A Shortcut Fusion Approach to Accumulations

Mónica Martínez[1], **Alberto Pardo**[1]

[1] Instituto de Computación, Facultad de Ingeniería
Universidad de la República, Montevideo, Uruguay

{mmartine,pardo}@fing.edu.uy

Abstract. *In functional programming it is common to write programs as composition of other simpler functions. This makes it possible to take advantage of the well-known benefits of modular programming. However, in many cases, the resulting programs have efficiency problems caused by the generation of data structures that are solely used for communication between the intervening functions in the compositions. Many of those intermediate structures can be eliminated by an appropriate combination of the codes of the involved functions using a technique called program fusion. In this work, we propose program fusion techniques for accumulations, which are recursive functions that use additional parameters for keeping intermediate results. Accumulations are known to be difficult to be fused because of the presence of the additional parameters and the fact that in many cases results are computed in those parameters. Our analysis is based on a shortcut fusion approach which turns out to be effective in the case of accumulations. We present benchmarks that illustrate the impact of shortcut fusion on accumulations.*

1. Introduction

In functional programming it is common to write programs as composition of comparatively simple functions. This makes it possible to take advantage of the well-known benefits of modular programming, such as readability and maintainability, but in many cases the resulting programs have efficiency problems, both in terms of execution time and memory consumption, due to the construction of intermediate data structures required for the communication between the functions intervening in the compositions. More efficient definitions can be obtained by the application of a program transformation technique, known as *fusion* or *deforestation*, aiming at the elimination of these intermediate data structures.

There exist well-known fusion methods that successfully perform such transformations for an ample set of functions. However, there is a class of functions that have been traditionally difficult to manipulate for program fusion. These are the so-called *accumulations*, which are functions that use additional parameters where they hold intermediate results or even build the final results. The difficulty with these functions relies in the fact that classical fusion techniques are in general not able to reach the accumulating parameters when accumulations appear as producer functions (i.e. as the ones that generate the intermediate structure), and therefore cannot fully eliminate the intermediate data structures in those cases. There are different proposals and approaches that manage to solve fusion involving accumulations for certain groups of functions, with different success and simplicity levels. Representative examples are [Hu et al. 1999, Gibbons 2000, Svenningsson 2002, Nishimura 2002, Pardo 2003, Voigtländer and Kühnemann 2004, Voigtländer 2004, Katsumata and Nishimura 2006].

In this work we focus on a fusion technique based on an approach known as *shortcut fusion* [Gill et al. 1993]. We pay special attention to the case when accumulations play the role of producer function and propose a particular treatment for this case in the style of shortcut fusion. The solution we present is simple, easy to use and automate. It is worth mentioning that, like standard shortcut fusion, the technique we introduce is generic, meaning that it can be defined in an uniform way for an ample class of datatypes.

Along with this proposal, we perform a set of benchmarks allowing running time comparisons between the transformed and the original programs, thus helping identify program groups for which the application of the technique is effective. Throughout we will use Haskell notation, assuming a cpo semantics (in terms of pointed cpos), but without the presence of the seq function [Johann and Voigtländer 2004].

The paper is organized as follows. We start in Section 2 with a review of the concept of shortcut fusion. In Section 3, by means of specific examples, we study the application of shortcut fusion for programs involving accumulations. The generic formulation of the concepts and laws developed in Sections 2 and 3 are presented in Section 4. Section 5 presents a benchmark that evaluates the application of shortcut fusion to accumulations. Finally, Section 6 concludes the paper.

2. Shortcut fusion

Shortcut fusion [Gill et al. 1993, Gill 1996] is a program transformation technique that allows the elimination of intermediate data structures generated in compositions of the form $c \circ p$ by a suitable combination of the definitions of c (the *consumer*) and p (the *producer*). This technique is a consequence of parametricity properties (known as "free theorems" [Wadler 1989]) associated to polymorphic functions.

For its application, shortcut fusion requires that both the consumer and producer functions satisfy certain (structural) restrictions. The consumer c is required to process all the elements of the intermediate data structure in a uniform way. Concretely, this condition is established by requiring that c is expressible as a *fold* [Bird 1998], a program scheme that captures function definitions by structural recursion. Regarding the producer p, it must be such that the generation of the intermediate data structure is performed using uniquely the constructors of the data type and not whatever values that are passed to it as arguments. For example, if p produces a list as result, then it must ensure that the list is generated only in terms of the list constructors $[\,]$ and $(:)$. To meet this condition the producer is required to be expressible in terms of a so-called *build* function [Gill et al. 1993], which is a function that carries a "template" that exhibits the occurrences of the constructors of the intermediate datatype. The essential idea of shortcut fusion is then to replace, in the template, the occurrences of the intermediate datatype's constructors by appropriate functions that are specified within the consumer. As a result, one obtains a definition that computes the same as the original composition $c \circ p$ but without building the intermediate data structure. Because it fuses compositions of *fold* with *build* this transformation is usually referred to as the *fold/build* law.

Next, we show the definition of this law for lists and leaf-labelled binary trees.

Lists As mentioned above, the consumer must be a structural recursive definition representable as a *fold*. In the case of lists fold has the following definition:

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

This function is equivalent to the well-known *foldr* function [Bird 1998]. It traverses the list and replaces $[\,]$ by the constant nil and the occurrences of $(:)$ by function cons. For example, the function that computes the length of a list:

$$\begin{array}{ll}
\mathit{length}\ [\,] & = 0 \\
\mathit{length}\ (a : \mathit{as}) & = 1 + \mathit{length}\ \mathit{as}
\end{array}$$

can be written as $\mathit{foldL}\ (0, \lambda a\ r \to 1 + r)$.

The producer, on the other hand, must be expressible in terms of a build function:

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

which takes as parameter a function g that abstracts the list constructors from the process that generates the intermediate list. For example, let us consider the function that given a list of values of type $\mathit{Maybe}\ a$ returns a list with the values of type a:

$$\mathbf{data}\ \mathit{Maybe}\ a = \mathit{Nothing} \mid \mathit{Just}\ a$$

$$\begin{array}{ll}
\mathit{collect} & :: [\mathit{Maybe}\ a] \to [a] \\
\mathit{collect}\ [\,] & = [\,] \\
\mathit{collect}\ (m : \mathit{ms}) & = \mathbf{case}\ m\ \mathbf{of} \\
& \qquad \mathit{Nothing} \to \mathit{collect}\ \mathit{ms} \\
& \qquad \mathit{Just}\ a \quad \to a : \mathit{collect}\ \mathit{ms}
\end{array}$$

It can be written as:

$$\begin{array}{l}
\mathit{collect} = \mathit{buildL}\ \mathit{gcoll} \\
\quad \mathbf{where} \\
\qquad \begin{array}{ll}
\mathit{gcoll}\ (\mathit{nil}, \mathit{cons})\ [\,] & = \mathit{nil} \\
\mathit{gcoll}\ (\mathit{nil}, \mathit{cons})\ (m : \mathit{ms}) & = \mathbf{case}\ m\ \mathbf{of} \\
& \qquad \mathit{Nothing} \to \mathit{gcoll}\ (\mathit{nil}, \mathit{cons})\ \mathit{ms} \\
& \qquad \mathit{Just}\ a \quad \to \mathit{cons}\ a\ (\mathit{gcoll}\ (\mathit{nil}, \mathit{cons})\ \mathit{ms})
\end{array}
\end{array}$$

Based on the forms required to the consumer and producer functions it is possible to define the following shortcut fusion law:

Law 1 (fold/build for lists)

$$\mathit{foldL}\ (\mathit{nil}, \mathit{cons}) \circ \mathit{buildL}\ g = g\ (\mathit{nil}, \mathit{cons})$$

To see an example of the application of shortcut fusion, consider the definition of function $\mathit{count} = \mathit{length} \circ \mathit{collect}$, which counts the number of elements of type a that occur in a list of values of type $Maybe\ a$. This function produces an intermediate list with the values of type a. A monolithic definition can be obtained by using Law 1: $count = gcoll\ (0, \lambda a\ r \rightarrow 1 + r)$. Inlining,

$$\begin{array}{ll} count\ [\,] & = 0 \\ count\ (m : ms) & = \mathbf{case}\ m\ \mathbf{of} \\ & \quad \begin{array}{ll} Nothing & \rightarrow count\ ms \\ Just\ a & \rightarrow 1 + count\ ms \end{array} \end{array}$$

Binary trees Leaf-labelled binary trees can be declared by:

$$\mathbf{data}\ Btree\ a = Leaf\ a \mid Join\ (Btree\ a)\ (Btree\ a)$$

The *fold* and *build* functions for this data type are given by:

$$\begin{array}{ll} foldB & :: (a \rightarrow b, b \rightarrow b \rightarrow b) \rightarrow Btree\ a \rightarrow b \\ foldB\ (leaf, join)\ (Leaf\ a) & = leaf\ a \\ foldB\ (leaf, join)\ (Join\ l\ r) & = join\ (foldB\ (leaf, join)\ l)\ (foldB\ (leaf, join)\ r) \end{array}$$

$$\begin{array}{l} buildB \quad :: (\forall\ b\ .\ (a \rightarrow b, b \rightarrow b \rightarrow b) \rightarrow c \rightarrow b) \rightarrow c \rightarrow Btree\ a \\ buildB\ g = g\ (Leaf, Join) \end{array}$$

The shortcut fusion law for this type is then the following:

Law 2 (fold/build for binary trees)

$$foldB\ (leaf, join) \circ buildB\ g = g\ (leaf, join)$$

3. Accumulations

Accumulations are recursive functions that use additional parameters for keeping intermediate values or even generating the final result. The use of this kind of functions is in general motivated by efficiency reasons since in many cases accumulative versions of functions turn out to be more efficient than nonaccumulative ones. In the context of program transformation, this fact has been well exploited by the so-called *accumulation technique* [Burstall and Darlington 1977, Bird 1998] that transforms recursive definitions by the introduction of additional arguments where to compute intermediate results.

A prototypical example of the benefits of accumulations is the definition of a linear-time reverse function. It is well-known that the usual definition of reverse:

$$\begin{array}{ll} reverse & :: [a] \rightarrow [a] \\ reverse\ [\,] & = [\,] \\ reverse\ (a : as) & = reverse\ as +\!\!+ [a] \end{array}$$

is quadratic in the length of the input list due to the presence of list append $(+\!\!+)$. In response to this problem an accumulative linear version can be introduced:[1]

[1]or even derived from the nonaccumulative version by the application of the accumulation technique.

$$reverse\ as = areverse\ (as, [\,])$$

$$\begin{aligned}
&areverse\ ([\,], xs) &&= xs\\
&areverse\ (a : as, xs) &&= areverse\ (as, a : xs)
\end{aligned}$$

In this section we analyse different situations involving accumulations and the effectiveness of shortcut fusion in each case. In Section 4 we present the generic formulation of the different constructions and laws treated here.

3.1. Accumulations as consumers

Accumulations may occur as consumer functions. However, due to the limitation on the form of the consumer imposed by the *fold/build* law, shortcut fusion can be applied in such a case only if the accumulation can be written as a $fold$. Therefore, as $fold$ does not handle accumulating parameters, the only alternative is to write the accumulation as a higher-order fold.

For example, let us consider the accumulative version of the $length$ function:

$$\begin{aligned}
&alength &&:: [a] \to Int \to Int\\
&alength\ [\,]\ \ z &&= z\\
&alength\ (a : as)\ z &&= alength\ as\ (1 + z)
\end{aligned}$$

such that $length\ as = alength\ as\ 0$. Function $alength$ can be written as a fold that returns a function of type $Int \to Int$:

$$alength = foldL\ (id, \lambda a\ r \to r \circ (1+))$$

The problem with this definition as a higher-order fold is that it returns a list of suspended function calls of same dimension as the consumed input list. That is, to compute the length of the input list, this $fold$ first builds a list of function calls, which is then applied to the initial value of the accumulator in order to produce the final integer. Such a computation is therefore less efficient than the one performed by the original definition of $alength$ on the same input.

Unfortunately, the loss of efficiency caused by the representation of accumulations as higher-order folds impacts negatively in the performance of the programs that are obtained when fusing accumulations with producer functions. That is, the resulting fused programs are expected to be, in general, less efficient than the original ones. This is because, as a consequence of the application of shortcut fusion, suspended function calls are inserted in the places corresponding to the constructors within the body of producers. Therefore, the resulting fused programs produce values of functional type which are isomorphic to the intermediate data structures eliminated by fusion.

To see an example, recall the definition of function $count$ and suppose we consider the accumulative definition of $length$. Then,

$$\begin{aligned}
count\ ms &= length\ (collect\ ms)\\
&= alength\ (collect\ ms)\ 0\\
&= foldL\ (id, \lambda a\ r \to r \circ (1+))\ (collect\ ms)\ 0
\end{aligned}$$

$$
\begin{aligned}
&= (\mathit{foldL}\ (\mathit{id}, \lambda a\ r \rightarrow r \circ (1+)) \circ \mathit{buildL}\ \mathit{gcoll})\ \mathit{ms}\ 0 \\
&= \mathit{gcoll}\ (\mathit{id}, \lambda a\ r \rightarrow r \circ (1+))\ \mathit{ms}\ 0
\end{aligned}
$$

Let $lc = \mathit{gcoll}\ (\mathit{id}, \lambda a\ r \rightarrow r \circ (1+))$. Inlining,

$$
\begin{aligned}
&lc\ [\,] &&= \mathit{id} \\
&lc\ (m : ms) &&= \mathbf{case}\ m\ \mathbf{of} \\
& && \quad \mathit{Nothing} \rightarrow lc\ ms \\
& && \quad \mathit{Just}\ a \quad\ \rightarrow lc\ ms \circ (1+)
\end{aligned}
$$

In the benchmarks presented in Section 5 we compare execution times between original and fused programs that involve accumulations. In those benchmarks it can be observed that in various cases (but not in all ones) less efficient programs are obtained from applying fusion in a *fold/build* style to programs that have accumulations as consumers.

In the context of shortcut fusion, there is an alternative treatment of accumulations as consumers using a dual law called *destroy/unfold* (see [Svenningsson 2002] for details).

3.2. Accumulations as producers

The effectiveness of fusion in situations where accumulations appear as producer functions depends on the kind of accumulation. In fact, two cases can be identified: (i) when the accumulating parameters are used to hold auxiliary values that are eventually used in the final result; and (ii) when the result of the function, or part of it, is constructed in the accumulating parameters and therefore they are of the same type as the result. Of course, there may be mixed situations of functions that have some accumulating parameters of kind (i) and others of kind (ii), but these will simply require a mixed treatment.

In case (i), the fact that the accumulating parameters and the result of the function are of a different type makes it possible to apply standard shortcut fusion without even being necessary to identify that the producer is an accumulation.

Let us see an example. Consider the function that computes the height of a binary tree in two steps: first, it replaces the value of each leaf by its depth, and then it computes the maximum of the tree.

$$
\begin{aligned}
&\mathit{height} &&:: \mathit{Btree}\ a \rightarrow \mathit{Int} \\
&\mathit{height}\ t &&= \mathit{aheight}\ (t, 0)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{aheight} :: (\mathit{Btree}\ a, \mathit{Int}) \rightarrow \mathit{Int} \\
&\mathit{aheight} = \mathit{maxBtree} \circ \mathit{depths}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{maxBtree} &&:: \mathit{Ord}\ a \Rightarrow \mathit{Btree}\ a \rightarrow a \\
&\mathit{maxBtree}\ (\mathit{Leaf}\ a) &&= a \\
&\mathit{maxBtree}\ (\mathit{Join}\ l\ r) &&= \mathit{max}\ (\mathit{maxBtree}\ l)\ (\mathit{maxBtree}\ r)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{depths} &&:: (\mathit{Btree}\ a, \mathit{Int}) \rightarrow \mathit{Btree}\ \mathit{Int} \\
&\mathit{depths}\ (\mathit{Leaf}\ a, n) &&= \mathit{Leaf}\ n \\
&\mathit{depths}\ (\mathit{Join}\ l\ r, n) &&= \mathit{Join}\ (\mathit{depths}\ (l, n+1))\ (\mathit{depths}\ (r, n+1))
\end{aligned}
$$

Since $maxBtree$ can be written as a $fold$ and $depths$ in terms of $build$,

$$maxBtree = foldB\ (id, max)$$

$$\begin{aligned}
&depths = buildB\ gdep \\
&\quad \mathbf{where} \\
&\qquad gdep\ (leaf, join)\ (Leaf\ a, n) = leaf\ n \\
&\qquad gdep\ (leaf, join)\ (Join\ l\ r, n) \\
&\qquad\quad = join\ (gdep\ (leaf, join)\ (l, n+1))\ (gdep\ (leaf, join)\ (r, n+1))
\end{aligned}$$

by Law 2 we can obtain the following monolithic definition of $aheight$:

$$\begin{aligned}
&aheight\ (Leaf\ a, n) &&= n \\
&aheight\ (Join\ l\ r, n) &&= max\ (aheight\ (l, n+1))\ (aheight\ (r, n+1))
\end{aligned}$$

In this case the fact that $depths$ is an accumulation has no impact in the way fusion is applied. This is because its accumulating parameter holds simply a integer, used to fill the leaves of the new tree. Therefore, there is nothing that fusion could change on that parameter; in fact, in the fused program it is used in the same way as it was in $depths$.

Now, let us analyse case (ii). This is the case where fusion methods have traditionally failed because of the difficulties to reach the accumulating parameters. The fact that accumulations compute their results in their accumulating parameters makes it necessary that fusion takes place also in those parameters. Accumulation in this category are, for example, function $areverse$, shown previously, and $aflatten$, which lists the leaves of a binary tree in left-to-right order.

$$\begin{aligned}
&aflatten &&:: (Btree\ a, [a]) \to [a] \\
&aflatten\ (Leaf\ a, xs) &&= a : xs \\
&aflatten\ (Join\ l\ r, xs) &&= aflatten\ (l, aflatten\ (r, xs))
\end{aligned}$$

To see a simple example of the difficulty in reaching the accumulating parameters, let us consider the following definition of a function that given a number represented as a list of digits $d_n \cdots d_1 d_0$ returns the corresponding integer.

$$\begin{aligned}
&number &&:: [Int] \to Int \\
&number\ ds &&= anumber\ (ds, [\,])
\end{aligned}$$

$$anumber = horner \circ areverse$$

$$\begin{aligned}
&horner\ [\,] &&= 0 \\
&horner\ (d : ds) &&= d + 10 * horner\ ds
\end{aligned}$$

First, let us try to derive a recursive definition for $anumber$ by case analysis:

$$\begin{aligned}
&anumber\ ([\,], xs) &&= horner\ (areverse\ ([\,], xs)) \\
& &&= horner\ xs
\end{aligned}$$

$$anumber\ (d : ds, xs) = horner\ (areverse\ (d : ds, xs))$$

$$
\begin{aligned}
&= \mathit{horner}\ (\mathit{areverse}\ (\mathit{ds}, d : \mathit{xs}))\\
&= \mathit{anumber}\ (\mathit{ds}, d : \mathit{xs})
\end{aligned}
$$

The definition obtained by this method is not satisfactory as it continues constructing an intermediate list (now internal to the function), holding the reverse of the list of digits, which still needs to be evaluated by the horner function once the base case of the empty list of digits is reached.

The problem with a direct approach like this is that it does not notice that the accumulating parameter should have a different type in the fused program. In fact, in this particular example, the accumulator should change from type [Int] to type Int in the fused program as it needs to hold the (partial) value of the integer being computed. That is, after applying fusion one should obtain the following definition, which is the natural tail recursive definition to this problem,

$$
\begin{array}{ll}
\mathit{anumber}' & :: ([\mathit{Int}], \mathit{Int}) \to \mathit{Int}\\
\mathit{anumber}'\ ([\,], z) & = z\\
\mathit{anumber}'\ (d : \mathit{ds}, z) & = \mathit{anumber}'\ (\mathit{ds}, d + 10 * z)
\end{array}
$$

together with the equation:

$$\mathit{anumber}\ (\mathit{ds}, \mathit{xs}) = \mathit{anumber}'\ (\mathit{ds}, \mathit{horner}\ \mathit{xs})$$

which states that the initial value of the “old” accumulator (the one of the producer) needs to converted to the format of the “new” accumulator (the one of the fused program). This conversion is performed by the consumer, the horner function.

In fact, the same equation holds in general for any producer given by an accumulation of this category. That is, given a consumer c and an accumulative producer p, its fusion fus satisfies that $c \circ p = \mathit{fus} \circ (\mathit{id} \times c)$, where $(f \times g)\ (x, y) = (f\ x, g\ y)$. This is the essential equation one has to have in mind when performing fusion involving this kind of accumulations.

The transformation shown above can be performed using shortcut fusion. The build function must now abstract the constructors that occur in the accumulating parameters in order to make them reachable for fusion. For the example, function $\mathit{areverse}$ can be written in terms of build as follows:

$$
\begin{array}{l}
\mathit{areverse} = \mathit{buildL}\ \mathit{grev}\\
\quad \textbf{where}\\
\qquad \mathit{grev}\ (\mathit{nil}, \mathit{cons})\ ([\,], \mathit{xs}) \qquad = \mathit{xs}\\
\qquad \mathit{grev}\ (\mathit{nil}, \mathit{cons})\ (a : \mathit{as}, \mathit{xs}) = \mathit{grev}\ (\mathit{nil}, \mathit{cons})\ (\mathit{as}, \mathit{cons}\ a\ \mathit{xs})
\end{array}
$$

However, this definition is not complete. In fact, to make the desired transformation possible the build function needs to include as part of its definition the format conversion of the initial value of the accumulator, which is performed by the consumer function. So we need to insert a call to the consumer (a fold) as part of the producer. The correct definition is thus as follows:

$$\begin{array}{l}
\mathit{areverse} = \mathit{buildL}\ \mathit{grev} \\
\quad \textbf{where} \\
\qquad \mathit{grev}\ (\mathit{nil}, \mathit{cons})\ (\mathit{as}, \mathit{xs}) \quad = \mathit{grev}'\ (\mathit{nil}, \mathit{cons})\ (\mathit{as}, \mathit{foldL}\ (\mathit{nil}, \mathit{cons})\ \mathit{xs}) \\
\qquad \mathit{grev}'\ (\mathit{nil}, \mathit{cons})\ ([\,], z) \qquad = z \\
\qquad \mathit{grev}'\ (\mathit{nil}, \mathit{cons})\ (a : \mathit{as}, z) = \mathit{grev}'\ (\mathit{nil}, \mathit{cons})\ (\mathit{as}, \mathit{cons}\ a\ z)
\end{array}$$

If we apply Law 1 we obtain the definitions of $\mathit{anumber}$ and $\mathit{anumber}'$ showed above.

An unsatisfactory aspect of the previous transformation in terms of shortcut fusion is the fact that the conversion that needs to be done on the accumulator of the producer is in all cases the same. That, it is always necessary to insert an application of the consumer function to the initial value of the producer´s accumulator as part of the definition of the build function. This means that to write the definition of the build function we have to inspect the definition of the producer in order to infer in which places to insert the applications of the consumer function. However, it is worth noting that it is possible to obtain automatically such applications of the consumer if we slightly modify the type of the template g carried by the build function. Concretely, the modification we propose is to make explicit in g the fact that the accumulating parameters are of the same type as the result of g. For instance, in the case of function $\mathit{areverse}$, instead of having grev of type,

$$\mathit{grev} :: (\forall\ b\ .\ (b, a \to b \to b) \to ([a], [a]) \to b)$$

our proposal is to manipulate a grev function of type

$$\mathit{grev} :: (\forall\ b\ .\ (b, a \to b \to b) \to ([a], b) \to b)$$

Having this modification, the desired applications of the consumer to the accumulators will appear automatically with no effort at all as a consequence of the free theorems associated with the new type of g. This implies of course the introduction of a modified version of build that we call builda. Next we show the definition of builda for lists and leaf-labelled binary trees.

In the case of lists the new build function has the following definition:

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

and an associated shortcut fusion law.

Law 3 (fold/builda for lists)

$$\mathit{foldL}\ (\mathit{nil}, \mathit{cons}) \circ \mathit{buildaL}\ g = g\ (\mathit{nil}, \mathit{cons}) \circ (\mathit{id} \times \mathit{foldL}\ (\mathit{nil}, \mathit{cons}))$$

Writing $\mathit{areverse}$ in terms of builda,

$$\begin{array}{l}
\mathit{areverse} = \mathit{builda}\ \mathit{grev} \\
\quad \textbf{where} \\
\qquad \mathit{grev}\ (\mathit{nil}, \mathit{cons})\ ([\,], z) \qquad = z \\
\qquad \mathit{grev}\ (\mathit{nil}, \mathit{cons})\ (a : \mathit{as}, z) = \mathit{grev}\ (\mathit{nil}, \mathit{cons})\ (\mathit{as}, \mathit{cons}\ a\ z)
\end{array}$$

the same definitions of $anumber$ and $anumber'$ seen before are obtained now by the application of Law 3.

For binary trees, $builda$ has the following definition:

$$
\begin{aligned}
&buildaB \quad :: (\forall\ b\ .\ (a \to b, b \to b \to b) \to (c, b) \to b) \to (c, Btree\ a) \to Btree\ a \\
&buildaB\ g = g\ (Leaf, Join)
\end{aligned}
$$

with a corresponding shortcut fusion law.

Law 4 (fold/builda for binary trees)

$$foldB\ (leaf, join) \circ buildaB\ g = g\ (leaf, join) \circ (id \times foldB\ (leaf, join))$$

A *fold/builda* law for similar binary trees, but which do not carry information in the leaves, i.e. a kind of shape trees, can be found in [Katsumata and Nishimura 2006].

To see an example, consider the following definitions:

$$
\begin{aligned}
&printAsc :: (Btree\ a, Btree\ a) \to Btree\ a \\
&printAsc = b2s \circ asc
\end{aligned}
$$

$$
\begin{aligned}
&b2s && :: Btree\ a \to String \\
&b2s\ (Leaf\ a) && = \texttt{"Leaf "} +\!\!+\ show\ a \\
&b2s\ (Join\ l\ r) && = \texttt{"Join ("} +\!\!+\ b2s\ l +\!\!+ \texttt{") ("} +\!\!+\ b2s\ r +\!\!+ \texttt{")"}
\end{aligned}
$$

$$
\begin{aligned}
&asc && :: (Btree\ a, Btree\ a) \to Btree\ a \\
&asc\ (Leaf\ a, t) && = Join\ t\ (Leaf\ a) \\
&asc\ (Join\ l\ r, t) && = asc\ (l, asc\ (r, t))
\end{aligned}
$$

Since $b2s$ can be written as a fold and asc in terms of $builda$:

$$
\begin{aligned}
&b2s = foldB\ (fleaf, fjoin) \\
&\quad \textbf{where} \\
&\qquad fleaf\ a \quad = \texttt{"Leaf "} +\!\!+\ show\ a \\
&\qquad fjoin\ x\ y = \texttt{"Join ("} +\!\!+\ x +\!\!+ \texttt{") ("} +\!\!+\ y +\!\!+ \texttt{")"}
\end{aligned}
$$

$$
\begin{aligned}
&asc = buildaB\ gasc \\
&\quad \textbf{where} \\
&\qquad gasc\ (leaf, join)\ (Leaf\ a, t) \quad = join\ t\ (leaf\ a) \\
&\qquad gasc\ (leaf, join)\ (Join\ l\ r, t) = gasc\ (leaf, join)\ (l, gasc\ (leaf, join)\ (r, t))
\end{aligned}
$$

we can apply Law 4 to fuse the parts, obtaining the following definition for $printAsc$,

$$printAsc\ (t, u) = printAsc'\ (t, b2s\ u)$$

$$
\begin{aligned}
&printAsc' && :: (Btree\ a, String) \to String \\
&printAsc'\ (Leaf\ a, s) && = \texttt{"Join ("} +\!\!+\ s +\!\!+ \texttt{") (Leaf"} +\!\!+\ show\ a +\!\!+ \texttt{")"} \\
&printAsc'\ (Join\ l\ r, s) && = printAsc'\ (l, printAsc'\ (r, s))
\end{aligned}
$$

4. Formalization

In this section, we show that the instances of fold, build, and shortcut fusion presented in the previous sections correspond to generic definitions valid for a wide class of datatypes.

4.1. Data types

The structure of datatypes can be captured using the concept of a *functor*. A functor consists of a type constructor F and a function $\mathit{map}_F :: (a \to b) \to (F\ a \to F\ b)$, which preserves identities and compositions: $\mathit{map}_F\ \mathit{id} = \mathit{id}$ and $\mathit{map}_F\ (f \circ g) = \mathit{map}_F\ f \circ \mathit{map}_F\ g$. A standard example of a functor is that formed by the list type constructor and the well-known map function.

Semantically, recursive datatypes are understood as least fixed points of functors. That is, given a datatype declaration it is possible to derive a functor F such that the datatype is the least solution to the equation $\tau \cong F\tau$. We write μF to denote the type corresponding to the least solution. The isomorphism between μF and $F\ \mu F$ is provided by two strict functions $\mathit{in}_F :: F\ \mu F \to \mu F$ and $\mathit{out}_F :: \mu F \to F\ \mu F$, inverses of each other. Function in_F packs the constructors of the datatype while out_F the destructors (for more details see e.g [Abramsky and Jung 1994, Gibbons 2002]).

In the case of lists, the structure is captured by a bifunctor L (a functor on two variables) because of the presence of the type paremeter. That is, $\mu(L\, a) \cong [a]$.

$$\mathbf{data}\ L\ a\ b = \mathit{FNil} \mid \mathit{FCons}\ a\ b$$

$$\begin{array}{ll}
\mathit{mapL} & :: (a \to c) \to (b \to d) \to L\ a\ b \to L\ c\ d \\
\mathit{mapL}\ f\ g\ \mathit{FNil} & = \mathit{FNil} \\
\mathit{mapL}\ f\ g\ (\mathit{FCons}\ a\ b) & = \mathit{FCons}\ (f\ a)\ (g\ b)
\end{array}$$

4.2. Fold

Let F be a functor that captures the structure of a datatype. Given a function $\varphi :: F\ a \to a$, *fold* [Gibbons 2002] is defined as the least function $\mathit{fold}_F\ \varphi :: \mu F \to a$ such that:

$$\mathit{fold}_F\ \varphi \circ \mathit{in}_F = \varphi \circ F\ (\mathit{fold}_F\ \varphi)$$

A function $\varphi :: F\ a \to a$ is called an F*-algebra*. For example, an algebra corresponding to the functor $L\ a$ is a function $\varphi :: L\ a\ b \to b$ of the form:

$$\varphi\ \mathit{FNil} = \mathit{nil} \qquad \text{and} \qquad \varphi\ (\mathit{FCons}\ a\ b) = \mathit{cons}\ a\ b$$

with $\mathit{nil} :: b$ and $\mathit{cons} :: a \to b \to b$. In the specific instance of fold for the list datatype [a] we wrote an algebra φ simply as a pair $(\mathit{nil}, \mathit{cons})$. The same can be applied to other datatypes.

4.3. Shortcut fusion

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

$$\begin{array}{ll}
\mathit{build}_F & :: (\forall\ a\ .\ (F\ a \to a) \to c \to a) \to c \to \mu F \\
\mathit{build}_F\ g = g\ \mathit{in}_F &
\end{array}$$

Notice that the abstraction of the datatype's constructors is given in terms of an F-algebra. Together with fold, build enjoys the following fusion law [Takano and Meijer 1995], which is an instance of a free theorem [Wadler 1989].

Law 5 (fold/build) *For strict φ,*[2]

$$\mathit{fold}_F\ \varphi \circ \mathit{build}_F\ g = g\ \varphi$$

4.4. Shortcut fusion for accumulations

In the same way as build, a generic definition of the builda function that captures accumulative producers can be formulated. The only difference with build is the explicit occurrence of the accumulator (of the same type as the result).

$$\begin{aligned}
&\mathit{builda}_F \quad :: (\forall\ a\ .\ (F\ a \to a) \to (c, a) \to a) \to (c, \mu F) \to \mu F\\
&\mathit{builda}_F\ g = g\ \mathit{in}_F
\end{aligned}$$

The shortcut fusion law in this case is the following.

Law 6 (fold/builda) *For strict φ,*

$$\mathit{fold}_F\ \varphi \circ \mathit{builda}_F\ g = g\ \varphi \circ (\mathit{id} \times \mathit{fold}_F\ \varphi)$$

Proof The free theorem associated with g's type states that, for all types a and b, algebras $\psi :: F\ a \to a$ and $\varphi :: F\ b \to b$, and strict function $f :: a \to b$, the following holds $f \circ \psi = \varphi \circ \mathit{map}_F\ f \Rightarrow f \circ g\ \psi = g\ \varphi \circ (\mathit{id} \times f)$. By considering $f = \mathit{fold}_F\ \varphi$ and $\psi = \mathit{in}_F$, we get $\mathit{fold}_F\ \varphi \circ g\ \mathit{in}_F = g\ \varphi \circ (\mathit{id} \times \mathit{fold}_F\ \varphi)$, because the premise of the implication holds by definition of fold. Finally, by applying the definition of builda_F we obtain the law. The strictness on φ is necessary for instantiation: if the algebra φ is strict, then so is $\mathit{fold}_F\ \varphi$, and we can instantiate f with $\mathit{fold}_F\ \varphi$. □

5. Benchmarks

With the aim at illustrating the impact of shortcut fusion on accumulations we present a benchmark comparing time performance of a set of example programs. The programs considered in the benchmark are simple but representative of different cases involving accumulations. All of them have an accumulation as producer function, and in some cases accumulations also appear as consumer functions.

Figure 1 presents the speedup percentages of comparing the running times of the original programs against those obtained by the application of shortcut fusion. The results were obtained by compiling the programs with the Glasgow Haskell Compiler (GHC) 6.8.2. In order to measure the effects of shortcut fusion in isolation, we compiled both the original and the transformed programs without activating the native optimisations of GHC. The transformed programs correspond to the recursive definitions that are obtained by inlining the expressions that result from the application of shortcut fusion. Those definitions were generated by hand.

The name of the programs considered in Figure 1 are formed by the concatenation of the names of the consumer and the producer functions, like e.g. "LengthFlatten". Consumer

[2]The strictness condition on φ was not mentioned in the concrete instances of the law shown in Section 2 because the algebras considered in those instances are all strict.

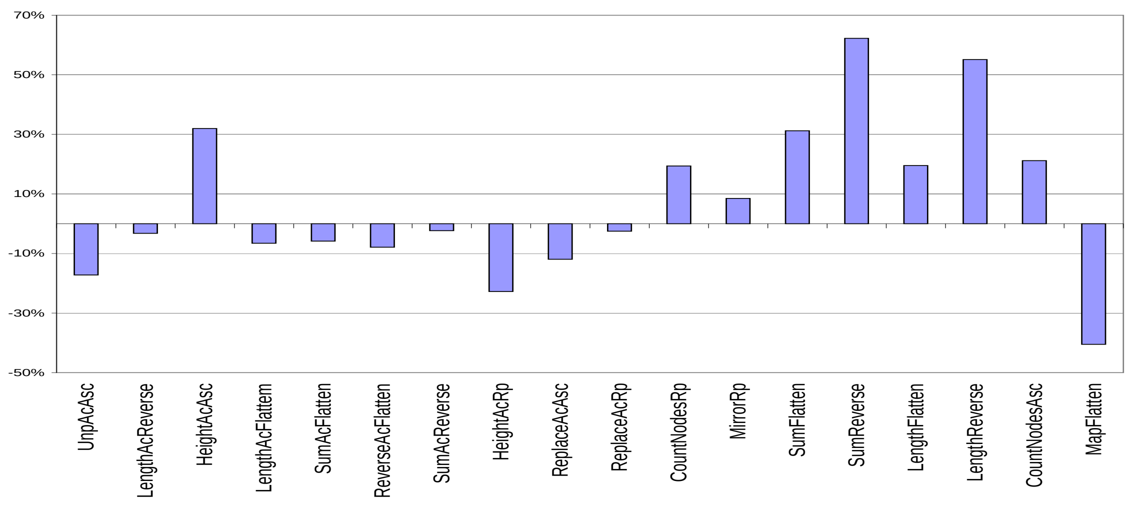

Figure 1. Percentage speedup

names with suffix ‘Ac’ correspond to accumulations. That is the case of “LengthAc”, for example. The names of most programs are self-explanatory, except for those with suffix “Rp”; this suffix corresponds to a function on shape trees (binary trees without information in the leaves) that replaces the rightmost leaf by a tree. The leftmost ten programs have accumulations as consumers. With the exception of program “HeightAcAsc”, the application of shortcut fusion to those programs results in slower ones as expected. On the other hand, for the programs with nonaccumulative functions as consumers (the rightmost eight ones), out of “MapFlatten”, the application of shortcut fusion results in more efficient programs.

A prototype implementation of shortcut fusion has been obtained using the rewrite rules mechanism (the RULES pragma) of GHC, which permits the specification of equational transformations within programs that are applied by the compiler during optimisation. To apply the rewrite rules it is necessary to write the consumer functions as a fold, and the producer functions as a build. So far this has been done by hand. We believe that the development of a procedure for the automatic derivation of fold and build representations for accumulations in the line of *warm fusion* [Launchbury and Sheard 1995] is possible. We performed some experiments using rewrite rules, but we do not present their results because the interplay between shortcut fusion and other optimisations in the context of accumulations deserves further analysis. In fact, some of the results have certain similarity with those presented in Figure 1, but there are others that are dramatically different.

6. Conclusions

This paper presented an study of the application of shortcut fusion to programs involving accumulations. In particular, we introduced a modified version of the *fold/build* law tailored to the cases when producer functions are given by accumulations that compute their results (or part of them) in accumulating parameters. The new law required the definition of a slightly modified build function, called $builda$, that explicitly exhibits in its type the occurrences of those accumulating parameters. Benchmarks showed the effectiveness of the new law for a variety of example programs.

References

Abramsky, S. and Jung, A. (1994). Domain theory. In *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Clarendon Press.

Bird, R. S. (1998). *Introduction to Functional Programming Using Haskell*. Prentice–Hall.

Burstall, R. M. and Darlington, J. (January 1977). A transformation system for developing recursive programs. In *Journal of the ACM*, volume 24(1), pages 44–67.

Gibbons, J. (2000). Generic Downwards Accumulations. *Science of Computer Programming*, 37(1–3):37–65.

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

Gill, A. (1996). *Cheap Deforestation for Non strict Functional Languages*. PhD thesis, University of Glasgow.

Gill, A., Launchbury, J., and Jones, S. L. P. (1993). A short cut to deforestation. In *Functional Programming Languages and Computer Architecture*, pages 223–232. ACM.

Hu, Z., Iwasaki, H., and Takeichi, M. (1999). Calculating accumulations. *New Generation Computing*, 17(2):153–173.

Johann, P. and Voigtländer, J. (2004). Free theorems in the presence of seq. In *31st Symposium on Principles of Programming Languages*, pages 99–110. ACM.

Katsumata, S.-Y. and Nishimura, S. (2006). Algebraic fusion of functions with an accumulating parameter and its improvement. In *ICFP'06*, pages 227–238. ACM.

Launchbury, J. and Sheard, T. (1995). Warm fusion: deriving build-catas from recursive definitions. In *Functional Programming Languages and Computer Architecture*, pages 314–323. ACM.

Nishimura, S. (2002). Deforesting in accumulating parameters via type–directed transformations. In *APLAS*, pages 145–159.

Pardo, A. (2003). Generic accumulations. In *IFIP TC2/WG2.1 Working Conference on Generic Programming*, pages 49–8. Kluwer, B.V.

Svenningsson, J. (2002). Shortcut fusion for accumulating parameters & zip-like functions. In *ICFP*, pages 124–132. ACM.

Takano, A. and Meijer, E. (1995). Shortcut to Deforestation in Calculational Form. In *Functional Programming Languages and Computer Architecture*, pages 306–313. ACM.

Voigtländer, J. (2004). Using circular programs to deforest in accumulating parameters. *Higher–Order and Symbolic Computation*, 17:129–163.

Voigtländer, J. and Kühnemann, A. (2004). Composition of functions with accumulating parameters. *Journal of Functional Programming*, 14(3):317–363.

Wadler, P. (1989). Theorems for free! In *Functional Progranning Languages and Computer Architecture*, pages 347–359. ACM.