A Shortcut Fusion Rule for Circular Program Calculation

João Paulo Fernandes *
Universidade do Minho, Portugal
jpaulo@di.uminho.pt

Alberto Pardo
Universidad de la Republica, Uruguay
pardo@fing.edu.uy

João Saraiva
Universidade do Minho, Portugal
jas@di.uminho.pt

Abstract

Circular programs are a powerful technique to express multiple traversal algorithms as a single traversal function in a lazy setting. In this paper, we present a shortcut deforestation technique to calculate circular programs. The technique we propose takes as input the composition of two functions, such that the first builds an intermediate structure and some additional context information which are then processed by the second one, to produce the final result. Our transformation into circular programs achieves intermediate structure deforestation and multiple traversal elimination. Furthermore, the calculated programs preserve the termination properties of the original ones.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors - Compilers, Optimization; D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructors - Program and Recursion Schemes

General Terms Languages, Theory, Algorithms

Keywords Circular Programming, Program Calculation, Shortcut Fusion, Deforestation

1. Introduction

Circular programs, as introduced by Bird (1984), are a famous example that demonstrates the power of lazy evaluation. Bird's work showed that any algorithm that performs multiple traversals over the same data structure can be expressed in a lazy language as a single traversal *circular function*, the *repmin* program being the reference example in this case. Such a (virtually) circular function may contain a *circular definition*, that is, an argument of a function call that is also a result of that same call. Although circular definitions always induce non-termination under a strict evaluation mechanism, they can sometimes be computed using a lazy evaluation strategy. The lazy engine is able to compute the right evaluation order, if that order exists.

Using the style of circular programming, the programmer does not have to concern him/herself with the definition and the scheduling of the different traversal functions. Moreover, because there is a single traversal function, the programmer does not have to define intermediate gluing data structures to convey information between traversals, either.

Bird's work showed the power of circular programming, not only as an optimization technique to eliminate multiple traversal of data, but also as a powerful, elegant and concise technique to express multiple traversal algorithms. For example, circular programs are used to express pretty-printing algorithms (Swierstra et al. 1999), breadth-first traversal strategies (Okasaki 2000), type systems (Dijkstra and Swierstra 2004), aspect-oriented compilers (de Moor et al. 2000), and, as Johnsson (1987) and Kuiper and Swierstra (1987) originally showed, circular programs are the natural representation of attribute grammars in a lazy setting (de Moor et al. 2000; Dijkstra 2005; Saraiva 1999; Swierstra and Azero 1998). Circular programs have also been studied in the context of partial evaluation (Lawall 2001) and continuations (Danvy and Goldberg 2002).

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

Several attempts have successfully been made to combine such compositions of two functions into a single function, eliminating the use of the intermediate structures (Gill et al. 1993; Ohori and Sasano 2007; Onoue et al. 1997; Wadler 1990). In those situations, circular programs have also been advocated suitable for deforesting intermediate structures in compositions of two functions with accumulating parameters (Voigtländer 2004).

On the other hand, when the second traversal requires additional information, besides the intermediate structure computed in the first traversal, in order to be able to produce its outcome, no such method produces satisfactory results. In fact, as a side-effect of deforestation, they introduce multiple traversals of the input structure. This is due to the fact that deforestation methods focus on eliminating the intermediate structure, without taking into account the computation of the additional information necessary for the second traversal.

Our motivation for the present work is then to transform programs of this kind into programs that construct no intermediate

* Supported by Fundação para a Ciência e Tecnologia (FCT), grant No. SFRH/BD/19186/2004

Haskell'07, September 30, 2007, Freiburg, Germany.

data-structure and that traverse the input structure only once. That is to say, we want to perform deforestation on those programs and, subsequently, to eliminate the multiple traversals that deforestation introduces. These goals are achieved by transforming the original programs into circular ones. We allow the first traversal to produce a completely general intermediate structure together with some additional context information. The second traversal then uses such context information so that, consuming the intermediate structure produced in the first traversal, it is able to compute the desired results.

The method we propose is based on a variant of the well-known *fold/build* rule (Gill et al. 1993; Launchbury and Sheard 1995). The standard *fold/build* rule does not apply to the kind of programs we wish to calculate as they need to convey context information computed in one traversal into the following one. The new rule we introduce, called *pfold/buildp*, was designed to support contextual information to be passed between the first and the second traversals and also the use of completely general intermediate structures. Like *fold/build*, our rule is also cheap and practical to implement in a compiler.

The *pfold/buildp* rule states that the composition of such two traversals naturally induces a circular program. That is, we calculate circular programs from programs that consist of function compositions of the form $f \circ g$, where g, the producer, builds an intermediate structure t and some additional information i, and where f, the consumer, defined by structural recursion over t, traverses t and, using i, produces the desired results. The circular programs we derive compute the same results as the two original functions composed together, but they do this by performing a single traversal over the input structure. Furthermore, and since that a single traversal is performed, the intermediate structures lose their purpose. In fact, they are deforested by our rule.

In this paper, we not only introduce a new calculation rule, but we also present the formal proof that such rule is correct. We also present formal evidence that this rule introduces no *real* circularity, i.e., that the circular programs it derives preserve the same termination properties as the original programs. Recall that Bird's approach to circular program derivation preserves partial correctness only: the circular programs it derives are not guaranteed to terminate, even that the original programs do.

The relevance of the rule we introduce in this paper may also be appreciated when observed in combination with other program transformation techniques. With our rule, we derive circular programs which most programmers would find difficult to write directly. Those programs can then be further transformed by applying manipulation techniques like, for example, the one presented by Fernandes and Saraiva (2007). This technique attempts to eliminate the performance overhead potentially introduced by circular definitions (the evaluation of such definitions requires the execution of a complex *lazy* engine) by transforming circular programs into programs that do not make essential use of lazyness. Furthermore, the programs obtained are completely data-structure free. So, they do not traverse, nor construct, any data structure.

This paper is organized as follows. In Section 2, we review Bird's method for deriving circular programs in the case of the *repmin* problem, and we contrast it with the (informal) derivation of the circular solution for the same problem following the method we propose. Like *fold/build*, our technique will be characterized by certain program schemes, which will be presented in Section 3 together with the algebraic laws necessary for the proof of the new rule. In Section 4 we formulate and prove the *pfold/buildp* rule; we also review the calculation of the circular program for the *repmin* problem, now in terms of the rule. Sections 5 and 6 illustrate the application of our method to other programming problems: Section 5 presents, in detail, the main steps of our transformation applied to a simple example and Section 6 presents the application of our method to a real example. Section 7 concludes the paper.

2. Circular Programs

Circular programs were first proposed by Bird (1984) as an elegant and efficient technique to eliminate multiple traversals of data structures. As the name suggests, circular programs are characterized by having what appears to be a circular definition: arguments in a function call depend on results of that same call. That is, they contain definitions of the form:

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

In order to motivate the use of circular programs, Bird introduces the following programming problem: consider the problem of transforming a binary leaf tree into a second tree, identical in shape to the original one, but with all the leaf values replaced by the minimum leaf value. This problem is widely known as *repmin*.

In a strict and purely functional setting, solving this problem would require a two traversal strategy: the first traversal to compute the original tree's minimum value, and the second traversal to replace all the leaf values by the minimum value, therefore producing the desired tree. This straightforward solution is as follows.

$$\begin{aligned}
&\mathbf{data}\ \mathit{LeafTree} = \mathit{Leaf}\ \mathit{Int}\\
&\qquad\qquad\quad \mid\ \mathit{Fork}\ (\mathit{LeafTree}, \mathit{LeafTree})\\
&\\
&\mathit{transform} :: \mathit{LeafTree} \to \mathit{LeafTree}\\
&\mathit{transform}\ t = \mathit{replace}\ (t, \mathit{tmin}\ t)\\
&\\
&\mathit{tmin} :: \mathit{LeafTree} \to \mathit{Int}\\
&\mathit{tmin}\ (\mathit{Leaf}\ n) \quad\ = n\\
&\mathit{tmin}\ (\mathit{Fork}\ (l, r)) = \mathit{min}\ (\mathit{tmin}\ l)\ (\mathit{tmin}\ r)\\
&\\
&\mathit{replace} :: (\mathit{LeafTree}, \mathit{Int}) \to \mathit{LeafTree}\\
&\mathit{replace}\ (\mathit{Leaf}\ _\quad\ , m) = \mathit{Leaf}\ m\\
&\mathit{replace}\ (\mathit{Fork}\ (l, r), m) = \mathit{Fork}\ (\mathit{replace}\ (l, m),\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{replace}\ (r, m))
\end{aligned}$$

However, a two traversal strategy is not essential to solve the *repmin* problem. An alternative solution can, on a single traversal, compute the minimum value and, at the same time, replace all leaf values by that minimum value.

2.1 Bird's method

Bird (1984) proposed a method for deriving single traversal programs from straightforward solutions, using tupling, folding-unfolding and circular programming. For example, using Bird's method, the derivation of a single traversal solution for *repmin* proceeds as follows.

Since functions $\mathit{replace}$ and tmin traverse the same data structure (a leaf tree) and given their common recursive pattern, we tuple them into one function repmin, which computes the same results as the previous two functions combined. Note that, in order to be able to apply such tupling step, it is essential that the two functions traverse the same data structure.

$$\mathit{repmin}\ (t, m) = (\mathit{replace}\ (t, m), \mathit{tmin}\ t)$$

We may now synthesize a recursive definition for repmin using the standard application of the fold-unfold method. Two cases have to be considered:

$$\begin{aligned}
&\mathit{repmin}\ (\mathit{Leaf}\ n, m)\\
&\quad = (\mathit{replace}\ (\mathit{Leaf}\ n, m), \mathit{tmin}\ (\mathit{Leaf}\ n))\\
&\quad = (\mathit{Leaf}\ m \qquad\qquad\ , n)
\end{aligned}$$

$$
\begin{array}{l}
repmin\ (Fork\ (l,r),m) \\
\quad = (replace\ (Fork\ (l,r),m), tmin\ (Fork\ (l,r))) \\
\quad = (Fork\ (replace\ (l,m) \\
\qquad\qquad , replace\ (r,m))\quad , min\ (tmin\ l)\ (tmin\ r)) \\
\quad = (Fork\ (l',r') \qquad\qquad\quad , min\ n_1\ n_2) \\
\quad\ \ \mathbf{where}\ \ (l',\ n_1) = repmin\ (l,m) \\
\qquad\qquad\ \ (r',n_2) = repmin\ (r,m)
\end{array}
$$

Finally, circular programming is used to couple the two components of the result value of $repmin$ to each other. Consequently, we obtain the following circular definition of $transform$.

$$
\begin{array}{l}
transform :: LeafTree \rightarrow LeafTree \\
transform\ t = nt \\
\quad \mathbf{where}\ (nt,m) = repmin\ (t,m)
\end{array}
$$

A single traversal is obtained because the function applied to the argument t of $transform$, the $repmin$ function, traverses t only once; this single traversal solution is possible due to the circular call of $repmin$: m is both an argument and a result of that call. This circularity ensures that the information on the minimum value is being used at the same time it is being computed.

Although the circular definitions seem to induce both cycles and non-termination of those programs, the fact is that using a *lazy* language, the *lazy* evaluation machinery is able to determine, at runtime, the right order to evaluate such circular definitions.

After the seminal paper by Bird, the style of circular programming became widely known. However, the approach followed by Bird does not guarantee termination of the resulting lazy program. In fact, Bird (1984) discusses this problem and presents an example of a non-terminating circular program obtained using his transformational technique.

2.2 Our method

The calculational method that we propose in this paper is, in particular, suitable for calculating a circular program that solves the $repmin$ problem. In this section, we calculate such a program.

Our calculational method is used to calculate circular programs from programs that consist in the composition $f \circ g$ of a producer g and a consumer f, where $g :: a \rightarrow (b,z)$ and $f :: (b,z) \rightarrow c$.

In order to be able to apply our method to $repmin$, we then need to slightly change the straightforward solution presented earlier. In that solution, the consumer (function $replace$) fits the desired structure; however, no explicit producer occurs, since the input tree is copied as an argument to function $replace$.

We then define the following solution to $repmin$:

$$
\begin{array}{l}
transform :: LeafTree \rightarrow LeafTree \\
transform\ t = replace \circ tmint\ \$\ t
\end{array}
$$

$$
\begin{array}{l}
tmint :: LeafTree \rightarrow (LeafTree, Int) \\
tmint\ (Leaf\ n) \qquad = (Leaf\ n \qquad , n) \\
tmint\ (Fork\ (l,r)) = (Fork\ (l',r'), min\ n_1\ n_2) \\
\qquad\quad \mathbf{where}\ (l',\ n_1) = tmint\ l \\
\qquad\qquad\qquad\ \ (r',n_2) = tmint\ r
\end{array}
$$

$$
\begin{array}{l}
replace :: (LeafTree, Int) \rightarrow LeafTree \\
replace\ (Leaf\ \_\quad , m) = Leaf\ m \\
replace\ (Fork\ (l,r),m) = Fork\ (replace\ (l,m), \\
\qquad\qquad\qquad\qquad\qquad\qquad replace\ (r,m))
\end{array}
$$

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

Although the original solution needs to be slightly modified, so that it is possible to apply our method to $repmin$, we present such a modified version, and the circular program we calculate from it, since $repmin$ is very intuitive, and, by far, the most well-known motivational example for circular programming. In the remaining of this paper we will present more realistic examples (in Sections 5 and 6), where the gluing trees need to grow from traversal to traversal. This fact forces the definition of new data-structures in order to glue the different traversals together. Therefore, our rule directly applies to them.

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

First we observe that we can rewrite the original definition of $transform$ as follows:

$$
\begin{array}{l}
transform\ t \\
\quad = replace\ (tmint\ t) \\
\quad = replace\ (\pi_1\ (tmint\ t), \pi_2\ (tmint\ t)) \\
\quad = replace' \circ \pi_1 \circ tmint\ \$\ t \\
\qquad \mathbf{where}\ replace'\ x = replace\ (x,m) \\
\qquad\qquad\qquad\ \ m = \pi_2\ (tmint\ t) \\
\quad = \pi_1 \circ (replace' \ \times\ id) \circ tmint\ \$\ t \\
\qquad \mathbf{where}\ replace'\ x = replace\ (x,m) \\
\qquad\qquad\qquad\ \ m = \pi_2\ (tmint\ t)
\end{array}
$$

where π_1 and π_2 are the projection functions and $(f\ \times\ g)\ (x,y) = (f\ x, g\ y)$. Therefore, we can redefine $transform$ as:

$$
\begin{array}{l}
transform\ t = nt \\
\quad \mathbf{where} \\
\qquad (nt,\_) = repm\ t \\
\qquad repm\ t = (replace' \ \times\ id) \circ tmint\ \$\ t \\
\qquad replace'\ x = replace\ (x,m) \\
\qquad m = \pi_2\ (tmint\ t)
\end{array}
$$

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

$$
\begin{array}{l}
transform\ t = nt \\
\quad \mathbf{where} \\
\qquad (nt,\_) = repm\ t \\
\qquad m = \pi_2\ (tmint\ t) \\
\qquad repm\ (Leaf\ n) = (Leaf\ m, n) \\
\qquad repm\ (Fork\ (l,r)) = \mathbf{let}\ (l',n_1)\ = repm\ l \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ (r',n_2) = repm\ r \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ (Fork\ (l',r'), min\ n_1\ n_2)
\end{array}
$$

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

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

$$\pi_2 \circ tmint = \pi_2 \circ repm \tag{1}$$

This may seem a particular observation for this specific case but it is a property that holds in general for all transformed programs of this kind. In fact, later on we will see that $tmint$ and $repm$ are both instances of a same polymorphic function and actually this

equality is a consequence of a free theorem (Wadler 1989) about that function. Using this equality we may substitute $tmint$ by $repm$ in the new version of $transform$, finally obtaining:

$$
\begin{array}{l}
transform\ t = nt \\
\quad \mathbf{where} \\
\quad\quad (nt, m) = repm\ t \\
\quad\quad repm\ (Leaf\ n) = (Leaf\ m, n) \\
\quad\quad repm\ (Fork\ (l, r)) = \mathbf{let}\ (l', n_1)\ = repm\ l \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (r', n_2) = repm\ r \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{in}\ (Fork\ (l', r'), min\ n_1\ n_2)
\end{array}
$$

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

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

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

3. Program schemes

Our method will be applied to a class of expressions that will be characterized in terms of program schemes. This will allow us to give a generic formulation of the transformation rule in the sense that it will be parametric in the structure of the intermediate data type involved in the function composition to be transformed.

In this section we describe two program schemes which capture structurally recursive functions and are relevant constructions in our transformation. Throughout we shall assume we are working in the context of a lazy functional language with a *cpo* semantics, in which types are interpreted as pointed cpos (complete partial orders with a least element $\perp$) and functions are interpreted as continuous functions between pointed cpos. However, our semantics differs from that of Haskell in that we do not consider lifted cpos. That is, unlike the semantics of Haskell, we do not consider lifted products and function spaces. As usual, a function f is said to be *strict* if it preserves the least element, i.e. $f\ \perp = \perp$.

3.1 Data types

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

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

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.

$$
\begin{array}{l}
map\ \ :: (a \to b) \to [a] \to [b] \\
map\ f\ [\,] \quad\quad = [\,] \\
map\ f\ (a : as) = f\ a : map\ f\ as
\end{array}
$$

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:

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

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:

$$
\begin{array}{l}
\pi_1 :: (a, b) \to a \\
\pi_1\ (a, b) = a \\
\\
\pi_2 :: (a, b) \to b \\
\pi_2\ (a, b) = b \\
\\
(\vartriangle) :: (c \to a) \to (c \to b) \to c \to (a, b) \\
(f\ \vartriangle\ g)\ c = (f\ c, g\ c)
\end{array}
$$

Among others properties, it holds that

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

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

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

Another case of 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.

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

$$
\begin{array}{l}
(+) :: (a \to c) \to (b \to d) \to (a + b) \to (c + d) \\
(f + g)\ (Left\ a) = Left\ (f\ a) \\
(f + g)\ (Right\ b) = Right\ (g\ b)
\end{array}
$$

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

$$
\begin{array}{l}
(\triangledown) :: (a \to c) \to (b \to c) \to (a + b) \to c \\
(f\ \triangledown\ g)\ (Left\ a) = f\ a \\
(f\ \triangledown\ g)\ (Right\ b) = g\ b
\end{array}
$$

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

We consider declarations of datatypes of the form:[1]

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

where each $\tau_{i,j}$ is restricted to be a constant type (like Int or $Char$), a type variable, a type constructor D applied to a type $\tau'_{i,j}$ or τ itself. Datatypes of this form are usually called regular. The derivation of a functor that captures the structure of the datatype essentially proceeds as follows: alternatives are regarded as sums ($\mid$ is replaced by $+$) and occurrences of τ 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:

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

[1] For simplicity we shall assume that constructors in a datatype declaration are declared uncurried.

where $\sigma_{i,j} = \tau_{i,j}[\tau := a]$[2]. The body of the corresponding mapping function $F :: (a \to b) \to (F\ a \to F\ b)$ is similar to that of $F\ a$, with the difference that the occurrences of the type variable a are replaced by a function $f :: a \to b$, and identities are placed in the other positions:

$$Ff = g_{1,1} \times \cdots \times g_{1,k_1} \ + \cdots + \ g_{n,1} \times \cdots \times g_{n,k_n}$$

with

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

where the D in the expression $D\ g'_{i,j}$ represents the map function $D :: (a \to b) \to (D\ a \to D\ b)$ corresponding to the type constructor D.

For example, for the type of leaf trees

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

we can derive a functor T given by

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

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

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

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

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

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

Every recursive datatype 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 datatype 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

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

$$\begin{aligned} &out_T :: \mathit{LeafTree} \to T\ \mathit{LeafTree} \\ &out_T\ (\mathit{Leaf}\ n) = \mathit{Left}\ n \\ &out_T\ (\mathit{Fork}\ (l, r)) = \mathit{Right}\ (l, r) \end{aligned}$$

3.2 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* (or *catamorphism*), denoted by $fold\ h :: \mu F \to a$, is defined as the least function f that satisfies the following equation:

$$f \circ in_F = h \circ F\ f$$

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

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

A function $h :: F\ a \to a$ is called an *F-algebra*.[3] The functor F plays the role of 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 $f \circ h = k \circ F\ f$. Notice that $fold\ h$ is a homomorphism between the algebras in_F and h.

For example, for leaf trees fold is given by:

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

For instance,

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

can be defined as:

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

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

LAW 3.1 (FOLD/BUILD RULE). *For h strict,*

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

where

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

The instance of this law for leaf trees is the following:

$$fold_T\ (h_1, h_2) \circ build_T\ g = g\ (h_1, h_2) \tag{5}$$

where

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

The assumption about the strictness of the algebra disappears because every algebra $h_1 \triangledown h_2$ is strict as so is every case analysis.

As an example, we can use this law to fuse:

$$tmm = tmin \circ mirror$$

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

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

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

To do so, first we have to express $mirror$ in terms of $build_T$:

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

Finally, by (5) we have that

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

Inlining,

$$
\begin{array}{l}
tmm\ (Leaf\ n) = n \\
tmm\ (Fork\ (l, r)) = min\ (tmm\ r)\ (tmm\ l)
\end{array}
$$

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

LAW 3.2 (FOLD/BUILDP RULE). *For h strict,*

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

where

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

Proof *From the polymorphic* **type of** $\$\ g\ \$$

we can deduce the following free theorem: for f strict,

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

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

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

$$(fold_T\ (h_1, h_2)\ \times\ id) \circ buildp_T\ g = g\ (h_1, h_2) \qquad (6)$$

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 (5).

To see an example of the application of this law, consider the function $ssqm$:

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

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

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

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

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

Hence, by (6),

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

Inlining,

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

Finally, the following property is an immediate consequence of Law 3.2.

LAW 3.3. *For any strict h,*

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

Proof

$$
\begin{array}{ll}
 & \pi_2 \circ g\ in_F \\
= & \qquad \{\ (3)\ \} \\
 & \pi_2 \circ (fold\ h\ \times\ id) \circ g\ in_F \\
= & \qquad \{\ \text{Law 3.2}\ \} \\
 & \pi_2 \circ g\ h \qquad\qquad\qquad\qquad \square
\end{array}
$$

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.

3.3 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:

$$
\begin{array}{l}
f :: (\mu F, z) \to a \\
f\ (t, z) = fold\ h\ t
\end{array}
$$

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

$$replace :: (LeafTree, Int) \to LeafTree$$

$$replace\ (Leaf\ n, m) = Leaf\ m$$
$$replace\ (Fork\ (l, r), m) = Fork\ (replace\ (l, m),\ replace\ (r, m))$$

which can be defined as:

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

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

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

This is an alternative we won't pursue in this paper.

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

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

that satisfies certain coherence axioms (see (Cockett and Fukushima 1992; Cockett and Spencer 1991; Pardo 2002) for 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:

$$\tau^T :: (T\ a, z) \rightarrow T\ (a, z)$$
$$\tau^T\ (Left\ n, z) = Left\ n$$
$$\tau^T\ (Right\ (a, a'), z) = Right\ ((a, z), (a', z))$$

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

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

Observe that now function h also accepts the value of the parameters. It is a function of the form $(h_1 \triangledown \ldots \triangledown h_n) \circ d$ where each $h_i :: (F_i\ a, z) \rightarrow a$ if $F\ a = F_1\ a + \cdots + F_n\ a$, and $d :: (x_1 + \cdots + x_n, z) \rightarrow (x_1, z) + \cdots + (x_n, z)$ 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.

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

$$pfold_T :: ((Int, z) \rightarrow a, ((a, a), z) \rightarrow a) \rightarrow (LeafTree, z) \rightarrow a$$
$$pfold_T\ (h_1, h_2) = p_T$$
$$\textbf{where}$$
$$p_T\ (Leaf\ n, z) = h_1\ (n, z)$$
$$p_T\ (Fork\ (l, r), z) = h_2\ ((p_T\ (l, z), p_T\ (r, z)), z)$$

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

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

The following equation shows one of the possible relationships between pfold and fold.

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

Like fold, pfold satisfies a set of algebraic laws. We don't show any of them here as they are not necessary for this paper. The interested reader may consult (Pardo 2002, 2001).

4. The pfold/buildp rule

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

The rule makes some natural assumptions about $cons$ and $prod$: t is a recursive data type μF, the consumer $cons$ is defined by structural recursion on t, 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: $pfold\ h \circ buildp\ g$.

LAW 4.1 (PFOLD/BUILDP RULE). *For any* $h = (h_1 \triangledown \ldots \triangledown h_n) \circ d$,

$$g :: \forall\ a\ .\ (F\ a \rightarrow a) \rightarrow c \rightarrow (a, z)$$
$$\Rightarrow$$
$$pfold\ h \circ buildp\ g\ \$\ c = v\ \ \textbf{where}\ (v, z) = g\ k\ c,\ k = k_1 \triangledown \ldots \triangledown k_n,\ k_i\ x = h_i\ (x, z)$$

Proof The proof will show in detail the two steps of our method. The first step corresponds to the application of deforestation, which is represented by Law 3.2. For that reason we need first to express the pfold as a fold.

$pfold\ h \circ buildp\ g\ \$\ c$
$=$ { definition of $buildp$ }
$pfold\ h \circ g\ in_F\ \$\ c$
$=$ { (4) }
$pfold\ h \circ (((\pi_1 \circ g\ in_F) \vartriangle (\pi_2 \circ g\ in_F)))\ \$\ c$
$=$ { (7) }
$fold\ k \circ \pi_1 \circ g\ in_F\ \$\ c\ \ \textbf{where}\ z = \pi_2 \circ g\ in_F\ \$\ c,\ k_i\ x = h_i\ (x, z)$
$=$ { (2) }
$\pi_1 \circ (fold\ k \times id) \circ g\ in_F\ \$\ c\ \ \textbf{where}\ z = \pi_2 \circ g\ in_F\ \$\ c,\ k_i\ x = h_i\ (x, z)$
$=$ { Law 3.2 }
$\pi_1 \circ g\ k\ \$\ c\ \ \textbf{where}\ z = \pi_2 \circ g\ in_F\ \$\ c$

$$k_i\ x = h_i\ (x, z)$$

Law 3.2 was applicable because by construction the algebra k is strict.

Once we have reached this point we observe that the resulting program is deforested, but it contains two traversals on c. The elimination of the multiple traversals is then performed by introducing a circular definition. The essential property that makes it possible the safe introduction of a circularity is Law 3.3, which states that the computation of the second component of type z is independent of the particular algebra that is passed to g. This is a consequence of the polymorphic type of g. Therefore, we can replace in_F by another algebra and we will continue producing the same value z. In particular, we can take k as this other algebra, and in that way we are introducing the circularity. It is this property that ensures that no terminating program is turned into a nonterminating one.

$$\begin{array}{l}
\pi_1 \circ g\ k\ \$\ c \\
\quad \textbf{where}\ z = \pi_2 \circ g\ in_F\ \$\ c \\
\qquad\quad k_i\ x = h_i\ (x, z) \\
= \quad \{\ \text{Law 3.3}\ \} \\
\pi_1 \circ g\ k\ \$\ c \\
\quad \textbf{where}\ z = \pi_2 \circ g\ k\ \$\ c \\
\qquad\quad k_i\ x = h_i\ (x, z) \\
= \quad \{\ (4)\ \} \\
v \\
\quad \textbf{where}\ (v, z) = g\ k\ c \\
\qquad\quad k_i\ x = h_i\ (x, z) \qquad \square
\end{array}$$

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

$$\begin{array}{l}
transform :: LeafTree \to LeafTree \\
transform\ t = replace \circ tmint\ \$\ t
\end{array}$$

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

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

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

Therefore, by applying Law 4.1 we get:

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

Inlining, we obtain the definition we showed previously in Section 2.2:

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

5. The Increase Average Merge-Sort Problem

In this section, we show the application of our method to another programming problem.

Consider the following problem over lists of numbers:

(i) We want to increase the elements of a list by the list's average value. For example, for the list $[8, 4, 6]$ we would produce the list $[14, 10, 12]$, as the average value is 6.

(ii) We want the output list to be returned in ascending order. Therefore, from the list $[8, 4, 6]$, we would have to produce the list $[10, 12, 14]$.

This problem may be understood as a variation of a sorting algorithm on lists that increases all elements in a list by the list's average. We call the problem *Increase Average Merge-Sort* (or *incavgMS*, for short) because of the use of merge-sort as sorting algorithm.

A straightforward solution to this problem would rely on the following strategy:

1. traverse the input list in order to compute its sum and length (these values are needed to compute the list's average);
2. following (Augusteijn 1998), implement merge-sort using a leaf tree that contains the numbers in the input list;
3. traverse the leaf tree, increasing all its elements by the input list's average (calculated using the sum and length already computed) while sorting the increased values.

In summary, this solution can be implemented in this form:

$$\begin{array}{l}
incavgMS :: (Ord\ b, Fractional\ b) \Rightarrow [Int] \to [b] \\
incavgMS\ [\,] = [\,] \\
incavgMS\ xs = incsort \circ ltreesumlen\ \$\ xs
\end{array}$$

According to the strategy, the function *ltreesumlen* must compute a leaf tree containing the elements of the input list. It is clear that the input list could also be used as the intermediate data structure that glues the two functions together. The reason for introducing a leaf tree is to obtain a $O(n \log n)$ sorting algorithm, instead of a quadratic solution one would obtain by using a list as the intermediate structure. However, to achieve the desired $O(n \log n)$ behavior, the elements occurring in the constructed leaf tree must be *evenly distributed*, i.e., the computed leaf tree must be balanced, under certain criteria.

In addition to computing such a leaf tree, *ltreesumlen* must also compute the sum and length values of the input list.

$$\begin{array}{l}
ltreesumlen :: [Int] \to (LeafTree, (Int, Int)) \\
ltreesumlen\ [x] \\
\quad = (Leaf\ x, (x, 1)) \\
ltreesumlen\ xs \\
\quad = \textbf{let}\ (xs_1, xs_2)\ \ = splitl\ xs \\
\qquad\quad (t_1, (s_1, l_1)) = ltreesumlen\ xs_1 \\
\qquad\quad (t_2, (s_2, l_2)) = ltreesumlen\ xs_2 \\
\qquad \textbf{in}\ (Fork\ (t_1, t_2), (s_1 + s_2, l_1 + l_2))
\end{array}$$

$$\begin{array}{l}
splitl :: [a] \to ([a], [a]) \\
splitl\ [\,] = ([\,], [\,]) \\
splitl\ (a : as) = (zs, a : ys) \\
\quad \textbf{where}\ (ys, zs) = splitl\ as
\end{array}$$

The auxiliary function $splitl$ splits a list xs into two sublists xs_1 and xs_2 such that:

$$xs_1 +\!\!+ xs_2 \text{ `}isPermutation\text{` } xs$$

$$length\ xs_1 \leqslant length\ xs_2 \leqslant length\ xs_1 + 1$$

The last property guarantees that the tree generated by $ltreesumlen$ is balanced.

Once we have computed the intermediate (balanced) leaf tree and the input list's sum and length, we traverse the leaf tree, increasing all its elements by the average value while sorting the list that is being produced as output. These actions are performed by function $incsort$.

$$\begin{aligned}
&incsort :: (Ord\ b, Fractional\ b)\\
&\qquad \Rightarrow (LeafTree, (Int, Int)) \rightarrow [b]\\
&incsort\ (Leaf\ n, (s, l)) = [n + s\ /\ l]\\
&incsort\ (Fork\ t_1\ t_2, p) = merge\ (incsort\ (t_1, p))\\
&\qquad\qquad\qquad\qquad\qquad\qquad (incsort\ (t_2, p))
\end{aligned}$$

$$\begin{aligned}
&merge :: (Ord\ a) \Rightarrow [a] \rightarrow [a] \rightarrow [a]\\
&merge\ [\,]\ m = m\\
&merge\ l\ [\,] = l\\
&merge\ (x : xs)\ (y : ys) \mid x < y \quad\ = x : merge\ xs\ (y : ys)\\
&\qquad\qquad\qquad\qquad\quad \mid otherwise = y : merge\ (x : xs)\ ys
\end{aligned}$$

To apply our method we first have to write $incsort$ as a $pfold_T$ and $ltreesumlen$ in terms of $buildp_T$:

$$\begin{aligned}
&incsort = pfold_T\ (h_1, h_2)\\
&\quad \textbf{where}\ h_1\ (n, (s, l)) = [n + s\ /\ l]\\
&\qquad\quad\ h_2\ ((ys, zs), _) = merge\ ys\ zs
\end{aligned}$$

$$\begin{aligned}
&ltreesumlen\\
&\quad = buildp_T\ g\\
&\quad\ \textbf{where}\ g\ (leaf, fork)\ [x]\\
&\qquad\qquad\quad = (leaf\ x, (x, 1))\\
&\qquad\qquad g\ (leaf, fork)\ xs\\
&\qquad\qquad\quad = \textbf{let}\ (xs_1, xs_2) \quad = splitl\ xs\\
&\qquad\qquad\qquad\qquad\ (t_1, (s_1, l_1)) = g\ (leaf, fork)\ xs_1\\
&\qquad\qquad\qquad\qquad\ (t_2, (s_2, l_2)) = g\ (leaf, fork)\ xs_2\\
&\qquad\qquad\qquad\ \textbf{in}\ (fork\ (t_1, t_2), (s_1 + s_2, l_1 + l_2))
\end{aligned}$$

By direct application of Law 4.1 to the $incavgMS\ \ xs$, for $x \neq [\,]$, we obtain the following program:

$$\begin{aligned}
&incavgMS\ xs = ys\\
&\quad \textbf{where}\\
&\qquad (ys, (s, l)) = gk\ xs\\
&\qquad gk\ [x] \quad\ = ([x + s\ /\ l], (x, 1))\\
&\qquad gk\ xs \quad\ = \textbf{let}\ (xs_1, xs_2) = splitl\ xs\\
&\qquad\qquad\qquad\ (ys1, (s_1, l_1)) = gk\ xs_1\\
&\qquad\qquad\qquad\ (ys2, (s_2, l_2)) = gk\ xs_2\\
&\qquad\qquad\ \textbf{in}\ (merge\ ys1\ ys2, (s_1 + s_2, l_1 + l_2))
\end{aligned}$$

We may observe that the leaf tree that was previously used to glue the functions $incsort$ and $ltreesumlen$ has been deforested. Furthermore, we observe that such deforestation did not introduce multiple traversals over the input list: it is traversed only once.

6. Algol 68 scope rules

In this section, we consider the application of our rule to a real example: the Algol 68 scope rules (de Moor et al. 2000; Saraiva 1999). These rules are used, for example, in the Eli system [4] (Kastens et al. 2007) to define a generic component for the name analysis task of a compiler.

We wish to construct a program to deal with the scope rules of a block structured language, the Algol 68. In this language a definition of an identifier x is visible in the smallest enclosing block, with the exception of local blocks that also contain a definition of x. In the latter 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:

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

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. According to these 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 an *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 is 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.

As a consequence, semantic errors resulting from duplicate definitions are 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

[4] A well known compiler generator toolbox.

the first to the second traversal of a block. This is also achieved by defining and constructing an intermediate data structure.

The abstract language may be described by the following recursive data type definitions:

$$\begin{aligned}\textbf{data}\ Its\ &= NilIts\ () \\ &\mid Use\ \ (Var, Its) \\ &\mid Decl\ \ (Var, Its) \\ &\mid Block\ (Its\ \ , Its)\end{aligned}$$

$$\textbf{type}\ Var = String$$

In order to pass the necessary information from the first to the second traversal of a block, we define the following intermediate data structure:

$$\begin{aligned}\textbf{data}\ Its_2 = &\ NilIts_2\ () \\ \mid &\ Use_2\ \ (Var\ \ \ \ , Its_2) \\ \mid &\ Decl_2\ \ ([\,Var\,]\ \ \ , Its_2) \\ \mid &\ Block_2\ ((Int, Its), Its_2)\end{aligned}$$

Errors resulting from duplicate declarations, computed in the first traversal, are passed to the second, using constructor $Decl_2$'s list of variables. The level of a nested block, as well as the instructions it defines, are passed to the second traversal using constructor $Block_2$'s pairs containing an integer and a sequence of instructions.

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

$$\begin{aligned}&semantics :: Its \to [\,Var\,] \\ &semantics = missing_{decls} \circ (duplicate_{decls}\ (0, [\,]))\end{aligned}$$

The function $duplicate_{decls}$ detects duplicate variable declarations by collecting all the declarations occurring in a block. It is defined as follows[5]:

$$\begin{aligned}&duplicate_{decls} :: (Int, [(Var, Int)]) \\ &\qquad \to Its \to (Its_2, [(Var, Int)]) \\ &duplicate_{decls}\ (lev, dcli)\ (Use\ (var, its)) \\ &\quad = (Use_2\ (var, its_2), dclo) \\ &\quad \textbf{where}\ (its_2, dclo) = duplicate_{decls}\ (lev, dcli)\ its\end{aligned}$$

$$\begin{aligned}&duplicate_{decls}\ (lev, dcli)\ (Decl\ (var, its)) \\ &\quad = (Decl_2\ (errs_1, its_2), dclo) \\ &\quad \textbf{where}\ errs_1 \qquad = mNBIn\ (var, lev, dcli) \\ &\qquad\ \ (its_2, dclo) = duplicate_{decls}\ (lev, (var, lev) : dcli)\ its\end{aligned}$$

$$\begin{aligned}&duplicate_{decls}\ (lev, dcli)\ (Block\ (nested, its)) \\ &\quad = (Block_2\ ((lev + 1, nested), its_2), dclo) \\ &\quad \textbf{where}\ (its_2, dclo) = duplicate_{decls}\ (lev, dcli)\ its\end{aligned}$$

$$\begin{aligned}&duplicate_{decls}\ (lev, dcli)\ (NilIts\ ()) \\ &\quad = (NilIts_2\ (), dcli)\end{aligned}$$

Besides detecting the invalid declarations, the $duplicate_{decls}$ function also computes a data structure, of type Its_2, that is later traversed in order to detect variables that are used without being declared. This detection is performed by function $missing_{decls}$, that is defined such as [6]:

$$\begin{aligned}&missing_{decls} :: (Its_2, [(Var, Int)]) \to [\,Var\,] \\ &missing_{decls}\ (Use_2\ (var, its_2), env) \\ &\quad = errs_1 +\!\!+ errs_2 \\ &\quad \textbf{where}\ errs_1 = mBIn\ (var, env) \\ &\qquad\ \ errs_2 = missing_{decls}\ (its_2, env)\end{aligned}$$

$$\begin{aligned}&missing_{decls}\ (Decl_2\ (errs_1, its_2), env) \\ &\quad = errs_1 +\!\!+ errs_2 \\ &\quad \textbf{where}\ errs_2 = missing_{decls}\ (its_2, env)\end{aligned}$$

$$\begin{aligned}&missing_{decls}\ (Block_2\ ((lev, its), its_2), env) \\ &\quad = errs_1 +\!\!+ errs_2 \\ &\quad \textbf{where} \\ &\qquad errs_1 = missing_{decls} \circ (duplicate_{decls}\ (lev, env))\ \$\ its \\ &\qquad errs_2 = missing_{decls}\ (its_2, env)\end{aligned}$$

$$missing_{decls}\ (NilIts_2\ (), \_) = [\,]$$

This solution uses an Its_2 data structure as gluing data structure. So, to apply our rule, we first have to express the functions $duplicate_{decls}$ and $missing_{decls}$ in terms of pfold and buildp for Its_2 structures, respectively. The functor that captures the structure of Its_2 trees is:

$$I\ a = () + (Var, a) + ([\,Var\,], a) + ((Int, Its), a)$$

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

Pfold and buildp for Its_2 trees are then given by:

$$\begin{aligned}&pfold_I :: \ (((), z) \to a, ((Var, a), z) \to a, (([\,Var\,], a), z) \to a, \\ &\qquad\qquad (((Int, Its), a), z) \to a) \to (Its_2, z) \to a \\ &pfold_I\ (h_1, h_2, h_3, h_4) = p_I \\ &\quad \textbf{where}\ p_I\ (NilIts_2\ (), env) \\ &\qquad\qquad\ \ = h_1\ ((), env) \\ &\qquad\qquad p_I\ (Use_2\ (var, its_2), env) \\ &\qquad\qquad\ \ = h_2\ ((var, p_I\ (its_2, env)), env) \\ &\qquad\qquad p_I\ (Decl_2\ (errs_1, its_2), env) \\ &\qquad\qquad\ \ = h_3\ ((errs_1, p_I\ (its_2, env)), env) \\ &\qquad\qquad p_I\ (Block_2\ ((lev, its), its_2), env) \\ &\qquad\qquad\ \ = h_4\ (((lev, its), p_I\ (its_2, env)), env)\end{aligned}$$

$$\begin{aligned}&buildp_I :: (\forall\ a\ .\ (() \to a, (Var, a) \to a, ([\,Var\,], a) \to a, \\ &\qquad\qquad ((Int, Its), a) \to a) \to c \to (a, z)) \to c \to (Its_2, z) \\ &buildp_I\ g = g\ (NilIts_2, Use_2, Decl_2, Block_2)\end{aligned}$$

We may now write $missing_{decls}$ and $duplicate_{decls}$ in terms of them:

$$\begin{aligned}&missing_{decls} = pfold_I\ (h_1, h_2, h_3, h_4) \\ &\quad \textbf{where} \\ &\qquad h_1\ ((), \_) = [\,] \\ &\qquad h_2\ ((var, errs_2), env) \quad = mBIn\ (var, env) +\!\!+ errs_2 \\ &\qquad h_3\ ((errs_1, errs_2), env) = errs_1 +\!\!+ errs_2 \\ &\qquad h_4\ (((lev, its), errs_2), env) \\ &\qquad\quad = \textbf{let}\ errs_1 \\ &\qquad\qquad\quad = missing_{decls} \circ (duplicate_{decls}\ (lev, env))\ \$\ its \\ &\qquad\quad\ \ \textbf{in}\ errs_1 +\!\!+ errs_2\end{aligned}$$

$$\begin{aligned}&duplicate_{decls} :: (Int, [(Var, Int)]) \\ &\qquad\qquad \to Its \to (Its_2, [(Var, Int)]) \\ &duplicate_{decls}\ (lev, dcli) = buildp_I\ (g\ (lev, dcli))\end{aligned}$$

$$\begin{aligned}&g\ (lev, dcli)\ (nil_2, use_2, decl_2, block_2)\ (Use\ (var, its)) \\ &\quad = (use_2\ (var, its_2), dclo)\end{aligned}$$

[5] The auxiliary function $mNBIn$ checks that an identifier *must not be in* a particular level, in the environment.

[6] The auxiliary function $mBIn$ checks that an identifier *must be in* the environment, at any level.

$$
\begin{aligned}
&\textbf{where } (its_2, dclo) = g\ (lev, dcli) \\
&\qquad\qquad (nil_2, use_2, decl_2, block_2)\ its
\end{aligned}
$$

$$
\begin{aligned}
&g\ (lev, dcli)\ (nil_2, use_2, decl_2, block_2)\ (Decl\ (var, its)) \\
&\quad = (decl_2\ (errs_1, its_2), dclo) \\
&\quad \textbf{where } errs_1 = mNBIn\ (var, lev, dcli) \\
&\qquad (its_2, dclo) = g\ (lev, (var, lev) : dcli) \\
&\qquad\qquad (nil_2, use_2, decl_2, block_2)\ its
\end{aligned}
$$

$$
\begin{aligned}
&g\ (lev, dcli)\ (nil_2, use_2, decl_2, block_2)\ (Block\ (nested, its)) \\
&\quad = (block_2\ ((lev + 1, nested), its_2), dclo) \\
&\quad \textbf{where } (its_2, dclo) = g\ (lev, dcli) \\
&\qquad\qquad (nil_2, use_2, decl_2, block_2)\ its
\end{aligned}
$$

$$
\begin{aligned}
&g\ (lev, dcli)\ (nil_2, use_2, decl_2, block_2)\ (NilIts\ ()) \\
&\quad = (nil_2\ (), dcli)
\end{aligned}
$$

Recall the definition we want to transform:

$$
\begin{aligned}
&semantics :: Its \rightarrow [\,Var\,] \\
&semantics = missing_{decls} \circ (duplicate_{decls}\ (0, [\,]))
\end{aligned}
$$

and notice that we have just given this composition an explicit $pfold \circ buildp$ form. By application of Law 4.1 to the above definition, we obtain the circular program:

$$
\begin{aligned}
&semantics\ its = errors \\
&\quad \textbf{where } (errors, env) = g\ (0, [\,])\ k\ its \\
&\qquad k_i\ x = h_i\ (x, env)
\end{aligned}
$$

The circularity that has been introduced by our law eliminates the construction of the intermediate data structure that was used to glue the two traversals toghether. In fact, it guarantees (except for its nested blocks) that the input sentence is traversed only once. However, we may still notice, from the definition of h_4,

$$
\begin{aligned}
&h_4\ (((lev, its), errs_2), env) \\
&\quad = \textbf{let } errs_1 \\
&\qquad\quad = missing_{decls} \circ (duplicate_{decls}\ (lev, env))\ \$\ its \\
&\qquad \textbf{in } errs_1 +\!\!+ errs_2
\end{aligned}
$$

that multiple traversals still occur in the calculated program, as well as the production of intermediate structures. So, there is still an opportunity to introduce a circularity in the program. In fact, functions $missing_{decls}$ and $duplicate_{decls}$ have already been written in terms of pfold and buildp. We may, then, directly apply Law 4.1 to such composition. We get a circular definition for h_4:

$$
\begin{aligned}
&h_4\ (((lev, its), errs_2), env) \\
&\quad = \textbf{let } (errs_1, env') = g\ (lev, env)\ k\ its \\
&\qquad\quad k_i\ x = h_i\ (x, env') \\
&\qquad \textbf{in } errs_1 +\!\!+ errs_2
\end{aligned}
$$

The introduction of this circularity eliminates the construction of the intermediate structure that was produced when a nested block was assigned the desired semantics. It guarantees that nested blocks are traversed only once, too.

As a consequence, the calculated circular program performs a single traversal and no gluing data structure is constructed.

We may now say that, in the circular version of the program that assigns semantics to block sentences (calculated using our rule), each sentence is traversed only once and no gluing intermediate data structure is constructed.

7. Conclusions

In this paper we have presented a new program transformation technique for intermediate structure elimination. The programs we are able of dealing with consist in the composition of a producer and a consumer functions. The producer constructs an intermediate structure that is later traversed by the consumer. Furthermore, we allow the producer to compute additional values that may be needed by the consumer. This kind of compositions is general enough to deal with a wide number of practical examples. Our approach is calculational, and proceeds in two steps: we apply standard deforestation methods to obtain intermediate structure-free programs and we introduce circular definitions to avoid multiple traversals that are introduced by deforestation. Since that, in the first step, we apply standard fusion techniques, the expressive power of our rule is then bound by deforestation.

We introduce a new calculational rule conceived using a similar approach to the one used in the *fold/build* rule: our rule is also based on parametricity properties of the functions involved. Therefore, it has the same benefits and drawbacks of *fold/build* since it assumes that the functions involved are instances of specific program schemes. Therefore, it could be used, like *fold/build*, in the context of a compiler. In fact, we have used the rewrite rules (RULES pragma) of the Glasgow Haskell Compiler (GHC) in order to obtain a prototype implementation of our fusion rule.

According to Danielsson et al. (2006), the calculation rule we present in this paper is morally correct *only*, in Haskell. In fact, in the formal proof of our rule, surjective pairing (Law (4)) is applied twice to the result of function g. However, (4) is not valid in Haskell: though it holds for defined values, it fails when the result of function g is undefined, because $\perp$ is different from $(\perp, \perp)$ as a consequence of lifted products. Therefore, (4) is morally correct *only* and, in the same sense, so is our rule. We may, however, argue that, for all cases with practical interest (the ones for which function g produces defined results), our rule directly applies in Haskell. Furthermore, due to the presence of seq in Haskell, further strictness pre-conditions may need to be defined in our rule in order to guarantee its correctness in Haskell (Johann and Voigtländer 2004).

The rule that we propose is easy to apply: in this paper, we have presented three examples that show that our rule is effective in its aim. The calculation of circular programs may be understood as an intermediate stage: the circular programs we calculate may be further transformed into very efficient, completely data structure free programs.

References

S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, 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, September 1998.

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

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

R.S. Bird and O. de Moor. *Algebra of Programming*. Prentice Hall, UK, 1997.

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

R. Cockett and D. 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 Press.

Olivier Danvy and Mayer Goldberg. There and back again. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 230–234, New York, NY, USA, 2002. ACM Press.

Oege de Moor, Kevin Backhouse, and S. Doaitse Swierstra. First-class attribute grammars. *Informatica (Slovenia)*, 24(3), 2000.

Oege de Moor, Simon Peyton-Jones, and Eric Van Wyk. Aspect-oriented compilers. *Lecture Notes in Computer Science*, 1799, 2000.

Atze Dijkstra. *Stepping through Haskell*. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, November 2005.

Atze Dijkstra and Doaitse Swierstra. Typing haskell with an attribute grammar (part i). Technical Report UU-CS-2004-037, Institute of Information and Computing Sciences, Utrecht University, 2004.

João Fernandes and João Saraiva. Tools and Libraries to Model and Manipulate Circular Programs. In *Proc. of the ACM SIGPLAN 2007 Workshop on Partial Evaluation and Program Manipulation (PEPM'07)*, pages 102–111. ACM Press, 2007.

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

A. 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.

Patricia Johann and Janis Voigtländer. Free theorems in the presence of seq. In Neil D. Jones and Xavier Leroy, editors, *31st Symposium on Principles of Programming Languages, Venice, Italy, Proceedings*, volume 39 of *SIGPLAN Notices*, pages 99–110. ACM Press, January 2004.

Thomas Johnsson. Attribute grammars as a functional programming paradigm. In *Functional Programming Languages and Computer Architecture*, pages 154–173, 1987.

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

Matthijs Kuiper and Doaitse Swierstra. Using attribute grammars to derive efficient functional programs. In *Computing Science in the Netherlands CSN'87*, November 1987.

John Launchbury and Tim Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA'95, La Jolla, San Diego, CA, USA, 25–28 June 1995*, pages 314–323. ACM Press, New York, 1995.

Julia L. Lawall. Implementing Circularity Using Partial Evaluation. In *Proceedings of the Second Symposium on Programs as Data Objects PADO II*, volume 2053 of *LNCS*, May 2001.

Atsushi Ohori and Isao Sasano. Lightweight fusion by fixed point promotion. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 143–154, New York, NY, USA, 2007. ACM Press.

Chris Okasaki. Breadth-first numbering: lessons from a small exercise in algorithm design. *ACM SIGPLAN Notices*, 35(9): 131–136, 2000.

Y. Onoue, Z. Hu, H. Iwasaki, and M. Takeichi. A Calculational Fusion System HYLO. In *IFIP TC 2 Working Conference on Algorithmic Languages and Calculi, Le Bischenberg, France*, pages 76–106. Chapman & Hall, February 1997.

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

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

João Saraiva. *Purely Functional Implementation of Attribute Grammars*. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, December 1999.

Doaitse Swierstra, Pablo Azero, and João Saraiva. Designing and Implementing Combinator Languages. In Doaitse Swierstra, Pedro Henriques, and José Oliveira, editors, *Third Summer School on Advanced Functional Programming*, volume 1608 of *LNCS Tutorial*, pages 150–206, September 1999.

S. Doaitse Swierstra and Pablo Azero. Attribute grammars in a functional style. In *Systems Implementation 2000*, Berlin, 1998. Chapman & Hall.

A. Takano and E. Meijer. Shortcut to Deforestation in Calculational Form. In *Functional Programming Languages and Computer Architecture'95*, 1995.

Janis Voigtländer. Using circular programs to deforest in accumulating parameters. *Higher-Order and Symbolic Computation*, 17:129–163, 2004. Previous version appeared in *ASIA-PEPM 2002*, Proceedings, pages 126–137, ACM Press, 2002.

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

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