesis, RWTH Aachen (October 2000)
6. Cockett, R., Spencer, D.: Strong categorical datatypes I. In: Seely, R.A.C. (ed.) International Meeting on Category Theory 1991. Canadian Mathematical Society Conference Proceedings, vol. 13, pp. 141–169 (1991)
7. Danielsson, N.A., Hughes, J., Jansson, P., Gibbons, J.: Fast and loose reasoning is morally correct. In: POPL'06: Proc. of the 33rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. ACM, New York (2006)
8. Danvy, O., Goldberg, M.: There and back again. In: ICFP'02: Proc. of the 7th ACM SIGPLAN Int. Conf. on Functional Programming. ACM, New York (2002)
9. de Moor, O., Peyton Jones, S.L., Van Wyk, E.: Aspect-oriented compilers. In: GCSE'99: Proc. of the 1st International Symposium on Generative and Component-Based Software Engineering, pp. 121–133. Springer, Berlin (2000)
10. Dijkstra, A., Swierstra, D.: Typing Haskell with an attribute grammar. Technical report, Inst. of Information and Computing Sciences, Utrecht University (2004)
11. Erkök, L., Launchbury, J.: A recursive do for Haskell. In: Haskell'02: Proceedings of the ACM SIGPLAN Haskell Workshop, pp. 29–37. ACM, New York (2002)
12. Fernandes, J.P.: Design, implementation and calculation of circular programs. Ph.D. thesis, Department of Informatics, University of Minho, Portugal (March 2009)
13. Fernandes, J.P., Saraiva, J.: Tools and libraries to model and manipulate circular programs. In: PEPM'07: Proc. of the ACM SIGPLAN 2007 Symposium on Partial Evaluation and Program Manipulation, pp. 102–111. ACM, New York (2007)
14. Fernandes, J.P., Pardo, A., Saraiva, J.: A shortcut fusion rule for circular program calculation. In: Proceedings of the ACM SIGPLAN Haskell Workshop, pp. 95–106. ACM, New York (2007)
15. Ghani, N., Johann, P.: Short cut fusion of recursive programs with computational effects. In: Symposium on Trends in Functional Programming (2008)
16. Gibbons, J.: Calculating functional programs. In: Algebraic and Coalgebraic Methods in the Mathematics of Program Construction. LNCS, vol. 2297, pp. 148–203. Springer, Berlin (2002)
17. Gill, A.: Cheap deforestation for non-strict functional languages. Ph.D. thesis, Department of Computing Science, University of Glasgow, UK (1996)
18. Gill, A., Launchbury, J., Peyton Jones, S.L.: A short cut to deforestation. In: Conference on Functional Programming Languages and Computer Architecture, pp. 223–232 (1993)
19. Hinze, R., Jeuring, J.: Generic Haskell: practice and theory. In: Summer School on Generic Programming (2002)
20. Hutton, G., Meijer, E.: Monadic parsing in Haskell. J. Funct. Program. **8**(4), 437–444 (1998)
21. Johann, P., Voigtländer, J.: Free theorems in the presence of seq. In: POPL'04: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 99–110. ACM, New York (2004)
22. Johnsson, T.: Attribute grammars as a functional programming paradigm. In: Functional Programming Languages and Computer Architecture (1987)
23. Jones, G., Gibbons, J.: Linear-time breadth-first tree algorithms an exercise in the arithmetic of folds and zips. Technical Report 71, Dept. of Computer Science, University of Auckland (1993)
24. Kastens, U., Pfahler, P., Jung, M.T.: The eli system. In: CC'98: Proceedings of the 7th International Conference on Compiler Construction, pp. 294–297. Springer, London (1998)
25. Kuiper, M., Swierstra, D.: Using attribute grammars to derive efficient functional programs. In: Computing Science in the Netherlands (1987)
26. Lawall, J.L.: Implementing circularity using partial evaluation. In: Proceedings of the Second Symposium on Programs as Data Objects (PADO). LNCS, vol. 2053. Springer, Berlin (2001)
27. Manzino, C., Pardo, A.: Shortcut fusion of monadic programs. J. Univers. Comput. Sci. **14**(21), 3431–3446 (2008)
28. Marlow, S., Peyton Jones, S.: The new GHC/Hugs Runtime System (1999)
29. Ohori, A., Sasano, I.: Lightweight fusion by fixed point promotion. In: POPL'07: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 143–154. ACM, New York (2007)
30. Okasaki, C.: Breadth-first numbering: lessons from a small exercise in algorithm design. ACM SIGPLAN Not. **35**(9), 131–136 (2000)

31. Onoue, Y., Hu, Z., Iwasaki, H., Takeichi, M.: A calculational fusion system HYLO. In: IFIP TC 2 Working Conference on Algorithmic Languages and Calculi, Le Bischenberg, France, pp. 76–106. Chapman & Hall, London (1997)
32. Paakki, J.: Attribute grammar paradigms—a high-level methodology in language implementation. ACM Comput. Surv. **27**(2), 196–255 (1995)
33. Pardo, A.: Generic accumulations. In: IFIP TC2/WG2.1 Working Conference on Generic Programming, pp. 49–78. Kluwer Academic, Dordrecht (2003)
34. Pardo, A.: A calculational approach to recursive programs with effects. Ph.D. thesis, Technische Universität Darmstadt (October 2001)
35. Pettorossi, A., Skowron, A.: The lambda abstraction strategy for program derivation. In: Fundamenta Informaticae XII, pp. 541–561 (1987)
36. Swierstra, D.: Tutorial on attribute grammars. In: Second International Conference on Generative Programming and Component Engineering (GPCE) (2003)
37. Swierstra, D., Chitil, O.: Linear, bounded, functional pretty-printing. J. Funct. Program. **19**(01), 1–16 (2009)
38. Takano, A., Meijer, E.: Shortcut deforestation in calculational form. In: Proc. Conference on Functional Programming Languages and Computer Architecture, pp. 306–313. ACM, New York (1995)
39. Voigtländer, J.: Semantics and pragmatics of new shortcut fusion rules. In: Proc. of the 2008 International Symposium on Functional and Logic Programming, pp. 163–179. Springer, Berlin (2008)
40. Voigtländer, J.: Using circular programs to deforest in accumulating parameters. High.-Order Symb. Comput. **17**, 129–163 (2004)
41. Wadler, P.: Theorems for free! In: 4th International Conference on Functional Programming and Computer Architecture. ACM, London (1989)
42. Wadler, P.: Deforestation: transforming programs to eliminate trees. Theor. Comput. Sci. **73**, 231–248 (1990)