Applicative Shortcut Fusion

Germán Andrés Delbianco[1], Mauro Jaskelioff[2], and Alberto Pardo[3]

[1] IMDEA Software Institute, Spain
[2] CIFASIS-CONICET/Universidad Nacional de Rosario, Argentina
[3] InCo, Universidad de la República, Uruguay

Abstract. In functional programming one usually writes programs as the composition of simpler functions. Consequently, the result of a function might be generated only to be consumed immediately by another function. This potential source of inefficiency can often be eliminated using a technique called shortcut fusion, which fuses both functions involved in a composition to yield a monolithic one. In this article we investigate how to apply shortcut fusion to applicative computations. Applicative functors provide a model of computational effects which generalise monads, but they favour an applicative programming style. To the best of our knowledge, this is the first time shortcut fusion is considered in an applicative setting.

1 Introduction

One of functional programming much advocated benefits is the possibility of easily constructing large and complex programs through the combination of smaller or simpler ones [12]. This modular approach, however, often results in programs which are quite inefficient when compared to their monolithic counterparts: compositional design often involves creating an *intermediate* data structure which is immediately consumed. In order to alleviate this problem, several formal techniques have been developed that allow the derivation of efficient programs from simpler modular ones. The way these techniques are usually discovered is by identifying common patterns in programs, analyzing these patterns, and obtaining algebraic laws for programs that fit the pattern [18].

Among these techniques lies *shortcut fusion* [11,20] which is concerned with the elimination of unnecessary list traversals. It is based on a single transformation: the `foldr/build` rule which fuses the application of a uniform list-consuming function, expressed as a *fold* on lists, to the result of a uniform list-generating function, expressed in terms of the *build* combinator. This *fusion rule* can be generalised to any inductive datatype, yielding the following generic rule:

$$fold\ k \circ build\ g = g\ k$$

Shortcut fusion has been extended to cope with cases where the intermediate structure is produced in certain contexts. For example, shortcut fusion has

been considered for monadic computations [6,13,14], unstructured functors [7], accumulations [15] and circular programs [5,19].

A recent development is the notion of applicative functor [16]. Applicative functors provide a novel manner in which effectful computations can be constructed that has gained a rapid acceptance among functional programmers. However, shortcut fusion under an applicative context has not yet been studied. Precisely, in this article, we investigate shortcut fusion under the context of an applicative computation, we identify common patterns in which many applicative programs are written, and give algebraic laws that apply to programs that fit those patterns. Concretely, the contributions of this article are:

- We show how to do shortcut fusion on applicative computations.
- We identify a common pattern in applicative programs which shows the importance and generality of traversals for generating applicative structures and their fundamental role in applicative shortcut fusion.
- We provide a combinator (*ifold*) which models the uniform consumption of applicative computations.

The paper is organised as follows. In Section 2 we review the concept of shortcut fusion. In Section 3 we present the notions of applicative and traversable functors. Section 4 develops the notions of applicative shortcut fusion and applicative structural recursion. In Sections 2 to 4 our motivating examples are on lists. In Section 5 we show the datatype-generic formulation of the concepts and laws presented in previous sections. Finally, in Section 6 we conclude and discuss future work.

Throughout the paper we asume we are working in the context of a functional language with a Haskell-like syntax and with a set-theoretic semantics in which types are interpreted as sets and functions as set-theoretic functions.

2 Shortcut fusion

Shortcut fusion [11] is a program transformation technique for the elimination of intermediate data structures generated in function compositions. It is a consequence of parametricity properties, known as "free theorems" [21], associated with polymorphic functions. Given a composition $fc \circ fp$, where fc is called the *consumer* and fp the *producer* of the intermediate structure, shortcut fusion requires for its application that both consumer and producer definitions conform to determinate structural requirements. Like other fusion laws of its kind, shortcut fusion requires that the consumer be expressible as a *fold* [4]. The producer, on the other hand, is required to build the intermediate data structure using uniquely the constructors of the datatype. This is expressed in terms of a function, called *build*, which carries a "template" that abstracts from the function body the occurrences of those constructors. For example, when the intermediate structure is a list, *fold* and *build* are given by the following definitions:

$$\begin{aligned} &foldr :: (a \to b \to b) \to b \to [a] \to b \\ &foldr\ f\ e\ [\,] \quad\ = e \end{aligned}$$

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

where *foldr* is a well-known function pattern in functional programming [4].

The essential idea of shortcut fusion is then to replace, in the template of *build*, the occurrences of the constructors of the intermediate structure ((:) and [] in the case of lists) by the corresponding operations carried by the *fold*. The second-order polymorphism of *build* ensures that the argument can only manufacture its result by using its two arguments. For lists, shortcut fusion is expressed by the following law, usually referred to as the *fold/build law*.

Law 1 (FOLDR/BUILD **[11])**

$$\mathit{foldr}\ f\ e \circ \mathit{build}\ g = g\ f\ e$$

As a result of the application of this law one obtains an equivalent definition that computes the same as the original consumer-producer composition but avoiding the construction of the intermediate data structure.

Example 1. To see an application of Law 1 we define a function that computes the sum of the positionwise differences between two lists of numbers.

$$
\begin{aligned}
&\mathit{sumDiff} \quad :: \mathit{Num}\ a \Rightarrow ([a], [a]) \to a\\
&\mathit{sumDiff}\ \mathit{ys} = \mathit{sum} \circ \mathit{diffList}\\
&\mathit{diffList} \quad :: \mathit{Num}\ a \Rightarrow ([a], [a]) \to [a]\\
&\mathit{diffList}\ (xs, [\,]) \quad = [\,]\\
&\mathit{diffList}\ ([\,], y : ys) \quad = [\,]\\
&\mathit{diffList}\ (x : xs, y : ys) = (x - y) : \mathit{diffList}\ (xs, ys)
\end{aligned}
$$

Function *sum* has the usual definition as a foldr: $\mathit{sum} = \mathit{foldr}\ (+)\ 0$. When applied to a pair of lists (xs, ys), *diffList* computes the list of differences between values in xs and ys, up to the shorter of the two lists. This function is a *good producer* in the sense that it can be expressed in terms of *build*:

$$
\begin{aligned}
&\mathit{diffList} = \mathit{build}\ \mathit{gdiff}\\
&\quad \textbf{where}\\
&\qquad \mathit{gdiff}\ \mathit{cons}\ \mathit{nil}\ (\_, [\,]) \quad = \mathit{nil}\\
&\qquad \mathit{gdiff}\ \mathit{cons}\ \mathit{nil}\ ([\,], \_) \quad = \mathit{nil}\\
&\qquad \mathit{gdiff}\ \mathit{cons}\ \mathit{nil}\ (x : xs, y : ys) = \mathit{cons}\ (x - y)\ (\mathit{gdiff}\ \mathit{cons}\ \mathit{nil}\ (xs, ys))
\end{aligned}
$$

Once we have consumer and producer expressed in terms of *foldr* and *build* we are in a position to apply Law 1, obtaining the following definition for *sumDiff*:

$$\mathit{sumDiff} = \mathit{gdiff}\ (+)\ 0$$

Inlining the definition,

$$
\begin{aligned}
&\mathit{sumDiff}\ (\_, [\,]) &&= 0\\
&\mathit{sumDiff}\ ([\,], \_) &&= 0\\
&\mathit{sumDiff}\ (x : xs, y : ys) &&= (x - y) + \mathit{sumDiff}\ (xs, ys)
\end{aligned}
$$

In this paper we are also interested in a generalised form of shortcut fusion which captures the case where the intermediate data structure is generated as part of another structure. This generalisation has been a fundamental tool for the formulation of shortcut fusion laws for monadic programs [14,7], and for the derivation of (monadic) circular and higher-order programs [19,5]. In this paper our aim is to analyse this generalisation in the case when the effects are given by applicative functors.

The generalisation of shortcut fusion [7] is based on an extended form of build. For lists, it has the following definition:

$$
\begin{aligned}
&\mathit{ebuild}\quad :: \mathit{Functor}\ f \Rightarrow (\forall b.(a \to b \to b) \to b \to c \to f\ b) \to c \to f\ [a]\\
&\mathit{ebuild}\ g = g\ (:)\ [\,]
\end{aligned}
$$

where f acts as a container of the generated list. The type requires f to be an instance of the *Functor* class, which ensures that f has an associated function $\mathit{fmap} :: (a \to b) \to f\ a \to f\ b$ that preserves composition and identity.

Law 2 (FOLDR/EBUILD **[7])**

$$\mathit{fmap}\ (\mathit{foldr}\ f\ e) \circ \mathit{ebuild}\ g = g\ f\ e$$

The use of *fmap* means that fusion acts on the occurrences of the list type within the context structure, maintaining the context structure unchanged.

3 Applicative Functors

An *applicative functor* (or *idiom*) [16] is a type constructor $f :: * \to *$, equipped with two operations:

$$
\begin{aligned}
&\textbf{class}\ (\mathit{Functor}\ f) \Rightarrow \mathit{Applicative}\ f\ \textbf{where}\\
&\quad \mathit{pure} :: a \to f\ a\\
&\quad (\circledast)\ \ :: f\ (s \to t) \to f\ s \to f\ t
\end{aligned}
$$

Intuitively, *pure* lifts a pure computation into the effectful context defined by f and $\circledast$ performs an effectful application. Instances of *pure* and $\circledast$ must verify certain laws (see e.g [16] for details).

Example 2 (Maybe). The *Maybe* applicative functor models *failure* as a computational effect.

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Applicative}\ \mathit{Maybe}\ \textbf{where}\\
&\quad \mathit{pure} &&= \mathit{Just}\\
&\quad (\mathit{Just}\ f) \circledast (\mathit{Just}\ x) &&= \mathit{Just}\ (f\ x)\\
&\quad \_ \qquad \circledast \_ &&= \mathit{Nothing}
\end{aligned}
$$

All monads are applicative functors, taking $\circledast$ to be monadic application and *pure* to be *return*. However, there are applicative functors which are not monads, such as the one in the following example.

Example 3 (Ziplists). The list functor has an *Applicative* instance other than the one obtained from the list monad [16]. This applicative functor models a *transposition* effect, and is defined as follows:

$$\begin{aligned}
&\textbf{instance}\ \mathit{Applicative}\ [\,]\ \textbf{where}\\
&\quad \mathit{pure}\ x \qquad\quad = x : \mathit{pure}\ x\\
&\quad (f : \mathit{fs}) \circledast (x : \mathit{xs}) = f\ x : (\mathit{fs} \circledast \mathit{xs})\\
&\quad _ \quad \circledast \quad _ \quad = [\,]
\end{aligned}$$

An *applicative action* is a function of type $a \to f\ b$ where f is an applicative functor. Applicative actions can be used to perform traversals over a certain class of data structures, threading an effect through the data structure. This class of data structures is called *Traversable*:

$$\begin{aligned}
&\textbf{class}\ (\mathit{Functor}\ t) \Rightarrow \mathit{Traversable}\ t\ \textbf{where}\\
&\quad \mathit{traverse} :: (\mathit{Applicative}\ f) \Rightarrow (a \to f\ b) \to t\ a \to f\ (t\ b)
\end{aligned}$$

Alternatively, this class can be defined by means of a distributive law $\mathit{dist} :: f\ (c\ a) \to c\ (f\ a)$ which pulls the effects out of the data structure. The functions *dist* and *traverse* are interdefinable, with $\mathit{dist} = \mathit{traverse}\ \mathit{id}$ and $\mathit{traverse}\ \iota = \mathit{dist} \circ \mathit{fmap}\ \iota$. The latter definition gives a concise description of what an effectful traversal does: first populate the structure with effects by mapping the applicative action and then collect them using the distributive law.

Example 4 (Lists). Lists are *Traversable* data structure, as witnessed by the following instance:

$$\begin{aligned}
&\textbf{instance}\ \mathit{Traversable}\ [\,]\ \textbf{where}\\
&\quad \mathit{traverse}\ \iota\ [\,] \qquad = \mathit{pure}\ [\,]\\
&\quad \mathit{traverse}\ \iota\ (x : \mathit{xs}) = \mathit{pure}\ (:) \circledast \iota\ x \circledast \mathit{traverse}\ \iota\ \mathit{xs}
\end{aligned}$$

Example 5 (Reciprocal List). We want to define a function that computes the reciprocals of a given list of numbers, failing if there is any 0 value in the list. We can think of the computation of the reciprocal of a value as an *applicative action*: if the value is nonzero then a computation that produces its reciprocal is returned, else we fail *via Nothing*.

$$\begin{aligned}
&\mathit{recipM} \quad :: \mathit{Fractional}\ a \Rightarrow a \to \mathit{Maybe}\ a\\
&\mathit{recipM}\ x = \textbf{if}\ (x \not\equiv 0)\ \textbf{then}\ \mathit{pure}\ (\mathit{recip}\ x)\ \textbf{else}\ \mathit{Nothing}
\end{aligned}$$

where $\mathit{recip} :: \mathit{Fractional}\ a \Rightarrow a \to a$ is such that $\mathit{recip}\ x = 1 \,/\, x$. We can use this applicative action to define *recipList* by structural recursion:

$$
\begin{aligned}
&\mathit{recipList} && :: \mathit{Fractional}\ a \Rightarrow [a] \to \mathit{Maybe}\ [a] \\
&\mathit{recipList}\ [\,] && = \mathit{pure}\ [\,] \\
&\mathit{recipList}\ (x : xs) && = \mathit{pure}\ (:) \circledast \mathit{recipM}\ x \circledast \mathit{recipList}\ xs
\end{aligned}
$$

In this definition, we recognise the application of *recipM* to each element in the list, and therefore it clearly can be expressed in terms of *traverse*:

$$\mathit{recipList} = \mathit{traverse}\ \mathit{recipM}$$

On lists as well as on other *Traversable* inductive datatypes function *traverse* can be seen both as a good consumer and good producer: similar to the map function on the datatype, it traverses its input and generates its output in a uniform way. In the remainder of this section we focus on its quality as a consumer; in the next section we show that it is a good producer as well.

Any *Traversable* inductive datatype is a good consumer because it can easily be defined as a *fold*. For example, for lists,

$$\mathit{traverse}\ \iota = \mathit{foldr}\ (\lambda x\ t \to \mathit{pure}\ (:) \circledast \iota\ x \circledast t)\ (\mathit{pure}\ [\,])$$

From this fact, we can state the following law in connection with *build*.

Law 3 (TRAVERSE/BUILD FOR LISTS)

$$\mathit{traverse}\ \iota \circ \mathit{build}\ g = g\ (\lambda x\ t \to \mathit{pure}\ (:) \circledast \iota\ x \circledast t)\ (\mathit{pure}\ [\,])$$

Proof. By the definition of *traverse* as a *fold* and Law 1. □

Example 6 (Hermitian transpose). Given a type for complex numbers *Comp*, we will define an algoritm which calculates the *Hermitian or conjugate transpose* of a complex matrix.

$$\mathbf{data}\ \mathit{Real}\ x \Rightarrow \mathit{Comp}\ x = x + x\,\mathbf{i}$$

The algorithm is quite simple: first calculate the conjugate matrix and then transpose it. The conjugate matrix is defined elementwise, taking the complex conjugate of each entry:

$$
\begin{aligned}
&\mathit{hermitian} :: (\mathit{Real}\ a) \Rightarrow [[\,\mathit{Comp}\ a\,]] \to [[\,\mathit{Comp}\ a\,]] \\
&\mathit{hermitian} = \mathit{transpose} \circ \mathit{map}\ (\mathit{map}\ \mathit{scalarconj}) \\
&\qquad\qquad \mathbf{where}\ \mathit{scalarconj}\ (a + b\,\mathbf{i}) = a + (-b)\,\mathbf{i}
\end{aligned}
$$

In Example 3, we stated that the ziplists applicative function models a transposition effect. In fact, matrix transposition is a traversal with the identity function i.e. $\mathit{transpose} = \mathit{traverse}\ \mathit{id}$ [16]. Then, by the application of Law 3 the following definition of the Hermitian transpose is obtained, avoiding the construction of the intemediate matrix:

$$
\begin{aligned}
&\mathit{hermitian} :: (\mathit{Real}\ a) \Rightarrow [[\,\mathit{Comp}\ a\,]] \to [[\,\mathit{Comp}\ a\,]] \\
&\mathit{hermitian} = \mathit{foldr}\ (\lambda xs\ xss \to \mathit{pure}\ (:) \circledast \mathit{fmap}\ \mathit{scalarconj}\ xs \circledast xss)\ (\mathit{pure}\ [\,]) \\
&\qquad\qquad \mathbf{where}\ \mathit{scalarconj}\ (a + b\,\mathbf{i}) = a + (-b)\,\mathbf{i}
\end{aligned}
$$

4 Applicative Shortcut Fusion

In this section we analyse situations where the production and consumption of a data structure is performed in the context of an applicative effect. Our aim is to obtain a shortcut fusion law for those cases. As with monads [14,7], the extension of shortcut fusion presented in Section 2 turns out to be an appropriate device to achieve this goal. Again, our development in this section is performed on lists; the datatype-generic constructions are shown in Section 5.

Applicative shortcut fusion works on those cases where the container of the generated intermediate data structure is an applicative functor. The *build* function in this case is simply an instance of the *extended build* that we call *ibuild* (for *idiomatic build*):

$$\begin{aligned} &\mathit{ibuild} \quad :: \mathit{Applicative}\ f \Rightarrow (\forall b.(a \to b \to b) \to b \to c \to f\ b) \to c \to f\ [a] \\ &\mathit{ibuild}\ g = g\ (:)\ [\,] \end{aligned}$$

The corresponding instance of extended shortcut fusion (Law 2) is the following:

Law 4 (FOLDR/IBUILD)

$$\mathit{fmap}\ (\mathit{foldr}\ f\ e) \circ \mathit{ibuild}\ g = g\ f\ e$$

Example 7 (traverse). As mentioned at the end of Section 3, function *traverse* may not only be considered a good consumer but also a good producer since it generates its output list in a uniform way as the result of an effectful computation. In fact, it is very simple to express *traverse* in terms of *ibuild*:

$$\begin{aligned} &\mathit{traverse}\ \iota = \mathit{ibuild}\ \mathit{gtrav} \\ &\quad \textbf{where} \\ &\qquad \mathit{gtrav}\ \mathit{cons}\ \mathit{nil}\ [\,] \qquad = \mathit{pure}\ \mathit{nil} \\ &\qquad \mathit{gtrav}\ \mathit{cons}\ \mathit{nil}\ (x : \mathit{xs}) = \mathit{pure}\ \mathit{cons} \circledast \iota\ x \circledast \mathit{gtrav}\ \mathit{cons}\ \mathit{nil}\ \mathit{xs} \end{aligned}$$

which is the same as,

$$\begin{aligned} &\mathit{traverse}\ \iota = \mathit{ibuild}\ \mathit{gtrav} \\ &\quad \textbf{where} \\ &\qquad \mathit{gtrav}\ \mathit{cons}\ \mathit{nil} = \mathit{foldr}\ (\lambda x\ t \to \mathit{pure}\ \mathit{cons} \circledast \iota\ x \circledast t)\ (\mathit{pure}\ \mathit{nil}) \end{aligned}$$

It is also interesting to see that the composition $\mathit{traverse}\ \iota \circ \mathit{build}\ g$, which is the subject of Law 3, can also be expressed as an *ibuild*:

$$\begin{aligned} &\mathit{traverse}\ \iota \circ \mathit{build}\ g = \mathit{ibuild}\ g' \\ &\quad \textbf{where}\ g'\ f\ e = g\ (\lambda x\ t \to \mathit{pure}\ f \circledast \iota\ x \circledast t)\ (\mathit{pure}\ e) \end{aligned}$$

A common pattern of computation using applicative functors is the one that applies a *fold* after having performed an applicative traversal over a data structure. We identify this pattern with a new program scheme that we call *idiomatic fold*, which specifies an applicative notion of structural recursion. For lists,

$$ifoldr :: Applicative\ f \Rightarrow (b \to c \to c) \to c \to (a \to f\ b) \to [a] \to f\ c$$
$$ifoldr\ f\ e\ \iota = fmap\ (foldr\ f\ e) \circ traverse\ \iota$$

Using the fact that *traverse* can be expressed as an *ibuild* we can apply Law 4 obtaining as result that an *ifoldr* is a *foldr*:

$$ifoldr\ f\ e\ \iota = foldr\ (\lambda x\ t \to pure\ f \circledast \iota\ x \circledast t)\ (pure\ e) \tag{1}$$

Inlining,

$$ifoldr\ f\ e\ \iota\ [\,] \quad = pure\ e$$
$$ifoldr\ f\ e\ \iota\ (x : xs) = pure\ f \circledast \iota\ x \circledast ifoldr\ f\ e\ \iota\ xs$$

Example 8 (Sum of reciprocal list). In Example 5 we defined the function *recipList* that computes the reciprocals of a list of numbers. We used the *Maybe* applicative functor to model the possibility of failure originated by the occurrence of some 0 in the input list. Now we want to compute the sum of the reciprocals of a list:

$$sumRecips :: Fractional\ a \Rightarrow [a] \to Maybe\ a$$
$$sumRecips = fmap\ sum \circ recipList$$

Since $sum = foldr\ (+)\ 0$ and $recipList = traverse\ recipM$, *sumRecips* corresponds to an *ifold*:

$$sumRecips = ifoldr\ (+)\ 0\ recipM$$

Inlining,

$$sumRecips\ [\,] \quad = pure\ 0$$
$$sumRecips\ (x : xs) = pure\ (+) \circledast recipM\ x \circledast sumRecips\ xs$$

Having introduced a notion of applicative structural recursion, we can state a shortcut fusion law associated with it.

Law 5 (IFOLDR/BUILD)

$$ifoldr\ f\ e\ \iota \circ build\ g = g\ (\lambda x\ y \to pure\ f \circledast \iota\ x \circledast y)\ (pure\ e)$$

Proof.

$$\begin{aligned}
&\quad ifoldr\ f\ e\ \iota \circ build\ g \\
&\equiv \{\ \text{definition } ifoldr\ \} \\
&\quad fmap\ (foldr\ f\ e) \circ traverse\ \iota \circ build\ g \\
&\equiv \{\ \text{Example 7, } g'\ f\ e = g\ (\lambda x\ t \to pure\ f \circledast \iota\ x \circledast t)\ (pure\ e)\ \} \\
&\quad fmap\ (foldr\ f\ e) \circ ibuild\ g' \\
&\equiv \{\ \text{Law 4}\ \} \\
&\quad g\ (\lambda x\ t \to pure\ f \circledast \iota\ x \circledast t)\ (pure\ e)\ \square
\end{aligned}$$

Example 9 (Sum of reciprocals of list differences). We now want to compose the function that calculates the sum of reciprocals of a list of numbers, given in Example 8, with the function that computes the differences of two list of numbers, given in Example 1.

$$\begin{aligned}&\mathit{sumRecipsDiff} :: \mathit{Fractional}\ a \Rightarrow ([a],[a]) \to \mathit{Maybe}\ a\\&\mathit{sumRecipsDiff} = \mathit{sumRecips} \circ \mathit{diffList}\end{aligned}$$

Since $\mathit{sumRecips} = \mathit{ifoldr}\ (+)\ 0\ \mathit{recipM}$ and $\mathit{diffList} = \mathit{build}\ \mathit{gdiff}$, by Law 5 we get a monolithic definition that avoids the construction of the intermediate lists:

$$\mathit{sumRecipsDiff} = \mathit{gdiff}\ (\lambda x\ t \to \mathit{pure}\ (+) \circledast \mathit{recipM}\ x \circledast t)\ (\mathit{pure}\ 0)$$

Inlining,

$$\begin{aligned}&\mathit{sumRecipsDiff}\ (\_,[\,]) &&= \mathit{pure}\ 0\\&\mathit{sumRecipsDiff}\ ([\,],\_) &&= \mathit{pure}\ 0\\&\mathit{sumRecipsDiff}\ (x:xs, y:ys) &&= \mathit{pure}\ (+) \circledast \mathit{recipM}\ (x-y)\\&&&\quad \circledast \mathit{sumRecipsDiff}\ (xs, ys)\end{aligned}$$

We conclude this section by showing an example that, unlike the previous one, does not fit the pattern *fold/traverse/build*: it is a case where we cannot factor an occurrence of *traverse*. The example, however, needs extra structure on the applicative functor, namely to be an instance of the *Alternative* class.

Example 10 (Parsing). Suppose we want to compute the exclusive OR of a sequence of bits that we parse from an input string. It is in the parsing phase that effects will come into play, as we will use an applicative parser.

$$\begin{aligned}&\textbf{newtype}\ \mathit{Parser}\ a = P\ \{\mathit{runP} :: \mathit{String} \to [(a, \mathit{String})]\}\\&\textbf{instance}\ \mathit{Functor}\ \mathit{Parser}\ \textbf{where}\\&\quad \mathit{fmap}\ f\ p = P\ \$\ \lambda cs \to [(f\ a, cs') \mid (a, cs') \leftarrow \mathit{runP}\ p\ cs]\\&\textbf{instance}\ \mathit{Applicative}\ \mathit{Parser}\ \textbf{where}\\&\quad \mathit{pure}\ a = P\ \$\ \lambda cs \to [(a, cs)]\\&\quad p \circledast q\ \ = P\ \$\ \lambda cs \to [(f\ v, cs'') \mid\ \ (f, cs') \leftarrow \mathit{runP}\ p\ cs\\&\qquad\qquad\qquad\qquad\qquad\qquad\quad ,\ \ (v, cs'') \leftarrow \mathit{runP}\ q\ cs']\\&\textbf{class}\ \mathit{Applicative}\ f \Rightarrow \mathit{Alternative}\ f\ \textbf{where}\\&\quad \mathit{empty} :: f\ a\\&\quad (\langle|\rangle)\ \ :: f\ a \to f\ a \to f\ a\\&\textbf{instance}\ \mathit{Alternative}\ \mathit{Parser}\ \textbf{where}\\&\quad \mathit{empty} = P\ \$\ \mathit{const}\ [\,]\\&\quad p\langle|\rangle q\ \ = P\ \$\ \lambda cs \to \textbf{case}\ \mathit{runP}\ p\ cs +\!\!+ \mathit{runP}\ q\ cs\ \textbf{of}\\&\qquad\qquad\qquad\qquad\quad [\,]\ \ \ \ \to [\,]\\&\qquad\qquad\qquad\qquad\quad x:xs \to [x]\\&\mathit{pSym} :: \mathit{Char} \to \mathit{Parser}\ \mathit{Char}\\&\mathit{pSym}\ x = P\ \$\ \lambda cs \to \textbf{case}\ cs\ \textbf{of}\end{aligned}$$

$$
\begin{array}{ll}
c : cs \mid x \equiv c & \rightarrow [(c, cs)] \\
otherwise & \rightarrow [\,]
\end{array}
$$

Alternatives are represented by a choice operator ($\langle|\rangle$), which, for simplicity, returns at most one result. The parser *pSym* parsers a determinate character.

Using these combinators we define parsers for bits and bit strings.

$$
\begin{array}{l}
bitstring = pure\ (:) \circledast bit \circledast bitstring \\
\qquad\qquad \langle|\rangle \\
\qquad\qquad pure\ [\,] \\
bit = pure\ (const\ False) \circledast pSym\ \texttt{'0'} \\
\qquad \langle|\rangle \\
\qquad pure\ (const\ True) \circledast pSym\ \texttt{'1'} \\
listXor \qquad\quad :: [Bool] \rightarrow Bool \\
listXor\ [\,] \qquad = False \\
listXor\ (b : bs) = b\ `xor`\ listXor\ bs \\
xor \qquad\quad :: Bool \rightarrow Bool \rightarrow Bool \\
b\ `xor`\ b' = (b \wedge \neg\ b') \vee (\neg\ b \wedge b')
\end{array}
$$

We want to compute the composition: $xorBits = fmap\ (listXor) \circ bitstring$. Since $listXor = foldr\ xor\ False$ and *bitstring* can be expressed as an *ibuild*:

$$
\begin{array}{l}
bitstring = ibuild\ gbits \\
\quad \textbf{where}\ gbits\ cons\ nil = pure\ cons \circledast bit \circledast gbits\ cons\ nil \\
\qquad\qquad\qquad\qquad\qquad\quad \langle|\rangle \\
\qquad\qquad\qquad\qquad\qquad\quad pure\ nil
\end{array}
$$

we can apply Law 4 obtaining that $xorBits = gbits\ xor\ False$. Inlining,

$$
\begin{array}{l}
xorBits = pure\ xor \circledast bit \circledast xorBits \\
\qquad\qquad \langle|\rangle \\
\qquad\qquad pure\ False
\end{array}
$$

5 The datatype-generic formulation

In the previous sections, we focused our presentation on the list datatype in order to give a comprehensive explanation of the main concepts. However, constructions such as *fold*, *build* and *ebuild*, and laws like shortcut fusion can be formulated for a wide class of datatypes using a datatype-generic approach [2,3,9].

5.1 Inductive Data types

The structure of data types can be captured using the concept of a *functor*. A functor consists of a type constructor f and a map function:

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

where *fmap* must preserves identities and compositions: $fmap\ id = id$ and $fmap\ (f \circ g) = fmap\ f \circ fmap\ g$. A standard example of a functor is that formed by the list type constructor and the well-known *map* function.

Inductive data types correspond to least fixed points of functors. Given a data type declaration it is possible to derive a functor f, which captures the structure of the type, such that the data type can be seen as the least solution of the equation $x \cong fx$ [1]. In Haskell, we can encode this isomorphism defining a type constructor $\mu :: (* \to *) \to *$ as follows:

```
newtype μ f = In { unIn :: f (μ f) }
```

Example 11 (Naturals). Given a data type for natural numbers,

```
data Nat = Zero | Succ Nat
```

its signature is given by a functor *FNat* defined as follows:

```
data FNat x = FZero | FSucc x

instance Functor FNat where
  fmap f FZero     = FZero
  fmap f (FSucc n) = FSucc (f n)
```

So, alternatively, we can say that $Nat = \mu\ FNat$.

For polymorphic types, it is necessary to use functors on multiple arguments to capture their signature in order to account for type parameters. For example, for types with one parameter we need a functor on two arguments, usually called a *bifunctor*, to represent their structure.

```
class Bifunctor f where
  bimap :: (a → b) → (c → d) → f a c → f b d
```

Example 12 (Lists). The structure of polymorphic lists, $[a]$, is captured by a bifunctor *FList*,

```
data FList a b = FNil | FCons a b

instance Bifunctor FList where
  bimap f g FNil        = FNil
  bimap f g (FCons a b) = FCons (f a) (g b)
```

By fixing the bifunctor argument corresponding to the type parameter a (the type of the list elements) we get a functor $FList\ a$ which represents the signature of lists of type a:

```
instance Functor (FList a) where
  fmap f FNil        = FNil
  fmap f (FCons a b) = FCons a (f b)
```

Thus, $[a] = \mu\ (FList\ a)$.

5.2 Fold

Given a functor f that captures the signature of a data type and a function $k :: f\ a \to a$ (called an *f-algebra*), we can define a program scheme, called *fold* [3], which captures function definitions by structural recursion on the type $\mu\, f$.

$$\begin{aligned} &\mathit{fold} :: \mathit{Functor}\ f \Rightarrow (f\ a \to a) \to \mu\, f \to a \\ &\mathit{fold}\ k = k \circ \mathit{fmap}\ (\mathit{fold}\ k) \circ \mathit{unIn} \end{aligned}$$

The signature corresponding to a type T with n constructors is a functor that has also n cases. The same occurs with the algebras for that functor; they are essentially a tuple $(k_1, \ldots, k_n)$ of n component operations, each one with the appropriate type. For example, an algebra for the functor $\mathit{FList}\ a$ is a function $k :: \mathit{FList}\ a\ b \to b$ of the form:

$$\begin{aligned} k\ \mathit{FNil} &= e \\ k\ (\mathit{FCons}\ a\ b) &= f\ a\ b \end{aligned}$$

with components $e :: b$ and $f :: a \to b \to b$. This is the reason why *foldr*, the *fold* for lists, has type $(a \to b \to b) \to b \to [\,a\,] \to b$.

5.3 Shortcut fusion

The shortcut-fusion law of Section 2 can be generalised from list to all datatypes expressible as the (least) fixpoint of a functor [8,20]. The generic *build* can be defined as follows.

$$\begin{aligned} &\mathit{build} \quad :: (\mathit{Functor}\ f) \Rightarrow (\forall a.(f\ a \to a) \to c \to a) \to c \to \mu\, f \\ &\mathit{build}\ g = g\ \mathit{In} \end{aligned}$$

Notice that the abstraction of the datatype's constructors is represented in terms of an f-algebra. As explained before, the idea of shortcut fusion is then to replace, in the producer, the occurrences of the abstracted constructors by corresponding operations in the algebra of the fold that appears as consumer. The datatype-generic *fold/build law* is then:

Law 6 (FOLD/BUILD [8,20])

$$\mathit{fold}\ k \circ \mathit{build}\ g = g\ k$$

5.4 Extended shortcut fusion

The generic formulation of the extended build [7] is as follows:

$$\begin{aligned} &\mathit{ebuild} :: (\mathit{Functor}\ f, \mathit{Functor}\ h) \Rightarrow (\forall a.(f\ a \to a) \to c \to h\ a) \to c \to h\ (\mu\, f) \\ &\mathit{ebuild}\ g = g\ \mathit{In} \end{aligned}$$

where h is a functor that represents the container structure of the generated datatype. As we saw for lists, this is a natural extension of the standard build function. Using *ebuild* we can state the extended shortcut fusion law:

Law 7 (EXTENDED FOLD/BUILD **[7,14]**)

$$fmap\ (fold\ k) \circ ebuild\ g = g\ k$$

Fusion acts on the occurrences of the internal structure, while the context structure is maintained unchanged.

5.5 Generic traversals

It is possible to define *datatype-generic* traversals for parametric data structures corresponding to fixpoints of a parametric bifunctors. In order to define *traverse* generically, we must first establish when the signature of a datatype can be traversed:

```
class Bifunctor s ⇒ Bitraversable s where
  bitraverse :: (Applicative f) ⇒
                (a → f c) → (b → f d) → s a b → f (s c d)
```

Gibbons and Oliveira [10] present an equivalent characterisation: a bifunctor s is *Bitraversable* if for any applicative functor c there exists a *natural transformation* $bidist :: s\ (c\ a)\ (c\ b) \to c\ (s\ a\ b)$ which serves as a distributive law between the signature bifunctor and the applicative functor. Such distributive law exists for any given regular datatype and it can be defined *polytipically* i.e. by induction on the structure of the signature bifunctor [2,17]. As in the case of *traverse* and *dist* above, *bitraverse* and *bidist* are also interdefinable as $bidist = bitraverse\ id\ id$ and $bitraverse\ f\ g = bidist \circ bimap\ f\ g$. Thus, *traverse* can be defined generically for all fixed points of *Bitraversable* functors.

```
traverse :: (Applicative f, Bitraversable s) ⇒
            (a → f b) → μ (s a) → f (μ (s b))
traverse ι = fold (fmap In ∘ bitraverse ι id)
```

Gibbons and Oliveira [10] also claim that the *traverse* operator captures *"the essence of the Iterator pattern"* and have studied some calculational properties of idiomatic traversals. In Section 4, we saw that traversals play an important role in the characterisation of some common applicative forms of computation, like applicative structural recursion, and are well suited for fusion because of the fact of being good producers and good consumers simultaneously.

5.6 Applicative shortcut fusion

We define an idiomatic build to be an extended build where the container is an applicative functor.

```
ibuild  :: (Applicative f) ⇒ (∀b.(s a → a) → c → a) → c → f (μ s)
ibuild g = g In
```

The corresponding instance of extended shortcut fusion (Law 7) results:

Law 8 (FOLD/IBUILD)

$$fmap\ (fold\ \phi) \circ ibuild\ g = g\ \phi$$

5.7 Applicative structural recursion

Given a bitraversable bifunctor s, an algebra $\phi :: s\ b\ c \to c$ for the functor $(s\ b)$ and an applicative action $\iota :: a \to f\ b$ for an applicative functor f, we define *ifold* by the following equation:

$$\begin{aligned} &\mathit{ifold} \quad :: (\mathit{Applicative}\ f, \mathit{Bitraversable}\ s) \Rightarrow \\ &\qquad (s\ b\ c \to c) \to (a \to f\ b) \to \mu\ (s\ a) \to f\ c \\ &\mathit{ifold}\ \phi\ \iota = \mathit{fmap}\ (\mathit{fold}\ \phi) \circ \mathit{traverse}\ \iota \end{aligned}$$

which in turn, is equivalent to the following generalization of (1):

$$\mathit{ifold}\ \phi\ \iota = \mathit{fold}\ (\mathit{fmap}\ \phi \circ \mathit{bitraverse}\ \iota\ \mathit{id}) \tag{2}$$

Associated with *ifold* we have the following shortcut fusion law which gives a monolithic expression for the pattern *fold/traverse/build*:

Law 9 (IFOLD/BUILD)

$$\begin{aligned} &\mathit{ifold}\ \phi\ \iota \circ \mathit{build}\ g && \text{(I)} \\ =\ &\mathit{fmap}\ (\mathit{fold}\ \phi) \circ \mathit{traverse}\ \iota \circ \mathit{build}\ g && \text{(II)} \\ =\ &g\ (\mathit{fmap}\ \phi \circ \mathit{bitraverse}\ \iota\ \mathit{id}) && \text{(III)} \end{aligned}$$

Proof. $(I) = (II)$ by the definition of *ifold*. By the definition of *ifold* in terms of *fold*, (2), and Law 6, $(I) = (III)$. □

Note that in the *fold/traverse/build* pattern there is no need to use generalised shortcut fusion. The traversal takes care of creating and collecting the extra structure.

5.8 Composite functors

Applicative Functors are closed under functor composition. Gibbons and Oliveira [10] exploit this fact to define the *sequential composition* of applicative actions:

$$\begin{aligned} &\mathbf{data}\ (m \boxdot n)\ a = \mathit{Comp}\ \{\mathit{unComp} :: m\ (n\ a)\} \\ &(\odot) \quad :: (\mathit{Functor}\ m, \mathit{Functor}\ n) \Rightarrow (b \to n\ c) \to (a \to m\ b) \to a \to (m \boxdot n)\ c \\ &f \odot g = \mathit{Comp} \circ \mathit{fmap}\ f \circ g \end{aligned}$$

The $\odot$ operator can not only be used to compose traversals but also to show they are, in fact, closed under sequential composition i.e.

$$\mathit{traverse}\ (f \odot g) = \mathit{traverse}\ f \odot \mathit{traverse}\ g \tag{3}$$

Using this equation, we can derive a shortcut fusion law for the sequential composition of *ifold* and *traverse* as follows.

Law 10 (IFOLD/$\odot$/TRAVERSE)

$$\mathit{ifold}\ \phi\ \iota \odot \mathit{traverse}\ \kappa = \mathit{ifold}\ \phi\ (\iota \odot \kappa)$$

Proof (Sketch). By expanding definitions of $\odot$ and *ifold*, using functoriality and composition of traversals (3).

6 Conclusions and Future Work

We have presented two approaches to shortcut fusion for applicative computations. One is based on the extended shortcut fusion law tailored to applicative computations. We aimed at obtaining a more structured fusion law that took into account the way applicative computations are written. By analysing several examples we found that traversals are at the core of applicative computations. Based on this fact we proposed the pattern *fold/traverse/build* as the core of structural applicative computations and introduced a law for those patterns. This pattern elegantly separates the pure part of the computation from the one producing computational effects. We also introduced a notion of applicative structural recursion as the composition of a fold with a traversal.

Future Work The proposed pattern arose as a result of the study of several examples found in the literature (e.g. [16,10]). Despite the elegance of the results, we would like to obtain a more theoretically founded justification for them such as an initial algebra semantics for *ifold*. Related to this is the notion of a category of applicative computations, but this notion is still missing. Additionally we would like to extend our results to applicative functors with extra structure, such as the one in Example 10.

Acknowledgements: We thank the anonymous reviewers for their helpful comments and suggestions.

References

1. Abramsky, S., and Jung, A. Domain Theory. In *Handbook of Logic in Computer Science Volume 3*, S. Abramsky, D. Gabbay, and T. S. E. Maibaum, Eds. Oxford University Press, 1994, pp. 1–168.
2. Backhouse, R., Jansson, P., Jeuring, J., and Meertens, L. Generic programming — an introduction. In *LNCS* (1999), vol. 1608, Springer-Verlag, pp. 28–115. Revised version of lecture notes for AFP'98.
3. Bird, R., and de Moor, O. *Algebra of programming.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.
4. Bird, R. S. *Introduction to Functional Programming Using Haskell.* Prentice-Hall, 1998.
5. Fernandes, J. P., Pardo, A., and Saraiva, J. A shortcut fusion rule for circular program calculation. In *Haskell* (2007), G. Keller, Ed., ACM, pp. 95–106.
6. Ghani, N., and Johann, P. Monadic augment and generalised short cut fusion. *Journal of Functional Programming 17*, 6 (2007), 731–776.

7. Ghani, N., and Johann, P. Short cut fusion of recursive programs with computational effects. In *Trends in Functional Programming* (2009), P. Achten, P. Koopman, and M. Morazán, Eds., vol. 9 of *Trends in Functional Programming*, Intellect, pp. 113–128. ISBN 978-1-84150-277-9.
8. Ghani, N., Uustalu, T., and Vene, V. Build, augment and destroy, universally. In *APLAS* (2004), W.-N. Chin, Ed., vol. 3302 of *Lecture Notes in Computer Science*, Springer, pp. 327–347.
9. Gibbons, J. Datatype-generic programming. In *Datatype-Generic Programming*, R. Backhouse, J. Gibbons, R. Hinze, and J. Jeuring, Eds., vol. 4719 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2007, ch. 1, pp. 1–71–71.
10. Gibbons, J., and d. S. Oliveira, B. C. The essence of the iterator pattern. *Journal of Functional Programming 19*, 3-4 (2009), 377–402.
11. Gill, A., Launchbury, J., and Peyton Jones, S. A short cut to deforestation. In *FPCA '93: Proceedings of the conference on Functional Programming Languages and Computer Architecture* (New York, NY, USA, 1993), ACM Press, pp. 223–232.
12. Hughes, J. Why functional programming matters. *Comput. J. 32*, 2 (1989), 98–107.
13. Johann, P., and Ghani, N. Monadic fold, monadic build, monadic short cut fusion. In *Proceedings of the 10th Symposium on Trends in Functional Programming (TFP'09)* (2009), pp. 9 – 23.
14. Manzino, C., and Pardo, A. Shortcut fusion of monadic programs. *Journal of Universal Computer Science 14*, 21 (2008), 3431–3446.
15. Martínez, M., and Pardo, A. A shortcut fusion approach to accumulations. In *Simpósio Brasileiro de Linguagens de Programacao (SBLP 2009)* (2009).
16. McBride, C., and Paterson, R. Applicative programming with effects. *Journal of Functional Programming 18*, 01 (2008), 1–13.
17. Meertens, L. Functor pulling. In *Proc. Workshop on Generic Programming* (1998), R. Backhouse and T. Sheard, Eds.
18. Meijer, E., Fokkinga, M. M., and Paterson, R. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture* (London, UK, 1991), Springer-Verlag, pp. 124–144.
19. Pardo, A., Fernandes, J. P., and Saraiva, J. Shortcut fusion rules for the derivation of circular and higher-order monadic programs. In *PEPM* (2009), G. Puebla and G. Vidal, Eds., ACM, pp. 81–90.
20. Takano, A., and Meijer, E. Shortcut deforestation in calculational form. In *In Proc. Conference on Functional Programming Languages and Computer Architecture* (1995), ACM Press, pp. 306–313.
21. Wadler, P. Theorems for Free! In *Proceedings of the 4th ACM Conference on Functional Programming Languages and Computer Architecture, FPCA'89* (New York, 1989), ACM Press, pp. 347–359.