

A Library Writer's Guide to Shortcut Fusion

Thomas Harper

Department of Computer Science, University of Oxford
tom.harper@cs.ox.ac.uk

Abstract

There are now a variety of shortcut fusion techniques in the wild for removing intermediate data structures in Haskell. They are often presented, however, specialised to a specific data structure and interface. This can make it difficult to transfer these techniques to other settings. In this paper, we give a roadmap for a library writer who would like to implement fusion for his own library. We explain shortcut fusion without reference to any specific implementation by treating it as an instance of data refinement. We also provide an example application of our framework using the features available in the Glasgow Haskell Compiler.

Categories and Subject Descriptors D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.3.4 [Programming Languages]: Optimisation

General Terms Languages, Algorithms

Keywords Deforestation, optimisation, program transformation, program fusion, functional programming, shortcut fusion

1. Introduction

When writing a library, a programmer often seeks to get the best performance possible out of his data structures and interface. However, even if the data structure is well-designed and the interface functions carefully tuned, it is still up to the compiler to optimise programs written by users of the library. In Haskell, programmers can compose simple functions to create a complex “pipeline” that transforms a data structure. For example, the program

$$f :: (Int, Int) \to Int$$
$$f = sum \circ map (+1) \circ filter odd \circ between$$

forms a pipeline of functions over a list. It starts with the function *between*, which generates an enumeration between two numbers as a list, *filters* out any even numbers, and then increments the remaining numbers before summing them together. This method allows us to write powerful programs over a data structure functions in a concise, modular way.

Because these are recursively defined functions over a recursive datatype, such a program produces *intermediate data structures*. Each function consumes a structure and produces a new one to pass on to the next function in the pipeline. Such structures glue the components together, but do not appear in the final result. They

nevertheless take up space in memory and ultimately affect the performance of the program.

It is possible, however, to rewrite such a program so that it uses a single loop:

$$f' :: (Int, Int) \to Int$$
$$f' (x, y) = loop\ x$$

where

$$loop\ x \mid x > y = 0$$
$$\mid otherwise = \mathbf{if}\ odd\ x$$
$$\mathbf{then}\ (x + 1) + loop\ (x + 1)$$
$$\mathbf{else}\ loop\ (x + 1)$$

This program is equivalent to the first, but produces no intermediate data structures. Transforming a program in this manner is known as *fusion*. Obviously, performing fusion by hand requires a programmer to know about the details of the functions and data structure, an unrealistic exception for the user of a library. It would be preferable if, instead, the compiler could perform this transformation for us.

Shortcut fusion allows a programmer to implement mechanised fusion for a specific data structure and interface. This is a term that originally referred to *foldr/build* [8] fusion, but has come to encompass other incarnations that take a similar approach. The programmer chooses a datatype and a recursive scheme over it. The chosen recursion pattern is encapsulated in combinators that signify the consumption and production of a data structure. Program transformations can then be phrased as simple syntactic rewrites that remove instances of the production combinator followed immediately by the consumption combinator. Any remaining work can be finished by compiler using its usual complement of optimisation techniques. This allows the programmer to implement fusion without touching compiler or worrying about the impact of program transformations on unrelated parts of the program. Instead, the programmer has the responsibility of doing the specialisation himself.

Although many have investigated shortcut fusion in a datatype-generic way, the presentation is often highly theoretical. On the other hand, more practical papers on the subject present shortcut fusion in the context of a specific data structure. This can lead to the view that many of the innovations are unique to a particular implementation, or at the least it can obscure the principles used to arrive at that implementation. We intend to join these two paths by giving a description of shortcut fusion that can be used by the aspiring library writer to implement fusion for his own library. We do this by treating shortcut fusion as an instance of data abstraction, which allows us to describe it without reference to a specific datatype, but provide a framework that can be instantiated the library writer straightforwardly.

The main contributions of this paper are as follows:

- A description of shortcut fusion as an instance of data abstraction (Section 3), which provides a concise description of the theory of shortcut fusion that can also be instantiated for a specific datatype.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Haskell'11, September 22, 2011, Tokyo, Japan.

Copyright © 2011 ACM 978-1-4503-0860-1/11/09...\$10.00

- A description of Church encodings and Cochurch encodings and a demonstration of how they can be used to implement shortcut fusion (Section 4).
- An example implementation using Church and Cochurch encodings to fuse an interface over leaf trees, providing a roadmap for the aspiring library author that demonstrates how to use the infrastructure in the Glasgow Haskell Compiler (GHC) to implement shortcut fusion (Section 5) and accompanying benchmarks of the interface (Section 6).

Additionally, Section 2 reviews some prerequisites for discussing Church and Cochurch encodings: initial algebras and folds, final coalgebras and unfolds, and natural transformations. We summarise related work in Section 7 and conclude in Section 8.

2. Background: Initial algebras and final coalgebras

2.1 Initial algebras and folds

Folds and unfolds provide a common pattern for defining functions over recursively defined datatypes. Category theory provides a setting for reasoning about folds and unfolds using initial algebras and final coalgebras. In this section, we will briefly review these concepts. In this description, we assume a basic knowledge of categories and functors.

For a category \mathbb{C} and a functor $F : \mathbb{C} \rightarrow \mathbb{C}$, an *algebra* is a pair (A, a) , where A is an object in \mathbb{C} and a is an arrow of type $F A \rightarrow A$. If we have another algebra (B, b) , then an *algebra homomorphism* is a function $h : A \rightarrow B$ such that the following diagram commutes:

$$\begin{array}{ccc} F A & \xrightarrow{F h} & F B \\ a \downarrow & & \downarrow b \\ A & \xrightarrow{h} & B \end{array}$$

Together, algebras and algebra homomorphisms form a category. The *initial algebra* is the initial object in such a category. The initial algebra is denoted $(\mu F, in)$ and has the property that there is a *unique* arrow from it to any algebra. For an algebra (A, a) , this arrow is called *fold f*, denoted (f) , which makes the following diagram commute:

$$\begin{array}{ccc} F(\mu F) & \xrightarrow{F(a)} & F A \\ in \downarrow & & \downarrow a \\ \mu F & \xrightarrow{(a)} & A \end{array}$$

The initiality of μF , and therefore uniqueness of (a) , is captured by the *universal property of folds*

$$h = (a) \iff h \circ in = a \circ F h \quad (1)$$

Initial algebras provide a semantics for consuming a recursive data structure. Consider the functor $L A B = 1 + A \times B$. The functor $L A$ is the *base functor* for lists with elements of type A . In Haskell, we can define this as

```
data List_ a b = Nil_ | Cons_ a b
instance Functor (List_ a) where
  fmap f Nil_      = Nil_
  fmap f (Cons_ a b) = Cons_ a (f b)
```

This corresponds to a datatype declaration for a list but with the recursive call abstracted away into an additional type parameter. (Note that we declare a datatype with an underscore to represent the base functor). The sum type is expressed as a type with a constructor for each summand, and a product type corresponds to a constructor with one field per operand. The initial algebra for this functor is the type $List\ a$, which is obtain by passing a recursive call as the argument to the functor:

```
data List a = Cons a (List a) | Nil
```

The corresponding function, *in*

```
in :: List_ a (List a) -> List a
in Nil_      = Nil
in (Cons_ x xs) = Cons x xs
```

specifies how to construct a $List$ either from the final object $Nil_$, or from a pair consisting of an element and an already-constructed $List$. The universal property provides us with a definition for *fold*; if we apply the left hand side to the right, we obtain the *computation law*

$$(a) \circ in = a \circ F (a) \quad (2)$$

which states that we place the recursive call over the tail of the list (determined by our definition of *fmap*), and then combine the results of this call with the head of the list using *a*:

```
fold :: (List_ a b -> b) -> List a -> b
fold a Nil_      = (a \circ fmap (fold a)) Nil_
fold a (Cons_ x xs) = (a \circ fmap (fold a)) (Cons_ x xs)
```

Thus, the base functor determines the shape of a data structure as well as the recursion pattern over it. This pattern gives us a way to combine elements of a data structure using an algebra, which is just a function of type $List_ a b \rightarrow b$, where a is the type of the elements of the list and b is the type of the result of the fold. While this definition illustrates the relationship between the base functor and the recursive pattern, we will hence use a more Haskellish style of explicitly placing the recursive call:

```
fold' :: (List_ a b -> b) -> List a -> b
fold' a Nil_      = a Nil_
fold' a (Cons_ x xs) = a (Cons_ x (fold' a xs))
```

The algebra describes a single step, while *fold* takes care of recursively applying it. If we fold the initial algebra over some data structure, we get the same structure back, as we are just swapping data constructors for themselves. This is called the *reflection law*

$$(in) = id \quad (3)$$

Finally, folds also come with a *fusion law*. As its name hints, it lets us combine a function with a fold, in this case “absorbing” a function that appears after a fold under certain conditions:

$$h \circ (a) = (b) \iff h \circ a = b \circ Fh \quad (4)$$

The precondition requires that h is an algebra homomorphism from a to b .

2.2 Final coalgebras and unfolds

Dual to initial algebras are *final coalgebras*, which give us a semantics for producing a data structure. For a functor $F : \mathbb{C} \rightarrow \mathbb{C}$, a coalgebra is a pair (c, C) consisting of an object C in \mathbb{C} , along with an arrow $c : C \rightarrow F C$. Given another coalgebra (d, D) , a *coalgebra homomorphism* is a function $h : C \rightarrow D$ such that the following diagram commutes:

$$\begin{array}{ccc}
C & \xrightarrow{h} & D \\
c \downarrow & & \downarrow d \\
F C & \xrightarrow{F h} & F D
\end{array}$$

Coalgebras and coalgebra homomorphisms also form a category. The *final coalgebra* is the final object in such a category. The final coalgebra of a functor F is denoted $(out, \nu F)$. As the final object, there exists a unique arrow from every coalgebra to it. For a coalgebra (c, C) , this arrow is called *unfold* c , denoted $[c]$, and makes the following diagram commute:

$$\begin{array}{ccc}
C & \xrightarrow{[c]} & \nu F \\
c \downarrow & & \downarrow out \\
F C & \xrightarrow{F [c]} & F (\nu F)
\end{array}$$

The finality of $(out, \nu F)$ is captured in the *universal property of unfolds*

$$h = [c] \iff out \circ h = F h \circ c \quad (5)$$

This provides us with another recursion scheme for recursive data types. We can continue our example using $List_a$ as a base functor; the carrier of the final coalgebra for this functor is also $List_a$. This is because, in Haskell, datatypes represented by μF and νF coincide, allowing us to phrase functions as both folds and unfolds over the same datatype. This property is known as *algebraic compactness* [4].

From the universal property of unfolds, we also get a computation law providing us with a definition for unfolds:

$$out \circ [c] = F [c] \circ c \quad (6)$$

An unfold applies a coalgebra to a seed, which produces an element of the data structure along with an F-shaped collection of successive seeds. These are then recursively unfolded to yield a data structure. For $List_a$, this has the definition

$$\begin{aligned}
unfold &:: (s \rightarrow List_a\ s) \rightarrow s \rightarrow List\ a \\
unfold\ c\ s &= \mathbf{case}\ c\ s\ \mathbf{of} \\
Nil_ &\rightarrow Nil \\
Cons_x\ s' &\rightarrow Cons\ x\ (unfold\ c\ s')
\end{aligned}$$

Here, we have already placed the recursive call explicitly. We then collect the results in a $List\ a$, thereby producing a data structure.

Unlike *in*, *out* describes how to deconstruct a data structure:

$$\begin{aligned}
out &:: List\ a \rightarrow List_a\ (List\ a) \\
out\ Nil &= Nil_ \\
out\ (Cons\ x\ xs) &= Cons_x\ xs
\end{aligned}$$

In the case of a list, *out* returns the head and tail as a pair. If we use a data structure as a seed and *out* as the coalgebra, we will just recursively split up a data structure and then recombine the results, meaning we get the original data structure back. Therefore, the reflection law for data unfolds is

$$[out] = id \quad (7)$$

There is also a fusion law for unfolds. It allows us to absorb a function to the right of an unfold, this time:

$$[c] = [d] \circ h \iff F h \circ c = d \circ h \quad (8)$$

In this case, h must be a coalgebra homomorphism from c to d .

Although we have explained folds and unfolds by modelling a specific datatype as an initial algebra and final coalgebra, the

power of these recursion schemes is that we can reason about them *without* considering the specifics of the data structure in question. The universal properties provide datatype-generic definitions of folds and unfolds and properties of them. We will use this to our advantage later by showing how folds and unfolds can be used in shortcut fusion and thereby giving a roadmap for doing so for any datatype.

2.3 Natural transformations

Natural transformations provide a way for mapping between functors. For functors $F, G : \mathbb{C} \rightarrow \mathbb{D}$, a natural transformation f is a collection of arrows, one for each object in \mathbb{C} , such that $f_A : F A \rightarrow G A$. Furthermore, for all arrows $h : A \rightarrow B$, a natural transformation possesses a *coherence property*, which states that

$$G h \circ f_A = f_B \circ F h \quad (9)$$

This guarantees that f respects the structure of the functors while mapping objects between them. The notion of natural transformations can be used to provide a semantics for functions that modify the elements of a recursive datatype but not the structure. For example, a *map* over lists can be viewed as $(in \circ (m f))$, where m takes a function $f : a \rightarrow b$ and $m f$ is a natural transformation that defines the transformation of a single element from $List_a$ to $List_b$, which are two different functors:

$$\begin{aligned}
m &:: (a \rightarrow b) \rightarrow List_a\ c \rightarrow List_b\ c \\
m\ f\ Nil_ &= Nil_ \\
m\ f\ (Cons_x\ xs) &= Cons_ (f\ x)\ xs
\end{aligned}$$

After this, *in* constructs a list from the transformed element and the tail xs , which is a list of already transformed elements.

3. Shortcut fusion as data abstraction

We use the term *fusion* to describe a program transformation that transforms the composition of a series of functions

$$f = f_n \circ \dots \circ f_1$$

defined over a recursive datatype into a single recursive pass f' such that $f = f'$. The purpose of this transformation is to remove the intermediate data structures that are passed from f_i to f_{i+1} . Removing the intermediate data structures from such a program poses some challenges for the compiler's usual arsenal of optimisations because they give up when they encounter recursive definitions.

The shortcut fusion solution to this problem is to convert values of recursive datatypes to a different representation that the optimiser can deal with. For example, assume that we want to convert values of the recursive datatype μF to values of a type C . The idea is that C can faithfully represent values of μF , but composed functions over C can be fused automatically. Instead of writing functions directly over μF , we define them in terms of functions over C along with conversion functions that convert between μF and C . We call these conversion functions $con : \mu F \rightarrow C$ and $abs : C \rightarrow \mu F$.

In data abstraction terms [12], μF is the *abstract* datatype over which the interface is defined, and C is the *concrete* datatype over which the interface is implemented. In order for C to be a faithful representation of μF , con and abs must have the property

$$abs \circ con = id_{\mu F} \quad (10)$$

This requires that C be capable of representing all values of μF uniquely.

A fusible function usually has a counterpart that can be written directly over the abstract datatype. This makes a fusible function a concrete refinement of the abstract function. Therefore, we want to ensure that this refinement actually implements the original

function. In order to define a fusible version of a function $f : \mu F \rightarrow \mu F$ in terms of C , we must define a function $f_C : C \rightarrow C$ such that the following diagram commutes:

$$\begin{array}{ccc} \mu F & \xleftarrow{\text{abs}} & C \\ f \downarrow & & \downarrow f_C \\ \mu F & \xleftarrow{\text{abs}} & C \end{array}$$

If we have implemented such a function, then we can define a function over the abstract type $f' = \text{abs} \circ f_C \circ \text{con}$. In isolation, this definition does not necessarily gain us anything. Indeed, the cost of converting between μF and C may even make this function *less* efficient than one that is defined directly over μF . Suppose, however, that we have functions $g : \mu F \rightarrow \mu F$ and $g_C : C \rightarrow C$ defined similarly and compose f and g :

$$\begin{array}{ccc} \mu F & \xrightarrow{\text{con}} & C \\ f \downarrow & & \downarrow f_C \\ \mu F & \xleftrightarrow[\text{con}]{\text{abs}} & C \\ g \downarrow & & \downarrow g_C \\ \mu F & \xleftrightarrow[\text{con}]{\text{abs}} & C \\ \mu F & \xleftarrow{\text{abs}} & C \end{array}$$

Down the left hand side of this diagram, we see the program as written over μF , in which f communicates with g using intermediate data structures. If this diagram commutes, however, we can move to the right hand side using con and chase the arrows down to the bottom, converting the result back using abs , thereby obtaining the fusible program

$$\text{abs} \circ g_C \circ f_C \circ \text{con}$$

Such a program is only correct, however, if we are able to discard the conversion $\text{con} \circ \text{abs}$ that arises. The simplest situation is the one in which $\text{con} \circ \text{abs} = \text{id}$, which means that we can remove it unconditionally. This is an extremely strong condition, demanding that not only does C faithfully represent values of μF , but also vice versa i.e. that $\mu F \cong C$. Instead, we can prove that if f_C and g_C are correct implementations of f and g i.e. that they satisfy the property stated above, then the fusible program preserves the meaning of the original:

$$\begin{aligned} & \text{abs} \circ f_C \circ g_C \circ \text{con} \\ = & \{ f \circ \text{abs} = \text{abs} \circ f_C \} \\ & f \circ \text{abs} \circ g_C \circ \text{con} \\ = & \{ g \circ \text{abs} = \text{abs} \circ g_C \} \\ & f \circ g \circ \text{abs} \circ \text{con} \\ = & \{ \text{abs} \circ \text{con} = \text{id}_{\mu F} \} \\ & f \circ g \end{aligned}$$

In addition to transformation functions that, like those above, consume a data structure to produce a new one, we can also fuse those that produce a recursive data structure from some seed and those that consume a recursive data structure to produce a value. As for transformations, a producer $p : S \rightarrow \mu F$ and a consumer $c : \mu F \rightarrow T$ will have fusible counterparts $p_C : S \rightarrow C$ and $c_C : C \rightarrow T$, respectively, such that

$$\begin{array}{ccc} S & & \\ p \downarrow & \searrow p_C & \\ \mu F & \xleftarrow{\text{abs}} & C \end{array} \quad \text{and} \quad \begin{array}{ccc} \mu F & \xrightarrow{\text{con}} & C \\ c \downarrow & \swarrow c_C & \\ T & & \end{array}$$

commute. This gives us the obligations $p = \text{abs} \circ p_C$ and $c = \text{con} \circ c_C$. Like the property for transformations, these conditions are sufficient to allow us to remove the unnecessary conversions. To tie it all together, consider a pipeline $c \circ f \circ g \circ p$, whose functions have types as above and meet the necessary obligations for their fusible counterparts. Diagrammatically, we can represent this pipeline as

$$\begin{array}{ccc} S & & \\ p \downarrow & \searrow p_C & \\ \mu F & \xleftrightarrow[\text{con}]{\text{abs}} & C \\ f \downarrow & & \downarrow f_C \\ \mu F & \xleftrightarrow[\text{con}]{\text{abs}} & C \\ g \downarrow & & \downarrow g_C \\ \mu F & \xleftrightarrow[\text{con}]{\text{abs}} & C \\ c \downarrow & \swarrow c_C & \\ T & & \end{array}$$

We can see that, by removing the $\text{con} \circ \text{abs}$ conversions, we can obtain a pipeline in which we no longer depend on the unfusable functions over a recursive datatype. Instead, the program $c_C \circ f_C \circ g_C \circ p_C$, when optimised by the compiler, will take a seed of type S and use it to produce some data that it transforms and consumes without producing any intermediate data structures in order to produce a single value, like the second example from the introduction.

So far, we have provided a generic setup for implementing shortcut fusion. We have established the central idea of representation change and how to use this to obtain fusible programs. We have been silent, however, about the sort of representations that can faithfully represent recursive datatypes, but nevertheless allow the compiler to fuse their functions automatically. We cover such representations and how they can fit into the above framework in the next section.

4. Concrete representations

As mentioned before, the issue at hand is that the compiler cannot fuse pipelines composed of recursive functions. It will, however, inline non-recursive functions and remove intermediate data structures from the resulting program. For shortcut fusion, we therefore need a type that faithfully represents a recursive datatype but, paradoxically, allows us to write functions with non-recursive definitions over it. In this section, we will describe two classes of such representations. These are the *Church encodings* and *Cochurch encodings* of recursive datatypes.

Church and Cochurch encodings are closely related to the concept of folds and unfolds, respectively. The Church encoding of a data structure represents it as a higher-order function that takes an algebra and returns the fold of that algebra over the data structure. For an endofunctor $F : \mathbb{C} \rightarrow \mathbb{C}$, the Church encoding is of type

$$\text{data Church } F = \text{Ch } (\forall A . (F A \rightarrow A) \rightarrow A)$$

Note the rank-2 polymorphic type; the parametricity of this function guarantees that it obtains the final value of type A by applying the algebra to elements of the underlying data structure.

Church encodings allow us to represent a recursive datatype such that the recursion is “built-in” i.e. we do not to specify the recursive call in our definitions, only a single step in the form of an algebra that has a non-recursive definition. A pipeline of such functions can therefore be fused by the usual complement of compiler optimisations.

To use this as a representation, we instantiate conversion functions $toCh$ and $fromCh$ for con and abs . We convert a data structure of type μF to its Church encoding by defining a function that takes an algebra and folds that algebra over the data structure:

$$\begin{aligned} toCh &:: \mu F \rightarrow Church\ F \\ toCh\ x &= Ch\ (\lambda a \rightarrow (a)\ x) \end{aligned}$$

To get the data structure back, we simply apply the Church encoding to the initial algebra $in : F\ (\mu F) \rightarrow \mu F$

$$\begin{aligned} fromCh &:: Church\ F \rightarrow \mu F \\ fromCh\ (Ch\ g) &= g\ in \end{aligned}$$

which defines how to construct the datatype. To prove that Church encodings faithfully represent their underlying datatype, we are required to prove $fromCh \circ toCh = id$. This is simply a consequence of the universal properties of folds:

$$\begin{aligned} &fromCh\ (toCh\ x) \\ &= \{ \text{definition of } toCh \} \\ &fromCh\ (Ch\ (\lambda a \rightarrow (a)\ x)) \\ &= \{ \text{definition of } fromCh \} \\ &(\lambda a \rightarrow (a)\ x)\ in \\ &= \{ \text{function application} \} \\ &(in)\ x \\ &= \{ \text{fold reflection law (3)} \} \\ &x \end{aligned}$$

We can also prove the other direction, however, because Church encodings are isomorphic to their underlying datatypes. We have just proved one direction of this isomorphism, but the other direction, $toCh \circ fromCh = id$, requires a different tactic:

$$\begin{aligned} &toCh\ (fromCh\ (Ch\ g)) = Ch\ g \\ \iff &\{ \text{definition of } fromCh \} \\ &toCh\ (g\ in) = (Ch\ g) \\ \iff &\{ \text{definition of } toCh \} \\ &Ch\ (\lambda a \rightarrow (a)\ (g\ in)) = (Ch\ g) \\ \iff &\{ \text{extensionality} \} \\ &\forall a . (a)\ (g\ in) = g\ a \end{aligned}$$

To satisfy the final condition, we must prove that constructing a recursive datatype from its Church encoding and then folding an algebra over it is the same as folding the algebra over the Church encoding itself. The proof of this rests on the free theorem [20] of the Church encoded datatype g , which is a function of type $\forall A . (F\ A \rightarrow A) \rightarrow A$:

$$h \circ b = c \circ F\ h \implies h\ (g\ b) = g\ c \quad (11)$$

The precondition requires that h be an algebra homomorphism from b to c (here, we omit the associated carriers of these algebras). If it is, applying g to b i.e. folding b over the underlying datatype encoded by g , and then passing the result to h , is the same as just folding c over the datatype. If we plug in $h = (a)$, which is,

by definition, an algebra homomorphism from in to a , then the theorem becomes

$$(a)\ (g\ in) = g\ a \quad (12)$$

which is precisely the condition we are required to prove.

As outlined in Section 3, we are obliged to prove that our fusible functions constitute implementations of the functions that they are replacing. Because we are using Church encodings, we know that we can only replace functions that take the form of folds. For consumers, this obligation is rather simple; a consumer simply applies the Church-encoded value to an algebra, so $c_C = \lambda(Ch\ g) \rightarrow g\ b$ for some algebra $b : F\ B \rightarrow B$. We can prove that this is equivalent to folding this algebra over the original data structure, which satisfies the proof obligation for consumers:

$$\begin{aligned} &(\lambda(Ch\ g) \rightarrow g\ b)\ (toCh\ x) \\ &= \{ \text{definition of } toCh \} \\ &(\lambda(Ch\ g) \rightarrow g\ b)\ (Ch\ (\lambda a \rightarrow (a)\ x)) \\ &= \{ \text{function application} \} \\ &(\lambda a \rightarrow (a)\ x)\ b \\ &= \{ \text{function application} \} \\ &(b)\ x \end{aligned}$$

For producers, the situation is less straightforward. Unlike consumers, these functions do not have the form of a fold. Instead, they construct a data structure recursively (we cover the limitations of which functions can have recursive definitions in Section 5). The fusible version also uses a recursive function, but this function takes as an argument an algebra that it puts where the constructors would go. Therefore, a producer has the form $(\lambda x \rightarrow Ch\ (\lambda a \rightarrow f\ a\ x))$ where f recursively creates elements, and folds them by putting a where the constructors belong. If we actually construct this data structure, we end up simply passing in to f :

$$\begin{aligned} &fromCh\ ((\lambda x \rightarrow Ch\ (\lambda a \rightarrow f\ a\ x))\ s) \\ &= \{ \text{function application} \} \\ &fromCh\ (Ch\ (\lambda a \rightarrow f\ a\ s)) \\ &= \{ \text{definition of } fromCh \} \\ &(\lambda a \rightarrow f\ a\ s)\ in \\ &= \{ \text{function application} \} \\ &f\ in\ s \end{aligned}$$

The proof obligation amounts to requiring that, if f is passed in , it must construct the same data structure as the producer that it is replacing.

Transformations, like consumers, involve recursing over a data structure, and are therefore folds. Rather than combine the elements, they transform the elements. Therefore, they have the form $(in \circ f)$, where f is natural transformation that transforms each element composed with the initial algebra. The Church-encoded version of this function uses the same transformation f , but it creates a new Church encoded value, applying original Church encoded value to f composed with the resulting Church encoding’s abstracted algebra. We can satisfy the proof obligation from Section 3 for transformations using Equation 12.

$$\begin{aligned} &fromCh\ ((\lambda Ch\ g \rightarrow Ch\ (\lambda a \rightarrow g\ (a \circ f)))\ (Ch\ xs)) \\ &= \{ \text{function application} \} \\ &fromCh\ (Ch\ (\lambda a \rightarrow xs\ (a \circ f))) \\ &= \{ \text{definition of } fromCh \} \\ &(\lambda a \rightarrow xs\ (a \circ f))\ in \\ &= \{ \text{function application} \} \\ &xs\ (in \circ f) \end{aligned}$$

$$\begin{aligned}
&= \{ \text{Equation 12} \} \\
&\quad (in \circ f) (xs \text{ in}) \\
&= \{ \text{definition of } fromCh \} \\
&\quad (in \circ f) (fromCh (Ch xs))
\end{aligned}$$

The concept of Church encodings dualises to Cochurch encodings, which encapsulate unfolds instead of folds. For a final coalgebra $(\nu F, out)$ with a base functor $F : \mathbb{C} \rightarrow \mathbb{C}$, the type of the Cochurch encoding is

$$\mathbf{data} \ CoChurch \ F = \exists S . \ CoCh \ (S \rightarrow F \ S) \ S$$

This representation consists of a stepper function and an initial seed. Applying the stepper function to the seed produces a value and new seeds. Recursively applying the stepper function to successive seeds unfolds the data structure. Dual to the universal quantification we saw in the Church encoding, Cochurch encodings use existential type quantification to enforce the requirement that the type of the seed and the type of the stepper function match up. Another characterisation of this type is that it encapsulates a stateful computation, containing an initial state and a transition function that can yield a result and a new state. We take advantage of this characterisation in Section 5.

To convert a data structure νF to its Cochurch encoding, we create a pair with $out : \nu F \rightarrow F (\nu F)$ as the stepper function and the data structure itself as the seed:

$$\begin{aligned}
toCoCh &:: \nu F \rightarrow CoChurch \ F \\
toCoCh \ x &= CoCh \ out \ x
\end{aligned}$$

Whereas Church encodings have the recursive pattern pre-defined in the encoding, the Cochurch encoding simply has a description of a single step. To get back to the data structure, we unfold the data structure using this stepper function by applying it to the initial seed (and recursively to any resulting ones):

$$\begin{aligned}
fromCoCh &:: CoChurch \ F \rightarrow \nu F \\
fromCoCh \ (CoCh \ h \ x) &= [h] \ x
\end{aligned}$$

Like Church encodings, Cochurch encodings are isomorphic to their underlying datatypes. The proofs proceed along similar lines, with the conversion to Cochurch encodings and back harnessing the universal property of unfolds:

$$\begin{aligned}
&fromCoCh \ (toCoCh \ x) \\
&= \{ \text{definition of } toCoCh \} \\
&\quad fromCoCh \ (CoCh \ out \ x) \\
&= \{ \text{definition of } fromCoCh \} \\
&\quad [out] \ x \\
&= \{ \text{unfold reflection law (7)} \} \\
&\quad x
\end{aligned}$$

The other direction requires a similar theorem to Equation 12. To start, we note that the type of a function $(\exists C . (C \rightarrow F \ C), C) \rightarrow D$, which unfolds a Cochurch-encoded data structure to produce a value of a fixed type D , is isomorphic to the type $\forall C . (C \rightarrow F \ C) \rightarrow C \rightarrow D$. The free theorem of this type is

$$Fh \circ c = d \circ h \implies f \ c = f \ d \circ h \quad (13)$$

where $f : \forall C . (C \rightarrow F \ C) \rightarrow C \rightarrow D$. The precondition requires that h be coalgebra homomorphism from c to d . If we plug in the unfold $[c]$, which is a coalgebra homomorphism from c to out , we obtain the equation

$$f \ c = f \ out \circ [c] \quad (14)$$

Unlike with Church encodings, we can't directly apply this rule to our representation of Cochurch encodings because we use the

isomorphic existential type. To get such a rule, we instantiate the constructor $CoCh$ for f :

$$CoCh \ c = (CoCh \ out) \circ [c] \quad (15)$$

This rule states that unfolding a Cochurch-encoded structure and then re-encoding it yields an equivalent structure. We can now prove the other direction of the isomorphism:

$$\begin{aligned}
toCoCh \ (fromCoCh \ (CoCh \ c \ x)) \\
&= \{ \text{definition of } fromCoCh \} \\
toCoCh \ ([c] \ x) \\
&= \{ \text{definition of } toCoCh \} \\
CoCh \ out \ ([c] \ x) \\
&= \{ \text{composition} \} \\
(CoCh \ out \circ [c]) \ x \\
&= \{ \text{Equation 15} \} \\
CoCh \ c \ x
\end{aligned}$$

As with Church encodings, we can rely on the fact that Cochurch functions will replace functions of a certain form when satisfying the proof obligations, this time unfolds instead of folds. As unfolds are dual to folds, the easiest case this time is the production of a data structure. If we have a producer phrased as an unfold $[c]$ applied to a seed s , the equivalent Cochurch version simply takes the seed and pairs it with the coalgebra:

$$\begin{aligned}
fromCoCh \ ((\lambda s \rightarrow CoCh \ c \ s) \ x) \\
&= \{ \text{function application} \} \\
fromCoCh \ (CoCh \ c \ x) \\
&= \{ \text{definition of } fromCoCh \} \\
[c] \ x
\end{aligned}$$

Folds are not naturally producers, so Church producers have to have recursive definitions that build in the recursion as well as placement of the abstracted algebra. Similarly, unfolds are not naturally consumers. This means that a Cochurch consumer will have the form $(\lambda CoCh \ c \ s \rightarrow f \ c \ s)$, where f is a recursive function that applies the stepper function to seeds and consumes the values along the way. If we want this to consume data structures in the same fashion as an abstract function, f should behave the same way as that function if it is passed out :

$$\begin{aligned}
(\lambda CoCh \ c \ s \rightarrow f \ c \ s) \ (toCoCh \ x) \\
&= \{ \text{definition of } toCoCh \} \\
(\lambda CoCh \ c \ s \rightarrow f \ c \ s) \ (CoCh \ out \ x) \\
&= \{ \text{function application} \} \\
f \ out \ x
\end{aligned}$$

Finally, transformers behave similarly to those in Church encodings in that they consist of a natural transformation $f : F \rightarrow G$ between two functors. A transformation consists of an unfold that first yields a value, then transforms it with f , so it has the form $f \circ out$. The Cochurch version returns a new $CoChurch$ datatype where the stepper function is composed with the transformation. This satisfies the obligation from Section 3:

$$\begin{aligned}
fromCoCh \ ((\lambda CoCh \ c \ s \rightarrow CoCh \ (f \circ c) \ s) \ (CoCh \ h \ x)) \\
&= \{ \text{function application} \} \\
fromCoCh \ (CoCh \ (f \circ h) \ x) \\
&= \{ \text{definition of } fromCoCh \} \\
[f \circ h] \ x \\
&= \{ \text{fusion law of unfolds (8)} \}
\end{aligned}$$

$[f \circ out] \circ [h]$

Here, we have invoked the fusion law, although instead of using it to fuse two folds together, we have used it to justify splitting one apart. Now, we must prove that $G [h] \circ f \circ h = f \circ out \circ [h]$. We do this by using the fact that f is a natural transformation, with the coherence property $f \circ Fh = Gh \circ f$:

$$\begin{aligned} & G [h] \circ f \circ h = f \circ out \circ [h] \\ \iff & \{ \text{naturality of } f \} \\ & f \circ F [h] \circ h = f \circ out \circ [h] \\ \iff & \{ \text{composition} \} \\ & F [h] \circ h = out \circ [h] \end{aligned}$$

The final statement is true according to the universality property of unfolds.

Church and Cochurch encodings allow us to write functions over a data type using the familiar recursion schemes of folds and unfolds while gaining the added benefit of fusibility. They are also a convenient representation for shortcut fusion because their correctness generalises to any datatype. Many library writers may be unsure what the appropriate concrete representation is for their interface; this choice depends on the data structure and the interface the author wishes to implement. Some functions can be phrased as either a fold or an unfold, whereas others are inherently one or the other (further discussion on this matter can be found in [6]).

5. Application: Fusing leaf trees

So far, we have established the central idea of shortcut fusion and proposed using Church and Cochurch encodings as representations of recursive datatypes. We have also shown how they can satisfy the proof obligations in Section 3. Now, we will show how to instantiate each of these representations for a specific data structure. For this example, we will use the Glasgow Haskell Compiler's (GHC) rewrite rules and inlining system, which is the tool of choice for implementing shortcut fusion in Haskell. In order to show how this works for a datatype outside the usual list of suspects, we apply the framework to leaf trees. This is a data structure that can act as a sequence type where *append* is a $\mathcal{O}(1)$ operation, which can be useful in certain applications.

We define leaf trees using following datatype declaration:

```
data Tree a = Empty
           | Leaf a
           | Fork (Tree a) (Tree a)
```

For instructional purposes, the interface will be developed by instantiating both Church and Cochurch encodings in parallel as concrete representations. This allows us to compare the two side-by-side and gives the reader the opportunity to see both of them in action. Usually, however, a programmer would choose a single representation.

5.1 Combinators

To begin, we must instantiate a concrete representation for leaf trees, along with appropriate conversion functions for *con* and *abs*. As discussed in Section 4, we can use the Church encoding by defining *toCh* and *fromCh*, or the Cochurch encoding by defining *toCoCh* and *fromCoCh*.

For the Church encoding version, we start by declaring the base functor for *Tree*, which is the same as the datatype declaration but with the recursive definition abstracted away:

```
data Tree_ a b = Empty_ | Leaf_ a | Fork_ b b
```

Because *Tree* is a polymorphic type, the base functor we are concerned with is *Tree_ a*. We are now able to define the Church encoding type over *Trees*:

```
data Tree† a = Tree† (∀ b . (Tree_ a b → b) → b)
```

Next, we provide the definition for our *con* combinator, *toCh*

```
toCh :: Tree a → Tree† a
toCh t = Tree† (λa → fold a t)
fold :: (Tree_ a b → b) → Tree a → b
fold a Empty_ = a Empty_
fold a (Leaf x) = a (Leaf_ x)
fold a (Fork l r) = a (Fork_ (fold a l)
                          (fold a r))
```

The *fold* function is simply the fold over *Trees*, which takes an algebra *a* and applies it accordingly. For Church encodings, this is where the recursion is built-in to the type.

For *abs*, we use *fromCh*, which reconstructs the tree by applying the Church encoding to the initial algebra *in*:

```
fromCh :: Tree† a → Tree a
fromCh (Tree† fold) = fold in
in :: Tree_ a (Tree a) → Tree a
in Empty_ = Empty
in (Leaf x) = Leaf x
in (Fork l r) = Fork l r
```

As with the list example in Section 2, *in* is defined by having one case per constructor. In this case, single step of construction simply swaps the constructors back to those of *Tree*. The built-in recursion of the *Tree[†]* takes care of applying *in* recursively.

Dually, we can instantiate *con* and *abs* with conversion functions to and from the Cochurch encoding. We can use the same base functor as for the Church encoding. An unfold, instead of recursively combining results, creates a tree by branching into two subtrees, yielding a value of the sequence, or simply stopping when the subtree is empty.

Again, from Section 4, we get the definition of the Cochurch encoding

```
data Tree‡ a = ∃ s . Tree‡ (s → Tree_ a s) s
```

as a pair consisting of a stepper function and an initial seed.

To convert a tree to its Cochurch encoding, we pair the original tree with a function that describes how to perform single step in the unfold:

```
toCoCh :: Tree a → Tree‡ a
toCoCh t = Tree‡ out t
out Empty_ = Empty_
out (Leaf a) = Leaf_ a
out (Fork l r) = Fork_ l r
```

Dual to *in*, *out* describes to destruct a recursive data structure into its components. Going back the other way, we recursively apply the stepper function to each successive seed.

```
fromCoCh :: Tree‡ a → Tree a
fromCoCh (Tree‡ h s) = unfold h s
unfold h s = case h s of
  Empty_ → Empty
  Leaf_ a → Leaf a
  Fork_ sl sr → Fork (unfold h sl) (unfold h sr)
```

We can again see the duality between these two representations. For Church encodings, the recursion pattern is completely baked into the representation as it is created. The Cochurch encoding, on

the other hand, provides a blueprint for how to construct a tree, but the recursion itself appears when the encoding is converted back to a regular tree. In both cases, the proof of correctness comes with the encoding for free, because we have proved that Church and Cochurch encodings and the associated fold and unfold semantics satisfy the proof obligations for fusion.

5.2 Rewrite rules

Now that we have defined representations and conversion functions, and discharged the associated proof obligations, it is time to implement the syntactic transformation. Luckily, GHC provides a relatively simple way to do this with the *RULES* pragma [14]. This pragma allows us to specify an equation in which we replace any instance of the left hand side by the right hand side. For example, we can specify how to remove instances of *fromCh* followed by *toCh* by declaring the rule

```
{-# RULES "toCh/fromCh fusion"
  forall x. toCh (fromCh x) = x #-}
```

The `{-# . . #-}` brackets signify that the text in between is a compiler pragma. The keyword *RULES* specifies the name of the pragma, and the string in quotes is simply a unique name for the rule, which can be used to identify it in compiler-generated statistics. The rewrite equation itself begins with the keyword *forall*, which allows us to universally quantify one or more variables in the following equation. Finally, the equation contains a left hand side that we wish to rewrite into the right hand side. In our case, we want to remove the unnecessary conversion of a value, so we remove the conversion functions in the expression replace it with the value itself. We can also define the analogous rule for our Cochurch encoding combinators:

```
{-# RULES "toCoCh/fromCoCh fusion"
  forall x. toCoCh (fromCoCh x) = x #-}
```

The programmer should be aware that the *RULES* pragma comes with almost no guarantees. Aside from checking that the types of the two sides match, the *RULES* pragma does nothing to check the correctness of the transformation. Furthermore, specifying this rule does not guarantee that the compiler will rewrite all (or even any) of the situations where this rule can be applied! This is because encountering these situations depends on GHC inlining functions in a pipeline definition in order to expose the conversion functions, but also not inline away the combinators themselves before the rule can be applied. We can address this issue by fine-tuning how GHC inlines our conversion functions, which we will discuss next.

The first case, ensuring that functions containing these combinators are inlined, is a simple matter of using the *INLINE* pragma to encourage the GHC inliner [16] to inline it, even if it might not otherwise. However, it is also possible to tell GHC *when* to inline a function, which is important in order to keep the conversion combinators visible long enough for the simplifier to see them and apply the rewrite rule. To accomplish this, we pass an integer that specifies a phase of the simplifier in the pragma. In doing so, we signal to the simplifier *not* to inline the given function until that phase has been reached. Phases are numbered in decreasing order, with the final being phase 0, so if we specify the pragmas

```
{-# INLINE [0] toCh #-}
{-# INLINE [0] fromCh #-}
```

for our Church encoding combinators and

```
{-# INLINE [0] toCoCh #-}
{-# INLINE [0] fromCoCh #-}
```

for our Cochurch encoding combinators, then we give GHC as long as possible to eliminate these functions. If they still remain by the final phase, we assume that they cannot be fused away and allow GHC to inline and optimise them.

The main principle of writing fusible functions is to use the recursion provided by the concrete representation; recall that the purpose of this whole exercise is to allow us provide non-recursive definitions for our interface functions so that the compiler does the low-level work for us—introducing extra recursion will stump the compiler. Previously, we divided such functions into three categories based upon their use of the conversion combinators. We also use these divisions in establishing guidelines for fusible functions and will deal with an example of each of them here.

5.3 Producers

Producers are functions that produce a data structure without consuming one. As an example of such a function, we use the function *between*, which takes a pair of integers and generates the enumeration from the first to the second, inclusive:

```
between :: (Int, Int) → Tree Int
between (x, y)
  | x > y = Empty
  | x == y = Leaf x
  | x < y = Fork (between (x, mid))
                (between (mid + 1, y))
  where
    mid = (x + y) `div` 2
```

Written as a Church encoding, nothing is going to build in the recursive pattern for us, as this is usually done by *toCh* when converting an already existing data structure. There, the Church-encoding-based producer must encapsulate the recursive pattern to create the $Tree^\dagger$ representation of this tree:

```
between† :: (Int, Int) → Tree† Int
between† (x, y) = Tree† (λa → loop a (x, y))
  where
    loop a (x, y)
      | x > y = a Empty_
      | x == y = a (Leaf_ x)
      | x < y = a (Fork_ (loop a (x, mid))
                       (loop a (mid + 1, y)))
    where
      mid = (x + y) `div` 2
```

As a Church producer, this is an example of a function that is allowed to have recursion. The *loop* function simultaneously describes how to construct the enumeration as well as the placement of the abstracted algebra *a*, which describes how to reduce the structure. To actually produce the tree, *fromCh* will apply the $Tree^\dagger$ function to *in*:

```
between' :: (Int, Int) → Tree Int
between' = fromCh ∘ between†
{-# INLINE between' #-}
```

If *fromCh* is fused away, however, the next algebra will be applied instead, which means the elements of the enumeration will be consumed without writing out the actual tree. Note the use of the unconditional *INLINE* pragma to ensure that we inline *between'* so that the *fromCh* is exposed and subsequently removed, if possible.

In the Cochurch encoding, *fromCoCh* encapsulates our recursive pattern, and therefore we supply a non-recursive coalgebra that describes how to construct a single step:

```
between‡ :: (Int, Int) → Tree‡ Int
between‡ (x, y) = Tree‡ h (x, y)
```


where

```

h (x, y)
| x > y = Empty_
| x == y = Leaf_ x
| x < y = Fork_ (x, mid) (mid + 1, y)

```

where

```

mid = (x + y) 'div' 2
between'' :: (Int, Int) → Tree Int
between'' = fromCoCh ∘ between†
{-# INLINE between'' #-}

```

We depend on *fromCoCh* to recursively apply *c* to successive seeds until the tree is fully constructed. If, however, the *fromCoCh* call is fused away, the next function has a description of how to build a tree, which it can use to do so, or, alternatively, consume the values as they are yielded to produce a single value.

5.4 Consumers

Whereas Church producers may be recursive and Cochurch producers may not be, the opposite is true for consumers.

Since the *toCh* function encapsulates the recursive pattern, which describes how to reduce the structure to a single value. For example, the function *sum*

```

sum :: Tree Int → Int
sum Empty_ = 0
sum (Leaf_ x) = x
sum (Fork_ x y) = sum x + sum y

```

combines a *Tree Int* into a single *Int* by adding the elements together.

The function *sum*[†] mirrors this form, but removes the recursive call because folds (and therefore Church encodings) build in the recursion for consumption. Instead, we just have to supply an algebra that describes how to deal with a single step:

```

sum† :: Tree† Int → Int
sum† (Tree† g) = g s
s :: Tree_ Int Int → Int
s Empty_ = 0
s (Leaf_ x) = x
s (Fork_ x y) = x + y
sum' :: Tree Int → Int
sum' = sum† ∘ toCh
{-# INLINE sum' #-}

```

Instead, it is the Cochurch encoding version, *sum*[‡], that must do the recursive work on its own:

```

sum‡ :: Tree‡ Int → Int
sum‡ (Tree‡ h s) = loop s
where
loop s = case h s of
  Empty_ → 0
  Leaf_ x → x
  Fork_ l r → loop l + loop r
sum'' :: Tree Int → Int
sum'' = sum‡ ∘ toCoCh
{-# INLINE sum'' #-}

```

To accomplish this task, *sum*[†] is armed with *h*, which it can apply to the initial seed *s*. At each step, the *loop* function consumes the result, either by unwrapping and returning an *Int* or recursively obtaining the results of consuming two of subtrees and adding them together.

We now know how to create trees and consume them using this framework. Now, we move on to functions that are both producers and consumers.

5.5 Transformations

A transformation is a function that consumes a tree in order to produce a new one, and is therefore both a producer and a consumer.

Because of this, they should not be recursive in either representation, instead always using the built-in recursive pattern.

As examples of transformations, we develop *Tree* versions of two familiar functions: *reverse* and *filter*. First, take the function *reverse*, which reverses the ordering of the elements of the tree:

```

reverse :: Tree a → Tree a
reverse Empty_ = Empty_
reverse (Leaf_ a) = Leaf_ a
reverse (Fork_ l r) = Fork_ r l

```

The *reverse* function recursive swaps all subtrees and leaves the leaf elements intact.

As discussed in Section 4, such functions take the form of a transformation composed with either an algebra or coalgebra. In the case of *reverse*, this transformation defines a single swap of subtrees:

```

r :: Tree_ a c → Tree_ a c
r Empty_ = Empty_
r (Leaf_ a) = Leaf_ a
r (Fork_ l r) = Fork_ r l

```

For the Church encoding, *reverse*[†] creates a new *Tree*[†] that applies the Church encoded input value to the abstracted algebra precomposed with *r*, i.e transforms swaps the subtrees of a *Fork* and then passes that result on to the next algebra:

```

reverse† :: Tree† a → Tree† a
reverse† (Tree† g) = Tree† (λa → g (a ∘ r))
reverse' :: Tree a → Tree a
reverse' = fromCh ∘ reverse† ∘ toCh
{-# INLINE reverse' #-}

```

Dually, the Cochurch encoding takes the input stepper function and simply postcomposes *r*, i.e. as each subtree is yielded, it is then transformed:

```

reverse‡ :: Tree‡ a → Tree‡ a
reverse‡ (Tree‡ h s) = Tree‡ (r ∘ h) s
reverse'' :: Tree a → Tree a
reverse'' = fromCoCh ∘ reverse‡ ∘ toCoCh
{-# INLINE reverse'' #-}

```

In a similar way, we can define the function *filter* over *Trees*

```

filter :: (a → Bool) → Tree a → Tree a
filter p Empty_ = Empty_
filter p (Leaf_ a) = if p a then Leaf_ a else Empty_
filter p (Fork_ l r) = append (filter p l) (filter p r)

```

which takes a predicate *p* and discards any elements that do not satisfy the predicate. Unlike a *filter* on linear sequences, discarding an element does not mean we have to recursively search for the next element. Instead, we simply return *Empty*.

Like *reverse*, *filter* is defined with a transformation that is composed with a (co)algebra

```

f :: (a → Bool) → Tree_ a c → Tree_ a c
f p Empty_ = Empty_
f p (Leaf_ x) = if p x then (Leaf_ x) else Empty_
f p (Fork_ l r) = Fork_ l r

```

which tests each of the elements using the predicate p , and returns *Empty* if they fail.

The definition for the Church encoding version of *filter* is then similar to that of *reverse*:

```
filter† :: (a → Bool) → Tree† a → Tree† a
filter† p (Tree† g) = Tree† (λa → g (a ∘ (f p)))
filter' :: (a → Bool) → Tree a → Tree a
filter' p = fromCh ∘ filter† p ∘ toCh
{-# INLINE filter' #-}
```

and likewise for the Cochurch encoding:

```
filter‡ :: (a → Bool) → Tree‡ a → Tree‡ a
filter‡ p (Tree‡ h s) = Tree‡ (f p ∘ h) s
filter'' :: (a → Bool) → Tree a → Tree a
filter'' p = fromCoCh ∘ filter‡ p ∘ toCoCh
{-# INLINE filter'' #-}
```

At this point, we are able to write fusible producers, consumers, and transformers by defining functions over our concrete representation and converting to and from it when necessary. The goal of this effort is to provide an implementation of an interface that is more efficient and therefore obtains better performance than the analogous functions over the abstract datatype. There may be cases, however, where the analogous function over the abstract datatype is more efficient in certain contexts.

5.6 Rewrite rules, revisited

One of the features of leaf trees is that appending two trees is a constant time, non-recursive operation:

```
append :: Tree a → Tree a → Tree a
append t1 Empty = t1
append Empty t2 = t2
append t1 t2 = Fork t1 t2
```

This is extremely efficient as it requires no recursion and no copying. The downside, however, is that, if this is in the middle of a recursive pipeline such as

```
sumApp (x, y) = sum (append (between (x, y))
  (between (x, y)))
```

the fusion of the pipeline breaks down because *between* has to write out an intermediate data structure for use by *append*, and then *sum* consumes the newly appended tree. We can, of course, define fusible versions of *append* using both Church encodings and Cochurch encodings. The Church encoding version applies the abstracted algebra h' to the two subtrees joined by a *Fork*, each of which is also applied to h' .

```
append† :: Tree† a → Tree† a → Tree† a
append† (Tree† g1) (Tree† g2) =
  Tree† (λa → a (Fork (g1 a) (g2 a)))
append' :: Tree a → Tree a → Tree a
append' t1 t2 = fromCh (append† (toCh t1) (toCh t2))
{-# INLINE append' #-}
```

The Cochurch encoding version uses a feature we have hitherto been silent about: state. We can think of the seed of a Cochurch encoding as a state, and the stepper function as an iterator. The iterator takes this state as an argument and uses it to perform a computation that produces values each time it is applied and an updated state. In our case, the possibilities are to halt the computation with no value using *Empty*, halt but yield a value using *Leaf*, or to branch into two new states using *Fork*. Our Cochurch encoding does not

specify a specific state type, only requiring that it match the argument type of the stepper function. Therefore, we can transform a Cochurch encoded tree by creating a stepper function that applies prior one, then wraps successive states in a new, more expressive state type. This new state type can then be used to determine what do to next. For the *append[‡]* function,

```
append‡ :: Tree‡ a → Tree‡ a → Tree‡ a
append‡ (Tree‡ h1 s1) (Tree‡ h2 s2) = Tree‡ h' Nothing
  where
    h' Nothing = Fork_ (Just (Tree‡ h1 s1))
                  (Just (Tree‡ h2 s2))
    h' (Just (Tree‡ h s)) = case h s of
      Empty_ → Empty_
      Leaf_ a → Leaf_ a
      Fork_ l r → Fork_ (Just (Tree‡ h l))
                        (Just (Tree‡ h r))
append'' :: Tree a → Tree a → Tree a
append'' t1 t2 = fromCoCh
  (append‡ (toCoCh t1) (toCoCh t2))
{-# INLINE append'' #-}
```

we wrap the original state in a *Maybe* type that signals whether the stepper function is being called the first time (the *Nothing* case), in which case it should yield its two arguments joined by a *Fork*. The *Nothing* case then wraps each resulting state so that it contains the seed and stepper function of one of the arguments. When the stepper function is applied to each of these states, it can then apply the correct stepper function for branch it is being called on.

For both Church and Cochurch encodings, these functions allow us to fuse *appends*, solving the problem within a pipeline. When unfused, however, these functions are extremely inefficient. Even if we make no changes to a tree, converting it to another representation and back again involves a full traversal of the structure and forces the entire tree to be copied. The original version, on the other hand, did not need to inspect either subtree and could join them without copying. When *append* is part of a pipeline that already does this, it actually improves performance by fusing and avoiding intermediate data structures. Otherwise, it actually creates even more intermediate data structures than the original.

We now have a situation where we would like to use different versions of *append* based on whether or not it appears in a pipeline. We could, of course, provide an interface that allows the programmer to choose between different implementations of *append*. It would be unacceptably arduous, however, to require that the programmer always choose the right one for a given situation, and for large programs this might not even be feasible. Luckily, the rewrite pragmas that we discussed in Section 5.2 allow us define two versions of a function and then leave it to the compiler to choose the appropriate one for us at compile time:

```
{-# RULES
"append -> fused" [~1] forall t1 t2.
  append t1 t2 =
    fromCh (append† (toCh t1) (toCh t2))
"append -> unfused" [1] forall t1 t2.
  fromCh (append† (toCh t1) (toCh t2)) =
  append t1 t2 #-}
```

These rules use the simplifier phase notation that we previously used with the *INLINE* pragma. The \sim symbol can be read as “before”. The first rule swaps the out the bog-standard definition of *append* for the Church encoded version whenever it is encountered before simplifier phase 1. Once the simplifier reaches this, its

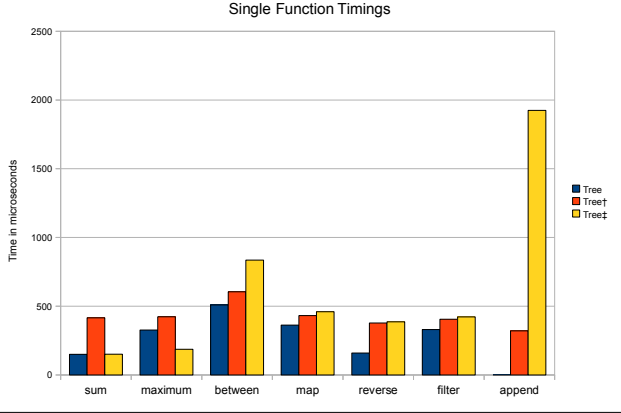


Figure 1. Single function timings

second to last phase, it checks to see if there are any instances of *append*[†] left that have not either of their conversion combinators fused away. If not, it puts the original *append* back, since it would be more efficient in this situation. We can, of course, implement the same sort of rules for the Cochurch encoding version by swapping the combinators and using *append*[‡] instead.

6. Benchmarks

Now that we have implemented our interface, it is time to test whether or not it achieves any speed up. We perform some microbenchmarks as a “sanity-check”, comparing functions in isolation with their performance in a fused context. We start by testing our interface functions in isolation and comparing them with the traditional versions of these functions. Our benchmarks cover the functions we have implemented so far, plus the familiar functions *map* and *maximum*, the former mapping a transformation function over the elements of a tree and the latter finding the maximum element of a tree. The program was compiled with GHC 7.0.2 using the `-O2` flag. The timings for these functions over a *Tree* of 10,000 elements is shown in Figure 1. In the case of *between*, we are measuring the time to create such a tree.

We can see that, in single-function tests, the use of shortcut fusion does not necessarily give any speedup. In fact, such implementations are sometimes even slower, especially the *append* function. This is not particularly surprising nor should it be cause for alarm; the purpose of this approach is to optimise pipelines, not single functions. If a particular function is often used its unfused form, and there is an intolerable slowdown in such cases, we can use rewrite rules to choose the correct version of the function automatically.

To test the performance of these functions when fused, we compose them to form the pipeline given in the introduction

$$\text{sum} \circ \text{map} (+1) \circ \text{filter odd} \circ \text{between}$$

and test this for an input of $(1, 10000)$. In addition, we test the function *sumApp* on the same input, both with the fusible forms of *append* and the non-recursive *Tree* version. The execution times for these pipelines are shown in Figure 2. As can be seen from the timings, the power of shortcut fusion shines clearly in this example. Both the Church and Cochurch representations achieve significant speedups over the conventional *Tree* example.

Finally, we provide more coverage by compositions of the functions we have tested singly. The results are shown in Figure 3. The results show a speedup, sometimes significant. Some what surprisingly, however, this is not completely consistent across all tests. There are still some cases in which the unfused functions outperform the fusible ones. This may be because the cost of conversion

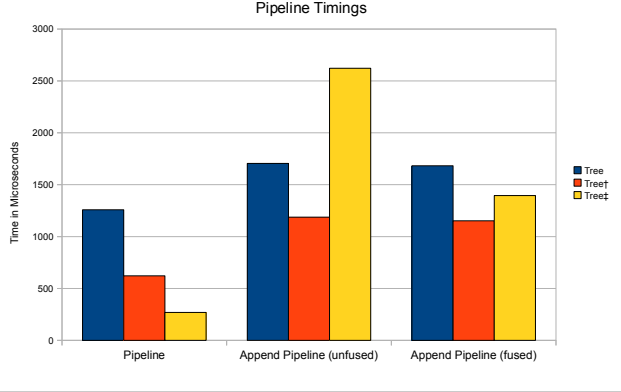


Figure 2. Pipeline timings

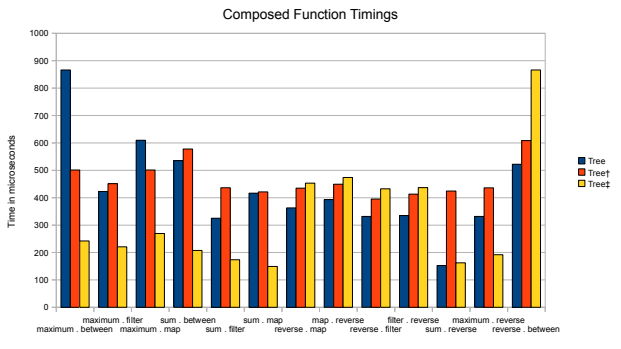


Figure 3. Fusion timings

is not “paid back” for a pipeline consisting of only two functions, or because the data set is too small. The worst performance seems to be with the *reverse* function, so this may be a case where fusion can only be of limited help. Interestingly, however, we note that Cochurch encodings consistently outperform Church encodings, sometimes by a significant margin. While we do not consider these results conclusive, we think that these results merit further investigation of this issue. It may be due to how GHC optimises code, or an issue that is specific to the fusion of tree or tree-like data structures.

Overall, our micro-benchmarks confirm the guidelines that we laid down in Section 5. We confirm that our shortcut fusion framework has the potential to provide a speedup, particularly when using Cochurch encodings. It would appear that, even when they do not provide a significant speedup, they do not decrease program performance as much as Church encodings. In a full-fledged library, benchmarking is an important part of the development process, and using shortcut fusion is not substitute. When such instances are identified, strategies such as the rewrite rule trick in Section 5.6 can be used to refine an implementation to provide the best performance.

7. Related work

Our work draws on prior shortcut fusion implementations, namely *foldr/build* [8], *destroy/unfoldr* [17], and stream fusion [1]. Of those, stream fusion introduces an explicit datatype that takes advantage of the fact that representations need not be isomorphic by adding an additional *Skip* constructor which allowed them to define more functions as unfolds. This was used to write fusible interfaces over arrays [2] and Unicode-encoded text representations [9]. A

similar setup has been provided by the worker-wrapper [7] transformation, which also proves a general set-up for implementing some optimisations, namely unboxing types.

The correctness and genericity of fusion has been explored in a variety of settings. Takano and Meijer [18] provided a calculational view of fusion using hylomorphisms. Ghani, Uustalu, and Vene have also given a “semantic footing” to *foldr/build* fusion and addressed the theoretical aspects of generalising it to arbitrary datatypes [5]. Johann and Ghani have also harnessed the concept of Church encodings in showing how to apply initial algebra semantics, and thus *foldr/build* fusion, to nested datatypes [13]. Voigtländer has also used free theorems to show correctness, specifically of a *destroy/build* rule [19] that suggests the possibility of mixing Church and CoChurch encodings within the same interface. We have also previously examined these fusion techniques in a categorical setting [11] in which we were able to compare previously incompatible fusion techniques within the same framework. These efforts, however, have remained largely in theoretical settings and left the pragmatic details relatively untouched.

The pragmatics of applying fusion to new datatypes has, however, been addressed in attempts to mechanise certain fusion techniques. Warm fusion attempts to derive *fold* and *build* combinators for a data type and automatically rewrite explicitly recursive functions [15]. The HFusion framework works similarly, although using hylomorphisms, which are more general [3]. Fusion is also accomplished by supercompilation [10], where it is not the goal but one of many consequences of method. Shortcut fusion is a less automated approach in the sense that it requires more setup from the programmer to get the fusion, but it is also a more *targeted* approach. The automated methods we mentioned either require modification of the compiler itself, or have to consider entire programs as a whole, or both. With shortcut fusion, a library writer is able to use his specialised knowledge of a data structure and interface to provide better performance without impacting other parts of a program. Additionally, shortcut fusion appears to offer a degree flexibility by allowing the author to choose a concrete representation to suit the needs of the data structure and interface. Such a comparison merits more investigation as automated methods, especially supercompilation, become more popular.

8. Conclusions

We have presented shortcut fusion as a method of providing better performance for functions written over recursive datatypes. Unlike prior approaches, we have moved away from depending on a specific recursion scheme or representation by showing how shortcut fusion is an instance of data refinement. We have shown we can instantiate shortcut fusion to a specific datatype and representation by fulfilling the specification we laid out. Using GHC’s compiler pragmas, we have given an example that shows how the aspiring library author can apply the same method to a new interface for a datatype. Our benchmarks give an example of some of the weak spots the programmer might look for in his own framework and we have shown possible ways of mitigating some common problems.

Now that we have introduced a new setup for implementing shortcut fusion, we would like to find new applications of shortcut fusion that reach out beyond the representations we discussed here. Our new “view” has more clearly specified requirements for shortcut fusion techniques, which will enable use to explore them more systematically. In particular, we are interested in those case where, unlike Church and Cochurch encodings, the concrete representation is not isomorphic but still faithfully represents the datatype. For example, stream fusion has shown that this can be useful for expanding the expressivity of shortcut fusion, in this case by introducing a *Skip* constructor. They have also shown that a “concrete” representation can serve as abstraction over another non-fusible

datatype, such as an array. This notion has remained rather confined, despite having possibly wider applications.

References

- [1] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion. *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming (ICFP '07)*, 42(9):315–326, Oct. 2007. ISSN 03621340.
- [2] D. Coutts, D. Stewart, and R. Leshchinskiy. Rewriting Haskell Strings. In *PADL '07*, volume 4354, pages 50–64. Springer-Verlag, 2007.
- [3] F. Domínguez. *HFusion: a fusion tool based on Acid Rain plus extensions*. Master thesis, Universidad de la República, 2009.
- [4] P. J. Freyd. Remarks on algebraically compact categories. In M. P. Fourman, P. T. Johnstone, and A. M. Pitts, editors, *Applications of Categories in Computer Science*, volume 177 of *LMS Lecture Note Series*, pages 95–106. Cambridge University Press, 1992.
- [5] N. Ghani, T. Uustalu, and V. Vene. Build, augment and destroy. Universally. pages 327–347. In *Asian Symposium on Programming Languages, Proceedings*, 2004.
- [6] J. Gibbons, G. Hutton, and T. Altenkirch. When is a function a fold or an unfold? In *Proceedings of the 4th International Workshop on Coalgebraic Methods in Computer Science*. Elsevier Science, 2001.
- [7] A. Gill and G. Hutton. The Worker Wrapper Transformation. *Journal of Functional Programming*, 19(2):227—251, 2009.
- [8] A. Gill, J. Launchbury, and S. L. Peyton Jones. *A short cut to deforestation*. ACM Press, New York, New York, USA, 1993.
- [9] T. Harper. Stream fusion on Haskell Unicode strings. In M. Morazán and S.-B. Scholz, editors, *IFL'09 Proceedings of the 21st international conference on Implementation and application of functional languages*, pages 125–140, Berlin, Sept. 2009. Springer-Verlag.
- [10] M. Heine, B. Sørensen, and R. Glück. Introduction to Supercompilation. In J. Hatcliff, T. Mogensen, and P. Thiemann, editors, *Partial Evaluation*, volume 1706 of *Lecture Notes in Computer Science*, pages 246–270. Springer Berlin / Heidelberg, 1999.
- [11] R. Hinze, D. W. James, and T. Harper. Theory and practice of fusion. In J. Hage, editor, *Pre-proceedings of the 22nd Symposium on the Implementation and Application of Functional Languages (IFL '10)*, pages 402–421, September 2010.
- [12] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972. ISSN 0001-5903.
- [13] P. Johann and N. Ghani. *Initial algebra semantics is enough!*, volume 4583 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.
- [14] S. P. Jones, A. Tolmach, and T. Hoare. Playing by the Rules: Rewriting as a practical optimisation technique in GHC. In *Haskell Workshop*, pages 203–233. ACM SIGPLAN, 2001.
- [15] J. Launchbury and T. Sheard. Warm fusion: deriving build-cats from recursive definitions. *Functional Programming Languages and Computer Architecture*, page 314, 1995.
- [16] S. L. Peyton-Jones and A. L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1-3): 3–47, Sept. 1998. ISSN 01676423.
- [17] J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming - ICFP '02*, volume 37, pages 124–132, New York, New York, USA, 2002. ACM Press.
- [18] A. Takano and E. Meijer. Shortcut deforestation in calculational form. *Functional Programming Languages and Computer Architecture*, page 306, 1995.
- [19] J. Voigtländer. Proving correctness via free theorems: the case of the destroy/build-rule. *ACM/SIGPLAN Workshop Partial Evaluation and Semantics-Based Program Manipulation*, 2008.
- [20] P. Wadler. Theorems for free! In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347—359, London, 1989. ACM.