Watch out for that tree!
A Tutorial on Shortcut Deforestation

João Paulo Fernandes[1,5], Jácome Cunha[2], João Saraiva[3,5], and Alberto Pardo[4]

[1] LISP-Release, Universidade da Beira Interior, Portugal
jpf@di.ubi.pt
[2] NOVA LINCS, DI, FCT, Universidade NOVA de Lisboa, Portugal
jacome@fct.unl.pt
[3] Universidade do Minho, Portugal
jas@di.uminho.pt
[4] Universidad de la República, Uruguay
pardo@fing.edu.uy
[5] HASLab/INESC TEC, Portugal

Abstract. Functional programmers are strong enthusiasts of modular solutions to programming problems. Since software characteristics such as readability or maintainability are often directly proportional to modularity, this programming style naturally contributes to the beauty of functional programs. Unfortunately, in return of this beauty we often sacrifice efficiency: modular programs rely, at runtime, on the creation, use and elimination of intermediate data structures to connect its components. In this tutorial paper, we study an advanced technique that attempts to retain the best of this two worlds: i) it allows programmers to implement beautiful, modular programs ii) it shows how to transform such programs, in a way that can be incorporated in a compiler, into programs that do not construct any intermediate structure.

1 Introduction

Functional programming languages are a natural setting for the development of modular programs. Features common in functional programming languages, like polymorphism, higher-order functions and lazy evaluation are ingredients particularly suitable to develop software in a modular way. In such a setting, a software engineering develops her/his software by combining a set of simple, reusable, and off-the-shelf library of generic components into more complex (and possibly reusable) software. Indeed, already in Hughes (1984) it is stressed that modularity is a fundamental reason contributing to successful programming, hence the expressive power and relevance of functional languages.

Let us consider, for example, that we wish to define a function, named *trail*, to compute the last n lines of a given text. The naive programmer will solve this problem by defining from scratch all that functionality in a single, monolithic function. Although such a function may be correct and may have an efficient

execution time, it may be harder to define and to understand. In a modular setting programmers tend to solve these problems by re-using simpler functions and to combine them in order to solve the problem under consideration.

For example, *trail* may be defined in an elegant way as follows:

$$trail :: Int \to Text \to Text$$
$$trail\ n\ t = (unlines \circ reverse \circ take\ n \circ reverse \circ lines)\ t$$

where several simple, well known, and well understood library functions are reused, namely, function *lines* that breaks a text in (a list containing) the lines that constitute it, function *reverse* that inverts the order of the elements in a list, function *take* n that selects the first n elements of a list, and function *unlines* that implements the inverse behavior of *lines*. Such functions are easily combined by using another reusable, higher-order construction: function composition, denoted by $\circ$.[6]

However, such a setting may also entail a drawback: as it encourages a compositional style of programming where non-trivial solutions are constructed composing simple functions, intermediate structures need to be constructed to serve as connectors of such functions.

In *trail*, for example, function *lines* produces a list of strings which is used by *reverse* to construct another list, which then feeds *take* n, and so on.

In practical terms, constructing, traversing and destroying these data structures may degrade the performance of the resulting implementations. And, in fact, the naive programmer surely agrees that the modular solution is more elegant, concise, and easy to understand, but may still be convinced that his monolithic solution is better simply because it may be more efficient!

In this tutorial we will study concrete settings where this drawback can be avoided. For this, we rely on a program transformation technique, usually referred to as program deforestation or program fusion (Wadler 1990; Gill et al. 1993), which is based on a certain set of calculation laws that can merge computations and thus avoid the construction of intermediate data structures. By the application of this technique a program $h = f \circ g$ is then transformed into an equivalent program that does not construct any intermediate structure.

In this tutorial we study a particular approach to the fusion technique known as shortcut fusion (Gill et al. 1993; Takano and Meijer 1995; Fernandes et al. 2007). The laws we present assume that it is possible to express the functions f and g, that occur in a composition $f \circ g$, in terms of well-known, higher-order, recursion patterns. As we will see later, while the applicability of such laws is certainly not universal, the fact is that the state of the art in shortcut fusion techniques can already deal with an extensive set of programs.

A remarkable observation that can be made about the programs that we calculate is that they often rely on either higher-order functions or on lazyness to be executed. So, these constructions, that Hughes (1984) identified as being essential to modularity, are in fact not only useful to increase modularity, but

[6] Program composition $(f \circ g)\ x$ is interpreted as $f\ (g\ x)$, and is left associative, i.e., $f \circ g \circ h = (f \circ g) \circ h$.

they can also be explored for reasoning about modular programs, including to increase their efficiency.

This paper is organized as follows. In Section 2 we introduce the programming language that is used in the examples throughout the paper, that is, Haskell, and review the concepts of that language that are necessary to follow our materials. In Section 3, we present concrete shortcut fusion rules that are used to achieve deforestation of intermediate structures in small examples that are also introduced. These rules are concrete instances of generic ones, whose definition we present in Section 4. In Section 5 we study the application of fusion rules to a realistic example, and in Section 6 we conclude the paper.

2 A Gentle Introduction to Haskell

In this tutorial, all the code that we present is written in the Haskell programming language (Peyton Jones et al. 1999; Peyton Jones 2003). Haskell is a modern, polymorphic, statically-typed, lazy, and pure functional programming language.

In this section, we introduce the constructions that are necessary for the reader to follow our tutorial. While some familiarity with functional programming is expected, we hope that the reader does not need to be proficient in Haskell to understand such materials.

Haskell provides a series of predefined types such as *Int* (typifying natural numbers), *Float* (typifying floating point numbers) or *Char* (typifying single characters), and natural solutions to well known problems can readily be expressed. This is the case of the following (recursive) implementation of *factorial*, that directly follows from its mathematical definition:

$$
\begin{aligned}
&\mathit{factorial} :: \mathit{Int} \to \mathit{Int}\\
&\mathit{factorial}\ 0 = 1\\
&\mathit{factorial}\ n = n * \mathit{factorial}\ (n-1)
\end{aligned}
$$

In Haskell, type judgments are of the form $e :: \tau$ and state that an expression e has type τ. In the case of *factorial* the type $\mathit{Int} \to \mathit{Int}$ indicates that it is a function from integers to integers.

Besides working with predefined types, we also have ways of constructing more complex data-types based on existent ones (either provided by Haskell itself or defined by the user). Indeed, in its prelude Haskell already defines (polymorphic) lists as:

$$\mathbf{data}\ [a] = [\,] \mid a : [a]$$

A concrete list of elements of type a, which is of type $[a]$, is then either empty, $[\,]$, or it has an element of type a followed by a list $[a]$. By polymorphic we mean that we are able of creating a list of any type, which is achieved by

instantiating the type variable a with that type. For example, the type *String* is simply defined as,[7]

$$\textbf{type } \mathit{String} = [\mathit{Char}]$$

and concrete lists can easily be defined:

$$l_1 :: \mathit{String}$$
$$l_1 = [\texttt{'c'}, \texttt{'e'}, \texttt{'f'}, \texttt{'p'}]$$

$$l_2 :: [\mathit{Int}]$$
$$l_2 = [2, 0, 1, 5]$$

For clarity, we have explicitly annotated l_1 and l_2 with their corresponding types (l_1 is a lisf of characters, or a *String*, and l_2 is a list on integers), but this is not strictly necessary. The definitions of l_1 e l_2 are simply syntactic sugar for the following definitions:

$$l_1 = \texttt{'c'} : \texttt{'e'} : \texttt{'f'} : \texttt{'p'} : [\,]$$

$$l_2 = 2 : 0 : 1 : 5 : [\,]$$

Notice that the operator : for constructing lists (usually pronounced cons) is an infix operator. It can be turned into a prefix operator by using parenthesis, i.e., by writing (:). Hence, $5 : [\,]$ and (:) 5 [] are equivalent expressions. The same can be done with any other infix operator.

This means that l_1 and l_2 can also be expressed as:

$$l_1 = (:)\ \texttt{'c'}\ ((:)\ \texttt{'e'}\ ((:)\ \texttt{'f'}\ ((:)\ \texttt{'p'}\ [\,])))$$

$$l_2 = (:)\ 2\ ((:)\ 0\ ((:)\ 1\ ((:)\ 5\ [\,])))$$

In this paper, we will use all these different notations for lists interchangeably.

Regarding l_1, and since it is a string, it could alternatively have been defined as:

$$l_1 = \texttt{"cefp"}$$

Regarding the manipulation of lists, we normally use its constructors [] and (:) to pattern match on a given list. Indeed, a function f defined as:

$$f\ [\,] = f_1$$
$$f\ (h : t) = f_2$$

defines that its behavior on an empty list is that of f_1 and that its behavior on a list whose first element is h and whose tail is t is that of f_2. Of course, h and t can be used in the definition of f_2.

[7] Note that type synonyms are declared with the keyword **type** and that new data-types are declared with **data**.

As an example, we may define the following function to compute the sum of all elements in a list of integers.[8]

$$sum\ [\,] = 0$$
$$sum\ (h : t) = h + sum\ t$$

Regarding this implementation, already in 1990 Hughes pinpointed that only the value 0 and the operation + are specific to the computation of *sum*. Indeed, if we replace 0 by 1 and + by $*$ in the above definition, we obtain a function that multiplies all the elements in a list of integers:

$$product\ [\,] = 1$$
$$product\ (h : t) = h * product\ t$$

This suggests that abstract/generic patterns for processing lists are useful. And in fact all modern functional languages allow the definition of such patterns relying on the concept of higher-order functions.

In Haskell functions are first-class citizens, in the sense that they can be passed as arguments to other functions and they can be the result produced by other functions. With this possibility in mind, we may define a well-know pattern named *fold*:[9]

$$fold :: (b, (a, b) \to b) \to [a] \to b$$
$$fold\ (nil, cons) = f$$
$$\textbf{where}\ f\ [\,] = nil$$
$$f\ (x : xs) = cons\ (x, f\ xs)$$

With this pattern at hand, we may now give unified, modular, definitions for *sum* and *product*:[10]

$$sum = fold\ (0, uncurry\ (+))$$

$$product = fold\ (1, uncurry\ (*))$$

Exercise 1. Implement a function $sort :: [Float] \to [Float]$ that sorts all the elements in a list of floating point numbers. For this, you can rely on function $insert :: Float \to [Float] \to [Float]$ that inserts a number in a list whose elements are in ascending order so that the ordering is preserved.

---

[8] This function is actually included in the Haskell *Prelude*.

[9] This definition of *fold* slightly differs from the definition of $foldr :: (a \to b \to b) \to b \to [a] \to b$ provided by Haskell, in that we rely on uncurried functions and we have changed the order of the expected arguments. We give this definition here as it will simplify our presentation later.
Also, for simplicity, we have omitted an argument on both sides of the equation $fold\ (nil, cons) = f$, that could have equally been given the definition $fold\ (nil, cons)\ l = f\ l$.

[10] *uncurry* takes a function $f :: a \to b \to c$ and produces a function $f' :: (a, b) \to c$.

$$
\begin{aligned}
&\mathit{insert} :: \mathit{Float} \to [\mathit{Float}] \to [\mathit{Float}] \\
&\mathit{insert}\ n\ [\,] = [n] \\
&\mathit{insert}\ n\ (h : t) = \textbf{if}\ (n < h) \\
&\qquad\qquad\qquad\quad \textbf{then}\ n : h : t \\
&\qquad\qquad\qquad\quad \textbf{else}\ h : \mathit{insert}\ n\ t
\end{aligned}
$$

a) Propose a(n explicitly) recursive solution for *sort*.

b) Re-implement your previous solution in terms of a *fold*. □

Now, suppose that we want to increment all elements of a list of integers by a given number:

$$
\begin{aligned}
&\mathit{increment} :: ([\mathit{Int}], \mathit{Int}) \to [\mathit{Int}] \\
&\mathit{increment}\ ([\,], \_) = [\,] \\
&\mathit{increment}\ (h : t, z) = (h + z) : \mathit{increment}\ (t, z)
\end{aligned}
$$

Just by looking at the types involved, we may see that it is not possible to express *increment* in terms of a *fold*. Indeed, *fold* allows us to define functions of type $[a] \to b$, while *increment* is of type $([\mathit{Int}], \mathit{Int}) \to \mathit{Int}$, and it is not possible to match $[a]$ with $([\mathit{Int}], \mathit{Int})$.

Still, the *fold* pattern can be generalized in many ways, one of them to deal with functions of type $([a], z) \to b$. For this we may define a new pattern, called *pfold*, that also traverses a list in a systematic fashion, but does so taking into account the additional parameter of type z:

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

Now, we are in conditions to give *increment* a modular definition, as we have done for *sum* and *product*:

$$
\begin{aligned}
&\mathit{increment} = \mathit{pfold}\ (\mathit{hnil}, \mathit{hcons}) \\
&\quad \textbf{where}\ \mathit{hnil}\ \_ = [\,] \\
&\qquad\qquad \mathit{hcons}\ ((h, r), z) = (h + z) : r
\end{aligned}
$$

Besides working with lists, in this tutorial we will often need to use binary trees, whose elements are in their leaves and are of type integer. For this purpose, we may define the following Haskell data-type:

$$
\begin{aligned}
&\textbf{data}\ \mathit{LeafTree} = \mathit{Leaf}\ \mathit{Int} \\
&\qquad\qquad\qquad\quad \mid\ \mathit{Fork}\ (\mathit{LeafTree}, \mathit{LeafTree})
\end{aligned}
$$

Similarly to what we have defined for lists, we may now define *fold* and *pfold* for leaf trees, that we will name fold_T and pfold_T, respectively.

$$
\begin{aligned}
&fold_T :: (Int \to a, (a, a) \to a) \to LeafTree \to a \\
&fold_T\ (h_1, h_2) = f_T \\
&\quad \textbf{where}\ f_T\ (Leaf\ n) = h_1\ n \\
&\qquad\quad f_T\ (Fork\ (l, r)) = h_2\ (f_T\ l, f_T\ r)
\end{aligned}
$$

$$
\begin{aligned}
&pfold_T :: ((Int, z) \to a, ((a, a), z) \to a) \to (LeafTree, z) \to a \\
&pfold_T\ (h_1, h_2) = p_T \\
&\quad \textbf{where}\ p_T\ (Leaf\ n, z) = h_1\ (n, z) \\
&\qquad\quad p_T\ (Fork\ (l, r), z) = h_2\ ((p_T\ (l, z), p_T\ (r, z)), z)
\end{aligned}
$$

And we can express the recursive function *tmin*, that computes the minimum value of a tree, [11]

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

in terms of a fold for leaf trees:

$$tmin = fold_T\ (id, uncurry\ min)$$

Similarly, we can express the recursive function *replace*, that places a concrete value in all the leaves of a tree:

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

in terms of a pfold for leaf trees:

$$replace = pfold_T\ (Leaf \circ \pi_2, Fork \circ \pi_1)$$

In the above implementation, we have used functions π_1 and π_2, whose (type-parametric) definition is as follows:

$$
\begin{aligned}
&\pi_1 :: (a, b) \to a \\
&\pi_1\ (a, b) = a
\end{aligned}
$$

$$
\begin{aligned}
&\pi_2 :: (a, b) \to b \\
&\pi_2\ (a, b) = b
\end{aligned}
$$

Now, suppose that we want to construct a function that replaces all the leaves in a leaf tree by the minimum leaf of that tree, a problem widely know as *repmin* (Bird 1984). An example of this transformation is given in Figure 1.

We may combine the above implementations of *replace* and *tmin* in a simple way to obtain a solution to *repmin*:

$$repmin\ t = replace\ (t, tmin\ t)$$

[11] Given two numbers, *min* will compute the minimum of both numbers.

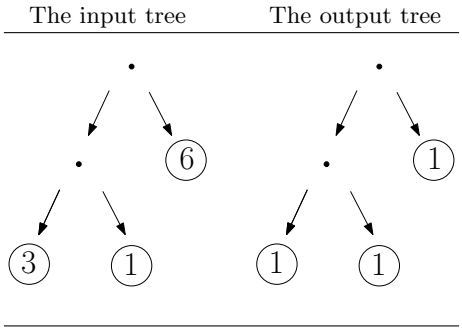

Fig. 1. An example of the use of *repmin*.

Regarding this implementation, Bird (1984) notices that t is traversed twice, and that in a lazy functional language this is not strictly necessary. In fact, Bird shows how to remove this multiple traversals by deriving circular programs from programs such as *repmin*.

Circular programs hold circular definitions, in which arguments in a function call depend on results of that same call. That is, they contain definitions such as:

$$(..., x, ...) = f\ (..., x, ...)$$

From the above *repmin* definition, Bird derives the following circular program:[12]

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

Although this circular definition seems to induce both a cycle and non-termination of this program, the fact is that using a *lazy* language, the *lazy* evaluation machinery is able to determine, at runtime, the right order to evaluate this circular definition. This reinforces the power of lazy evaluation strategy.

Deriving circular programs, however, is not the only way to eliminate multiple traversals of data structures. In particular, the straightforward *repmin* solution shown earlier may also be transformed, by the application of a well-known technique called lambda-abstraction (Pettorossi and Skowron 1987), into a higher-order program.

[12] 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).

This reinforces the power of higher-order features, and as a result, we obtain[13]:

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

Regarding this new version of *repmin*, we may notice that it is a higher-order program, since nt, the first component of the result produced by the call $\mathit{repm}\ t$, is now a function. Later, nt 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.

3 Shortcut Fusion

Having introduced the concepts of Haskell that are necessary to understand the remainder of this paper, in this section we introduce shortcut fusion by example.

We start by introducing simple programming problems whose solutions can be expressed as programs that rely on intermediate structures. That is, we consider programs such as:

$$
\begin{aligned}
&\mathit{prog} :: a \to c\\
&\mathit{prog} = \mathit{cons} \circ \mathit{prod}
\end{aligned}
$$

Then, we present specific shortcut fusion rules that are applicable to each such example.

In Section 3.1, we demonstrate with programs whose producer and consumer functions are of type $\mathit{prod} :: a \to b$ and $\mathit{cons} :: b \to c$, respectively.

In Section 3.2, we extend the applicability of such rules, considering programs whose producer and consumer functions are of type $\mathit{prod} :: a \to (b, z)$ and $\mathit{cons} :: (b, z) \to c$, respectively.

3.1 Standard Shortcut Fusion

In order to illustrate how deforestation can be achieved in practice, let us start by considering an alternative to the *factorial* implementation given in Section 2.

For a given, assumed positive, number n, this alternative creates a list with all the integers from n down to 1:

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

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

The elements of such a list then need to be multiplied, which can be achieved with function *product* that we have also already seen earlier:

$$\begin{aligned}&product :: [Int] \to Int\\&product\ [\,] = 1\\&product\ (h : t) = h * product\ t\end{aligned}$$

Now, in order to implement *factorial* it suffices to combine these functions appropriately:

$$\begin{aligned}&factorial :: Int \to Int\\&factorial\ n = product\ (down\ n)\end{aligned}$$

which is equivalent to:

$$factorial\ n = (product \circ down)\ n$$

or simply:

$$factorial = product \circ down$$

While this implementation is equivalent to the original one, it is creating an intermediate list of numbers which is clearly not necessary, and this affects its running performance. Of course, in this simple example, the original solution is at least as simple to implement as this alternative one, but in general, decomposing a problem in the sub-problems that constitute it contributes to increasing modularity and facilitates the programming and debugging tasks.

Regarding the above implementation of *factorial*, we see that if we ask for the value of *factorial* 0, an empty list is produced by *down*, which is replaced by the value 1, as defined in *product*. Similarly, we see that for *factorial* n, the list $n : down\ (n-1)$ is created which is later transformed into the expression $n * product\ (down\ (n-1))$. So, in this simple example, we can straightforwardly reason about the definition of a version of *factorial* that does not construct the intermediate list.

In order to derive this more efficient version of *factorial* in a systematic way we may proceed using shortcut fusion, and namely the *fold/build* rule(Gill et al. 1993; Takano and Meijer 1995; Gill 1996) that can be stated for the case when a list is the intermediate structure used to compose two functions:

Law 1 (FOLD/BUILD RULE FOR LISTS)

$$fold\ (h_1, h_2) \circ build\ g = g\ (h_1, h_2)$$

where

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

Function *build* allows us to abstract from the concrete list constructors that are used to build the intermediate structure. This abstraction is realized in function g. In this way, and given that $fold\ (h_1, h_2)$ replaces, in a list, all the occurrences of $[\,]$ by h_1 and all the occurrences of $(:)$ by h_2, deforestation proceeds by anticipating this replacement. This is precisely what is achieved in the definition $g\ (h_1, h_2)$.

In order to apply Law 1 to *factorial*, we first need to express *down* in terms of *build* and *product* in terms of *fold*:

$$product = fold\ (1, uncurry\ (*))$$

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

We then follow a simple equational reasoning to obtain:

$$\begin{aligned} &\quad factorial \\ &= \qquad \{\ \text{definition of } factorial\ \} \\ &\quad product \circ down \\ &= \qquad \{\ \text{definition of } product \text{ and } down\ \} \\ &\quad fold\ (1, uncurry\ (*)) \circ build\ g \\ &= \qquad \{\ \text{Law 1}\ \} \\ &\quad g\ (1, uncurry\ (*)) \end{aligned}$$

Finally, by inlining the above definition, we obtain the original formulation of *factorial*:

$$\begin{aligned} factorial\ 0 &= g\ (1, uncurry\ (*))\ 0 \\ &= 1 \\ factorial\ n &= g\ (1, uncurry\ (*))\ n \\ &= uncurry\ (*)\ (n, factorial\ (n-1)) \\ &= n * factorial\ (n-1) \end{aligned}$$

In the following exercise, we encourage the reader to apply Law 1 to another concrete example.

Exercise 2. Imagine that you are given the list of grades (the scale is $[0\,..\,10]$) obtained by a set of students in an university course, such as:

$$l = [(6,8), (4,5), (9,7)]$$

Each pair holds the grades of a particular student; its first component holds the grade obtained by the student in the exam, and its second component the grade obtained in the project.

Implement a function $average :: [(Float, Float)] \to [Float]$ that computes the average of the grades obtained by each student. As an example, $average\ l$ is expected to produce the list $[7.0, 4.5, 8.0]$.

a) Propose a(n explicitly) recursive solution for *average*.

b) Re-implement your previous solution in terms of a *build*.

c) Obtain a function $sort_{avgs} :: [(Float, Float)] \to [Float]$ simply by composing functions *sort* (from Exercise 1) and *average*.

d) Notice that function $sort_{avgs}$ relies on an intermediate structure of type $[Float]$, which can be eliminated. Apply Law 1 to obtain a deforested program, say $dsort_{avgs}$ that is equivalent to $sort_{avgs}$. □

Law 1 deals specifically with programs such as *factorial*, that rely on an intermediate list to convey results between the producer and the consumer functions.

A similar reasoning can, however, be made for programs relying on arbitrary data types as intermediate structures. This is, for example, the case of programs that need to construct an intermediate *LeafTree*, and Law 2, as follows, deals precisely with such type of programs.

Law 2 (FOLD/BUILD RULE FOR LEAF TREES)

$$fold_T\ (h_1, h_2) \circ build_T\ g = g\ (h_1, h_2)$$

where

$$build_T :: (\forall\ a\ .\ (Int \to a, (a, a) \to a) \to c \to a) \to c \to LeafTree$$
$$build_T\ g = g\ (Leaf, Fork)$$

As an example, we can use this law to fuse the following program, that computes the minimum value of a *mirrored* leaf tree.

$$tmm = tmin \circ mirror$$

$$mirror :: LeafTree \to LeafTree$$
$$mirror\ (Leaf\ n) = Leaf\ n$$
$$mirror\ (Fork\ (l, r)) = Fork\ (mirror\ r, mirror\ l)$$

Since we had already expressed *tmin* in terms of $fold_T$ in Section 2, as

$$tmin = fold_T\ (id, uncurry\ min)$$

we now need to express *mirror* in terms of $build_T$:

$$\begin{aligned} mirror &= build_T\ g \\ &\textbf{where}\ g\ (leaf, fork)\ (Leaf\ n) = leaf\ n \\ &\qquad g\ (leaf, fork)\ (Fork\ (l, r)) = fork\ (g\ (leaf, fork)\ r, \\ &\qquad\qquad\qquad\qquad\qquad\qquad g\ (leaf, fork)\ l) \end{aligned}$$

Finally, by Law 2 we have that

$$tmm = g\ (id, uncurry\ min)$$

Inlining, we have

$$tmm\ (Leaf\ n) = n$$
$$tmm\ (Fork\ (l, r)) = min\ (tmm\ r)\ (tmm\ l)$$

As expected, this function does not construct the intermediate mirror tree.

3.2 Extended Shortcut Fusion

In this section, we move on to study shortcut fusion for programs defined as the composition of two functions that, besides an intermediate structure, need to communicate using an additional parameter. That is, we focus on programs such as $prog = cons \circ prod$, where $prod :: a \to (b, z)$ and $cons :: (b, z) \to c$.

We start by deriving circular programs from such type of function compositions and then we derive higher-order programs from the same programs.

We illustrate with examples relying on intermediate structures of type *LeafTree* only. This is because a realistic example based on intermediate lists will be given in Section 5.

Deriving Circular Programs We start by introducing a new law, whose generic version was originally provided by Fernandes et al. (2007), and which is similar to Law 2. This law, however, applies to the extended form of function compositions we are now considering.

Law 3 (PFOLD/BUILDP RULE FOR LEAF TREES) [14]

$$\begin{array}{l} pfold_T\ (h_1, h_2) \circ buildp_T\ g\ \$\ c = v \\ \quad \textbf{where}\ (v, \boxed{z}) = g\ (k_1, k_2)\ c \\ \qquad\quad k_1\ n = h_1\ (n, \boxed{z}) \\ \qquad\quad k_2\ (l, r) = h_2\ ((l, r), \boxed{z}) \end{array}$$

where

$$buildp_T :: (\forall\ a\ .\ (Int \to a, (a, a) \to a) \to c \to (a, z)) \to c \to (LeafTree, z)$$
$$buildp_T\ g = g\ (Leaf, Fork)$$

Notice that the consumer is now assumed to be given in terms of a $buildp_T$ and that the consumer function is now expected to be given as a *pfold*. This is precisely to accommodate the additional parameter of type z.

To illustrate the application of this law in practice, recall the *repmin* problem that was introduced in Section 2 and its initial solution:

$$repmin\ t = replace\ (t, tmin\ t)$$

[14] We have used $(\$) :: (a \to b) \to a \to b$ in the expression $pfold_T\ (h_1, h_2) \circ buildp_T\ g\ \$\ c$ to avoid the use of parenthesis. The same expression could be defined as $(pfold_T\ (h_1, h_2) \circ buildp_T\ g)\ c$.

An alternative solution to such problem can be given by an explicit composition of two functions, where the first computes the minimum of a tree and the second replaces all leaf values by such minimum:[15]

$$
\begin{aligned}
&\mathit{transform} :: \mathit{LeafTree} \to \mathit{LeafTree}\\
&\mathit{transform} = \mathit{replace} \circ \mathit{tmint}
\end{aligned}
$$

where

$$
\begin{aligned}
&\mathit{tmint} :: \mathit{LeafTree} \to (\mathit{LeafTree}, \mathit{Int})\\
&\mathit{tmint}\ (\mathit{Leaf}\ n) = (\mathit{Leaf}\ n, n)\\
&\mathit{tmint}\ (\mathit{Fork}\ (l, r)) = (\mathit{Fork}\ (l', r'), \mathit{min}\ n_1\ n_2)\\
&\qquad \textbf{where}\ (l',\ n_1) = \mathit{tmint}\ l\\
&\qquad\qquad\ \ (r',\ n_2) = \mathit{tmint}\ r
\end{aligned}
$$

and *replace* remains unchanged:

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

To apply the rule, first we have to express *replace* and *tmint* in terms of pfold_T and buildp_T for leaf trees, respectively:

$$
\begin{aligned}
&\mathit{replace} = \mathit{pfold}_T\ (\mathit{Leaf} \circ \pi_2, \mathit{Fork} \circ \pi_1)
\end{aligned}
$$

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

Therefore, by applying Law 3 we get:

$$
\begin{aligned}
&\mathit{transform}\ t = nt\\
&\quad \textbf{where}\ (nt, \boxed{\mathrm{m}}) = g\ (k_1, k_2)\ t\\
&\qquad\quad\ k_1\ \_ = \mathit{Leaf}\ \boxed{\mathrm{m}}\\
&\qquad\quad\ k_2\ (l, r) = \mathit{Fork}\ (l, r)
\end{aligned}
$$

Inlining, we obtain the definition we showed previously in Section 2:

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

[15] Here, we needed to introduce an explicit function composition since one is needed in order to apply the rule. In practice, intermediate structures need to be more informative that the input ones, so the latter *must* be *bigger* than the former, and we are *forced* to define and manipulate intermediate structures. This means that solutions as function compositions are natural ones.

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

Next, we propose another concrete example where Law 3 is applicable.

Exercise 3. Our goal is to implement a function $\mathit{transform} = \mathit{add} \circ \mathit{convert}$, that takes a list of integers and produces a balanced leaf tree whose elements are the elements of the input list incremented by their sum. So, if the input list is $[1, 2, 3]$ we want to produce a balanced leaf tree whose elements are 7, 8 and 9.

a) Implement a function $\mathit{convert} :: [\mathit{Int}] \to (\mathit{LeafTree}, \mathit{Int})$ that produces a height-balanced leaf tree containing all the elements of a list. Function *convert* must also produce the sum of all elements of the list.

b) Implement a function $\mathit{add} :: (\mathit{LeafTree}, \mathit{Int}) \to \mathit{LeafTree}$ that adds to all the elements of a leaf tree a given number.

c) Write *convert* in terms of buildp_T and *add* in terms of pfold_T.

d) Apply Law 3 to derive a circular program that does not construct the intermediate leaf tree. □

Deriving Higher-Order Programs Next, we introduce a new law, Law 4, that applies to the same type of programs as Law 3, but that instead of deriving circular programs derives higher-order ones. The specific case of this law that deals with programs relying on intermediate lists instead of leaf trees was originally given by Voigtländer (2008) and its generic formulation was later given by Pardo et al. (2009).

Law 4 (HIGHER-ORDER PFOLD/BUILDP RULE FOR LEAF TREES)

$$
\begin{aligned}
&\mathit{pfold}_T\ (h_1, h_2) \circ \mathit{buildp}_T\ g\ \$\ c = f\ z \\
&\quad \mathbf{where}\ (f, z) = g\ (\varphi_{h_1}, \varphi_{h_2})\ c \\
&\qquad\quad \varphi_{h_1}\ n = \lambda z \to h_1\ (n, z) \\
&\qquad\quad \varphi_{h_2}\ (l, r) = \lambda z \to h_2\ ((l\ z, r\ z), z)
\end{aligned}
$$

where

$$
\begin{aligned}
&\mathit{buildp}_T :: (\forall\ a\ .\ (\mathit{Int} \to a, (a, a) \to a) \to c \to (a, z)) \to c \to (\mathit{LeafTree}, z) \\
&\mathit{buildp}_T\ g = g\ (\mathit{Leaf}, \mathit{Fork})
\end{aligned}
$$

To see an example of the application of Law 4, we consider again the straightforward solution to the *repmin* problem:

$$\mathit{transform} = \mathit{replace} \circ \mathit{tmint}$$

$$\mathit{replace} = \mathit{pfold}_T\ (\mathit{Leaf} \circ \pi_2, \mathit{Fork} \circ \pi_1)$$

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

In order to apply Law 4 to *transform*, we need the expressions of φ_{h_1} and φ_{h_2}. For φ_{h_1}, we have that:

$$
\begin{aligned}
&\varphi_{h_1}\ n \\
=&\quad \{\ \text{definition of } \varphi_{h_1} \text{ in Law 4}\ \} \\
&\lambda z \to h_1\ (n, z) \\
=&\quad \{\ \text{definition of } h_1\ \} \\
&\lambda z \to (\mathit{Leaf} \circ \pi_2)\ (n, z) \\
=&\quad \{\ \text{definition of function composition, definition of } \pi_2\ \} \\
&\lambda z \to \mathit{Leaf}\ z
\end{aligned}
$$

and similarly for φ_{h_2}, we obtain that $\varphi_{h_2}\ (l, r) = \lambda z \to \mathit{Fork}\ (l\ z, r\ z)$.

Then, by direct application of Law 4 to *transform*, we obtain:

$$
\begin{aligned}
&\mathit{transform}\ t = \mathit{nt}\ m \\
&\quad \textbf{where}\ (\mathit{nt}, m) = g\ (\varphi_{h_1}, \varphi_{h_2})
\end{aligned}
$$

Inlining the above definition, we obtain the higher-order solution to *repmin* that we had already presented in Section 2:

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

Exercise 4. Recall the solution to $\mathit{transform} = \mathit{add} \circ \mathit{convert}$ of Exercise 3.

a) Apply Law 4 to derive a higher-order program that does not construct the intermediate leaf tree. □

4 Generalized Shortcut Fusion

In the previous section, we have used concrete examples to demonstrate the applicability and interest of different types of shortcut fusion rules. In this section, we show that the concrete rules we have introduced before can actually be given uniform, generic formulations, that are applied to a wide range of programs characterized in terms of certain program schemes. The generic formulations

of the rules described here are parametric in the structure of the intermediate data-type involved in the function composition to be transformed.

Throughout the section we shall assume we are working in the context of a lazy functional language with a *cpo* (Complete Partial Order) semantics, in which types are interpreted as pointed cpos (complete partial orders with a least element $\perp$) and functions are interpreted as continuous functions between pointed cpos.

While this semantics closely resembles the semantics of Haskell, for now we do not consider lifted cpos. That is, unlike the semantics of Haskell, we do not consider lifted products and function spaces. The precise implications of these semantics differences are studied in Section 4.5.

As usual, a function f is said to be *strict* if it preserves the least element, i.e. $f \perp = \perp$.

4.1 Data-types

The structure of data-types 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:

$$map_F\ id = id \tag{1}$$

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

A standard example of a functor is that formed by the list type constructor and the well-known *map* function, which applies a function to the elements of a list, building a new list with the results.

$$\begin{aligned} &map && :: (a \to b) \to [a] \to [b] \\ &map\ f\ [\,] && = [\,] \\ &map\ f\ (a : as) && = f\ a : map\ f\ as \end{aligned}$$

Another example of a functor is the product functor, which is a case of a bifunctor, a functor on two arguments. On types its action is given by the type constructor for pairs. On functions its action is defined by:

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

Semantically, we assume that pairs are interpreted as the cartesian product of the corresponding cpos. Associated with the product we can define the following functions, corresponding to the projections and the split function:

$$\begin{aligned} &\pi_1 :: (a, b) \to a \\ &\pi_1\ (a, b) = a \end{aligned}$$

$$\begin{aligned} &\pi_2 :: (a, b) \to b \\ &\pi_2\ (a, b) = b \end{aligned}$$

$$(\vartriangle) :: (c \to a) \to (c \to b) \to c \to (a, b)$$
$$(f \ \vartriangle \ g) \ c = (f \ c, g \ c)$$

Among other properties, it holds that

$$f \circ \pi_1 = \pi_1 \circ (f \ \times \ g) \tag{3}$$

$$g \circ \pi_2 = \pi_2 \circ (f \ \times \ g) \tag{4}$$

$$f = ((\pi_1 \circ f) \vartriangle (\pi_2 \circ f)) \tag{5}$$

Another case of a bifunctor is the sum functor, which corresponds to the disjoint union of types. Semantically, we assume that sums are interpreted as the separated sum of the corresponding cpos.

data $a + b = \mathit{Left} \ a \mid \mathit{Right} \ b$

$$(+) :: (a \to c) \to (b \to d) \to (a + b) \to (c + d)$$
$$(f + g) \ (\mathit{Left} \ a) = \mathit{Left} \ (f \ a)$$
$$(f + g) \ (\mathit{Right} \ b) = \mathit{Right} \ (g \ b)$$

Associated with the sum we can define the case analysis function, which has the property of being strict in its argument of type $a + b$:

$$(\triangledown) :: (a \to c) \to (b \to c) \to (a + b) \to c$$
$$(f \triangledown g) \ (\mathit{Left} \ a) = f \ a$$
$$(f \triangledown g) \ (\mathit{Right} \ b) = g \ b$$

Product and sum can be generalized to n components in the obvious way.

We consider declarations of data-types of the form[16]:

$$\textbf{data} \ \tau \ (\alpha_1, \cdots, \alpha_m) = C_1 \ (\tau_{1,1}, \cdots, \tau_{1,k_1}) \ \mid \cdots \mid \ C_n \ (\tau_{n,1}, \cdots, \tau_{n,k_n})$$

where each $\tau_{i,j}$ is restricted to be a constant type (like *Int* or *Char*), a type variable α_t, a type constructor D applied to a type $\tau'_{i,j}$ or $\tau \ (\alpha_1, \cdots, \alpha_m)$ itself. Data-types of this form are usually called regular. The derivation of a functor that captures the structure of the data-type essentially proceeds as follows: alternatives are regarded as sums ($\mid$ is replaced by $+$) and constructors C_i are omitted. Every $\tau_{i,j}$ that consists of a type variable α_t or of a constant type remain unchanged and occurrences of $\tau \ (\alpha_1, \cdots, \alpha_m)$ are substituted by a type variable a in every $\tau_{i,j}$. In addition, the unit type () is placed in the positions corresponding to constant constructors (like e.g. the empty list constructor). As a result, we obtain the following type constructor F:

$$F \, a = (\sigma_{1,1}, \cdots, \sigma_{1,k_1}) + \cdots + (\sigma_{n,1}, \cdots, \sigma_{n,k_n})$$

where $\sigma_{i,j} = \tau_{i,j}[\tau \ (\alpha_1, \cdots, \alpha_m) := a]$[17]. The body of the corresponding mapping function $\mathit{map}_F :: (a \to b) \to (F \ a \to F \ b)$ is similar to that of $F \ a$, with the

[16] For simplicity we shall assume that constructors in a data-type declaration are declared uncurried.

[17] By $s[t := a]$ we denote the replacement of every occurrence of t by a in s.

difference that the occurrences of the type variable a are replaced by a function $f :: a \to b$:

$$map_F f = g_{1,1} \times \cdots \times g_{1,k_1} \; + \cdots + \; g_{n,1} \times \cdots \times g_{n,k_n}$$

with

$$g_{i,j} = \begin{cases} f & \text{if } \sigma_{i,j} = a \\ id & \text{if } \sigma_{i,j} = t, \text{ for some type } t \\ & \text{or } \sigma_{i,j} = a', \text{ for some type variable } a' \text{ other than } a \\ map_D \; g'_{i,j} & \text{if } \sigma_{i,j} = D \; \sigma'_{i,j} \end{cases}$$

where map_D represents the *map* function $map_D :: (a \to b) \to (D \; a \to D \; b)$ corresponding to the type constructor D.

For example, for the type of leaf trees

$$\begin{aligned} &\textbf{data } \mathit{LeafTree} = \mathit{Leaf} \; \mathit{Int} \\ &\qquad\qquad\quad | \; \mathit{Fork} \; (\mathit{LeafTree}, \mathit{LeafTree}) \end{aligned}$$

we can derive a functor T given by

$$T \; a = \; \mathit{Int} + (a, a)$$

$$\begin{aligned} &map_T \quad :: \; (a \to b) \to (T \; a \to T \; b) \\ &map_T \; f = id + f \; \times \; f \end{aligned}$$

The functor that captures the structure of the list data-type needs to reflect the presence of the type parameter:

$$L_a \; b = () + (a, b)$$

$$\begin{aligned} &map_{L_a} \quad :: \; (b \to c) \to (L_a \; b \to L_a \; c) \\ &map_{L_a} \; f = id + id \; \times \; f \end{aligned}$$

This functor reflects the fact that lists have two constructors: one is a constant and the other is a binary operation.

Every recursive data-type is then understood as the least fixed point of the functor F that captures its structure, i.e. as the least solution to the equation $\tau \cong F \; \tau$. We will denote the type corresponding to the least solution as μF. The isomorphism between μF and $F \; \mu F$ is provided by the strict functions $in_F :: F \; \mu F \to \mu F$ and $out_F :: \mu F \to F \; \mu F$, each other inverse. Function in_F packs the constructors of the data-type while function out_F packs its destructors. Further details can be found in (Abramsky and Jung 1994; Gibbons 2002).

For instance, in the case of leaf trees we have that $\mu T = \mathit{LeafTree}$ and

$$\begin{aligned} &in_T :: \; T \; \mathit{LeafTree} \to \mathit{LeafTree} \\ &in_T = \; \mathit{Leaf} \; \triangledown \; \mathit{Fork} \end{aligned}$$

$$\begin{aligned} &out_T :: \mathit{LeafTree} \to T \; \mathit{LeafTree} \\ &out_T \; (\mathit{Leaf} \; n) = \mathit{Left} \; n \end{aligned}$$

$$out_T\ (Fork\ (l, r)) = Right\ (l, r)$$

4.2 Fold

Fold (Bird and de Moor 1997; Gibbons 2002) 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 (Bird 1998).

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

$$f \circ in_F = h \circ map_F\ f$$

Because out_F is the inverse of in_F, this is the same as:

$$\begin{aligned} &fold\ \ \ :: (F\ a \to a) \to \mu F \to a \\ &fold\ h = h \circ map_F\ (fold\ h) \circ out_F \end{aligned}$$

A function $h :: F\ a \to a$ is called an *F-algebra*[18]. 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 $h :: F\ a \to a$ 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 $f \circ h = k \circ map_F\ f$. Notice that $fold\ h$ is a homomorphism between the algebras in_F and h.

The concrete instance of *fold* for the case when $F = T$ and $\mu F = LeafTree$ is given by the definition we had already presented in Section 2:

$$\begin{aligned} &fold_T :: (Int \to a, (a, a) \to a) \to LeafTree \to a \\ &fold_T\ (h_1, h_2) = f_T \\ &\quad \textbf{where}\ f_T\ (Leaf\ n) = h_1\ n \\ &\qquad\qquad f_T\ (Fork\ (l, r)) = h_2\ (f_T\ l, f_T\ r) \end{aligned}$$

In the same way, the concrete instance of *fold* for the case when $F = L_a$ and $\mu F = [\,a\,]$ is the definition we had also given in Section 2:

$$\begin{aligned} &fold :: (b, (a, b) \to b) \to [\,a\,] \to b \\ &fold\ (nil, cons) = f \\ &\quad \textbf{where}\ f\ [\,] = nil \\ &\qquad\qquad f\ (x : xs) = cons\ (x, f\ xs) \end{aligned}$$

Notice that, for simplicity, we are overloading *fold* both as the name of the generic recursion pattern and its instance for lists. This will also be the case for other constructions given in this paper, but it should be clear from every context whether we are referring to the generic or the specific case.

[18] When showing specific instances of fold for concrete data-types, we will write the operations in an algebra $h_1 \triangledown \cdots \triangledown h_n$ in a tuple $(h_1, \ldots, h_n)$.

4.3 The fold/build rule

Fold enjoys many algebraic laws that are useful for program transformation (Augusteijn 1998). A well-known example is *shortcut fusion* (Gill et al. 1993; Gill 1996; Takano and Meijer 1995) (also known as the *fold/build* rule), which is an instance of a free theorem (Wadler 1989).

Law 5 (FOLD/BUILD RULE) *For h strict,*

$$\begin{array}{ll} & g :: \forall\ a\ .\ (F\ a \to a) \to c \to a \\ \Rightarrow & \\ & fold\ h \circ build\ g = g\ h \end{array}$$

where

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

Laws 1 and 2, that we have presented in Section 3.1 are particular instances of Law 5. In that section, when we presented their formulation, notice that the assumption about the strictness of the algebra disappears. This is because every algebra $h_1 \triangledown h_2$ is strict as so is every case analysis.

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

Law 6 (FOLD/BUILDP RULE) *For h strict,*

$$\begin{array}{ll} & g :: \forall\ a\ .\ (F\ a \to a) \to c \to (a, z) \\ \Rightarrow & \\ & (fold\ h\ \times\ id) \circ buildp\ g = g\ h \end{array}$$

where

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

For example, the instance of this law for leaf trees is the following:

$$(fold_T\ (h_1, h_2)\ \times\ id) \circ buildp_T\ g = g\ (h_1, h_2) \tag{6}$$

where

$$\begin{array}{l} buildp_T :: (\forall\ a\ .\ (Int \to a, (a, a) \to a) \to c \to (a, z)) \\ \qquad \to c \to (LeafTree, z) \\ buildp_T\ g = g\ (Leaf, Fork) \end{array}$$

The assumption about the strictness of the algebra disappears by the same reason as for (2).

To see an example of the application of this law, consider the program *ssqm*: it replaces every leaf in a tree by its square while computing the minimum value of the tree; later, it sums all the (squared) elements of an input tree.

$$
\begin{aligned}
&ssqm :: LeafTree \to (Int, Int)\\
&ssqm = (sumt \times id) \circ gentsqmin
\end{aligned}
$$

$$
\begin{aligned}
&sumt :: LeafTree \to Int\\
&sumt\ (Leaf\ n) = n\\
&sumt\ (Fork\ (l, r)) = sumt\ l + sumt\ r
\end{aligned}
$$

$$
\begin{aligned}
&gentsqmin :: LeafTree \to (LeafTree, Int)\\
&gentsqmin\ (Leaf\ n) = (Leaf\ (n * n), n)\\
&gentsqmin\ (Fork\ (l, r)) = \mathbf{let}\ (l', n_1) = gentsqmin\ l\\
&\qquad\qquad\qquad\qquad\qquad\quad (r', n_2) = gentsqmin\ r\\
&\qquad\qquad\qquad\qquad\quad \mathbf{in}\ (Fork\ (l', r'), min\ n_1\ n_2)
\end{aligned}
$$

To apply Law (6) we have to express *sumt* as a fold and *gentsqmin* in terms of $buildp_T$:

$$
\begin{aligned}
&sumt \quad\ = fold_T\ (id, uncurry\ (+))\\
&gentsqmin = buildp_T\ g\\
&\quad \mathbf{where}\ g\ (leaf, fork)\ (Leaf\ n) = (leaf\ (n * n), n)\\
&\qquad\quad g\ (leaf, fork)\ (Fork\ (l, r)) = \mathbf{let}\ (l', n_1) = g\ (leaf, fork)\ l\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad (r', n_2) = g\ (leaf, fork)\ r\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ (fork\ (l', r'), min\ n_1\ n_2)
\end{aligned}
$$

Hence, by (6), we have

$$ssqm = g\ (id, uncurry\ (+))$$

Inlining, we obtain

$$
\begin{aligned}
&ssqm\ (Leaf\ n) = (n * n, n)\\
&ssqm\ (Fork\ (l, r)) = \mathbf{let}\ (s_1, n_1) = ssqm\ l\\
&\qquad\qquad\qquad\qquad\quad (s_2, n_2) = ssqm\ r\\
&\qquad\qquad\qquad\quad \mathbf{in}\ (s_1 + s_2, min\ n_1\ n_2)
\end{aligned}
$$

Finally, the following property is an immediate consequence of Law 6.

Law 7 *For any strict h,*

$$
\begin{aligned}
&g :: \forall\ a\ .\ (F\ a \to a) \to c \to (a, z)\\
\Rightarrow\\
&\pi_2 \circ g\ in_F = \pi_2 \circ g\ h
\end{aligned}
$$

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.

4.4 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 analyze 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 because 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. Definitions of this kind can be given in terms of a fold:

$$f :: (\mu F, z) \to a$$
$$f\ (t, z) = fold\ h\ t$$

such that the context information contained in z may eventually be used in the algebra h. This is the case of, for example, the function *replace*:

$$replace :: (LeafTree, Int) \to LeafTree$$
$$replace\ (Leaf\ n, m) = Leaf\ m$$
$$replace\ (Fork\ (l, r), m) = Fork\ (replace\ (l, m), replace\ (r, m))$$

which can be defined as:

$$replace\ (t, m) = fold_T\ (\lambda n \to Leaf\ m, Fork)\ t$$

Another alternative is the use of currying, which gives a function of type $\mu F \to (z \to a)$. The curried version can then be defined as a higher-order fold. For instance, in the case of *replace* it holds that

$$curry\ replace = fold_T\ (\lambda n \to Leaf, \lambda(f, f') \to Fork \circ (f \vartriangle f'))$$

This is an alternative we will study in detail in Section 4.6.

A third alternative is to define the function $f :: (\mu F, z) \to a$ in terms of a program scheme, called *pfold* (Pardo 2001, 2002), 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:

$$\tau^F :: (F\ a, z) \to F\ (a, z)$$

that satisfies the coherence axioms:

$$map_F\ \pi_1 \circ \tau^F = \pi_1$$

$$map_F\ \alpha \circ \tau^F = \tau^F \circ (\tau^F\ \times\ id) \circ \alpha$$

where $\alpha :: (a, (b, c)) \to ((a, b), c)$ is the product associativity (see (Pardo 2002; Cockett and Spencer 1991; Cockett and Fukushima 1992) 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 corresponding to functor T is given by:

$$\tau^T :: (T\ a, z) \to T\ (a, z)$$
$$\tau^T\ (\mathit{Left}\ n, z) = \mathit{Left}\ n$$
$$\tau^T\ (\mathit{Right}\ (a, a'), z) = \mathit{Right}\ ((a, z), (a', z))$$

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 $\mathit{pfold}\ h :: (\mu F, z) \to a$, is defined as the least function f that satisfies the following equation:

$$f \circ (in_F\ \times\ id) = h \circ (((map_F\ f \circ \tau^F) \vartriangle \pi_2))$$

Observe that now function h also accepts the value of the parameter. It is a function of the form $(h_1\ \triangledown\ \ldots\ \triangledown\ h_n) \circ d$ where each $h_i :: (F_i\ \ a, z) \to a$ if $F\ a = F_1\ a + \cdots + F_n\ a$, and $d :: (x_1 + \cdots + x_n, z) \to (x_1, z) + \cdots + (x_n, z)$ is the distribution of product over sum. When showing specific instances of pfold we will simply write the tuple of functions $(h_1, \ldots, h_n)$ instead of h.
The following equation shows one of the possible relationships between pfold and fold.

$$\mathit{pfold}\ h\ (t, z) = \mathit{fold}\ k\ t\ \ \textbf{where}\ \ k_i\ x = h_i\ (x, z) \tag{7}$$

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 thesis. The interested reader may consult (Pardo 2001, 2002).

4.5 The pfold/buildp rule

In this section, we present a generic formulation of a transformation rule that takes compositions of the form $\mathit{cons} \circ \mathit{prod}$, where a producer $\mathit{prod} :: a \to (t, z)$ is followed by a consumer $\mathit{cons} :: (t, z) \to b$, and returns an equivalent deforested circular program that performs a single traversal over the input value.

The rule, which was first introduced in Fernandes et al. (2007) and further studied in Fernandes (2009) and Pardo et al. (2011), 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, 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 as the instance of a polymorphic function by abstracting out the constructors of the type t from the body of *prod*. In other words, *prod* should be expressed in terms of the *buildp* function corresponding to the type t. The fact that the consumer *cons* is assumed to be structurally recursive leads us to consider that it is given by a pfold. In summary, the rule is applied to compositions of the form: $\mathit{pfold}\ h \circ \mathit{buildp}\ g$.

Law 8 (PFOLD/BUILDP RULE**)** *For any* $h = (h_1 \triangledown \ \ldots \ \triangledown h_n) \circ d$, [19]

$$
\begin{array}{ll}
 & g :: \forall\ a\ .\ (F\ a \to a) \to c \to (a, z) \\
\Rightarrow & \\
 & \mathit{pfold}\ h \circ \mathit{buildp}\ g\ \$\ c \\
 & = v \\
 & \textbf{where}\ (v, \boxed{z}) = g\ k\ c \\
 & \qquad\quad k = k_1 \triangledown \ \ldots \ \triangledown k_n \\
 & \qquad\quad k_i\ \bar{x} = h_i\ (\bar{x}, \boxed{z})
\end{array}
$$

Semantics of the pfold/buildp rule According to Danielsson et al. (2006), Law 8 is morally correct *only*, in Haskell. In fact, the formal proof of our rule, that the interested reader may consult in (Fernandes 2009; Pardo et al. 2011), relies on surjective pairing (Law (5)). However, (5) 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, (5) is morally correct *only* and, in the same sense, so is our rule.

Following our work, Voigtländer (2008) performed a rigorous study on various shortcut fusion rules, for languages like Haskell. In particular, the author presents semantic and pragmatic considerations on Law 8. As a first result, pre-conditions are added to our rule, so that its total correctness can be established.

The definition of Law 8 becomes:

Law 9 (HASKELL VALID PFOLD/BUILDP RULE**)** *For any*
$h = (h_1 \triangledown \ \ldots \ \triangledown h_n) \circ d$,
$\forall\ i\ \in\ \{1, \ldots, n\}\ .\ h_i\ ((\bot, ..., \bot), \bot)\ \neq\ \bot$

$$
\begin{array}{ll}
 & g :: \forall\ a\ .\ (F\ a \to a) \to c \to (a, z) \\
\Rightarrow & \\
 & \mathit{pfold}\ h \circ \mathit{buildp}\ g\ \$\ c \\
 & = v \\
 & \textbf{where}\ (v, \boxed{z}) = g\ k\ c \\
 & \qquad\quad k = k_1 \triangledown \ \ldots \ \triangledown k_n \\
 & \qquad\quad k_i\ \bar{x} = h_i\ (\bar{x}, \boxed{z})
\end{array}
$$

It is now possible to prove total correctness of Law 9 (Voigtländer 2008). However, although Law 9 is the one that guarantees totally correct transformations, in the semantics of Haskell, it is somewhat *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 (2008) presents an example where such pre-condition is violated, causing no harm in the calculated equivalent program. We review here such an example.

Consider the following programming problem: from the initial part of an input list before a certain predicate holds for the first time, return those elements

[19] We denote by $\bar{x}$ a tuple of values $(x_1, \cdots, x_{r_i})$.

that are repeated afterwards. The specification of a natural solution to this problem is as follows:

$$
\begin{array}{l}
\mathit{repeatedAfter} :: \mathit{Eq}\ b \Rightarrow (b \to \mathit{Bool}) \to [b] \to [b] \\
\mathit{repeatedAfter}\ p\ bs = (\mathit{pfilter}\ \mathit{elem}) \circ (\mathit{splitWhen}\ p)\ \$\ bs
\end{array}
$$

$$
\begin{array}{l}
\mathit{pfilter} :: (b \to z \to \mathit{Bool}) \to ([b], z) \to [b] \\
\mathit{pfilter}\ \_\ ([\,], \_) = [\,] \\
\mathit{pfilter}\ p\ (b : bs, z) = \mathbf{let}\ bs' = \mathit{pfilter}\ p\ (bs, z) \\
\qquad\qquad\qquad\qquad \mathbf{in\ if}\ p\ b\ z \\
\qquad\qquad\qquad\qquad\quad \mathbf{then}\ b : bs' \\
\qquad\qquad\qquad\qquad\quad \mathbf{else}\ bs'
\end{array}
$$

$$
\begin{array}{l}
\mathit{splitWhen} :: (b \to \mathit{Bool}) \to [b] \to ([b], [b]) \\
\mathit{splitWhen}\ p\ bs \\
\quad = \mathbf{case}\ bs\ \mathbf{of}\ [\,] \to ([\,], bs) \\
\qquad\qquad\qquad b : bs' \to \mathbf{if}\ p\ b \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{then}\ ([\,], bs) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else\ let}\ (xs, ys) = \mathit{splitWhen}\ p\ bs' \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ (b : xs, ys)
\end{array}
$$

This definition uses a list as the intermediate structure that serves the purpose of gluing the two composed functions. This intermediate list can be eliminated using Law 8. However, in order to apply that law to the *repeatedAfter* program, *pfilter* and *splitWhen p* must first be given in terms of pfold and buildp for lists (the type of the intermediate structure), respectively. The definition of pfold and buildp for lists is as follows.

$$
\begin{array}{l}
\mathit{buildp} :: (\forall\ b\ .\ (b, (a, b) \to b) \to c \to (b, z)) \to c \to ([a], z) \\
\mathit{buildp}\ g = g\ ([\,], \mathit{uncurry}\ (:))
\end{array}
$$

$$
\begin{array}{l}
\mathit{pfold} :: (z \to b, ((a, b), z) \to b) \to ([a], z) \to b \\
\mathit{pfold}\ (\mathit{hnil}, \mathit{hcons}) = p_L \\
\quad \mathbf{where}\ p_L\ ([\,], z) \quad\ = \mathit{hnil}\ z \\
\qquad\qquad p_L\ (a : as, z) = \mathit{hcons}\ ((a, p_L\ (as, z)), z)
\end{array}
$$

Now, we write *pfilter* and *splitWhen p* in terms of them:

$$
\begin{array}{l}
\mathit{splitWhen}\ p = \mathit{buildp}\ \mathit{go} \\
\quad \mathbf{where}\ \mathit{go}\ (\mathit{nil}, \mathit{cons})\ bs \\
\qquad\qquad = \mathbf{case}\ bs\ \mathbf{of}\ [\,] \to\ (\mathit{nil}, bs) \\
\qquad\qquad\qquad\qquad\qquad b : bs' \to \mathbf{if}\ p\ b \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{then}\ (\mathit{nil}, bs) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else\ let}\ (xs, ys) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad = \mathit{go}\ (\mathit{nil}, \mathit{cons})\ bs' \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ (\mathit{cons}\ (b, xs), ys)
\end{array}
$$

$$
\begin{array}{l}
\mathit{pfilter}\ p = \mathit{pfold}\ (\mathit{hnil}, \mathit{hcons}) \\
\quad \textbf{where}\ \mathit{hnil}\ \_ = [\,] \\
\qquad\quad \mathit{hcons}\ ((b, bs), z) = \textbf{if}\ (p\ b\ z)\ \textbf{then}\ (b : bs)\ \textbf{else}\ bs
\end{array}
$$

Regarding this example, we may notice that $\mathit{hcons}\ ((\bot, \bot), \bot) = \bot$, given that (**if** $\mathit{elem}\ \bot\ \bot$ **then** $\bot : \bot$ **else** $\bot$) equals $\bot$. This means that the pre-condition $\forall\ i\ .\ h_i\ ((\bot, ..., \bot), \bot)\ \neq\ \bot$, newly added to Law 8, fails. However, it is still possible to use Law 8 to calculate a correct circular program equivalent to the *repeatedAfter* program presented earlier:

$$
\begin{array}{l}
\mathit{repeatedAfter}\ p\ bs = a \\
\quad \textbf{where}\ (a, \boxed{z}) = go'\ bs \\
\qquad\quad go'\ bs = \textbf{case}\ bs\ \textbf{of}\ [\,] \rightarrow ([\,], bs) \\
\qquad\qquad\qquad\qquad\quad b : bs' \rightarrow \textbf{if}\ p\ b \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ ([\,], bs) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else let}\ (xs, ys) = go'\ bs' \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ (\textbf{if}\ \mathit{elem}\ b\ \boxed{z} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ b : xs \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ xs, ys)
\end{array}
$$

It is in this sense that we say Law 9 is *pessimistic*. However, this Law is the most general one can present, so far, in terms of total correctness.

In the next section, we will present an alternative way to transform compositions between pfold and buildp such that, instead of circular programs, higher-order programs are obtained as result. A good thing about the new transformation is that its total correctness can be established defining fewer pre-conditions than the ones defined in Law 9.

4.6 The higher-order pfold/buildp rule

In the previous section, we have presented the generic formulation of a calculation rule for deriving circular programs. There exists, however, an alternative way to transform compositions between *pfold* and *buildp*. Indeed, in this section we derive higher-order programs from such compositions, instead of the circular programs we derived before.

The alternative transformation presented in this section is based on the fact that every *pfold* can be expressed in terms of a higher-order fold: For $h :: (F\ a, z) \rightarrow a$,

$$\mathit{pfold}\ h = \mathit{apply} \circ (\mathit{fold}\ \varphi_h \times \mathit{id}) \tag{8}$$

where $\varphi_h :: F\ (z \rightarrow a) \rightarrow (z \rightarrow a)$ is given by

$$\varphi_h = \mathit{curry}\ (h \circ ((\mathit{map}_F\ \mathit{apply} \circ \tau^F) \vartriangle \pi_2))$$

and $\mathit{apply} :: (a \rightarrow b, a) \rightarrow b$ by $\mathit{apply}\ (f, x) = f\ x$. Therefore, $\mathit{fold}\ \varphi_h :: \mu F \rightarrow (z \rightarrow a)$ is the curried version of $\mathit{pfold}\ h$.

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 (Pettorossi and Skowron 1987). The specific case of this law when lists are the intermediate structure was introduced by Voigtländer (2008) and its generic formulation was given in Pardo et al. (2009).

Law 10 (HIGHER-ORDER PFOLD/BUILDP**)** *For left-strict h,*[20]

$$pfold\ h \circ buildp\ g = apply \circ g\ \varphi_h$$

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. That is,

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

5 Algol 68 scope rules

In Section 3 we have applied concrete fusion rules to small, but illustrative examples, and in Section 4 we have shown that such rules can be given generic definitions. In this section, we consider the application of shortcut fusion to a real example: the Algol 68 scope rules. These rules are used, for example, in the Eli system[21] (Kastens et al. 1998; Waite et al. 2007) to define a generic component for the name analysis task of a compiler.

The problem we consider is as follows: 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 scope rules via our favorite (toy) language: the *Block* language, which consists of programs of the following form:

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

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

$$\begin{array}{ll}\mathbf{type}\ Prog = [It] & \mathbf{data}\ It = Use\ Var \\ & \quad \mid Decl\ Var \\ \mathbf{type}\ Var = String & \quad \mid Block\ Prog\end{array}$$

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 these

[20] By left-strict we mean strict on the first argument, that is, $h\ (\bot, z) = \bot$.
[21] A well known compiler generator toolbox.

rules the above program contains two errors: at the outer level, the variable x has been declared twice and the use of the variable w, at the inner level, has no binding occurrence at all.

We aim to develop a program that analyses *Block* programs and computes a list containing the identifiers which do not obey to the rules of the language. In order to make the problem more interesting, and also to make it easier to detect which identifiers are being incorrectly used in a *Block* program, we require that the list of invalid identifiers follows the sequential structure of the input program. Thus, the semantic meaning of processing the example sentence is $[w, x]$.

Because we allow a *use-before-declare* discipline, a conventional implementation of the required analysis naturally leads to a program which traverses the abstract syntax tree twice: once for accumulating the declarations of identifiers and constructing the environment, and once for checking the uses of identifiers, according to the computed environment. The uniqueness of names can be detected in the first traversal: for each newly encountered declaration it is checked whether that identifier has already been declared at the current level. In this case an error message is computed. Of course the identifier might have been declared at a global level. Thus we need to distinguish between identifiers declared at different levels. We use the level of a block to achieve this. The environment is a partial function mapping an identifier to its level of declaration:

$$\textbf{type } \mathit{Env} = [(\mathit{Var}, \mathit{Int})]$$

Semantic errors resulting from duplicate definitions are then computed during the first traversal of a block and errors resulting from missing declarations in the second one. In a straightforward implementation of this program, this strategy has two important effects: the first is that a *"gluing"* data structure has to be defined and constructed to pass explicitly the detected errors from the first to the second traversal, in order to compute the final list of errors in the desired order; the second is that, in order to be able to compute the missing declarations of a block, the implementation has to explicitly pass (using, again, an intermediate structure), from the first traversal of a block to its second traversal, the names of the variables that are used in it.

Observe also that the environment computed for a block and used for processing the use-occurrences 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 first and second traversals are intermingled. Furthermore, the information on its nested blocks (the instructions they define and the blocks' level) has to be explicitly passed from the first to the second traversal of a block. This is also achieved by defining and constructing an intermediate data structure. In order to pass the necessary information from the first to the second traversal of a block, we define the following intermediate data structure:

$$\textbf{type } \mathit{Prog}_2 = [\mathit{It}_2] \qquad \textbf{data } \mathit{It}_2 = \mathit{Block}_2\ (\mathit{Int}, \mathit{Prog}) \mid \mathit{Dupl}_2\ \ \mathit{Var} \mid \mathit{Use}_2\ \ \mathit{Var}$$

Errors resulting from duplicate declarations, computed in the first traversal, are passed to the second, using constructor $Dupl_2$. The level of a nested block, as well as the instructions it defines, are passed to the second traversal using constructor $Block_2$'s pair containing an integer and a sequence of instructions.

According to the strategy defined earlier, computing the semantic errors that occur in a block sentence consists of:

$$
\begin{aligned}
&semantics :: Prog \to [\,Var\,] \\
&semantics = missing \circ (duplicate\ 0\ [\,])
\end{aligned}
$$

The function *duplicate* detects duplicate variable declarations by collecting all the declarations occurring in a block. It is defined as follows:

$$
\begin{aligned}
&duplicate :: Int \to Env \to Prog \to (Prog_2, Env) \\
&duplicate\ lev\ ds\ [\,] = ([\,], ds)
\end{aligned}
$$

$$
\begin{aligned}
&duplicate\ lev\ ds\ (Use\ var : its) \\
&\quad = \mathbf{let}\ (its_2, ds') = duplicate\ lev\ ds\ its \\
&\qquad \mathbf{in}\ \ (Use_2\ var : its_2, ds')
\end{aligned}
$$

$$
\begin{aligned}
&duplicate\ lev\ ds\ (Decl\ var : its) \\
&\quad = \mathbf{let}\ (its_2, ds') = duplicate\ lev\ ((var, lev) : ds)\ its \\
&\qquad \mathbf{in}\ \ \mathbf{if}\ ((var, lev) \in ds)\ \mathbf{then}\ (Dupl_2\ var : its_2, ds')\ \mathbf{else}\ (its_2, ds')
\end{aligned}
$$

$$
\begin{aligned}
&duplicate\ lev\ ds\ (Block\ nested : its) \\
&\quad = \mathbf{let}\ (its_2, ds') = duplicate\ lev\ ds\ its \\
&\qquad \mathbf{in}\ \ (Block_2\ (lev + 1, nested) : its_2, ds')
\end{aligned}
$$

Besides detecting the invalid declarations, the *duplicate* function also computes a data structure, of type $Prog_2$, that is later traversed in order to detect variables that are used without being declared. This detection is performed by function *missing*, that is defined such as:

$$
\begin{aligned}
&missing :: (Prog_2, Env) \to [\,Var\,] \\
&missing\ ([\,], \_) = [\,]
\end{aligned}
$$

$$
\begin{aligned}
&missing\ (Use_2\ var : its_2, env) \\
&\quad = \mathbf{let}\ errs = missing\ (its_2, env) \\
&\qquad \mathbf{in\ if}\ (var \in map\ \pi_1\ env)\ \mathbf{then}\ errs\ \mathbf{else}\ var : errs
\end{aligned}
$$

$$
\begin{aligned}
&missing\ (Dupl_2\ var : its_2, env) \\
&\quad = var : missing\ (its_2, env)
\end{aligned}
$$

$$
\begin{aligned}
&missing\ (Block_2\ (lev, its) : its_2, env) \\
&\quad = \mathbf{let}\ errs_1 = missing \circ (duplicate\ lev\ env)\ \$\ its \\
&\qquad\quad errs_2 = missing\ (its_2, env) \\
&\qquad \mathbf{in}\ errs_1 +\!\!+ errs_2
\end{aligned}
$$

The construction and traversal of an intermediate data structure, however, is not essential to implement the semantic analysis described. Indeed, in the next section we will transform *semantics* into an equivalent program that does not construct any intermediate structure.

5.1 Calculating a circular program

In this section, we calculate a circular program equivalent to the *semantics* program presented in the previous section. In our calculation, we will use the specific instance of Law 8 for the case when the intermediate structure gluing the consumer and producer functions is a list:

Law 11 *(*PFOLD/BUILDP RULE FOR LISTS)

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

where the schemes *pfold* and *buildp* have already been defined as:

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

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

Now, if we write *missing* in terms of *pfold*,

$$\begin{aligned}
&\mathit{missing} = \mathit{pfold}\ (\mathit{hnil}, \mathit{hcons}) \\
&\quad \textbf{where}\ \mathit{hnil}\ _ = [\,] \\
&\qquad\qquad \mathit{hcons}\ ((\mathit{Use}_2\ \mathit{var}, \mathit{errs}), \mathit{env}) \\
&\qquad\qquad\quad = \textbf{if}\ (\mathit{var} \in \mathit{map}\ \pi_1\ \mathit{env})\ \textbf{then}\ \mathit{errs}\ \textbf{else}\ \mathit{var} : \mathit{errs} \\
&\qquad\qquad \mathit{hcons}\ ((\mathit{Dupl}_2\ \mathit{var}, \mathit{errs}), \mathit{env}) \\
&\qquad\qquad\quad = \mathit{var} : \mathit{errs} \\
&\qquad\qquad \mathit{hcons}\ ((\mathit{Block}_2\ (\mathit{lev}, \mathit{its}), \mathit{errs}), \mathit{env}) \\
&\qquad\qquad\quad = \textbf{let}\ \mathit{errs}_1 = \mathit{missing} \circ (\mathit{duplicate}\ \mathit{lev}\ \mathit{env})\ \$\ \mathit{its} \\
&\qquad\qquad\quad\ \ \textbf{in}\ \mathit{errs}_1 +\!\!+ \mathit{errs}
\end{aligned}$$

and *duplicate* in terms of *buildp*,

$$\begin{aligned}
&\mathit{duplicate}\ \mathit{lev}\ \mathit{ds} = \mathit{buildp}\ (g\ \mathit{lev}\ \mathit{ds}) \\
&\quad \textbf{where}\ g\ \mathit{lev}\ \mathit{ds}\ (\mathit{nil}, \mathit{cons})\ [\,] = (\mathit{nil}, \mathit{ds})
\end{aligned}$$

$$
\begin{array}{l}
\quad g\ lev\ ds\ (nil, cons)\ (Use\ var : its) \\
\quad\quad = \textbf{let}\ (its_2, ds') = g\ lev\ ds\ (nil, cons)\ its \\
\quad\quad\quad \textbf{in}\ \ (cons\ (Use_2\ var, its_2), ds') \\
\\
\quad g\ lev\ ds\ (nil, cons)\ (Decl\ var : its) \\
\quad\quad = \textbf{let}\ (its_2, ds') = g\ lev\ ((var, lev) : ds)\ (nil, cons)\ its \\
\quad\quad\quad \textbf{in}\ \ \textbf{if}\ ((var, lev) \in ds)\ \textbf{then}\ (cons\ (Dupl_2\ var, its_2), ds')\ \textbf{else}\ (its_2, ds') \\
\\
\quad g\ lev\ ds\ (nil, cons)\ (Block\ nested : its) \\
\quad\quad = \textbf{let}\ (its_2, ds') = g\ lev\ ds\ (nil, cons)\ its \\
\quad\quad\quad \textbf{in}\ \ (cons\ (Block_2\ (lev + 1, nested), its_2), ds')
\end{array}
$$

we can apply Law 11 to the program $semantics = missing \circ (duplicate\ 0\ [\,])$, since this program has just been expressed as an explicit composition between a *pfold* and a *buildp*. We obtain a deforested circular definition, which, when inlined, gives the following program:

$$
\begin{array}{l}
semantics\ p = errs \\
\quad \textbf{where} \\
\quad\quad (errs, \boxed{\text{env}}) = gk\ 0\ [\,]\ p \\
\\
\quad\quad gk\ lev\ ds\ [\,] = ([\,], ds) \\
\\
\quad\quad gk\ lev\ ds\ (Use\ var : its) \\
\quad\quad\quad = \textbf{let}\ (errs, ds') = gk\ lev\ ds\ its \\
\quad\quad\quad\quad \textbf{in}\ (\textbf{if}\ (var \in map\ \pi_1\ \boxed{\text{env}})\ \textbf{then}\ errs\ \textbf{else}\ var : errs, ds') \\
\\
\quad\quad gk\ lev\ ds\ (Decl\ var : its) \\
\quad\quad\quad = \textbf{let}\ (errs, ds') = gk\ lev\ ((var, lev) : ds)\ its \\
\quad\quad\quad\quad \textbf{in if}\ ((var, lev) \in ds)\ \textbf{then}\ (var : errs, ds')\ \textbf{else}\ (errs, ds') \\
\\
\quad\quad gk\ lev\ ds\ (Block\ nested : its) \\
\quad\quad\quad = \textbf{let}\ (errs_2, ds') = gk\ lev\ ds\ its \\
\quad\quad\quad\quad \textbf{in}\ (\textbf{let}\ errs_1 = missing \circ (duplicate\ (lev + 1)\ \boxed{\text{env}})\ \$\ nested \\
\quad\quad\quad\quad\quad\ \textbf{in}\ errs_1 +\!\!+ errs_2, ds')
\end{array}
$$

We may notice that the above program is a circular one: the environment of a *Block* program (variable *env*) is being computed at the same time it is being used. The introduction of this circularity made it possible to eliminate some intermediate structures that occurred in the program we started with: the intermediate list of instructions that was computed in order to glue the two traversals of the outermost level of a *Block* sentence has been eliminated by application of Law 11. We may also notice, however, that, for nested blocks:

$$
\begin{array}{l}
gk\ lev\ ds\ (Block\ nested : its) \\
\quad = \textbf{let}\ (errs_2, ds') = gk\ lev\ ds\ its \\
\quad\quad \textbf{in}\ (\textbf{let}\ errs_1 = missing \circ (duplicate\ (lev + 1)\ env)\ \$\ nested \\
\quad\quad\quad\ \textbf{in}\ errs_1 +\!\!+ errs_2, ds')
\end{array}
$$

an intermediate structure is still being used in order to glue functions *missing* and *duplicate* together. This intermediate structure can easily be eliminated, since we have already expressed function *missing* in terms of *pfold*, and function *duplicate* in terms of *buildp*. Therefore, by direct application of Law 11 to the above function composition, we obtain:

$$
\begin{array}{l}
gk\ lev\ ds\ (Block\ nested : its) \\
\quad = \textbf{let}\ (errs_2, ds') = gk\ lev\ ds\ its \\
\quad\quad \textbf{in}\ (\textbf{let}\ (errs_1, \boxed{env_2}) = g\ (lev+1)\ env\ (knil, kcons)\ nested \\
\quad\quad\quad\quad \textbf{where}\ knil = hnil\ \boxed{env_2} \\
\quad\quad\quad\quad\quad\quad\quad kcons\ x = hcons\ (x, \boxed{env_2}) \\
\quad\quad\quad\quad \textbf{in}\ errs_1 +\!\!+ errs_2, ds')
\end{array}
$$

Again, we could inline the definition of function g into a new function, for example, into function gk'. However, the definition of gk' would exactly match the definition of gk, except for the fact that where gk searched for variable declarations in enviornment env, gk' needs to search them in environment env_2.

In order to use the same function for both gk and gk', we add an extra argument to function gk. This argument will make it possible to use circular definitions to pass the appropriate environment variable to the appropriate block of instructions (the top level block or a nested one).

We should notice that, in general, this extra effort is not necessary. In this particular example, this manipulation effort was made since it is possible to calculate two circular definitions from the straightforward solution and both circular functions share almost the same definition. In all other cases, inlining the calculated circular program is enough to derive an elegant and efficient *lazy* program from a function composition between a pfold and a buildp.

We finally obtain the program:

$$
\begin{array}{l}
semantics\ p = errs \\
\quad \textbf{where}\ (errs, \boxed{\text{env}}) = gk\ 0\ [\,]\ \boxed{\text{env}}\ p \\
\\
\quad\quad gk\ lev\ ds\ env\ [\,] = ([\,], ds) \\
\\
\quad\quad gk\ lev\ ds\ env\ (Use\ var : its) \\
\quad\quad\quad = \textbf{let}\ (errs, ds') = gk\ lev\ ds\ env\ its \\
\quad\quad\quad\quad \textbf{in}\ (\textbf{if}\ (var \in map\ \pi_1\ env)\ \textbf{then}\ errs\ \textbf{else}\ var : errs, ds') \\
\\
\quad\quad gk\ lev\ ds\ env\ (Decl\ var : its) \\
\quad\quad\quad = \textbf{let}\ (errs, ds') = gk\ lev\ ((var, lev) : ds)\ env\ its \\
\quad\quad\quad\quad \textbf{in if}\ ((var, lev) \in ds)\ \textbf{then}\ (var : errs, ds')\ \textbf{else}\ (errs, ds') \\
\\
\quad\quad gk\ lev\ ds\ env\ (Block\ nested : its) \\
\quad\quad\quad = \textbf{let}\ (errs_2, ds') = gk\ lev\ ds\ env\ its \\
\quad\quad\quad\quad \textbf{in}\ (\textbf{let}\ (errs_1, \boxed{env_2}) = gk\ (lev+1)\ env\ \boxed{env_2}\ nested \\
\quad\quad\quad\quad\quad\quad \textbf{in}\ errs_1 +\!\!+ errs_2, ds')
\end{array}
$$

Regarding the above program, we may notice that it has two circular definitions. One such definition occurs in the *semantics* function, and makes it possible for the environment of the outer level of a block program to be used while still being constructed. For the example sentence that we have considered before,

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

this circularity makes the environment $[(\texttt{"x"}, 0), (\texttt{"x"}, 0), (\texttt{"y"}, 0)]$ available to the function that traverses the outer block. The other circular definition, occurring in the last definition of function gk, is used so that, for every traversal of a nested sequence of instructions, its environment may readily be used. This means that the function traversing the nested block in the above example sentence may use the environment $[(\texttt{"x"}, 0), (\texttt{"x"}, 0), (\texttt{"y"}, 0), (\texttt{"y"}, 1)]$ even though it still needs to be constructed.

The introduction of these circularities, by the application of our calculational method, completely eliminated the intermediate lists of instructions that were used in the straightforward *semantics* solution we started with. Furthermore, the derivation of this circular program made it possible to obtain a *semantics* program that computes the list of errors that occur in a *Block* program by traversing it only once.

5.2 Calculating a higher-order program

In this section we study the application of Law 10 to the *semantics* program given earlier:

$$semantics = missing \circ (duplicate\ 0\ [\,])$$

As we have stated, this definition constructs an intermediate list of instructions, that again we woud like to eliminate with fusion. For this purpose, we will now use the specific instance of Law 10 for the case where the intermediate structure is a list:

Law 12 (HIGHER-ORDER PFOLD/BUILDP FOR LISTS)

$$pfold\ (hnil, hcons) \circ buildp\ g = apply \circ g\ (\varphi_{hnil}, \varphi_{hcons})$$

where $(\varphi_{hnil}, \varphi_{hcons})$ is the algebra of the higher-order fold which corresponds to the curried version of $pfold\ (hnil, hcons)$.

We have already expressed function *missing* in terms of *pfold*,

$$\begin{array}{l}missing = pfold\ (hnil, hcons)\\ \quad \textbf{where}\ hnil\ \_ = [\,]\\ \qquad\quad hcons\ ((Use_2\ var, errs), env)\\ \qquad\qquad = \textbf{if}\ (var \in map\ \pi_1\ env)\ \textbf{then}\ errs\ \textbf{else}\ var : errs\end{array}$$

$$
\begin{aligned}
&\mathit{hcons}\ ((\mathit{Dupl}_2\ \mathit{var}, \mathit{errs}), \mathit{env}) \\
&\quad = \mathit{var} : \mathit{errs}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{hcons}\ ((\mathit{Block}_2\ (\mathit{lev}, \mathit{its}), \mathit{errs}), \mathit{env}) \\
&\quad = \mathbf{let}\ \mathit{errs}_1 = \mathit{missing} \circ (\mathit{duplicate}\ \mathit{lev}\ \mathit{env})\ \$\ \mathit{its} \\
&\qquad \mathbf{in}\ \mathit{errs}_1 +\!\!+ \mathit{errs}
\end{aligned}
$$

and function *duplicate* in terms of *buildp*.

$$
\begin{aligned}
&\mathit{duplicate}\ \mathit{lev}\ \mathit{ds} = \mathit{buildp}\ (g\ \mathit{lev}\ \mathit{ds}) \\
&\quad \mathbf{where}\ g\ \mathit{lev}\ \mathit{ds}\ (\mathit{nil}, \mathit{cons})\ [\,] = (\mathit{nil}, \mathit{ds})
\end{aligned}
$$

$$
\begin{aligned}
&g\ \mathit{lev}\ \mathit{ds}\ (\mathit{nil}, \mathit{cons})\ (\mathit{Use}\ \mathit{var} : \mathit{its}) \\
&\quad = \mathbf{let}\ (\mathit{its}_2, \mathit{ds}') = g\ \mathit{lev}\ \mathit{ds}\ (\mathit{nil}, \mathit{cons})\ \mathit{its} \\
&\qquad \mathbf{in}\ \ (\mathit{cons}\ (\mathit{Use}_2\ \mathit{var}, \mathit{its}_2), \mathit{ds}')
\end{aligned}
$$

$$
\begin{aligned}
&g\ \mathit{lev}\ \mathit{ds}\ (\mathit{nil}, \mathit{cons})\ (\mathit{Decl}\ \mathit{var} : \mathit{its}) \\
&\quad = \mathbf{let}\ (\mathit{its}_2, \mathit{ds}') = g\ \mathit{lev}\ ((\mathit{var}, \mathit{lev}) : \mathit{ds})\ (\mathit{nil}, \mathit{cons})\ \mathit{its} \\
&\qquad \mathbf{in}\ \ \mathbf{if}\ ((\mathit{var}, \mathit{lev}) \in \mathit{ds})\ \mathbf{then}\ (\mathit{cons}\ (\mathit{Dupl}_2\ \mathit{var}, \mathit{its}_2), \mathit{ds}') \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ (\mathit{its}_2, \mathit{ds}')
\end{aligned}
$$

$$
\begin{aligned}
&g\ \mathit{lev}\ \mathit{ds}\ (\mathit{nil}, \mathit{cons})\ (\mathit{Block}\ \mathit{nested} : \mathit{its}) \\
&\quad = \mathbf{let}\ (\mathit{its}_2, \mathit{ds}') = g\ \mathit{lev}\ \mathit{ds}\ (\mathit{nil}, \mathit{cons})\ \mathit{its} \\
&\qquad \mathbf{in}\ \ (\mathit{cons}\ (\mathit{Block}_2\ (\mathit{lev} + 1, \mathit{nested}), \mathit{its}_2), \mathit{ds}')
\end{aligned}
$$

Therefore, in order to apply Law 12 to the *semantics* program, we now only need the expression of the algebra $(\varphi_{hnil}, \varphi_{hcons})$ of the curried version of *missing*:

$$
\begin{aligned}
&\mathit{missing}_{ho} = \mathit{fold}\ (\varphi_{hnil}, \varphi_{hcons}) \\
&\quad \mathbf{where}\ \varphi_{hnil} = \backslash\_ \to [\,]
\end{aligned}
$$

$$
\begin{aligned}
&\varphi_{hcons}\ (\mathit{Use}_2\ \mathit{var}, \mathit{ferrs}) \\
&\quad = \lambda \mathit{env} \to \mathbf{if}\ (\mathit{var} \in \mathit{map}\ \pi_1\ \mathit{env})\ \mathbf{then}\ \mathit{ferrs}\ \mathit{env} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \mathit{var} : (\mathit{ferrs}\ \mathit{env})
\end{aligned}
$$

$$
\begin{aligned}
&\varphi_{hcons}\ (\mathit{Dupl}_2\ \mathit{var}, \mathit{ferrs}) \\
&\quad = \lambda \mathit{env} \to \mathit{var} : (\mathit{ferrs}\ \mathit{env})
\end{aligned}
$$

$$
\begin{aligned}
&\varphi_{hcons}\ (\mathit{Block}_2\ (\mathit{lev}, \mathit{its}), \mathit{ferrs}) \\
&\quad = \lambda \mathit{env} \to \mathbf{let}\ \mathit{errs}_1 = \mathit{missing} \circ (\mathit{duplicate}\ \mathit{lev}\ \mathit{env})\ \$\ \mathit{its} \\
&\qquad\qquad\qquad \mathbf{in}\ \mathit{errs}_1 +\!\!+ (\mathit{ferrs}\ \mathit{env})
\end{aligned}
$$

After inlining the definition that we calculate by directly applying Law 12 to the *semantics* program, we obtain the program presented in the next page.

$$
\begin{array}{l}
semantics\ p = ferrs\ env \\
\quad \textbf{where}\ (ferrs, env) = g_\varphi\ 0\ [\,]\ p \\
\qquad g_\varphi\ lev\ ds\ [\,] = (\lambda env \rightarrow [\,], ds) \\
\\
\qquad g_\varphi\ lev\ ds\ (Use\ var : its) \\
\qquad\quad = \textbf{let}\ (ferrs, ds') = g_\varphi\ lev\ ds\ its \\
\qquad\qquad \textbf{in}\ (\lambda env \rightarrow \textbf{if}\ var \in map\ \pi_1\ env \\
\qquad\qquad\qquad\qquad\qquad \textbf{then}\ ferrs\ env \\
\qquad\qquad\qquad\qquad\qquad \textbf{else}\ var : (ferrs\ env), ds') \\
\\
\qquad g_\varphi\ lev\ ds\ (Decl\ var : its) \\
\qquad\quad = \textbf{let}\ (ferrs, ds') = g_\varphi\ lev\ ((var, lev) : ds)\ its \\
\qquad\qquad \textbf{in if}\ ((var, lev) \in ds) \\
\qquad\qquad\quad \textbf{then}\ (\lambda env \rightarrow var : (ferrs\ env), ds') \\
\qquad\qquad\quad \textbf{else}\ (ferrs, ds') \\
\\
\qquad g_\varphi\ lev\ ds\ (Block\ nested : its) \\
\qquad\quad = \textbf{let}\ (ferrs_2, ds') = g_\varphi\ lev\ ds\ its \\
\qquad\qquad \textbf{in}\ (\lambda env \rightarrow \textbf{let}\ errs_1 = missing \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \circ\ (duplicate\ (lev + 1) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad env)\ \$\ nested \\
\qquad\qquad\qquad\qquad\quad \textbf{in}\ errs_1 +\!\!+ ferrs_2\ env, ds')
\end{array}
$$

Notice that the first component of the result produced by the call $g_\varphi\ 0\ [\,]\ p$ is now a function, instead of a concrete value. When this function is applied to *env*, it produces the list of variables that do not obey to the semantic rules of the language. The program we have calculated is, therefore, a higher-order program.

Regarding the above program, we may notice that it maintains the construction of an intermediate structure. This situation already occurred in Section 5.1. Again, an intermediate structure is constructed whenever a nested sequence of instructions is traversed, in the definition presented next.

$$
\begin{array}{l}
g_\varphi\ lev\ ds\ (Block\ nested : its) \\
\quad = \textbf{let}\ (ferrs_2, ds') = g_\varphi\ lev\ ds\ its \\
\qquad \textbf{in}\ (\lambda env \rightarrow \textbf{let}\ errs_1 = missing \circ (duplicate\ (lev + 1)\ env)\ \$\ nested \\
\qquad\qquad\quad \textbf{in}\ errs_1 +\!\!+ ferrs_2\ env, ds')
\end{array}
$$

The *missing* $\circ$ *duplicate* composition in the above definition, however, may be eliminated by direct application of Law 12. This is due to the fact that functions *missing* and *duplicate* have already been expressed in terms of the appropriate program schemes. We obtain:

$$
\begin{array}{l}
g_\varphi\ lev\ ds\ (Block\ nested : its) \\
\quad = \textbf{let}\ (ferrs_2, ds') = g_\varphi\ lev\ ds\ its \\
\qquad \textbf{in}\ (\lambda env \rightarrow \textbf{let}\ (ferrs_1, env_1) = g_\varphi\ (lev + 1)\ env\ nested \\
\qquad\qquad\quad \textbf{in}\ ferrs_1\ env_1 +\!\!+ ferrs_2\ env, ds')
\end{array}
$$

The higher-order version of *semantics* that we calculate in this section, by applying Law 12, twice, to the original *semantics* program avoids the construction of any intermediate structure. Furthermore, in this program, the appropriate (local or global) environment is passed to the correct block of instructions. Notice that, in order for this to happen, it was not necessary to post-process the calculated program, as it was in Section 5.1. The execution of the higher-order *semantics* program is not restricted to a lazy execution setting. Recall that the intermediate structure free program that we calculated in Section 5.1 may only be executed in a lazy setting: it holds two circular definitions.

6 Conclusions

In this tutorial, we revised a systematic technique for the deforestation of intermediate data structures. These data structures enable a compositional style of programming, which contributes to an increased modularity, but their use may degrade the overall running efficiency of the resulting implementations.

As programmers, we would always like to deal with modular programs, but as software users we favour runtime performance. In the context of this tutorial, this opens up two questions:

1. *Is it possible to automatically derive the programs we have manually calculated here?*
   This derivation is indeed possible, for example within the Glasgow Haskell Compiler (GHC), using rewrite rules (RULES pragma). For the reader interested in further details, we suggest (Fernandes 2009).
2. *How do the types of programs we calculate here compare in terms of runtime performance?*
   This issue is particularly relevant for the circular and higher-order programs we have calculated, and we have in the past performed a first attempt on such comparison (Fernandes 2009). While in the examples we considered, higher-order programs as we propose to calculate in Section 5.2 were the most efficient, it would be interesting to conduct a detailed and representative benchmark to assess whether this observation holds in general.

In this tutorial, we have focused on programs consisting of the composition of two functions. Recently, we have however followed a similar approach to derive shortcut fusion rules that apply to programs consisting of an arbitrary number of function compositions (Pardo et al. 2013).

Here, we have also focused on the practical and pragmatical aspects of the fusion rules that were studied. In this line, we have chosen not to present their formal proofs, that the interested reader may obtain in (Fernandes 2009; Pardo et al. 2011).

As we have highlighted before, in the techniques we revise, lazy evaluation and higher-order programming are crucial.

Bibliography

Samson Abramsky and Achim Jung. Domain theory. In *Handbook of Logic in Computer Science*, pages 1–168. Clarendon Press, 1994.

Lex Augusteijn. Sorting morphisms. In Doaitse Swierstra, Pedro Henriques, and José Oliveira, editors, *Third Summer School on Advanced Functional Programming*, volume 1608 of *LNCS*, pages 1–27. Springer-Verlag, September 1998.

Richard Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21:239–250, 1984.

Richard Bird. *Introduction to Functional Programming using Haskell,* 2nd edition. Prentice-Hall, UK, 1998.

Richard Bird and Oege de Moor. *Algebra of Programming*, volume 100 of *Prentice-Hall Inernational Series in Computer Science*. Prentice-Hall, 1997.

Robin Cockett and Tom Fukushima. About Charity. Technical Report 92/480/18, University of Calgary, June 1992.

Robin Cockett and Dwight Spencer. Strong Categorical Datatypes I. In R.A.C. Seely, editor, *International Meeting on Category Theory 1991,* volume 13 of *Canadian Mathematical Society Conference Proceedings*, pages 141–169, 1991.

Nils Anders Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons. Fast and loose reasoning is morally correct. In *POPL'06: Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 206–217, New York, NY, USA, 2006. ACM.

João Paulo Fernandes. *Design, Implementation and Calculation of Circular Programs.* PhD thesis, Deparment of Informatics, University of Minho, Portugal, 2009.

João Paulo Fernandes, Alberto Pardo, and João Saraiva. A shortcut fusion rule for circular program calculation. In *Haskell'07: Proceedings of the ACM SIGPLAN Haskell Workshop*, pages 95–106, New York, NY, USA, 2007. ACM Press.

Jeremy Gibbons. Calculating Functional Programs. In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction,* LNCS 2297, pages 148–203. Springer-Verlag, January 2002.

Andrew Gill. *Cheap Deforestation for Non-strict Functional Languages.* PhD thesis, Department of Computing Science, University of Glasgow, UK, 1996.

Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, June 1993.

John Hughes. Why functional programming matters. *The Computer Journal*, 32:98–107, 1984.

Uwe Kastens, Peter Pfahler, and Matthias T. Jung. The Eli System. In *CC '98: Proceedings of the 7th International Conference on Compiler Construction*, pages 294–297, London, UK, 1998. Springer-Verlag.

Alberto Pardo. *A Calculational Approach to Recursive Programs with Effects.* PhD thesis, Technische Universität Darmstadt, October 2001.

Alberto Pardo. Generic Accumulations. In *IFIP WG2.1 Working Conference on Generic Programming*, Dagstuhl, Germany, July 2002.

Alberto Pardo, João Paulo Fernandes, and João Saraiva. Shortcut fusion rules for the derivation of circular and higher-order monadic programs. In *PEPM'09: Proceedings of the 2009 ACM SIGPLAN Symposium on Partial Evaluation and Program Manipulation*, pages 81–90. ACM Press, 2009.

Alberto Pardo, João Paulo Fernandes, and João Saraiva. Shortcut fusion rules for the derivation of circular and higher-order programs. *Higher-Order and Symbolic Computation*, 24(1-2):115–149, 2011. ISSN 1388-3690.

Alberto Pardo, João Paulo Fernandes, and João Saraiva. Multiple intermediate structure deforestation by shortcut fusion. In *SBLP'13: Proceedings of the 17th Brazilian Symposium on Programming Languages*, volume 8129 of *Lecture Notes in Computer Science*, pages 120–134. Springer, 2013.

Alberto Pettorossi and Andrzej Skowron. The lambda abstraction strategy for program derivation. In *Fundamenta Informaticae XII*, pages 541–561, 1987.

Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003. Also in *Journal of Functional Programming*, 13(1).

Simon Peyton Jones, John Hughes, Lennart Augustsson, et al. Report on the programming language Haskell 98. Technical report, February 1999.

Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In *In Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 306–313. ACM Press, 1995.

Janis Voigtländer. Semantics and pragmatics of new shortcut fusion rules. In *FLOPS '08: Proceedings of the 2008 International Symposium on Functional and Logic Programming*, pages 163–179. Springer-Verlag, 2008.

Philip Wadler. Theorems for free! In *4th International Conference on Functional Programming and Computer Architecture*, London, 1989.

Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

William Waite, Uwe Kastens, and Anthony M. Sloane. *Generating Software from Specifications*. Jones and Bartlett Publishers, Inc., USA, 2007.