Theory and Practice of Fusion

Ralf Hinze, Thomas Harper, and Daniel W. H. James

Computing Laboratory, University of Oxford
Wolfson Building, Parks Road, Oxford, OX1 3QD, England
`ralf.hinze,tom.harper,daniel.james@comlab.ox.ac.uk`

Abstract. There are a number of approaches for eliminating intermediate data structures in functional programs—this elimination is commonly known as *fusion*. Existing fusion strategies are built upon various, but related, recursion schemes, such as folds and unfolds. We use the concept of *recursive coalgebras* as a unifying theoretical and notational framework to explore the foundations of these fusion techniques. We first introduce the calculational properties of recursive coalgebras and demonstrate their use with proofs and derivations in a calculational style, then provide an overview of fusion techniques by bringing them together in this setting. We also showcase these developments with examples in Haskell.

1 Introduction

Functional programmers love modular programs. It is easy for them to create clear, concise, and reusable code by composing functions. Consider the following Haskell program as an example:

$$\begin{aligned} &f \;:\; (\mathit{Integer}, \mathit{Integer}) \to \mathit{Integer} \\ &f = \mathit{sum} \cdot \mathit{map}\;\mathit{sq} \cdot \mathit{filter}\;\mathit{odd} \cdot \mathit{between} \;\;. \end{aligned}$$

The program takes a pair of integers representing an interval and returns the sum of the squared odd integers in this interval. We have expressed this a composition of four functions: *between* generates an enumeration between two natural numbers as a list, *filter odd* removes any even numbers, *map sq* squares the remaining (odd) numbers, and *sum* adds them together. Unfortunately, the clarity of this program comes at a cost. The constituent functions of this program communicate with each other using intermediate data structures, the production and immediate consumption of which carries an obvious performance penalty. Yet, because these definitions are recursive, eliminating the need for these transient structures is beyond the reach of a typical compiler.

Nonetheless, such a transformation is possible. We can manually construct a program that is equivalent to f, but without the intermediate data structures

```
f′ (m, n) = go m
  where go m | m > n      = 0
             | otherwise  = go (m + 1) + if odd m then sq m else 0  .
```

This new program has lost the desirable qualities of the original—our concise, modular and declarative code has been hammered into a long, opaque and specialised function. In doing so, however, we have accomplished our goal of removing the intermediate data structures by transforming the numerous recursive traversals into a single one. This process is called *fusion*.

Fusing programs by hand quickly becomes infeasible for those of non-trivial length. Furthermore, it can be difficult to manually pinpoint all the opportunities for fusion. Instead, such a transformation should be performed automatically. Difficulties arise, however, in automatically fusing functions defined using general recursion. Specifically, such transformations often have proof obligations that cannot be discharged by the compiler.

One remedy is to standardise the way data structures are produced or consumed by encapsulating the recursion scheme in a higher-order function. The arguments to these functions are the non-recursive 'steps'. Simple syntactic transformations can then fuse many recursive traversals into a single one, and then non-recursive steps can be optimised using conventional methods. This approach is known as *shortcut fusion*. Different incarnations of this technique utilise different recursion schemes, e.g. *folds* for consumers or *unfolds* for producers. The steps of such a scheme are known as algebras or coalgebras, respectively.

The implementation of these fusion techniques is usually described syntactically, by giving a definition of the production and consumption combinators and accompanying rewrite rules. This alone does not really explain the underlying fusion mechanism. Furthermore, it is difficult to construct correctness proofs, or relate various fusion approaches to one another, despite the fact that such close relations exist. In this paper, we move fusion to a clearer setting, where the syntactic details of fusion fall away.

Category theory provides the tools we need to tackle the semantics of the recursion schemes. While some fusion techniques have been individually given this treatment before, our focus is to bring them all under one roof. In this paper, we propose using *recursive coalgebras* as that roof. We will show how recursive coalgebras enable us to explain the fusion rules underlying the various fusion techniques and give short, simple proofs of correctness.

Some proofs and source code examples have been elided from this paper. Any reader who desires further material should consult the associated technical report [10].

2 Background: Algebras and Coalgebras

The category theory concept of an *initial algebra* is key in the theory of functional programming languages, specifically for giving a semantics to recursive datatypes [15]. The remainder of this section will refresh the salient details. For the more functional-programming-minded reader, we will parallel these developments with examples in Haskell, where possible.

Let $\mathsf{F} : \mathbb{C} \to \mathbb{C}$ be a functor. An F-*algebra* is a pair $\langle a, A \rangle$ consisting of an object $A : \mathbb{C}$ and an arrow $a : \mathsf{F}\, A \to A : \mathbb{C}$. An F-*algebra homomorphism* between algebras $\langle a, A\rangle$ and $\langle b, B\rangle$ is an arrow $h : A \to B : \mathbb{C}$ such that $h \cdot a = b \cdot \mathsf{F}\, h$. The fact that functors preserve identity and composition entails that identity is a homomorphism and that homomorphisms compose. Consequently, F-algebras and their homomorphisms form a category, called $\mathbf{Alg}(\mathsf{F})$. We abbreviate a homomorphism, $h : \langle a, A\rangle \to \langle b, B\rangle : \mathbf{Alg}(\mathsf{F})$, by $h : a \to b : \mathbf{Alg}(\mathsf{F})$ if the objects are obvious from the context, or simply by $h : a \to b$ if the functor F is also obvious.

In Haskell, we model a functor as a datatype whose constructors describe its action on data (i.e. objects). Its action on arrows is defined by making that data type an instance of the *Functor* typeclass

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

We can then simply treat the concept of an F-algebra as a function, where F is a datatype that is an instance of the *Functor* class. The F-algebra $\langle a, A\rangle$ is simply a function $a : \mathsf{F}\, A \to A$. An F-algebra homomorphism between a and $b : \mathsf{F}\, B \to B$ is a function $h : A \to B$ that satisfies the side condition $h \cdot a = b \cdot \mathit{fmap}\, h$. This property cannot, however, be deduced from the type and must be checked by the programmer.

If the category $\mathbf{Alg}(\mathsf{F})$ has an initial object, then we call it $\langle \mathit{in}, \mu\mathsf{F}\rangle$. Initiality means that there is a *unique* arrow from $\langle \mathit{in}, \mu\mathsf{F}\rangle$ to any other F-algebra $\langle a, A\rangle$. This arrow, called *fold*, is written $(\!|a|\!) : \mathit{in} \to a$. We construct elements of $\mu\mathsf{F}$ using in and deconstruct them using $(\!|a|\!)$. We can think of $(\!|a|\!) : \mathit{in} \to a$ as replacing constructors by functions, represented by the algebras in and a, respectively. Initiality is captured by the *uniqueness property* of folds:[1]

$$h = (\!|a|\!) \quad \iff \quad h : \mathit{in} \to a \quad \iff \quad h \cdot \mathit{in} = a \cdot \mathsf{F}\, h \ . \tag{1}$$

It is important to note that we have omitted the quantification of the names that appear in property (1); we have done so for presentational succinctness and we will continue in this manner. In this case we will spell out implicit quantification: the uniqueness property holds for all functors F, where the category $\mathbf{Alg}(\mathsf{F})$ has an initial object named $\langle \mathit{in}, \mu\mathsf{F}\rangle$, and for all F-algebras $\langle a, A\rangle$, and for all F-algebra homomorphisms $h : \mathit{in} \to a$.

This property provides us with a general definition of $(\!|-|\!)$ in Haskell. First, we define the μ datatype, which takes a functor to its least fixed point:

```
data μf = in { out : f (μf) }
```

The constructor *in* allows us to construct a structure of type μf out of something of type $f\ (\mu f)$ and *out* deconstructs it. The relationship between *in* and *out* in our setting is discussed further in Sections 3 and 4.

[1] The formula $P \iff Q \iff R$ has to be read *conjunctively* as $P \iff Q \wedge Q \iff R$. Likewise, $P \Longleftarrow Q \iff R$ is shorthand for $P \Longleftarrow Q \wedge Q \iff R$.

We can define $(\!|-|\!)$ as a higher-order function that takes an algebra $f\ a \to a$ and returns a function $\mu f \to a$, according to the uniqueness property:

$$
\begin{aligned}
&(\!|-|\!) : (\mathit{Functor}\, f) \Rightarrow (f\ a \to a) \to (\mu f \to a)\\
&(\!|a|\!) = a \cdot \mathit{fmap}\ (\!|a|\!) \cdot \mathit{out}
\end{aligned}
$$

By allowing us to substitute $(\!|a|\!)$ for h, we see that the uniqueness property provides us with a definition of $(\!|-|\!)$ that recursively replaces occurrences of *in* by some algebra a. The placement of the recursive call for a given structure μf is determined by the definition *fmap*.

We will not employ the uniqueness property in an example proof just yet. In fact the uniqueness property is rarely used in its raw form; instead, there are a number of specific forms that we will introduce now.

If we set h to the identity *id* and a to the initial algebra *in*, then we obtain the *reflection law*: $(\!|\mathit{in}|\!) = \mathit{id}$. If we substitute the left-hand side into the right-hand side, then we obtain the *computation law*: $(\!|a|\!) : \mathit{in} \to a$, or expressed in terms of the base category, $(\!|a|\!) \cdot \mathit{in} = a \cdot \mathsf{F}\,(\!|a|\!)$.

The most important consequence of the uniqueness property is the *fusion law* for fusing an arrow with a fold to form a new fold.

$$
h \cdot (\!|a|\!) = (\!|b|\!) \quad \Longleftarrow \quad h : a \to b \quad \Longleftrightarrow \quad h \cdot a = b \cdot \mathsf{F}\, h \tag{2}
$$

As its name would suggest, the fusion law is closely related to the program transformation techniques described in the introduction. It allows a fold to absorb a function on its left, thereby producing a single fold. The law also shows the difficulty of mechanising this process; in order to produce the fused program, we must invent a new algebra b that satisfies the precondition.

Folds enjoy an additional fusion law. Whereas fusion allows us to absorb an additional function on the left, the *functor fusion law* allows us to absorb a function on the right. In order to formulate it, we have to turn μ into a higher-order functor of type $\mathbb{C}^{\mathbb{C}} \to \mathbb{C}$. The object part of this functor maps a functor to its initial algebra. (This is only well-defined for functors that have an initial algebra.) The arrow part maps a natural transformation $\alpha : \mathsf{F} \dot{\to} \mathsf{G}$ to an arrow $\mu\alpha : \mu\mathsf{F} \to \mu\mathsf{G} : \mathbb{C}$. It is defined as $\mu\alpha = (\!|\mathit{in} \cdot \alpha\,(\mu\mathsf{F})|\!)$. To reduce clutter, we will henceforth omit the argument of the natural transformation α. From these definitions we obtain the *functor fusion law* (we have annotated $(\!|-|\!)$ with the underlying functors):

$$
(\!|b|\!)_{\mathsf{G}} \cdot \mu\alpha \;=\; (\!|b \cdot \alpha|\!)_{\mathsf{F}} \ . \tag{3}
$$

It states that a fold after a map can be fused into a single fold — the map $\mu\alpha$ can be seen as a “base changer”.

We can also provide a Haskell definition of μ as a functor. The action on data is given by its datatype declaration. The action on functions is given by:

$$
\begin{aligned}
&\mu - : (\mathit{Functor}\, f) \Rightarrow (\forall\, a\ .\ f\ a \to g\ a) \to (\mu f \to \mu g)\\
&\mu\alpha = (\!|\mathit{in} \cdot \alpha|\!)
\end{aligned}
$$

Note that we use a rank-2 polymorphic type to express the idea that μ maps natural transformation from f to g to a function between their fixpoints.

Finally, the initial algebra $\mu\mathsf{F}$ is the least fixed point of F — this is known as Lambek's Lemma [13]. One direction of the isomorphism $\mathsf{F}\,(\mu\mathsf{F}) \cong \mu\mathsf{F}$ is given by *in*, its inverse is $in^{\circ} = (\!|\mathsf{F}\,in|\!)$. Lambek's Lemma is the key to giving a semantics to recursively defined datatypes. To illustrate this, the recursive definition of lists of natural numbers

$$\mathbf{data}\,\mathsf{List} = Nil \mid Cons\,(\mathbb{N}, \mathsf{List})$$

implicitly defines an underlying functor $\mathsf{L}\,X = 1 + \mathbb{N} \times X$, the so-called base functor of List. (This notation is a categorical rendering of sum-of-products algebraic datatypes, and defines a functor L with an argument X, where 1 denotes the terminal object of the underlying category.) Since the initial object $\mu\mathsf{L}$ satisfies the equation $X \cong \mathsf{L}\,X$, we can use it to assign meaning to the recursive datatype definition. (As an aside, the fold of the List datatype is a specialisation of Haskell's library function *foldr*.)

The initial F-algebra is the least solution of the equation $X \cong \mathsf{F}\,X$. If we dualise the development above, we obtain another canonical solution, namely the greatest one. In category theory, dualisation is denoted by the prefix "co-".

An F-*coalgebra* is a pair $\langle C, c\rangle$ consisting of an object $C : \mathbb{C}$ and an arrow $c : C \to \mathsf{F}\,C : \mathbb{C}$. An F-*coalgebra homomorphism* between coalgebras $\langle C, c\rangle$ and $\langle D, d\rangle$ is an arrow $h : C \to D : \mathbb{C}$ such that $\mathsf{F}\,h \cdot c = d \cdot h$. Coalgebras and coalgebra homomorphisms also form a category, called $\mathbf{Coalg}(\mathsf{F})$. The dual of the initial algebra is the final coalgebra, whose carrier $\nu\mathsf{F}$ is the greatest fixed point of F. Finality means that, for any other coalgebra, there is a unique arrow from it to the final coalgebra. Whereas a fold consumes a data structure, an unfold produces some data structure from a given seed.

Unfortunately, least and greatest fixed points are different beasts in general. In the category **Set** of sets and total functions, $\mu\mathsf{L}$ is the set of finite lists, whereas $\nu\mathsf{L}$ also contains infinite lists. This means that folds and unfolds are incompatible, in general. In the following section, we will focus on a restricted species of coalgebras, enabling us to work with folds and unfolds under the same roof.

3 Recursive Coalgebras

In this section we will introduce *recursive* coalgebras. We follow the work of Capretta et al. [2], who motivate the use of hylomorphisms based on recursive coalgebras as a structured recursion scheme. We shall continue to parallel our developments with examples in Haskell.

A coalgebra $\langle C, c\rangle$ is called *recursive* if for *every* algebra $\langle a, A\rangle$ the equation in the unknown $h : A \leftarrow C$,

$$h \;=\; a \cdot \mathsf{F}\,h \cdot c\ , \tag{4}$$

has a *unique* solution. The equation captures *divide-and-conquer*: a problem is divided into sub-problems (c), the sub-problems are solved recursively ($\mathsf{F}\,h$), and finally the sub-solutions are combined into a single solution (a). The uniquely defined function h is called a *hylomorphism* or *hylo* for short and is written $(\!| a \leftarrow c |\!)_{\mathsf{F}} : A \leftarrow C$. The notation is meant to suggest that h takes a coalgebra to an algebra. We omit the subscripted functor name if it is obvious from the context. Uniqueness of h is captured by the following property.

$$h = (\!| a \leftarrow c |\!) \quad \iff \quad h = a \cdot \mathsf{F}\, h \cdot c \tag{5}$$

In Haskell, $(\!| - \leftarrow - |\!)$ becomes a function that takes an algebra and a recursive coalgebra (which, dual to algebras, is a function of type $c \rightarrow f\ c$) as arguments and returns resulting hylo according to the definition in the universal property:

$$\begin{aligned}
&(\!| - \leftarrow - |\!) : (\mathit{Functor}\, f) \Rightarrow (f\ a \rightarrow a) \rightarrow (c \rightarrow f\ c) \rightarrow (c \rightarrow a)\\
&(\!| a \leftarrow c |\!) = a \cdot \mathit{fmap}\, (\!| a \leftarrow c |\!) \cdot c \ \ .
\end{aligned}$$

This function takes an algebra and a recursive coalgebra, yielding a hylo. Note that the type of this function does not guarantee that c is a *recursive* coalgebra and therefore does not guarantee that the resulting hylo has a unique solution; the programmer needs to discharge this obligation by some other means.

The category of recursive coalgebras and coalgebra homomorphisms forms a full subcategory of **Coalg**(F), called **Rec**(F). If the latter category has a final object $\langle F, \mathit{out} \rangle$, then there is a unique arrow from any other *recursive* coalgebra $\langle C, c \rangle$ to $\langle F, \mathit{out} \rangle$. This arrow, called *unfold*, is written $[\![c]\!] : c \rightarrow \mathit{out}$. Finality is captured by the following uniqueness property.

$$h = [\![c]\!] \quad \iff \quad h : c \rightarrow \mathit{out} \quad \iff \quad \mathsf{F}\, h \cdot c = \mathit{out} \cdot h \tag{6}$$

This is the usual property of unfolds, except that we are working in the category **Rec**(F), *not* **Coalg**(F). As with folds, we can draw out a Haskell definition of unfolds from the uniqueness property:

$$\begin{aligned}
&[\![-]\!] : (\mathit{Functor}\, f) \Rightarrow (c \rightarrow f\ c) \rightarrow (c \rightarrow \mu f)\\
&[\![c]\!] = \mathit{in} \cdot \mathit{fmap}\, [\![c]\!] \cdot c \ \ .
\end{aligned}$$

In contrast to folds, we are *creating* a structure of type μf from a seed value. The recursion, similarly, is determined by the form of the underlying functor f through the use of *fmap*. The uniqueness property for unfolds, like the one for folds, implies the *reflection law*, $[\![\mathit{out}]\!] = \mathit{id}$, the *computation law*, $\mathsf{F}[\![c]\!] \cdot c = \mathit{out} \cdot [\![c]\!]$, and the *fusion law*:

$$[\![c]\!] = [\![d]\!] \cdot h \quad \Longleftarrow \quad h : c \rightarrow d \quad \iff \quad \mathsf{F}\, h \cdot c = d \cdot h \ \ . \tag{7}$$

The definition of a hylomorphism does not assume that the initial F-algebra exists. The powerset functor, for instance, admits no fixed points. However, if the initial algebra exists, then it coincides with the final recursive coalgebra and, furthermore, folds and unfolds emerge as special cases of hylos. We can state this more formally:

Theorem 1. *Initial* F*-algebras and final recursive* F*-coalgebras coincide: (1) If* $\langle C, \mathit{out} \rangle$ *is the final recursive* F*-coalgebra, then* $\langle \mathit{out}^{\circ}, C \rangle$ *is the initial* F*-algebra. Furthermore,* $(\!| a |\!) = (\!| a \leftarrow \mathit{out} |\!)$*. (2) If* $\langle \mathit{in}, A \rangle$ *is the initial* F*-algebra, then* $\langle A, \mathit{in}^{\circ} \rangle$ *is the final recursive* F*-coalgebra. Furthermore,* $[\![c]\!] = (\!| \mathit{in} \leftarrow c |\!)$*.*

Theorem 1 allows us to treat folds and unfolds in the same setting—note that an unfold produces an element of an initial algebra! An alternative is to work in a setting where $\mu\mathsf{F}$ and $\nu\mathsf{F}$ coincide; an *algebraically compact* category is such a setting [8]. Haskell's ambient category $\mathbf{Cpo}_\perp$ serves as the standard example. This is the usual approach [7], however, the downside is that the hylo equation (4) only has a canonical, least solution, not a unique solution, so (5) does not hold.

4 Calculational Properties

In this section we will cover the calculational properties of our hylomorphisms. In a similar fashion to folds and unfolds, hylomorphisms have an identity law and a computation law, and they follow similarly from the uniqueness property (5).

Identity law Setting $h := id$, we obtain the *identity law*

$$(\!|a \leftarrow c|\!) = id \quad \iff \quad a \cdot c = id \ . \tag{8}$$

Computation law Substituting the left-hand side into the right-hand side gives the *computation law*:

$$(\!|a \leftarrow c|\!) \;=\; a \cdot \mathsf{F}\,(\!|a \leftarrow c|\!) \cdot c \ . \tag{9}$$

For hylomorphisms, we have *three* fusion laws: algebra fusion, coalgebra fusion, and composition.

Algebra fusion An algebra homomorphism after a hylo can be fused to form a single hylo.[2]

$$h \cdot (\!|a \leftarrow c|\!) = (\!|b \leftarrow c|\!) \quad \Longleftarrow \quad h : a \to b \quad \iff \quad h \cdot a = b \cdot \mathsf{F}\,h \tag{10}$$

For the proof we appeal to the uniqueness property and show that $h \cdot (\!|a \leftarrow c|\!)$ satisfies the recursion equation of $(\!|b \leftarrow c|\!)$. The obligation is discharged as follows:

$$\begin{aligned}
& h \cdot (\!|a \leftarrow c|\!) \\
= \quad & \{ \text{ hylo computation (9) } \} \\
& h \cdot a \cdot \mathsf{F}\,(\!|a \leftarrow c|\!) \cdot c \\
= \quad & \{ \text{ assumption: } h : a \to b \ \} \\
& b \cdot \mathsf{F}\,h \cdot \mathsf{F}\,(\!|a \leftarrow c|\!) \cdot c \\
= \quad & \{ \ \mathsf{F} \text{ functor } \} \\
& b \cdot \mathsf{F}\,(h \cdot (\!|a \leftarrow c|\!)) \cdot c \ .
\end{aligned}$$

Coalgebra fusion Dually, we can fuse a coalgebra homomorphism before a hylo to form a single hylo.

$$(\!|a \leftarrow c|\!) = (\!|a \leftarrow d|\!) \cdot h \quad \Longleftarrow \quad h : c \to d \quad \iff \quad \mathsf{F}\,h \cdot c = d \cdot h \tag{11}$$

[2] Note that h appears as both an algebra homomorphism in $\mathbf{Alg}(\mathsf{F})$ and as the underlying arrow in the underlying category.

Like the law, the proof is the dual of that for algebra fusion.

Composition law A composition of hylos can be merged into a single one if the coalgebra of the hylo on the left inverts the algebra of the right hylo.

$$(\!| a \leftarrow c |\!) \cdot (\!| b \leftarrow d |\!) = (\!| a \leftarrow d |\!) \quad \Longleftarrow \quad c \cdot b = id \tag{12}$$

Composition is, in fact, a simple consequence of algebra fusion as the hylomorphism $(\!| a \leftarrow c |\!) : b \rightarrow a$ is simultaneously an F-algebra homomorphism.

$$\begin{aligned}
& (\!| a \leftarrow c |\!) \cdot b \\
= \; & \{ \text{ hylo computation (9) } \} \\
& a \cdot \mathsf{F}\, (\!| a \leftarrow c |\!) \cdot c \cdot b \\
= \; & \{ \text{ assumption: } c \cdot b = id \ \} \\
& a \cdot \mathsf{F}\, (\!| a \leftarrow c |\!)
\end{aligned}$$

Alternatively, we can derive the composition law from coalgebra fusion by showing that $(\!| b \leftarrow d |\!) : d \rightarrow c$ is an F-coalgebra homomorphism. The composition law, together with the next law, generalises the functor fusion law of folds.

Hylo shift law or base change law If we have a natural transformation $\alpha : \mathsf{G} \dot{\rightarrow} \mathsf{F}$, then

$$(\!| a \cdot \alpha\, A \leftarrow c |\!)_{\mathsf{G}} \;=\; (\!| a \leftarrow \alpha\, C \cdot c |\!)_{\mathsf{F}} \;. \tag{13}$$

In fact, the statement can be strengthened: if c is recursive, then $\alpha\, C \cdot c$ is recursive, as well.

$$\begin{aligned}
& h = a \cdot \mathsf{F}\, h \cdot \alpha\, C \cdot c \\
\Longleftrightarrow \; & \{ \ \alpha \text{ natural } \} \\
& h = a \cdot \alpha\, A \cdot \mathsf{G}\, h \cdot c \\
\Longleftrightarrow \; & \{ \text{ uniqueness property of hylos (5) } \} \\
& h = (\!| a \cdot \alpha\, A \leftarrow c |\!)_{\mathsf{G}}
\end{aligned}$$

It is worth pointing out that the laws stated thus far are independent of the existence of initial algebras. Only the following law makes this assumption.

Fold/unfold law A fold after an unfold is a hylo.

$$(\!| a |\!) \cdot [\![c]\!] \;=\; (\!| a \leftarrow c |\!) \tag{14}$$

From left to right we are performing fusion and thus deforesting an intermediate data structure. From right to left we are turning a control structure into a data structure. The *fold/unfold law* is a direct consequence of Theorem 1 and any of the fusion laws.

5 Fusion

In the previous sections we have introduced the fusion laws that we will now use to help us explain a collection of specific fusion techniques. We collectively brand these techniques *shortcut fusion*, as they share the common characteristic of standardising the way data structures are recursively consumed and produced. Where shortcut fusion techniques differ is in their choice of recursion scheme. By using recursive coalgebras, we can clearly lay out and compare these approaches within the *same* framework.[3] This allows us to examine the relationships among these fusion approaches which are not readily apparent when examining their individual implementations.

5.1 Warm-up: Type Functors

We have seen in §2 that μ is a functor, whose action on arrows is defined $\mu\alpha = (\!|in \cdot \alpha|\!)$. Using Theorem 1 and the hylo shift law (13) we can actually express $\mu\alpha$ as a fold, an unfold or a hylo.

$$\mu\alpha \;=\; (\!|in \cdot \alpha|\!) \;=\; (\!|in \cdot \alpha \leftarrow out|\!) \;=\; (\!|in \leftarrow \alpha \cdot out|\!) \;=\; [\![\alpha \cdot out]\!] \ .$$

In §5.5 we shall see a key use of μ for stream fusion. For now, let us show a use of μ with the base functor of parametrized List

$$\mathbf{data}\,\mathsf{L}\;a\;b = \mathit{Nil} \mid \mathit{Cons}\,(a, b) \ .$$

This is a higher-order functor of type $\mathsf{L} : \mathbb{C} \to \mathbb{C}^{\mathbb{C}}$ that takes objects to functors and arrows to natural transformations. In Haskell, we can make this datatype an instance of the *Functor* class:

```
instance Functor (L a) where
  fmap f Nil             = Nil
  fmap f (Cons (a, b)) = Cons (a, f b)  .
```

We define this instance for the functor obtained by applying L to some type a. Haskell allows us to define this polymorphically for all a. The list datatype defined in terms of its base functor is $\mathsf{List}\,A = \mu(\mathsf{L}\,A)$. The parametric type List is itself a functor, a so-called type functor, whose action on arrows is Haskell's *map* function, defined in this setting by $\mathsf{List}\,f = \mu(\mathsf{L}\,f)$. Note that μ expects a natural transformation and that L delivers one.

5.2 Generalised *foldr/build* Fusion

We now move on to the main target of our new setting: shortcut fusion. The original shortcut fusion technique is a fold-centric approach called *foldr/build* fusion [9]. As its name would suggest, its original intention was to provide fusion for list functions written in terms of *foldr* and an additional combinator *build*. In this section, we will explore the foundations of this technique.

The mother of all fusion rules is algebra fusion (10). It allows us to fuse a hylo followed by an algebra homomorphism into a single hylo. It is similar to

[3] Previously these recursion schemes were only compatible for analysis by restricting the working category to one that is algebraically compact, such as $\mathbf{Cpo}_\perp$.

fold fusion in the sense that to use this law, we must construct a new algebra that satisfies a pre-condition. To illustrate this, the pipeline $sum \cdot filter\ odd$ can be expressed as a composition of two folds: $(\!|\mathfrak{s}|\!) \cdot (\!|\mathfrak{f}|\!)$. The algebras $\mathfrak{s}$ and $\mathfrak{f}$ are given by

$$\begin{aligned}
&\mathfrak{f} : \mathsf{L}\,\mathbb{N}\,(\mu(\mathsf{L}\,\mathbb{N})) \to \mu(\mathsf{L}\,\mathbb{N}) \\
&\mathfrak{f}\,Nil = in\,Nil \\
&\mathfrak{f}\,(Cons\,(x,y)) = \mathbf{if}\ odd\ x \\
&\qquad \mathbf{then}\ in\,(Cons\,(x,y)) \\
&\qquad \mathbf{else}\ y
\end{aligned}
\qquad
\begin{aligned}
&\mathfrak{s} : \mathsf{L}\,\mathbb{N}\,\mathbb{N} \to \mathbb{N} \\
&\mathfrak{s}\,Nil = 0 \\
&\mathfrak{s}\,(Cons\,(x,y)) = x + y \ .
\end{aligned}$$

To be able to apply algebra fusion (10), we have to show that $(\!|\mathfrak{s}|\!)$ is an algebra homomorphism from $\mathfrak{f}$ to some unknown algebra $\mathfrak{sf}$. By hand, it is not hard to derive $\mathfrak{sf}$ so that $(\!|\mathfrak{s}|\!) \cdot \mathfrak{f} = \mathfrak{sf} \cdot \mathsf{F}\,(\!|\mathfrak{s}|\!)$.

$$\begin{aligned}
&\mathfrak{sf} : \mathsf{L}\,\mathbb{N}\,\mathbb{N} \to \mathbb{N} \\
&\mathfrak{sf}\,Nil = \mathfrak{s}\,Nil \\
&\mathfrak{sf}\,(Cons\,(x,y)) = \mathbf{if}\ odd\ x\ \mathbf{then}\ \mathfrak{s}\,(Cons\,(x,y))\ \mathbf{else}\ y
\end{aligned}$$

Since $(\!|\mathfrak{s}|\!)$ replaces *in* by $\mathfrak{s}$, we simply have to replace the occurrences of *in* in $\mathfrak{f}$ by $\mathfrak{s}$. While this is an easy task to perform by hand, it is potentially difficult to mechanise as it requires analysis of the body of $\mathfrak{f}$; within it, the constructor *in* could easily have any name and conversely any function could be named *in*. Also, $\mathfrak{f}$ could contain unrelated occurrences of *in*. This transformation is therefore not purely syntactic, but also involves some further analysis of the source program; this is not an approach we wish to pursue.

The central idea of *foldr*/*build* fusion is to expose *in* so that replacing it by the algebra a is simple to implement. Consider fold fusion (2) again.

$$h \cdot (\!|a|\!) = (\!|b|\!) \quad \Longleftarrow \quad h : a \to b$$

A fold $(\!|-|\!)$ is a transformation that takes an algebra to a homomorphism. Assume that we have another such transformation, say, β that satisfies

$$h \cdot \beta\,a = \beta\,b \quad \Longleftarrow \quad h : a \to b \ . \tag{15}$$

The generalisation of *foldr*/*build* from lists to arbitrary datatypes, the so-called *acid rain rule* [19], is then

$$(\!|a|\!) \cdot \beta\,in \ = \ \beta\,a \ . \tag{16}$$

Using β we expose *in* so that replacing *in* by a is achieved through a simple function application. Instead of building a structure and then folding over it, we eliminate the *in* and pass a directly to β. The proof of correctness is painless.

$$\begin{aligned}
&(\!|a|\!) \cdot \beta\,in = \beta\,a \\
\Longleftarrow\ &\{\ \text{assumption (15)}\ \} \\
&(\!|a|\!) : in \to a
\end{aligned}$$

But, have we made any progress? After all, before we can apply (16), we have to prove (15). Fold satisfies this property, but this instance of (16) is trivial:

$(\!|a|\!) \cdot (\!|in|\!) = (\!|a|\!)$. Now, it turns out that in a *relationally parametric* programming language [16], the proof obligation (15) amounts to the *free theorem* [21] of the polymorphic type

$$\beta : \forall A \,.\, (\mathsf{F}\,A \to A) \to (B \to A) \ , \tag{17}$$

where B is some fixed type. In other words, in such a language the proof obligation can be discharged by the type checker.

Returning to our example, we redefine *filter odd* as $(\lambda\, a \,.\, (\!|\phi\, a|\!))\, in$ where

$$\begin{aligned}
&\phi : (\mathsf{L}\,\mathbb{N}\,b \to b) \to (\mathsf{L}\,\mathbb{N}\,b \to b)\\
&\phi\, a\, \mathit{Nil} &&= a\,\mathit{Nil}\\
&\phi\, a\, (\mathit{Cons}\,(x,y)) &&= \mathbf{if}\ \mathit{odd}\,x\ \mathbf{then}\ a\,(\mathit{Cons}\,(x,y))\ \mathbf{else}\ y \ .
\end{aligned}$$

We derived ϕ from the algebra $\mathfrak{f}$ by abstracting away from in. The reader should convince herself that $\lambda\, a \,.\, (\!|\phi\, a|\!)$ has indeed the desired polymorphic type (17). We can then invoke the acid rain rule (16) to obtain

$$(\!|\mathfrak{s}|\!) \cdot (\lambda\, a \,.\, (\!|\phi\, a|\!))\, in = (\lambda\, a \,.\, (\!|\phi\, a|\!))\, \mathfrak{s} = (\!|\phi\, \mathfrak{s}|\!) \ .$$

The example also shows that the *acid rain* rule is somewhat unstructured in that a hylo is hidden inside the abstraction $\lambda\, a$. Without performing an additional beta-reduction, we can apply the rule only once. We obtain a more structured rule if we shift the abstraction to the algebra and achieve *cata-hylo fusion:* If τ is a transformation that takes F-algebras to G-algebras satisfying

$$h : \tau\, a \to \tau\, b : \mathbf{Alg}(\mathsf{G}) \quad \Longleftarrow \quad h : a \to b : \mathbf{Alg}(\mathsf{F}) \ , \tag{18}$$

then

$$(\!|a|\!)_{\mathsf{F}} \cdot (\!|\tau\, in \leftarrow c|\!)_{\mathsf{G}} \quad = \quad (\!|\tau\, a \leftarrow c|\!)_{\mathsf{G}} \ . \tag{19}$$

If τ is $\lambda\, a \,.\, a$, then this is just the fold/unfold law (14). For $\tau\, a = a \cdot \alpha$, this is essentially functor fusion (3). The proof of correctness is straightforward.

$$\begin{aligned}
&(\!|a|\!)_{\mathsf{F}} \cdot (\!|\tau\, in \leftarrow c|\!)_{\mathsf{G}} = (\!|\tau\, a \leftarrow c|\!)_{\mathsf{G}}\\
=\ &\{ \text{ algebra fusion (10) } \}\\
&(\!|a|\!)_{\mathsf{F}} : \tau\, in \to \tau\, a : \mathbf{Alg}(\mathsf{G})\\
=\ &\{ \text{ assumption (18) } \}\\
&(\!|a|\!)_{\mathsf{F}} : in \to a : \mathbf{Alg}(\mathsf{F})
\end{aligned}$$

The proof obligation (18) once again amounts to a theorem for free, this time of the polymorphic type

$$\tau : \forall A \,.\, (\mathsf{F}\,A \to A) \to (\mathsf{G}\,A \to A) \ .$$

Using cata-hylo fusion, the running example simplifies to

$$(\!|\mathfrak{s}|\!) \cdot (\!|\phi\, in|\!) = (\!|\phi\, \mathfrak{s}|\!) \ .$$

We can now also fuse a composition of folds:

$$(\!|a|\!) \cdot (\!|\tau_1\, in|\!) \cdot \ldots \cdot (\!|\tau_n\, in|\!) \cdot [\![c]\!] = (\!|(\tau_n \cdot \ldots \cdot \tau_1)\, a \leftarrow c|\!) \ .$$

This demonstrates how the rewrite rule is able to achieve fusion over an entire pipeline of functions.

5.3 Generalised *destroy*/*unfoldr* Fusion

The *foldr*/*build* brand of shortcut fusion, and its generalisation to algebraic datatypes, is *fold-centric*. This limits the kind of functions that we can fuse, simply because some functions such as *zip* or *take* are not folds, or are not naturally written as folds. We can dualise *foldr*/*build* fusion to achieve an *unfold-centric* approach, called *destroy*/*unfoldr* [18]. To illustrate, consider the simple pipeline $\mathit{take}\,5 \cdot \mathit{between}$, where $\mathit{take}\,n$ takes n elements (if available) from a list. It can be written as an unfold after an initialisation step: $\mathit{take}\,n = [\![\mathfrak{t}]\!] \cdot \mathit{start}\,n$, where $\mathit{start}\,n = (\lambda\, l \,.\, (n, l))$, and where the coalgebra $\mathfrak{t}$ is given by

$$\begin{aligned}
&\mathbf{type}\,\mathsf{State}\,a = (\mathbb{N}, a)\\
&\mathfrak{t} : \mathsf{State}\,(\mu(\mathsf{L}\,a)) \to \mathsf{L}\,a\,(\mathsf{State}\,(\mu(\mathsf{L}\,a)))\\
&\mathfrak{t}\,(0, x) \qquad\;\; = \mathit{Nil}\\
&\mathfrak{t}\,(n+1, x) = \mathbf{case}\;\mathit{out}\,x\;\mathbf{of}\;\mathit{Nil} \to \mathit{Nil};\,\mathit{Cons}\,(a, y) \to \mathit{Cons}\,(a, (n, y))\;\;.
\end{aligned}$$

Here we make explicit the notion that an unfold models the steps of a stateful computation. The coalgebra takes a state as an argument and uses it to produce a value and a new state. In this example, the state type pairs the input list with a natural number, enabling us to track the overall number of values produced. The number of elements to take, paired with the list where the values are to be taken from, forms the initial state.

We can dualise the *acid rain rule* to fuse the pipeline. If β is a transformation that satisfies

$$\beta\,c = \beta\,d \cdot h \quad \Longleftarrow \quad h : c \to d\;\;, \tag{20}$$

then

$$\beta\,c \;\;=\;\; \beta\,\mathit{out} \cdot [\![c]\!]\;\;. \tag{21}$$

Previously we exposed *in*, now we expose *out*. To apply the dual of acid rain we redefine $\mathit{take}\,n$ as $(\lambda\, c \,.\, [\![\gamma\,c]\!] \cdot \mathit{start}\,n)\,\mathit{out}$, where

$$\begin{aligned}
&\gamma : (c \to \mathsf{L}\,a\,c) \to (\mathsf{State}\,c \to \mathsf{L}\,a\,(\mathsf{State}\,c))\\
&\gamma\,c\,(0, x)\;\, = \mathit{Nil}\\
&\gamma\,c\,(n, x) = \mathbf{case}\;c\,x\;\mathbf{of}\;\mathit{Nil} \to \mathit{Nil};\,\mathit{Cons}\,(a, y) \to \mathit{Cons}\,(a, (n-1, y))\;\;.
\end{aligned}$$

The transformation γ is derived from $\mathfrak{t}$ by abstracting away from *out*. We can now tackle our example:

$$(\lambda\, c \,.\, [\![\gamma\,c]\!] \cdot \mathit{start}\,5)\,\mathit{out} \cdot [\![\mathfrak{b}]\!] = (\lambda\, c \,.\, [\![\gamma\,c]\!] \cdot \mathit{start}\,5)\,\mathfrak{b} = [\![\gamma\,\mathfrak{b}]\!] \cdot \mathit{start}\,5\;\;.$$

The proof obligation (20) corresponds to the free theorem of

$$\beta : \forall\, C \,.\, (C \to \mathsf{F}\,C) \to (C \to D)\;\;, \tag{22}$$

where D is fixed. And, indeed, $\lambda\, c \,.\, [\![\gamma\, c]\!] \cdot \mathit{start}\, 5$ has the required type.

Similarly, we can dualise our more structured *cata-hylo fusion* to achieve *hylo-ana fusion:* If τ is a transformation that takes recursive F-coalgebras to recursive G-coalgebras satisfying

$$h : \tau\, c \to \tau\, d : \mathbf{Rec}(\mathsf{G}) \quad \Longleftarrow \quad h : c \to d : \mathbf{Rec}(\mathsf{F}) \ , \tag{23}$$

then

$$(\!| a \leftarrow \tau\, c |\!)_{\mathsf{G}} \ = \ (\!| a \leftarrow \tau\, \mathit{out} |\!)_{\mathsf{G}} \cdot [\![c]\!]_{\mathsf{F}} \ . \tag{24}$$

This time the proof obligation (23) cannot be discharged by the type checker alone as τ has to transform a *recursive* coalgebra into a *recursive* coalgebra! As an aside, the new rule cannot handle our running example as the two unfolds are separated by the initialisation function *start*.

Our example has focused on fusing the list parameter of *take*, yet if we admit to the fact that natural numbers are an inductive datatype, then *take* is really a function that consumes *two* data structures. The aforementioned *zip* is another function that consumes two data structures, and therefore has the potential to be fused with both of these inputs. Let us employ the expression $\mathit{zip} \cdot (\mathit{between} \times \mathit{between})$ as another example that can be written in terms of unfolds: $[\![\mathfrak{z}]\!] \cdot ([\![\mathfrak{b}]\!] \times [\![\mathfrak{b}]\!])$. The algebra $\mathfrak{z}$ is given by

$$\begin{array}{l}
\mathfrak{z} : (\mu(\mathsf{L}\, a_1), \mu(\mathsf{L}\, a_2)) \to \mathsf{L}\, (a_1, a_2)\, (\mu(\mathsf{L}\, a_1), \mu(\mathsf{L}\, a_2)) \\
\mathfrak{z}\, (x_1, x_2) = \mathbf{case}\, (\mathit{out}\, x_1, \mathit{out}\, x_2)\, \mathbf{of} \\
\qquad\qquad\qquad (\mathit{Cons}\, (a_1, b_1), \mathit{Cons}\, (a_2, b_2)) \to \mathit{Cons}\, ((a_1, a_2), (b_1, b_2)) \\
\qquad\qquad\qquad \mathit{otherwise} \qquad\qquad\qquad\qquad\quad\;\; \to \mathit{Nil} \ .
\end{array}$$

Our rules (21) and (24) are not applicable as we have *two* producers to the right of *zip*. Now, to fuse such a function, we need to employ *parallel hylo-ana fusion:* If τ satisfies,

$$\begin{array}{l}
h_1 \times h_2 : \tau\, (c_1, c_2) \to \tau\, (d_1, d_2) : \mathbf{Rec}(\mathsf{G}) \\
\qquad\qquad \Longleftarrow \quad h_1 : c_1 \to d_1 : \mathbf{Rec}(\mathsf{F}_1) \ \wedge \ h_2 : c_2 \to d_2 : \mathbf{Rec}(\mathsf{F}_2) \ ,
\end{array} \tag{25}$$

then

$$(\!| a \leftarrow \tau\, (c_1, c_2) |\!)_{\mathsf{G}} \ = \ (\!| a \leftarrow \tau\, (\mathit{out}, \mathit{out}) |\!)_{\mathsf{G}} \cdot ([\![c_1]\!]_{\mathsf{F}_1} \times [\![c_2]\!]_{\mathsf{F}_2}) \ . \tag{26}$$

Using this rule, we are now able to fuse the *zip* example:

$$[\![\zeta\, (\mathit{out}, \mathit{out})]\!] \cdot ([\![\mathfrak{b}]\!] \times [\![\mathfrak{b}]\!]) = [\![\zeta\, (\mathfrak{b}, \mathfrak{b})]\!] \ ,$$

where the transformation ζ is defined

$$\begin{array}{l}
\zeta : (b_1 \to \mathsf{L}\, a_1\, b_1, b_2 \to \mathsf{L}\, a_2\, b_2) \to (b_1, b_2) \to \mathsf{L}\, (a_1, a_2)\, (b_1, b_2) \\
\zeta\, (c_1, c_2)\, (x_1, x_2) = \mathbf{case}\, (c_1\, x_1, c_2\, x_2)\, \mathbf{of} \\
\qquad\qquad\qquad\qquad (\mathit{Cons}\, (a_1, b_1), \mathit{Cons}\, (a_2, b_2)) \to \mathit{Cons}\, ((a_1, a_2), (b_1, b_2)) \\
\qquad\qquad\qquad\qquad \mathit{otherwise} \qquad\qquad\qquad\qquad\quad\;\; \to \mathit{Nil} \ .
\end{array}$$

The proofs of correctness for (parallel) hylo-ana fusion are contained in extended version of this paper.

5.4 Church and Co-Church Encodings

In the two previous sections we have studied generalisations of *foldr*/*build* and *destroy*/*unfoldr* fusion. We have noted that $(\!|-|\!)$ generalises the list function *foldr*, and, likewise, $[\![-]\!]$ generalises *unfoldr*. We have been silent, however, about their counterparts *build* and *destroy*. It is time to break that silence, and in the process, provide a fresh perspective on recursive datatypes. For simplicity, we assume that we are working in **Set**.[4]

Consider again the polymorphic type of β (17) repeated below.

$$\forall A \,.\, (\mathsf{F}\,A \to A) \to (B \to A) \;\cong\; B \to (\forall A \,.\, (\mathsf{F}\,A \to A) \to A)$$

We have slightly massaged the type to bring B to the front. The universally quantified type on the right is known as the *Church encoding* of $\mu\mathsf{F}$ [4]. The type is quite remarkable as it encodes a recursive type without using recursion. One part of the isomorphism $\mu\mathsf{F} \cong \forall A \,.\, (\mathsf{F}\,A \to A) \to A$ is given by the acid rain rule (16). The following derivation, which infers the isomorphisms, makes this explicit—the initial equation is (16) with the arguments of β swapped.

$$\begin{aligned}
& \forall a \,.\, (\!|a|\!)\,(\beta\,b\,in) = \beta\,b\,a \\
\iff & \quad \{\text{ change of variables } \beta\,b = \gamma \;\} \\
& \forall a \,.\, (\!|a|\!)\,(\gamma\,in) = \gamma\,a \\
\iff & \quad \{\text{ extensionality }\} \\
& \lambda\,a \,.\, (\!|a|\!)\,(\gamma\,in) = \gamma \\
\iff & \quad \{\text{ define } \mathit{toChurch}\,x = \lambda\,a \,.\, (\!|a|\!)\,x \;\} \\
& \mathit{toChurch}\,(\gamma\,in) = \gamma \\
\iff & \quad \{\text{ define } \mathit{fromChurch}\,\gamma = \gamma\,in \;\} \\
& \mathit{toChurch}\,(\mathit{fromChurch}\,\gamma) = \gamma
\end{aligned}$$

The isomorphism *toChurch*, creates a function whose argument is an algebra and which folds that algebra over the given data structure. Its converse *fromChurch*, commonly called *build*, applies this function to the *in* algebra. Going back and forth, we get back the original structure: $\mathit{fromChurch}\,(\mathit{toChurch}\,s) = s$. This is the other part of the isomorphism, which follows directly from fold reflection.

As to be expected, everything nicely dualises. The polymorphic type (22) gives rise to the *co-Church encoding*.

$$\forall C \,.\, (C \to \mathsf{F}\,C) \to (C \to D) \;\cong\; (\exists C \,.\, (C \to \mathsf{F}\,C) \times C) \to D$$

Think of the co-Church encoding $\exists C \,.\, (C \to \mathsf{F}\,C) \times C$ as the type of state machines encapsulating a transition function $C \to \mathsf{F}\,C$ and the current state C.

The conversion to (co-)Church-encoding types are central to the concept of shortcut fusion. By changing representations to one with the recursion "built-in", we can write our transformations as non-recursively-defined (co-)algebras. Unlike recursive programs, compositions of these (co-)algebras can be optimised

[4] The development can be generalised using ends and coends [14].

by the compiler to remove any intermediate allocations. All that remains is for the programmer to instruct the compiler to remove any unnecessary conversions, i.e. cases of $\mathit{toChurch} \cdot \mathit{fromChurch}$. Removing these transformations preserves the semantics of the program because we can prove the isomorphism between these representations. More importantly, however, prevents us from producing a data structure only to immediately consume it. The co-Church encoding also underlies the original formulation of stream fusion, which we consider next.

5.5 Stream Fusion

The *foldr*/*build* flavour of fusion is fold-centric, in that it requires all functions that are intended to be fusible to be written as folds; similarly, *destroy*/*unfoldr* is unfold-centric. The boundaries of these world views are fuzzy. A *zip* can be written as a fold, the snag is that only one of the two inputs can be fused [9, §9]. Along a similar vein, a *filter* for the odd natural numbers, which we wrote before as a fold, can also be written as an unfold: $[\![\mathfrak{f}]\!]$ where

$$\mathfrak{f} : \mu(\mathsf{L}\,\mathbb{N}) \to \mathsf{L}\,\mathbb{N}\,(\mu(\mathsf{L}\,\mathbb{N}))$$
$$\mathfrak{f}\,x = \mathbf{case}\ \mathit{out}\,x\ \mathbf{of}\ \mathit{Nil} \to \mathit{Nil};\ \mathit{Cons}\,(x,y) \to \mathbf{if}\ \mathit{odd}\,x\ \mathbf{then}\ \mathit{Cons}\,(x,y)\ \mathbf{else}\ \mathfrak{f}\,y \ .$$

The coalgebra $\mathfrak{f}$ is recursive and thus theoretically fine, but it is also recursive in its definition and this is a practical problem. A coalgebra must be non-recursively defined for it to be fused with others. We have two definitions and are caught between two worlds; is it possible to free ourselves?

Perhaps surprisingly, the answer is yes. Let us first try to eliminate the recursion from the definition above—the rest will then fall out. The idea is to use a different base functor, one that allows us to skip list elements. We draw inspiration from stream fusion [5] here:

$$\mathbf{data}\,\mathsf{S}\ a\ b = \mathit{Done} \mid \mathit{Yield}\,(a,b) \mid \mathit{Skip}\ b \ .$$

$$\begin{array}{l} \mathbf{instance}\ \mathit{Functor}\,(\mathsf{S}\ a)\ \mathbf{where} \\ \quad \begin{array}{lcl} \mathit{fmap}\,f\ \mathit{Done} & = & \mathit{Done} \\ \mathit{fmap}\,f\ (\mathit{Skip}\ b) & = & \mathit{Skip}\,(f\ b) \\ \mathit{fmap}\,f\ (\mathit{Yield}\,(a,b)) & = & \mathit{Yield}\,(a, f\ b) \end{array} \end{array}$$

The *filter* coalgebra can now be written as a composition of *out* with

$$\begin{array}{lcl} \mathfrak{f} : \mathsf{S}\,\mathbb{N}\,b \to \mathsf{S}\,\mathbb{N}\,b & & \\ \mathfrak{f}\,\mathit{Done} & = & \mathit{Done} \\ \mathfrak{f}\,(\mathit{Skip}\ y) & = & \mathit{Skip}\ y \\ \mathfrak{f}\,(\mathit{Yield}\,(x,y)) & = & \mathbf{if}\ \mathit{odd}\,x\ \mathbf{then}\ \mathit{Yield}\,(x,y)\ \mathbf{else}\ \mathit{Skip}\ y \ . \end{array}$$

So, $\mathit{filter} = [\![\mathfrak{f} \cdot \mathit{out}]\!]$. Something interesting has happened: since $\mathfrak{f}$ is a natural transformation, we also have $\mathit{filter} = (\!|\mathit{in} \cdot \mathfrak{f}|\!)$. We are unstuck; *filter* is both a fold and an unfold. Moreover, it is an application of a mapping function: $\mathit{filter} = \mu\mathfrak{f}$.

In general, consumers are folds, transformers are maps, and producers are unfolds. An entire pipeline of these an be fused into a single hylo:

$$(\!|a|\!) \cdot \mu\alpha_1 \cdot \cdots \cdot \mu\alpha_n \cdot [\![c]\!] = (\!|a \cdot \alpha_1 \cdot \cdots \cdot \alpha_n \leftarrow c|\!) \ .$$

Inspecting the types, the rule is clear:

$$A \xleftarrow{(\!|a|\!)} \mu\mathsf{F}_0 \xleftarrow{\mu\alpha_1} \mu\mathsf{F}_1 \quad \cdots \quad \mu\mathsf{F}_{n-1} \xleftarrow{\mu\alpha_n} \mu\mathsf{F}_n \xleftarrow{[\![c]\!]} C \ .$$

In a sense, the introduction of *Skip* keeps the recursion in sync. Each transformation consumes a token and produces a token. Before, *filter* possibly consumed several tokens before producing one. We are finally in a position to deal with the example from the introduction, written in terms of the combinators we have

$$(\!|\mathfrak{s}|\!) \cdot \mu(\mathfrak{m}\, sq) \cdot \mu(\mathfrak{f}\, odd) \cdot [\![\mathfrak{b}]\!] = (\!|\mathfrak{s} \cdot \mathfrak{m}\, sq \cdot \mathfrak{f}\, odd \leftarrow \mathfrak{b}|\!) \ .$$

Utilising streams in this fashion is an instance of data abstraction; although we wish to present the List type using $\mu(\mathsf{L}\,a)$, we intend to do all the work using $\mu(\mathsf{S}\,a)$. We have functions $\rightharpoondown\mathsf{S}$ and $\leftharpoondown\mathsf{S}$ to convert to and from streams, respectively. They are defined as an algebra and a coalgebra that allow us to consume streams using a fold and produce them using an unfold:

$$\begin{aligned}
&\leftharpoondown\mathsf{S} : \mathsf{S}\,a\,(\mu(\mathsf{L}\,a)) \to (\mu(\mathsf{L}\,a)) \\
&\leftharpoondown\mathsf{S}\ \mathit{Done} &&= \mathit{in}\ \mathit{Nil} \\
&\leftharpoondown\mathsf{S}\ (\mathit{Skip}\ xs) &&= xs \\
&\leftharpoondown\mathsf{S}\ (\mathit{Yield}\ (x, xs)) &&= \mathit{in}\ (\mathit{Cons}\ (x, xs)) \\
&\rightharpoondown\mathsf{S} : \mu(\mathsf{L}\,a) \to \mathsf{S}\,a\,(\mu(\mathsf{L}\,a)) \\
&\rightharpoondown\mathsf{S}\ (\mathit{in}\ \mathit{Nil}) &&= \mathit{Done} \\
&\rightharpoondown\mathsf{S}\ (\mathit{in}\ (\mathit{Cons}\ (x, xs))) &&= \mathit{Yield}\ (x, xs) \ .
\end{aligned}$$

We must prove that our stream implementations, together with the conversion functions, fulfil the same specification as the analogous functions over $\mu(\mathsf{L}\,a)$ (*cf.* Lemma 1 and Theorem 3 in [23]). This is called the *data abstraction property*. In our framework, this obligation is expressed as a simple equality between a conventional list function definition and its associated stream version composed with our conversion functions. For example, for *filter* we must prove

$$\mathit{filter} = (\!|\leftharpoondown\mathsf{S}|\!) \cdot \mu\mathfrak{f} \cdot [\![\rightharpoondown\mathsf{S}]\!] \ ,$$

Because we can phrase these functions as folds, unfolds, and natural transformations, the proof is straightforward, using the laws we have set out in previous sections. We leave it as an exercise to the reader.

Just as for lists, every datatype can be extended with a *Skip*. Although stream fusion is the first to make use of this augmentation, we note its relation to Capretta's representation of general recursion in type theory [1], which proposes adding a "computation step" constructor to coinductive types.

6 Related Work

Wadler first introduced the idea of simplifying the fusion problem with his deforestation algorithm [22]. This was limited to so-called *treeless* programs, a subset of first-order programs. The fusion transformation proposed by Chin [3] generalises Wadler's deforestation. It uses a program annotation scheme to recognise the terms that can be fused and skip the terms that cannot. Sheard and Fegaras focus on the use of folds over algebraic types as a recursion scheme [17]. Their algorithm for normalising the nested application of folds is based on the fold fusion law. Their recursion schemes are suitably general to handle functions such as *zip* that recurse over multiple data structures simultaneously [6].

Gill et al. first introduced the notion of shortcut fusion with *foldr/build* fusion [9] for Haskell. This allowed programs written as folds to be fused. It was subsequently introduced into the List library for Haskell in GHC. Takano and Meijer [19] provided a calculational view of fusion and generalised it to arbitrary data structures. It generalised the fusion law by using hylomorphisms and also noted the possibility of dualising *foldr/build* fusion. They worked in the setting of **Cpo**, however, where hylomorphisms do not have unique solutions, only canonical ones. Takano and Meijer claimed that, even when restricted to lists, their method is more powerful than that of Gill et al. as theirs could fuse both parameters of *zip*. This was incorrect, and the need for an additional parallel rule for *zip* was pointed out later by Hu et al. [11]. Their extension is what we present as the parallel hylo-ana rule.

Svenningson provided an actual implementation of *destroy/unfoldr* fusion [18], where he showed how *filter*-like functions could be expressed as unfolds. Svenningson did not, however, solve the issue of recursion in the coalgebras of such functions, which could therefore not be fused even though they could be written as unfolds. This was addressed by Coutts et al., who presented stream fusion [5], which introduced the *Skip* constructor as a way to encode non-productive computation steps, similar to Capretta's work on encoding general recursion in type theory [1].

The correctness and generalisation of fusion has been explored in many different settings. In addition to the work of Takano and Meier, Ghani et al. generalised *foldr/build* to work with datatypes "induced by inductive monads". Johann and Ghani further showed how to apply initial algebra semantics, and thus *foldr/build* fusion, to nested datatypes [12]. Voigtländer has also used free theorems to show correctness, specifically of the *destroy/build* rule [20].

7 Conclusions

We have presented a framework that has allowed us to bring three fusion techniques into the same setting. We have exploited recursive coalgebras and hylomorphisms as 'the rug that ties the room together'. This enabled us to formally describe and reason about these fusion techniques. In doing so, we have exposed their underlying foundations, including the importance of Church and co-Church encodings. The fact that our hylomorphisms have unique solutions plays a central rôle. The knock-on effect is that we gain clear, short proofs thanks to the calculational properties available to us.

References

1. Capretta, V.: General recursion via coinductive types. Logical Methods in Computer Science 1(2), 1–28 (2005)
2. Capretta, V., Uustalu, T., Vene, V.: Recursive coalgebras from comonads. Information and Computation 204(4), 437–468 (2006)
3. Chin, W.N.: Safe Fusion of Functional Expressions. In: LISP and functional programming. pp. 11–20 (1992)
4. Church, A.: The calculi of lambda-conversion. Annals of Mathematics Studies No. 6, Princeton University Press (1941)
5. Coutts, D., Leshchinskiy, R., Stewart, D.: Stream Fusion: From Lists to Streams to Nothing At All. In: ICFP '07. pp. 315–326 (2007)
6. Fegaras, L., Sheard, T., Zhou, T.: Improving Programs which Recurse over Multiple Inductive Structures. In: PEPM'94 (June 1994)
7. Fokkinga, M.M., Meijer, E.: Program calculation properties of continuous algebras. Technical Report CS-R9104, CWI, Amsterdam (Jan 1991)
8. Freyd, P.J.: Remarks on algebraically compact categories. In: Fourman, M.P., Johnstone, P.T., Pitts, A.M. (eds.) Applications of Categories in Computer Science, LMS Lecture Note Series, vol. 177, pp. 95–106. Cambridge University Press (1992)
9. Gill, A., Launchbury, J., Peyton Jones, S.L.: A Short Cut to Deforestation. In: Functional programming languages and computer architecture. pp. 223–232 (1993)
10. Hinze, R., Harper, T., James, D.W.H.: Theory and Practice of Fusion. Tech. Rep. CS-RR-11-01, Oxford University Computing Laboratory (2011)
11. Hu, Z., Iwasaki, H., Takeichi, M.: An Extension of The Acid Rain Theorem. In: Functional and Logic Programming. pp. 91–105 (1996)
12. Johann, P., Ghani, N.: Initial algebra semantics is enough! In: Typed Lambda Calculi and Applications. LNCS, vol. 4583, pp. 207–222 (2007)
13. Lambek, J.: A fixpoint theorem for complete categories. Math. Zeitschr. 103, 151–161 (1968)
14. Mac Lane, S.: Categories for the Working Mathematician. Graduate Texts in Mathematics, Springer-Verlag, Berlin, 2nd edn. (1998)
15. Meijer, E., Fokkinga, M., Paterson, R.: Functional programming with bananas, lenses, envelopes and barbed wire. In: FPCA. LNCS, vol. 523, pp. 124–144 (1991)
16. Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: Mason, R.E.A. (ed.) Information Processing '83, pp. 513–523. North-Holland, Amsterdam (1983)
17. Sheard, T., Fegaras, L.: A Fold for All Seasons. In: Functional programming languages and computer architecture. pp. 233–242 (1993)
18. Svenningsson, J.: Shortcut fusion for Accumulating Parameters & Zip-like Functions. In: ICFP '02. pp. 124–132 (2002)
19. Takano, A., Meijer, E.: Shortcut deforestation in calculational form. In: Functional programming languages and computer architecture. pp. 306–313 (1995)
20. Voigtländer, J.: Proving correctness via free theorems: the case of the destroy/-build-rule. In: Partial Eval. and Semantics-Based Prog. Manip. pp. 13–20 (2008)
21. Wadler, P.: Theorems for free! In: FPCA. pp. 347–359 (1989)
22. Wadler, P.: Deforestation: transforming programs to eliminate trees. Theoretical Computer Science 73(2), 231 – 248 (1990)
23. Wang, M., Gibbons, J., Matsuda, K., Hu, Z.: Gradual refinement: Blending pattern matching with data abstraction. In: Bolduc, C., Desharnais, J., Ktari, B. (eds.) Mathematics of Program Construction. LNCS, vol. 6120, pp. 397–426 (2010)