Master's Thesis

Eben Rogers

June 4, 2024

Contents

1 Introduction

When writing functional code, we often use functions (or other data structures) to 'glue' multiple pieces of data together. Take, as an example, the following function in the programming language Haskell, as introduced by **?**:

```
all :: (a → Bool) → [a] → Bool
all p = and . map p
```

The function `map p` traverses across the input list, applying the predicate `p` to each element, resulting in a new boolean list. Then, the function `and` takes this resulting, intermediate, boolean list and consumes it by 'and-ing' together all the boolean values.

Being able to compose functions in this fashion is part of what makes functional programming so attractive, but it comes at the cost of computational overhead: Each time allocating a list cell, only to subsequently deallocate it once the value has been read. We could instead rewrite `all` in the following fashion:

```
all' p xs = h xs
  where h [] = True
        h (x : xs) = p x ∧ h xs
```

This function, instead of traversing the input list, producing a new list, and then subsequently traversing that intermediate list, traverses the input list only once; immediately producing a new answer. Writing code in this fashion is far more performant, at the cost of read- and write-ability. Can you write a high-performance, single-traversal, version of the following function (**?**)?

```
f :: (Int, Int) → Int
f = sum . map (+1) . filter odd . between
```

With some (more) effort and optimization, one could arrive at the following solution:

```
f' :: (Int, Int) → Int
f' (x, y) = loop x
  where loop x | x > y = 0
               | otherwise = if odd x
                             then (x + 1) + loop (x + 1)
                             else loop (x + 1)
```

Doing this by hand every time, to get from the nice, elegant, compositional style of programming to the higher-performance, single-traversal style, gets old very quick. Especially if this needs to be done, by hand, **every** time you compose any two functions. Is there some way to automate this process?

Fusion The answer is yes*, but it comes with an asterisk attached, namely that the functions that we are working with are folds or unfolds. The form of optimization that we are looking for is called fusion: The process of taking multiple list producing/consuming functions and turning (or fusing) them into just one.

Much work already exists, which is discussed in detail in **??**. My thesis focuses on a specific form of fusion called shortcut fusion through the use of (Co)Church encodings as described by **?** and asks the following two questions:

1. To implement (Co)Church encodings, what is necessary to make the code reliably fuse? This leads to the following sub-questions:
   - What transformations are used within Haskell to enable fusion to work?
   - What tools are used to get Haskell's compiler to cooperate and trigger fusion?
2. Are the transformations used to enable fusion safe? Meaning:
   - Do the transformations in Haskell preserve the semantics of the langaugage?
   - If the mathematics and the encodings are implemented in a dependently typed langauge, can the transformation be proved correct?

My thesis centers on formalizing, replicating, and expanding upon **?**'s work and makes two crucial contributions, answering the two questions above:

1. The Church and Cochurch encodings' implementation in Haskell, as described by **?** are replicated and investigated further as to their performance characteristics. In this process, a weakness was found in Haskell's optimizer, and further practical insights were gleaned as to how to get these encodings to properly fuse as well (especially for Cochurch encodings) and what optimizations enable shortcut fusion to do its work.

   This is important as **?** gave a good pragmatic explanation as to how to implement the (Co)Church encodings in Haskell, gave an example implementation, and benchmarked that implementation. He did not dive into too much detail as to *why* they work stating, "Interestingly, however, we note that Cochurch encodings consistently outperform Church encodings, sometimes by a significant margin. While we do consider these results conclusive, we think that these results merit further investigation." (**?**). This is what my research has (partially) set out to look into. This is discussed in detail in **??**.

2. The Church and Cochurch encodings described are formalized and implemented, including the relevant category theory, in Agda, in as a general fashion as possible, leveraging containers (**?**) to represent strictly positive functors. Furthermore, the functions that are described (producing, transforming, and consuming) are also implemented in a general fashion and shown to be equal to regular folds (i.e., catamorphisms and anamorphisms).

   This is important because there currently does not seem to exist a formalization of the work. Formally verifying the mathematics will strengthen the work done by **?**, perhaps also aiding in understanding in how the different pieces of mathematics relate. This is discussed in detail in **??**.

2 Background

Before discussing the work that I have done, it is important to describe the necessary background. My work builds on a body of existing work, namely foldr/build fusion (**?**), some category theory, Church encodings (**?**), Containers (**?**), parametricity also known as free theorems **?**, and some optimizations in Haskell's optimization pipeline that are relevant for fusion.

I will be describing each of these works briefly. After that, in the next sections, I will describe the work that I have done that builds on these topics.

2.1 Foldr/build fusion (on lists)

Starting with the basics of fusion. In **?**'s paper the original 'schortcut deforestation' technique was described. The core idea is described here as follows:

In functional programming lists are (often) used to store the output of one function such that it can then be consumed by another function. To co-opt **?**'s example:

$$all\ p\ xs = and\ (map\ p\ xs)$$

`map p xs` applies `p` to all of the elements, producing a boolean list, and `and` takes that new list and "ands" all of them together to produce a resulting boolean value. "The intermediate list is discarded, and eventually recovered by the garbage collector" (**?**).

This generation and immediate consumption of an intermediate datastructure introduces a lot of computation overhead. Allocating resources for each cons datatype instance, storing the data inside of that instance, and then reading back that data, all take time. One could instead write the above function like this:

$$
\begin{aligned}
&all'\ p\ xs = h\ xs \\
&\quad \textbf{where}\ h\ [\,] = True \\
&\qquad\quad\ \ h\ (x : xs) = p\ x \wedge h\ xs
\end{aligned}
$$

Now no intermediate datastructure is generated at the cost of more programmer involvement. We've made a custom, specialized version of `and . map p`. The compositional style of programming that function programming languages enable (such as Haskell) would be made a lot more difficult if, for every composition, the programmer had to write a specialized function. Can this be automated?

?'s key insight was to note that when using a `foldr k z xs` across a list, the effect of its application "is to replace each `cons` in the list `xs` with k and replace the `nil` in `xs` with `z`. By abstracting list-producing functions with respect to their connective datatype (`cons` and `nil`), we can define a function `build`:

$$build\ g = g\ (:)\ [\,]$$

Such that:

$$foldr\ k\ z\ (build\ g) = g\ k\ z$$

?."

? dubbed this the `foldr/build` rule. For its validity `g` needs to be of type:

$$g : \forall\ \beta : (A \to \beta \to \beta) \to \beta \to \beta$$

Which can be proved to be true through the use of g's free theorem à la **?**. For more information on free theorems see **??**

2.1.1 An example

Take the function from, that takes two numbers and produces a list of all the numbers from the first to the second:

$$
\begin{aligned}
from\ a\ b = \ &\textbf{if}\ a > b \\
&\textbf{then}\ [\,] \\
&\textbf{else}\ a : from\ (a + 1)\ b
\end{aligned}
$$

To arrive at a suitable `g` we must abstract over the connective datatypes:

$$
\begin{aligned}
from'\ a\ b = \lambda c\ n \to \ &\textbf{if}\ a > b \\
&\textbf{then}\ n \\
&\textbf{else}\ c\ a\ (from\ (a + 1)\ b\ c\ n)
\end{aligned}
$$

This is obviously a different function, we now redefine `from` in terms of `build` (**?**):

$$from\ a\ b = build\ (from'\ a\ b)$$

With some inlining and β reduction, one can see that this definition is identical to the original `from` definition. Now for the killer feature (**?**):

$$
\begin{aligned}
&sum\ (from\ a\ b) \\
&\quad = foldr\ (+)\ 0\ (build\ (from'\ a\ b)) \\
&\quad = from'\ a\ b\ (+)\ 0
\end{aligned}
$$

Notice how we can apply the `foldr/build` rule here to prevent an intermediate list being produced. Any adjacent `foldr/build` pair "cancel away". This is an example of shortcut fusion.

One can rewrite many functions in terms of `foldr` and `build` such that this fusion can be applied. This can be seen in Figure **??**. See **?**'s work, specifically the end of section 3.3 (`unlines`) for a more expansive example of how fusion, β reduction, and inlining can combine to fuse a pipeline of functions down an as efficcient minimum as can be expected.

$$
\begin{aligned}
&\mathit{map}\ f\ \mathit{xs} = \mathit{build}\ (\lambda c\ n \to \mathit{foldr}\ (\lambda a\ b \to c\ (f\ a)\ b)\ n\ \mathit{xs})\\
&\mathit{filter}\ f\ \mathit{xs} = \mathit{build}\ (\lambda c\ n \to \mathit{foldr}\ (\lambda a\ b \to \textbf{if}\ f\ a\ \textbf{then}\ c\ a\ b\ \textbf{else}\ b)\ n\ \mathit{xs})\\
&\mathit{xs} +\!\!+ \mathit{ys} = \mathit{build}\ (\lambda c\ n \to \mathit{foldr}\ c\ (\mathit{foldr}\ c\ n\ \mathit{ys})\ \mathit{xs})\\
&\mathit{concat}\ \mathit{xs} = \mathit{build}\ (\lambda c\ n \to \mathit{foldr}\ (\lambda x\ Y \to \mathit{foldr}\ c\ y\ x)\ n\ \mathit{xs})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{repeat}\ x = \mathit{build}\ (\lambda c\ n \to \textbf{let}\ r = c\ x\ r\ \textbf{in}\ r)\\
&\mathit{zip}\ \mathit{xs}\ \mathit{ys} = \mathit{build}\ (\lambda c\ n \to \textbf{let}\ \mathit{zip}'\ (x : \mathit{xs})\ (y : \mathit{ys}) = c\ (x, y)\ (\mathit{zip}'\ \mathit{xs}\ \mathit{ys})\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{zip}'\ \_\ \_ = n\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ \mathit{zip}'\ \mathit{xs}\ \mathit{ys})
\end{aligned}
$$

$$
\begin{aligned}
&[\,] = \mathit{build}\ (\lambda c\ n \to n)\\
&x : \mathit{xs} = \mathit{build}\ (\lambda c\ n \to c\ x\ (\mathit{foldr}\ c\ n\ \mathit{xs}))
\end{aligned}
$$

Figure 1: Examples of functions rewritten in terms of `foldr/build`. (**?**)

2.1.2 Generalization to recursive datastructues

This foldr/build fusion works for lists, but it has several limitations. One is that it only works on lists, this is alleviated using Church encodings and is described by **?**. Secondly, the functions that we are writing need to be expressible in terms of compositions of foldrs and builds. What if we want to implement to converse approach? This exists and is destroy/unfoldr fusion and is described by **?**. This work generalized by Cochurch encodings, also described by **?**.

The generalization by Harper leverages (Co)Church, which uses definitions from category such as F-algebras and initiality. Don't know what they are? Read on in the next section, where I give these category theory definitions.

2.2 The category theory

In order to explain what an initial/terminal F-(co)algebra is, I'll first need to explain what a functor is and, more pressingly, what a category is. The concept of cata- and anamorphisms (folds and unfolds) will follow suit. If you're familiar with category theory and these concepts, you can skip this section. The mathematics described here are based on the lecture notes by **?**.

2.2.1 A Category

A **category** $\mathcal{C}$ is a collection of four pieces of data satisfying three proofs:

1. A collection of objects, denoted by $\mathcal{C}_0$
2. For any given objects $X, Y \in \mathcal{C}_0$, a collection of morphisms from X to Y, denoted by $\mathtt{hom}_{\mathcal{C}}(X, Y)$, which is called a *hom-set*.
3. For each object $X \in \mathcal{C}_0$, a morphism $\mathtt{Id}_X \in \mathtt{hom}_{\mathcal{C}}(X, X)$, called the *identity morphism* on X.
4. A binary operation: $(\circ)_{X,Y,Z} : \mathtt{hom}_{\mathcal{C}}(Y, Z) \to \mathtt{hom}_{\mathcal{C}}(X, Y) \to \mathtt{hom}_{\mathcal{C}}(X, Z)$, called the *composition operator*, and written infix without the indices X, Y, Z as in $g \circ f$.

These pieces of data should satisfy the following three properties:

1. (**Left unit law**) For any morphism $f \in \mathtt{hom}_{\mathcal{C}}(X, Y)$:
$$f \circ \mathtt{Id}_X = f$$
2. (**Right unit law**) For any morphism $f \in \mathtt{hom}_{\mathcal{C}}(X, Y)$:
$$\mathtt{Id}_Y \circ f = f$$
3. (**Associative law**) For any morphisms $f \in \mathtt{hom}_{\mathcal{C}}(X, Y), g \in \mathtt{hom}_{\mathcal{C}}(Y, Z)$, and $h \in \mathtt{hom}_{\mathcal{C}}(Z, W)$:
$$h \circ (g \circ f) = (h \circ g) \circ f$$

2.2.2 Initial/Terminal Objects

Categories can contain objects that have certain (useful) properties. Two of these properties are summarized below:

initial Let $\mathcal{C}$ be a category. An object $A \in \mathcal{C}_0$ is **initial** if there is exactly one morphism from A to any object $B \in \mathcal{C}_0$:

$$\forall A, B \in \mathcal{C}_0 : \exists! \texttt{hom}_{\mathcal{C}}(A, B) \implies \textbf{initial}(A)$$

terminal Let $\mathcal{C}$ be a category. An object $A \in \mathcal{C}_0$ is **terminal** if there is exactly one morphism from any object $B \in \mathcal{C}_0$ to A:

$$\forall A, B \in \mathcal{C}_0 : \exists! \texttt{hom}_{\mathcal{C}}(B, A) \implies \textbf{terminal}(A)$$

The proofs of initality and terminality require a proof that is split into two steps: A proof of existence (The $\exists$ part of $\exists!$) and a proof of uniqueness (The ! part of $\exists!$). The former is usually done by construction, giving an example of a function that satisfies the property and the latter is usually done my assuming that another $\texttt{hom}_{\mathcal{C}}(A, B)$ (for the initial case) exists and showing that it must be equal to the one constructed.

2.2.3 Functors

For a given category $\mathcal{C}, \mathcal{D}$, a **functor** from $\mathcal{C}$ to $\mathcal{D}$ consists of two pieces of data and three proofs:

1. A function mapping objects in $\mathcal{C}$ to $\mathcal{D}$:

$$\mathcal{C}_0 \to \mathcal{D}_0$$

2. For each $X, Y \in \mathcal{C}_0$, a function mapping morphisms in $\mathcal{C}$ to morphisms in $\mathcal{D}$:

$$\texttt{hom}_{\mathcal{C}}(X, Y) \to \texttt{hom}_{\mathcal{D}}(F(X), F(Y))$$

These pieces of data should satisfy these two properties:

1. (**Composition law**) for any two morphisms $f \in \texttt{hom}_{\mathcal{C}}(X, Y), g \in \texttt{hom}_{\mathcal{C}}(Y, Z)$:

$$F(g \circ f) = Fg \circ Ff$$

2. (**Identity law**) For any $X \in \mathcal{C}_0$, we have:

$$F(\texttt{Id}_X) = \texttt{Id}_{F(X)}$$

An **endofunctor** is a functor that maps objects back to the category itself, i.e. $F : \mathcal{C} \to \mathcal{C}$

2.2.4 (Category of) F-(Co)Algebras

Given an endofunctor $F : \mathcal{C} \to \mathcal{C}$:
An **F-Algebra** consists of two pieces of data:

1. An object $C \in \mathcal{C}_0$
2. A morphism $\phi \in \texttt{hom}_{\mathcal{C}}(F(C), C)$

An **F-Algebra Homomorphism** is, given two F-Algebras $(C, \phi), (D, \psi)$, a morphism $f \in \texttt{hom}_{\mathcal{C}}(C, D)$, such that the following diagram commutes (i.e. $f \circ \phi = \psi \circ Ff$):

$$\begin{array}{ccc} FC & \xrightarrow{\phi} & C \\ {\scriptstyle Ff}\downarrow & & \downarrow{\scriptstyle f} \\ FD & \xrightarrow{\psi} & D \end{array}$$

The **category of F-Algebras** denoted by $\mathcal{A}lg(F)$ consists of (the needed) four pieces of data:

1. The objects are F-Algebras

2. The morphisms are F-Algebra homomorphisms
3. The identity on (C, ϕ) is given by the identity $\texttt{Id}_C$ in $\mathcal{C}$
4. The composition is given by the composition of morphisms in $\mathcal{C}$

These pieces of data should satisfy the usual category laws: left/right unit law and composition law. Note how $\mathcal{A}lg(F)$ makes use of the underlying category $\mathcal{C}$ of the functor to define its objects. An $\mathcal{A}lg(F)$ implicitly contains an underlying category in which its objects are embedded.

An **F-Coalgebra** consists of two pieces of data:

1. An object $C \in \mathcal{C}_0$
2. A morphism $\phi \in \texttt{hom}_{\mathcal{C}}(C, F(C))$

F-Coalgebra homomorphisms and $\mathcal{C}o\mathcal{A}lg(F)$ can be defined conversely as done for F-Algebras.

2.2.5 Cata- and Anamorphisms

Given (if it exists) an initial F-Algebra (μ^F, in) in $\mathcal{A}lg(F)$. We can know that (by definition), that for any other F-Algebra (C, ϕ), there exists a *unique* morphism $(\!|\phi|\!) \in \texttt{hom}_{\mathcal{C}}(\mu^F, C)$ such that the following diagram commutes:

$$\begin{array}{ccc} F\mu^F & \xrightarrow{in} & \mu^F \\ {\scriptstyle F(\!|\phi|\!)}\downarrow & & \downarrow{\scriptstyle (\!|\phi|\!)} \\ FC & \xrightarrow{\phi} & C \end{array}$$

A morphism of the form $(\!|\phi|\!)$ is called a **catamorphism**.

An analagous definition of for terminal objects in $\mathcal{C}o\mathcal{A}lg(F)$ exists, called **anamorphisms**, denoted by $[\![\phi]\!]$

2.2.6 Fusion property

Now for the definition we've been waiting for, **fusion**: Given an endofunctor $F : \mathcal{C} \to \mathcal{C}$ and an initial algebra (μ^F, in) in $\mathcal{A}lg(F)$. For any two F-Algebras (C, ϕ) and (D, ψ) and morphism $f \in \texttt{hom}_{\mathcal{C}}(C, D)$ we have a **fusion property**:

$$f \circ \phi = \psi \circ F(f) \Longrightarrow f \circ (\!|\phi|\!) = (\!|\psi|\!)$$

In English, if f is an F-Algebra homomorphism, we can know that $f \circ (\!|\psi|\!) = (\!|\psi|\!)$. We can fuse two functions into one! This is summarized in the following diagram:

$$\begin{array}{ccc} F\mu^F & \xrightarrow{in} & \mu^F \\ {\scriptstyle F(\!|\phi|\!)}\downarrow & & \downarrow{\scriptstyle (\!|\phi|\!)} \\ FC & \xrightarrow{\phi} & C \\ {\scriptstyle Ff}\downarrow & & \downarrow{\scriptstyle f} \\ FD & \xrightarrow{\psi} & D \end{array} \qquad \begin{array}{l} F(\!|\psi|\!) \\ \\ \\ \\ (\!|\psi|\!) \end{array}$$

An converse definition of fusion can be made for terminal object in $\mathcal{C}o\mathcal{A}lg(F)$

2.3 Library Writer's Guide to Shortcut Fusion

Now that the sufficient category theory has been explained, it is possible to describe **?**'s paper, which my thesis centers on called "A Library Writer's Guide to Shortcut Fusion".

In the work, **?** explain the concept of Church and CoChurch encodings in four steps: The necessary underlying category theory, the concepts of encodings and the proof obligations necessary for ensuring correctness of the encodings, the concepts of (Co)Church encodings with the proof of correctness, and finally an example implementation for leaf trees. I will now go through each step briefly.

2.3.1 Category Theory

For the full overview of the category theory, see **??**. The main concepts that **?** explains are the *universal property of (un)folds*, the *fusion law*, and the *reflection law*; all of which can be derived from the category theory already described earlier.

The universal property of folds is as follows:

$$h = (\!|a|\!) \iff h \circ in = a \circ Fh$$

The fusion law as:

$$h \circ (\!|a|\!) = (\!|b|\!) \impliedby h \circ a = b \circ Fh$$

And the reflection law as:

$$(\!|in|\!) = id$$

I formalized and proved all of these properties in my Agda formalization. It is also interesting to note that, for the universal property of unfolds, the forward direction is the proof of existence and the backward direction the proof of uniqueness, for the proof of initiality of an algebra. Converse definitions exist for terminal coalgebras, but I will not cover them in this section. They do exist in my formalization.

2.3.2 Encodings

The purpose of the encodings is to encode recursive functions, which are not inlined by Haskell's optimizer, into ones that are capable of being inlined and therefore fused: "For example, assume that we want to convert values of the recursive datatype μ`F` to values of a type `F`. The idea is that `C` can faithfully represent values of μ`F`, but composed functions over `C` can be fused automatically" (**?**).

Now, instead of writing functions over μ`F`, we write functions over `C`, along with two conversion functions `con:` μ`F` $\to$ `C` (converst) and `abs :` `C` $\to$ μ`F` (abstract). In order for the datatype `C` to faithfully represent μ`F`, we need $abs \circ con = id_{\mu F}$. I.e. that `C` can represent all values of μ`F` uniquely.

In total there are four main proof obligations, the one mentioned above, as well as the commutation of the following three diagrams:

$$\begin{array}{ccc} \mu F & \xleftarrow[abs]{} & C \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle f_C} \\ \mu F & \xleftarrow[abs]{} & C \end{array} \qquad \begin{array}{ccc} S & & \\ {\scriptstyle p}\downarrow & \searrow^{p_C} & \\ \mu F & \xleftarrow[abs]{} & C \end{array} \qquad \begin{array}{ccc} \mu F & \xrightarrow{con} & C \\ {\scriptstyle c}\downarrow & \swarrow_{c_C} & \\ T & & \end{array}$$

Where, in the second diagram, `p` is a producer function, generating a recursive data structure from a seed of type `S`, and, in the third diagram, `c` is a consumer function, consuming a recursive data structure to produce a value of type `T`.

2.3.3 (Co)Church Encodings

Next, **?** proposes two encodings, `Church` and `CoChurch`.

Church `Church` is defined (abstractly) as the following datatype:

data *Church* F = *Ch* $(\forall\ A \Rightarrow (F\ A \to A) \to A)$

`Church` contains a recursion principle (often referred to as `g` throughout this thesis). With conversion and abstraction functions `toCh` and `fromCh`:

toCh :: *mu* $F \to$ *Church* F
toCh x = *Ch* $(\lambda a \to$ *fold* $a\ x)$
fromCh :: *Church* $F \to$ *mu* F
fromCh (*Ch* g) = g **in**

From these definitions, Harper proves the four proof obligations along with a fifth proof, proving the other composition of `con` and `abs` to be equal to `id`, thereby showing isomorphism. For the proof of transformers and `con ∘ abs = id`, Harper makes use of the free theorem for the polymorphic recursion principle `g`. In all the five proofs for Church encodings, Harper does not use the fusion property.

Cochurch `CoChurch` is defined (abstractly) as the following datatype:

$$\textbf{data}\ \mathit{CoChurch'}\ F = \exists\ S \Rightarrow \mathit{CoCh}\ (S \to F\ S)\ S$$

An isomorphic definition which Harper later uses is the one I end up using in my formalization:

$$\textbf{data}\ \mathit{CoChurch}\ F = \forall\ S \Rightarrow \mathit{CoCh}\ (S \to F\ S)\ S$$

The Cochurch encoding encodes a coalgebra and a seed value together. The conversion and abstraction functions, `toCoCh` and `fromCoCh`:

$$\begin{aligned}&\mathit{toCoCh} :: \mathit{nu}\ F \to \mathit{CoChurch}\ F\\ &\mathit{toCoCh}\ x = \mathit{CoCh}\ \mathit{out}\ x\\ &\mathit{fromCoCh} :: \mathit{CoChurch}\ F \to \mathit{nu}\ F\\ &\mathit{fromCoCh}\ (\mathit{CoCh}\ h\ x) = \mathit{unfold}\ h\ x\end{aligned}$$

Similarly to his description of Church encodings, Harper proves the four proof obligations as well as the additional fifth one. The `con ∘ abs = id` proof, leverages the free theorem for the corecursion principle of the type `CoChurch`. The proof for natural transformations, however, does not use the free theorem and instead uses the fusion property for unfolds.

2.3.4 Example implementation

To tie it all together, Harper gives an example implementation of how one would implement the encodings described so far. For this he uses Leaf Trees. He implements four functions, `between`, `filter`, `concat`, and `sum`, as a normal, recursive function, in Church encoded form, and in Cochurch encoded form.

In doing so, he shows exactly how one goes from using the normal, recursive datatypes and function that are typically used in Haskell, to Church and Cochurch encoded versions. To conclude the performance of different compositions of functions are compared to show the performance benefits and differences between the three different variants of functions.

2.4 Theorems for Free

? in work 'Theorems for Free', describes a way of getting theorems from a polymorphic function only by looking at its type. In his paper, he uses the trick of reading types as relations (instead of sets) in order to derive a lemma called *parametricity*.

From this it is possible to derive a theorem that a type satisfies, without looking at its definition. These free theorems can be used to make claims about polymorphic functions. This is also done in **?**'s work; namely a theorem about the polymorphic induction principle and coinduction principle function types.

For example the free theorem of the following polymorphic function (**?**):

$$g : \forall\ A\ .\ (F\ A \to A) \to A$$

is the theorem stating that:

$$h\ .\ b = c\ .\ F\ h \Rightarrow h\ (g\ b) = g\ c$$

For functions `b : F B -> B`, `c : F C -> C`, `h : B -> C`.

Within Agda, proving that the free theorems of the polymorphic function types are correct is something that is currently not possible without extensions. Recent work by **?** does exist, that extends cubical Agda with a `-bridges` extension that makes it possible to derive free theorems from within Agda. While it might be possible to leverage this implementation, the work is very new, having come out after the start of this thesis project. Instead, I have opted to postulate the free theorems needed, which is only on two locations.

2.5 Containers

In my formalization I needed to represent functors somehow. While a `RawFunctor` datatype does exist, it does not provide the necessary structure such that proofs can be done over it, such as the functor laws.

Instead, I have opted to use Containers to represent strictly positive functors as described by **?**. The definition of a container is as follows:

```
record Container (s p : Level) : Set (suc (s ⊔ p)) where
  constructor _▷_; field
    Shape : Set s
    Position : Shape → Set p
```

A container contains an index set, called `Shape` and also a `Position`, which represent the recursive elements of the container.

Containers can be given a semantics (or extension) in the following manner:

```
⟦_⟧ : ∀ {s p ℓ} → Container s p → Set ℓ → Set (s ⊔ p ⊔ ℓ)
⟦ S ▷ P ⟧ X = Σ[ s ∈ S ] (P s → X)
```

The `X` represents the type of the recursive elements of the container.

The main benefit of leveraging containers to represent functions is that positivity is maintained as well as that the functor laws are true by definition. Deriving the container from a given (polynomial) functor is done in a couple of steps:

1. Analyze how many constructors your functor has, take as an example 2.
2. For the left side of the container take the coproduct of types that store the non-recursive sub-elements (such as const).
3. Count the amount of recursive elements in the constructor, the return type should include that many elements.

Taking an example:

`List` Taking the base functor for `List`: $\texttt{F}_A$ `X := 1 + A × X`.

For the `Shape` we take the coproduct of `Fin 1` and `const A`, corresponding to the '`nil`' and '`cons a _`' part, respectively.

For the `Position`, we have one constructor that is non-recursive and one that contains one recursive element, so we have: `0 → Fin 0` and `const n → Fin 1`. The Fin 1 refers to the recursive X that is present in the base functor (or the '`cons _ as`' part of cons).

`Binary tree` Taking the base functor for `Tree`: $\texttt{F}_A$ `X := 1 + X × A × X`.

For the `Shape` we take the coproduct of `Fin 1` and `const A`.

For the `Position`, we have one constructor that is non-recursive and one that contains two recursive elements, so we have: `0 → Fin 0` and `const n → Fin 2`.

The above description is summarized below in a table:

	`List`	`Binary Tree`
Base functor	$\texttt{F}_A$ `X := 1 + (A × X)`	$\texttt{F}_A$ `X := 1 + (X × A × X)`
Shape	`Fin 1 + const A`	`Fin 1 + const A`
Position	`nil → Fin 0 and const n → Fin 1`	`nil → Fin 0 and const n → Fin 2`

For a concrete example of how a datatype is implemented, see **??**.

2.6 Haskell's optimization pipeline

In order to understand how fusion works, it is important to understand a few other concepts that fusion works in tandem with. Namely, beta reduction, inlining, case-of-case, and tail call optimization. I will give a brief description of each.

2.6.1 Beta reduction

Beta reduction is simply the rule where an expression of the form `(λ x . a[x]) y` can get transformed into `a[y]`. For example `(λ x . x + x) y` would become `y + y`.

2.6.2 Inlining

Inlining is the process of taking a function expression and unfolding it into its definition. If we take the function `f = (+2)` and an expression `f 5`, we could inline `f` such that we get `(+2) 5`; which we could inline again to obtain `5 + 2`.

2.6.3 Case of case, and known-case elimination

As discussed by **?**, case of case optimization is the transformation of the following pattern:

case (
  case *C* **of**
    B1 → *F1*
    B2 → *F2*
) **of**
    A1 → *E1*
    A2 → *E2*

To the following[1]:

case *C* **of**
  B1 → **case** *F1* **of**
    A1 → *E1*
    A2 → *E2*
  B2 → **case** *F2* **of**
    A1 → *E1*
    A2 → *E2*

Where the branches of the outer case are pushed into the branches of the inner. Furthermore:

case *V* **of**
  V → *Expr*
  ...

Is case-of-known-constructor, we can simplify it to:

Expr

Together, these optimizations can often lead to the removal of unecessary computations. Take as an example (**?**):

if (¬ *x*) **then** *E1* **else** *E2*

"No decent compiler would actually negate the value of **x** at runtime! [...] After desugaring the conditional, and inlining the definition of **not**, we get" (**?**):

case (**case** *x* **of**
  True → *False*
  False → *True*
) **of**
  True → *E1*
  False → *E2*

With case-of-case transformation this gets transformed to:

case *x* **of**
  True → **case** *False* **of**
    True → *El*
    False → *E2*
  False → **case** *True* **of**
    True → *El*
    False → *E2*

[1] This specific example was retrieved from: https://stackoverflow.com/questions/35815503/what-ghc-optimization-is-responsible-for-duplicating-case-expressions

Then the case-of-known-constructor transformation gets us:

```
case x of
  True → E2
  False → E1
```

No more runtime evaluation of not!

ADD A (LARGE) EXAMPLE AS TO WHY THIS IS SO IMPORTANT FOR THE FUSED PIPELINES

3 Haskell Optimizations

In **?**'s work there were still multiple open questions left regarding the exact mechanics of what Church and Cochurch encodings did while making their way through the compiler. Why are Cochurch encodings faster in some pipelines, but slower in others?

In this section I'll describe my work replicating the fused Haskell code of the **?**'s work and further optimization opportunities that were discovered along the way.

I'll start off with the existing working code, followed by a discussion of the discoveries made throughout the process of writing, replicating, and further optimization of **?**'s example code.

3.1 Leaf Trees

In this section, the replication of **?**'s code is described. We start with his motivating example at the begginning of the paper, followed by the 'fused' version that we want the pipeline to become, once compiled:

```
f :: (Int, Int) → Int
f = sum1 . map1 (+1) . filter1 odd . between1

f' :: (Int, Int) → Int
f' (x, y) = loop x
  where
    loop x | x > y = 0
           | otherwise = if odd x
                         then (x + 1) + loop (x + 1)
                         else loop (x + 1)
```

Datatypes In his paper **?** implemented his example functions using leaf trees, this is defined as `Tree` below. Furthermore, the base functor of `Tree` was defined, as `Tree_`, with the recursive positions of the functor turned into a paramater of the datatype:

```
data Tree a = Empty | Leaf a | Fork (Tree a) (Tree a)
data Tree_ a b = Empty_ | Leaf_ a | Fork_ b b
```

Church-encoding The Church encoding of the `Tree` datatype is defined, using the base functor:

```
data TreeCh a = TreeCh (∀ b . (Tree_ a b → b) → b)
```

Next, the conversion functions `toCh` and `fromCh` are defined, using two auxillary functions `fold` and `in'`:

```
toCh :: Tree a → TreeCh a
toCh t = TreeCh (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))
fromCh :: TreeCh a → Tree a
fromCh (TreeCh 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
```

From here, the fusion rule is defined using a `RULES` pragma. Along with a couple of other rules, this core construct is responsible for doing the actual 'fusion'. The `INLINE` pragmas are also included, to delay any inlining of the `toCh/fromCh` functions to the latest possible moment, maximising the opportunity for fusion throughout the compilation process:

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

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

A generalized natural transformation function is defined:

```
natCh :: (∀ c . Tree_ a c → Tree_ b c) → TreeCh a → TreeCh b
natCh f (TreeCh g) = TreeCh (λa → g (a . f))
```

Cochurch-encoding Conversely, the cochurch encoding is defined, again using the base functor for `Tree`:

```
data TreeCoCh a = ∀ s . TreeCoCh (s → Tree_ a s) s
```

Next, the conversion functions `toCoCh` and `fromCoCh` are again defined, using two auxillary functions `out` and `unfold`:

```
toCoCh :: Tree a → TreeCoCh a
toCoCh = TreeCoCh out
out Empty = Empty_
out (Leaf a) = Leaf_ a
out (Fork l r) = Fork_ l r
fromCoCh :: TreeCoCh a → Tree a
fromCoCh (TreeCoCh 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)
```

Similar to Church-encodings, the proper pragmas are included to enable fusion:

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

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

A generalized natural transformation function is defined:

```
natCoCh :: (∀ c . Tree_ a c → Tree_ b c) → TreeCoCh a → TreeCoCh b
natCoCh f (TreeCoCh h s) = TreeCoCh (f . h) s
```

Between Three between functions are implemented: One regular, one church-encoded, and one cochurch encoded. Note how all three final functions are accompanied by an `INLINE` pragma. This inlining enables pairs of `toCh ∘ fromCh` to be revealed to the compiler for fusion. The regular one is implemented recursively in a fashion appropriate for leaf trees:

```
between1 :: (Int, Int) → Tree Int
between1 (x, y) = case compare x y of
  GT → Empty
  EQ → Leaf x
  LT → Fork (between1 (x, mid))
            (between1 (mid + 1, y))
  where mid = (x + y) `div` 2
```

The church-encoded version leverages the implementation of a recursion principle `b` for the between function of leaf trees:

```
b :: (Tree_ Int b → b) → (Int, Int) → b
b a (x, y) = case compare x y of
  GT → a Empty_
  EQ → a (Leaf_ x)
  LT → a (Fork_ (b a (x, mid))
                (b a (mid + 1, y)))
  where mid = (x + y) `div` 2
betweenCh :: (Int, Int) → TreeCh Int
betweenCh (x, y) = TreeCh (λa → b a (x, y))
between2 :: (Int, Int) → Tree Int
between2 = fromCh . betweenCh
 {-# INLINE between2 #-}
```

The cochurch-encoded version leverages the implementation of a coalgebra `h` for the between function of leaf trees:

```
h :: (Int, Int) → Tree_ Int (Int, Int)
h (x, y) = case compare x y of
  GT → Empty_
  EQ → Leaf_ x
  LT → Fork_ (x, mid) (mid + 1, y)
  where mid = (x + y) `div` 2
between3 :: (Int, Int) → Tree Int
between3 = fromCoCh . TreeCoCh h
 {-# INLINE between3 #-}
```

Filter Again three versions, similar to between. The regular implementation is as to be expected, leveraging an implementation of append:

```
filter1 :: (a → Bool) → Tree a → Tree a
filter1 p Empty = Empty
filter1 p (Leaf a) = if p a then Leaf a else Empty
filter1 p (Fork l r) = append1 (filter1 p l) (filter1 p r)
```

While for the (co)church-encoded versions a natural transformation `filt` is constructured. This is used to both implement both the church and cochurch-encoded function:

```
filt :: (a → Bool) → Tree_ a c → Tree_ a c
filt p Empty_ = Empty_
filt p (Leaf_ x) = if p x then Leaf_ x else Empty_
filt p (Fork_ l r) = Fork_ l r
filter2 :: (a → Bool) → Tree a → Tree a
filter2 p = fromCh . natCh (filt p) . toCh
 {-# INLINE filter2 #-}
filter3 :: (a → Bool) → Tree a → Tree a
filter3 p = fromCoCh . natCoCh (filt p) . toCoCh
 {-# INLINE filter3 #-}
```

Map The map function is implemented similarly to filter: A simple implementation for the non-encoded version and a single natural transformation that is leveraged in both the church- and cochurch-encoded versions:

```
map1 :: (a → b) → Tree a → Tree b
map1 f Empty = Empty
map1 f (Leaf a) = Leaf (f a)
map1 f (Fork l r) = append1 (map1 f l) (map1 f r)
m :: (a → b) → Tree_ a c → Tree_ b c
m f Empty_ = Empty_
m f (Leaf_ a) = Leaf_ (f a)
m f (Fork_ l r) = Fork_ l r
```

```
map2 :: (a → b) → Tree a → Tree b
map2 f = fromCh . natCh (m f) . toCh
 {-# INLINE map2 #-}
map3 :: (a → b) → Tree a → Tree b
map3 f = fromCoCh . natCoCh (m f) . toCoCh
 {-# INLINE map3 #-}
```

Sum The sum function is again more interesting, it is again implemented in three different ways: The non-encoded version is again as would normally be expected for leaf trees:

```
sum1 :: Tree Int → Int
sum1 Empty = 0
sum1 (Leaf x) = x
sum1 (Fork x y) = sum1 x + sum1 y
```

The church encoded version leverages an alagebra `s`:

```
s :: Tree_ Int Int → Int
s Empty_ = 0
s (Leaf_ x) = x
s (Fork_ x y) = x + y
sumCh :: TreeCh Int → Int
sumCh (TreeCh g) = g s
sum2 :: Tree Int → Int
sum2 = sumCh . toCh
 {-# INLINE sum2 #-}
```

The cochurch encoding is defined using a coinduction principle. Note that it is possible to implement this function using an accumulator of a list datatype (used like a queue), but it currently does not seem to provide a fused Core AST, for a more expansive discussion on tail-recursive cochurch-encoded pipelines, see **??**:

```
sumCoCh :: TreeCoCh Int → Int
sumCoCh (TreeCoCh h s') = loop s'
  where loop s = case h s of
                   Empty_ → 0
                   Leaf_ x → x
                   Fork_ l r → loop l + loop r
sum3 :: Tree Int → Int
sum3 = sumCoCh . toCoCh
 {-# INLINE sum3 #-}
```

Pipelines Finally the pipelines, whose performance can be measure or Core representation inspected, are defined below:

```
pipeline1 = sum1 . map1 (+2) . filter1 odd . between1
pipeline2 = sum2 . map2 (+2) . filter2 odd . between2
pipeline3 = sum3 . map3 (+2) . filter3 odd . between3

input = (1, 10000)
main = print (pipeline3 input)
```

3.2 Lists

In this section further replication of **?**'s work is described, but Lists are implemented instead of Leaf trees.

This was done to see how the descriptions in **?**'s work generalize and to have a simpler datastructure on which to perform analysis; seeing how and when the fusion works and when it doesn't.

We again start with the datatype descriptions. We use `List'` instead of `List` as there is a namespace collision with GHC's `List` datatype:

```
import GHC.List
data List' a = Nil | Cons a (List' a)
data List_ a b = Nil_ | Cons_ a b
```

(Co)Church-encodings The church encoding, proper encoding and decoding functions, and fusion pragmas are defined:

```
data ListCh a = ListCh (∀ b . (List_ a b → b) → b)
toCh :: List' a → ListCh a
toCh t = ListCh (λa → fold a t)
fold :: (List_ a b → b) → List' a → b
fold a Nil = a Nil_
fold a (Cons x xs) = a (Cons_ x (fold a xs))
fromCh :: ListCh a → List' a
fromCh (ListCh fold') = fold' in'
in' :: List_ a (List' a) → List' a
in' Nil_ = Nil
in' (Cons_ x xs) = Cons x xs
```

We introduce three important pragmas. One is the actual fusion rule, taking two functions and removing them from the compilation process. The second and third are to make sure that the `toCh` and `fromCh` functions are inlined as late as possible; maximising the time that they can be fused during the compilation process:

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

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

A generalized natural transformation function is defined:

```
natCh :: (∀ c . List_ a c → List_ b c) → ListCh a → ListCh b
natCh f (ListCh g) = ListCh (λa → g (a . f))
```

The cochurch encodings are defined similarly, including pragmas necessary for fusion:

```
data ListCoCh a = ∀ s . ListCoCh (s → List_ a s) s
toCoCh :: List' a → ListCoCh a
toCoCh = ListCoCh out
out :: List' a → List_ a (List' a)
out Nil = Nil_
out (Cons x xs) = Cons_ x xs
fromCoCh :: ListCoCh a → List' a
fromCoCh (ListCoCh h s) = unfold h s
unfold :: (b → List_ a b) → b → List' a
unfold h s = case h s of
  Nil_ → Nil
  Cons_ x xs → Cons x (unfold h xs)
{-# RULES "toCh/fromCh fusion" forall x. toCoCh (fromCoCh x) = x #-}

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

A generalized natural transformation function is defined:

```
natCoCh :: (∀ c . List_ a c → List_ b c) → ListCoCh a → ListCoCh b
natCoCh f (ListCoCh h s) = ListCoCh (f . h) s
```

Between The between function is defined in three different fashions: Normally, with the Church-encoding, and with the Cochurch encoding. We leverage `INLINE` pragmas to make sure that the fusion pragmas can effectively work. For the non-encoded implementation, we simply leverage recursion:

```
between1 :: (Int, Int) → List' Int
between1 (x, y) = case x > y of
  True → Nil
  False → Cons x (between1 (x + 1, y))
 {-# INLINE between1 #-}
```

For the Church-encoded version we define a recursion principle `b` and use that to define the encoded church function:

```
b :: (List_ Int b → b) → (Int, Int) → b
b a (x, y) = loop x
  where loop x = case x > y of
          True → a Nil_
          False → a (Cons_ x (loop (x + 1)))
betweenCh :: (Int, Int) → ListCh Int
betweenCh (x, y) = ListCh (λa → b a (x, y))
between2 :: (Int, Int) → List' Int
between2 = fromCh . betweenCh
 {-# INLINE between2 #-}
```

For the Cochurch-encoded version we define a coalgebra:

```
betweenCoCh :: (Int, Int) → List_ Int (Int, Int)
betweenCoCh (x, y) = case x > y of
  True → Nil_
  False → Cons_ x (x + 1, y)
between3 :: (Int, Int) → List' Int
between3 = fromCoCh . ListCoCh betweenCoCh
 {-# INLINE between3 #-}
```

Filter The filter function is, again, implemented in three different ways: In a non-encoded fashion, using a church-encoding, and using a cochurch-encoding. The non-encoded function simply uses recursion:

```
filter1 :: (a → Bool) → List' a → List' a
filter1 _ Nil = Nil
filter1 p (Cons x xs) = if p x then Cons x (filter1 p xs) else filter1 p xs
 {-# INLINE filter1 #-}
```

For the Church and Cochurch encoding see the extended discussion in **??**.

Map Contrary to filter, it is possible to implement the map function as a natural transformation. Again three implementations, the latter two of which leverage the defined natural transformation `m`:

```
map1 :: (a → b) → List' a → List' b
map1 _ Nil = Nil
map1 f (Cons x xs) = Cons (f x) (map1 f xs)
 {-# INLINE map1 #-}
m :: (a → b) → List_ a c → List_ b c
m f (Cons_ x xs) = Cons_ (f x) xs
m _ Nil_ = Nil_
map2 :: (a → b) → List' a → List' b
map2 f = fromCh . natCh (m f) . toCh
 {-# INLINE map2 #-}
map3 :: (a → b) → List' a → List' b
map3 f = fromCoCh . natCoCh (m f) . toCoCh
 {-# INLINE map3 #-}
```

Sum We define our sum function in, *again* three different ways: non-encoded, church-encoded, and cochurch-encoded. The non-encoded leverages simple recursion:

```
sum1 :: List' Int → Int
sum1 Nil = 0
sum1 (Cons x xs) = x + sum1 xs
 {-# INLINE sum1 #-}
```

The church-encoded function leverages an algebra and applies that the existing recursion principle:

```
su :: List_ Int Int → Int
su Nil_ = 0
su (Cons_ x y) = x + y
sumCh :: ListCh Int → Int
sumCh (ListCh g) = g su
sum2 :: List' Int → Int
sum2 = sumCh . toCh
 {-# INLINE sum2 #-}


su2' :: List_ Int (Int → Int) → (Int → Int)
su2' Nil_ s = s
su2' (Cons_ x y) s = y (s + x)
sumCh' :: ListCh Int → (Int → Int)
sumCh' (ListCh g) = g su2'
sum7 :: List' Int → Int
sum7 = flip sumCh' 0 . toCh
 {-# INLINE sum7 #-}
```

The cochurch-encoded function implements a corecursion principle and applies the existing coalgebra (and input) to it:

```
 {- TAIL RECURSION!!! -}
su2 :: (s → List_ Int s) → s → Int
su2 h s = loopt s 0
  where loopt s' sum = case h s' of
            Nil_ → sum
            Cons_ x xs → loopt xs (x + sum)
su3 :: (s → List_ Int s) → s → Int
su3 h s = loop s
  where loop s' = case h s' of
            Nil_ → 0
            Cons_ x xs → x + loop xs
sumCoCh :: ListCoCh Int → Int
sumCoCh (ListCoCh h s) = su2 h s
sumCoCh2' :: ListCoCh Int → Int
sumCoCh2' (ListCoCh h s) = su3 h s
sum3 :: List' Int → Int
sum3 = sumCoCh . toCoCh
 {-# INLINE sum3 #-}
sum8 :: List' Int → Int
sum8 = sumCoCh . toCoCh
 {-# INLINE sum8 #-}
```

Note that two subfunctions are provided to `su'`, the `loop` and the `loopt` function. The former function is implement as one would naively expect. The latter, interestingly, is implemented using tail-recursion. Because this `loopt` function constitutes a corecursion principle, all the algebras (or natural transformations) applied to it, will be inlined in such a way that the resultant function is also tail recursive, in some cases providing a significant speedup! For more details, see the discussion in **??**.

Pipelines and GHC list fusion

```
trodd :: Int → Bool
trodd n = n `rem` 2 ≡ 0
```

```
{-# INLINE trodd #-}


pipeline1 = sum1 . map1 (+2) . filter1 trodd . between1
pipeline2 = sum2 . map2 (+2) . filter2 trodd . between2
pipeline7 = sum7 . map2 (+2) . filter2 trodd . between2
pipeline3 = sum3 . map3 (+2) . filter3 trodd . between3
pipeline8 = sum8 . map3 (+2) . filter3 trodd . between3
pipeline4 (x, y) = loop x 0
  where loop z sum = if z > y
                     then sum
                     else if trodd z
                          then loop (z + 1) (sum + z + 2)
                          else loop (z + 1) sum

between5 :: (Int, Int) -> [Int]
between5 (x, y) = [x .. y]
 {-# INLINE between5 #-}
filter5 :: (Int -> Bool) -> [Int] -> [Int]
filter5 f xs = build (λc n -> foldr (λa b -> if f a then c a b else b) n xs)
 {-# INLINE filter5 #-}
map5 :: ∀ a b . (a -> b) -> [a] -> [b]
map5 f xs = build (λc n -> foldr (λa b -> c (f a) b) n xs)
 {-# INLINE map5 #-}
sum5 :: [Int] -> Int
sum5 = foldl' (λa b -> a + b) 0
 {-# INLINE sum5 #-}
pipeline5 = sum5 . map5 (+2) . filter5 trodd . between5
pipeline6 = sum6 . map6 (+2) . filter6 trodd . between6
pipeline9 = sum9 . map6 (+2) . filter6 trodd . between6
```

3.2.1 The Filter Problem

I have moved the discussion for Church and Cochurch encoded Lists down here, as I think it warrants more discussion and illustrates a few interesting points. There are multiple ways of implementing it, none of them trivial according to **?**'s description of how it should be implemented as a natural transformation.

When replicating **?**'s code for lists, there is one major limitation on natrual transformation functions: How to represent filter as a natural transformation for both Church and Cochurch encodings? In his work he implemented, using Leaf trees, a natural transformation for the filter function in the following manner:

```
filt :: (a -> Bool) -> Tree_ a c -> Tree_ a c
filt p Empty_ = Empty_
filt p (Leaf_ x) = if p x then Leaf_ x else Empty_
filt p (Fork_ l r) = Fork_ l r
filter2 :: (a -> Bool) -> Tree a -> Tree a
filter2 p = fromCh . natCh (filt p) . toCh
filter3 :: (a -> Bool) -> Tree a -> Tree a
filter3 p = fromCoCh . natCoCh (filt p) . toCoCh
```

This `filt` function was then subsequently used in the Church and Cochurch encoded function. Let's try this for the `List` datatype:

```
filt :: (a -> Bool) -> List_ a c -> List_ a c
filt p Nil_ = Nil_
filt p (Cons_ x xs) = if p x then Cons_ x xs else?
```

The question is, what should be in the place of the `?` above? Initially you might say `xs`, as the `Cons_ x` part should be filtered away, and this would be conceptually correct except for the fact that `xs` is of type `c`, and not of type `List_ a c`. Filling in `xs` gives a type error. Let's change the type annotation then,

right? Well no, if we did that we wouldn't have the type of a transformation anymore, so we can't do that either.

There are two solutions: One that modifies the definition of `filter2` and `filter3`, such that the definition is still possible, without leveraging transformations. The other modifies the definition of the underlying type such that the filter function is still possible to express as a transformation.

Solution 1: Abandoning Natural Transformations

Church Whereas before we wanted to implement our `filter` function in the following manner:

```
filterCh :: (∀ c . List_ a c → List_ b c) → ListCh a → ListCh b
filterCh p (ListCh g) = ListCh (λa → g (a . (filt p)))
filter2 :: (a → Bool) → List a → List a
filter2 p = fromCh . filterCh p . toCh
```

We now need to modify the `filterCh` function such that we can still express a filter function *witout* using a natural transformation:

```
filterCh :: (∀ c . List_ a c → List_ b c) → ListCh a → ListCh b
filterCh p (ListCh g) = ListCh (λa → g?)
```

Replacing the `?` above such that we apply the `a` selectively we can yield:

```
filterCh :: (a → Bool) → ListCh a → ListCh a
filterCh p (ListCh g) = ListCh (λa → g (λcase
    Nil_ → a Nil_
    Cons_ x xs → if (p x) then a (Cons_ x xs) else xs
  ))
filter2 :: (a → Bool) → List' a → List' a
filter2 p = fromCh . filterCh p . toCh
 {-# INLINE filter2 #-}
```

Notice how we do not apply `a` to `xs`, and, in doing so, can put `xs` in the place where wanted to. The definition of `filterCh` was too restrictive in always postcomposing `a`.

The astute observer will note that this solution is just a beta reduced form of a build/foldr composition pair!

Cochurch Whereas before we wanted to implement our `filter` function in the following manner:

```
filter3 :: (a → Bool) → List a → List a
filter3 p = fromCoCh . natCoCh (filt p) . toCoCh
```

For the cochurch-encoding, a natural transformation can be defined, but it is not a simple algebra, instead it is a recursive function.[2] The core idea is: we combine the natural transformation and postcomposition again, but this time we make the function recursively grab elements from the seed until we find one that satisfies the predicate.

```
filt :: (a → Bool) → (s → List_ a s) → s → List_ a s
filt p h s = go s
  where go s = case h s of
          Nil_ → Nil_
          Cons_ x xs → if p x then Cons_ x xs else go xs
filterCoCh :: (a → Bool) → ListCoCh a → ListCoCh a
filterCoCh p (ListCoCh h s) = ListCoCh (filt p h) s
filter3 :: (a → Bool) → List' a → List' a
filter3 p = fromCoCh . filterCoCh p . toCoCh
 {-# INLINE filter3 #-}
```

The `go` subfunction is recursive, so it does not inline (fuse) neatly into the main function body in the way that the rest of the pipeline does. There is existing work, called join-point optimization that should enable this function to still fully fuse, but it does not at the moment. There are existing issues in GHC's issue tracker that describe this problem.[3]

[2] And not necessarily guaranteed to terminate, the seed could generate an infinite structure.

[3] https://gitlab.haskell.org/ghc/ghc/-/issues/22227

Solution 2: go back and modify the underlying type It is possible to implement filter using a natural transformation, but this requires us to modify the type of the base functor. We can add a new constructor to the datatype that allows us to null out the value of our datatype: `ConsN'_ xs`. This way we can write the `filt` function in the following fashion:

$$
\begin{aligned}
&\mathit{filt'} :: (a \to \mathit{Bool}) \to \mathit{List'}\ \_\ a\ c \to \mathit{List'}\ \_\ a\ c \\
&\mathit{filt'}\ p\ \mathit{Nil'}\ \_ = \mathit{Nil'}\ \_ \\
&\mathit{filt'}\ p\ (\mathit{ConsN'}\ \_\ \mathit{xs}) = \mathit{ConsN'}\ \_\ \mathit{xs} \\
&\mathit{filt'}\ p\ (\mathit{Cons'}\ \_\ x\ \mathit{xs}) = \textbf{if}\ p\ x\ \textbf{then}\ \mathit{Cons'}\ \_\ x\ \mathit{xs}\ \textbf{else}\ \mathit{ConsN'}\ \_\ \mathit{xs}
\end{aligned}
$$

Now we do need to modify all of our already defined functions to take into account this modified datatype. The astute among you might notice that this technique is actually *stream fusion* as described by **?**. The `ConsN_` constructor is analogous to the `Skip` constructor. Therefore, this is a known and understood technique, motivated by the limitations of the techniques described by Harper.

So why was it possible to implement `filt` without modifying the datatype of leaf trees? Because leaf trees already have this consideration of being able to null the datatype in-place by chaining a `Leaf_ x` into an `Empty_`. `filt` is able to remove a value from the datastructure without changing the structure of the data. I.e. it is still a natural transformation. By chaning the list datatype such that this nullability is also possible, we can also write `filt` as a natural transformation.

This technique could be broader than a modification to just lists. By modifying (making nullable) any datatype, it might be possible to broaden the class of functions that can be represented as a natural transformation. One other example of this is already the difference between a `Binary Tree` and a `Leaf Tree` datatype:

$$
\begin{aligned}
&\textbf{data}\ \mathit{BinTree}\ a = \mathit{Leaf}\ a \mid \mathit{Fork}\ (\mathit{BinTree}\ a)\ (\mathit{BinTree}\ a) \\
&\textbf{data}\ \mathit{LeafTree}\ a = \mathit{Empty} \mid \mathit{Leaf}\ a \mid \mathit{Fork}\ (\mathit{LeafTree}\ a)\ (\mathit{LeafTree}\ a)
\end{aligned}
$$

The `Leaf` constructor of `BinTree` is also made nullable. I will leave the following question to future work: Is this generalizable?

3.2.2 Tail Recursion

Definition We call a recursive function tail-recursive, if all its recursive calls are returned immediately upon completion i.e., they don't do any additional calculations upon the result of the recursive call before returning a result.

When a function is tail-recursive, it is possible to reuse the stack frame of the current function call, reducing a lot of memory overhead. Haskell is able to identify tail-recursive functions and optimize the compiled byte code accordingly.

Example The following code, when applying fusion, case-of-case, and case-of-known-case optimization:

$$\mathit{sumCoCh}\ .\ \mathit{mapCoCh}\ (+2)\ .\ \mathit{filterCoCh}\ \mathit{odd}\ .\ \mathit{ListCoCh}\ \mathit{betweenCoCh}$$

Reduces to (See **??** for derivation):

$$
\begin{aligned}
&\mathit{loop}\ (x, y) = \textbf{if}\ (x > y) \\
&\qquad\qquad \textbf{then}\ 0 \\
&\qquad\qquad \textbf{else if}\ (\mathit{odd}\ x) \\
&\qquad\qquad\qquad \textbf{then}\ (x + 2) + \mathit{loop}\ (x + 1, y) \\
&\qquad\qquad\qquad \textbf{else}\ \mathit{loop}\ (x + 1, y) \\
&\mathit{loop}\ (x, y)
\end{aligned}
$$

This definition is not tail recursive as the `then (x + 2) + loop (x+1, y)` line includes some calcualtions that still need to made upon completion of the recursive `loop` call; i.e. the `loop` function is not in tail position.

If we tweak the definition of sum, such that it is tail recursive we get a different derivation (See **??** for derivation):

$$\mathit{sumCoCh}\ .\ \mathit{mapCoCh}\ (+2)\ .\ \mathit{filterCoCh}\ \mathit{odd}\ .\ \mathit{ListCoCh}\ \mathit{betweenCoCh}$$

Reduces to:

$$
\begin{aligned}
&\mathit{loop}\ (x, y)\ \mathit{acc} = \textbf{if}\ (x > y) \\
&\qquad\qquad\qquad\quad \textbf{then}\ \mathit{acc} \\
&\qquad\qquad\qquad\quad \textbf{else if}\ (\mathit{odd}\ x) \\
&\qquad\qquad\qquad\qquad\quad \textbf{then}\ \mathit{loop}\ (x + 1, y)\ ((x + 2) + \mathit{acc}) \\
&\qquad\qquad\qquad\qquad\quad \textbf{else}\ \mathit{loop}\ (x + 1, y) \\
&\mathit{loop}\ (x, y)\ 0
\end{aligned}
$$

Which is identical except for the fact that `loop` is tail-recursive. All that has been changed is the recursion principle `su'`.

Cochurch encodings better lend themselves to having fully tail-recursive fused pipelines, as writing a coinduction principle that is tail-recursive is easier than writing a recursion principle that is. For a further discussion on this, see **?**'s work.

3.3 Performance Comparison

PERFORMANCE DISCUSSION

Throughout the development process I made use of the tool tastybench[4] and analyzed the dumped core representation generated by GHC[5]. The process went through the following steps:

- Implement List Church encoding
- Modify type to make the list 'nullable', such that filter can be implemented as a natural transformation.
- Implement Leaf Tree Church encoding.
- Analyze core representation of (partially) fused list functions. Notice that (co)recursion principle for the function pipelines ends up being the recursion structure that ends up being represented in the Core repesentation.
- Modify sum function to be tail-recursive for Cochurch encodings. Notice 40% speedup.

4 Formalization

In **?**'s work "A Library Writer's Guide to Shortcut Fusion", the practice of implementing Church and CoChurch encodings is described, as well a paper proof necessary to show that the encodings optimizations employed are correct.

In this section the work I have done to formalize these proofs in the programming language Agda is discussed, as well as additional proofs to support the claims made in the paper.

The code can be neatly presented in roughly 2 parts:

- The proofs of the category theory truths described by **?**.
- The proofs about the (Co)Church encodings, again as described by **?**.

4.1 Category Theory: Initiality

This section is about my formalization of **?**'s work that describes the needed category theory, to be leveraged later on in the fusion part of the formalization. This module defines the category of F-Algebras, initiality of μ, the universal properties of folds, and the fusion properties.

[4] https://hackage.haskell.org/package/tasty-bench

[5] https://downloads.haskell.org/ghc/latest/docs/users_guide/debugging.html#core-representation-and-simplification

4.1.1 Universal properties of catamorphisms and initiality

This module defines a function and shows it to be a catamorphism in the category of F-Agebras, by module proving some properties of catamorphisms and is showing that `(μ F, in')` is initial.

```
module agda.init.initial where
open import Data.W using () renaming (sup to in'; foldr to (|_|)) public
```

A shorthand for the Category of F-Algebras.

```
C[_]Alg : (F : Container 0ℓ 0ℓ) → Cat (lsuc 0ℓ) 0ℓ 0ℓ
C[ F ]Alg = F-Algebras F[ F ]
```

A candidate function is defined, this will be proved to be a catamorphism through the proof of initiality:

```
-(|_|) : {F : Container 0ℓ 0ℓ}{X : Set} → (⟦ F ⟧ X → X) → μ F → X
-(| a |) (in' (op , ar)) = a (op , (| a |) ∘ ar)
```

It is shown that any (|_|) is a valid F-Algebra homomorphism from `in'` to any other object `a`; i.e. the forward direction of the *universal property of folds* (**?**). This constitutes a proof of existence:

```
univ-to : {F : Container 0ℓ 0ℓ}{X : Set}{a : ⟦ F ⟧ X → X}{h : μ F → X} →
                  h ≡ (| a |) → h ∘ in' ≡ a ∘ map h
univ-to refl = refl
```

It is shown that any other valid F-Algebra homomorphism from `in'` to `a` is equal to the (|_|) function defined; i.e. the backwards direction of the *universal property of folds* (**?**). This constitutes a proof of uniqueness:

```
univ-from : {F : Container 0ℓ 0ℓ}{X : Set}(a : ⟦ F ⟧ X → X)(h : μ F → X) →
                                  h ∘ in' ≡ a ∘ map h → (x : μ F) → h x ≡ (| a |) x
univ-from a h eq (in' x@(op , ar)) = begin
    (h ∘ in') x
  ≡⟨ cong-app eq x ⟩
    (a ∘ map h) x
  ≡⟨⟩
    a (op , h ∘ ar)
  ≡⟨ cong (λ f → a (op , f)) (funext $ univ-from a h eq ∘ ar) ⟩
    a (op , (| a |) ∘ ar)
  ≡⟨⟩
    (a ∘ map (| a |)) x
  ≡⟨⟩
    ((| a |) ∘ in') x
  ■
```

The two previous proofs, constituting a proof of existence and uniqueness, are combined to show that `(μ F, in')` is initial:

```
initial-in : {F : Container 0ℓ 0ℓ} → IsInitial C[ F ]Alg (to-Algebra in')
initial-in = record { ! = λ {A} →
                                  record {
                                        f = (| α A |)
                                        ; commutes = λ {x} → cong-app (univ-to {_}{_}{α A} refl) x }
                    ; !-unique = λ {A} fhom {x} → sym $ univ-from (α A) (f fhom) (funext (λ y → commutes fhom
```

The *computation law* (**?**):

```
comp-law : {F : Container 0ℓ 0ℓ}{A : Set}(a : ⟦ F ⟧ A → A) → (| a |) ∘ in' ≡ a ∘ map (| a |)
comp-law a = refl
```

The *reflection law* (**?**):

```
reflection : {F : Container 0ℓ 0ℓ}(y : μ F) → (| in' |) y ≡ y
reflection y@(in' (op , ar)) = begin
    (| in' |) y
  ≡⟨⟩ - Dfn of (|_|)
    in' (op , (| in' |) ∘ ar)
  ≡⟨ cong (λ x → in' (op , x)) (funext (reflection ∘ ar)) ⟩
    y
  ■

reflection-law : {F : Container 0ℓ 0ℓ} → (| in' |) ≡ id
reflection-law {F} = funext (reflection {F})
```

4.1.2 Initial F-Algebra fusion

This module proves the categorical fusion property (see **??**). From it, it extracts the 'fusion law' as it was declared by **?**; which is easier to work with. This shows that the fusion law does follow from the fusion property.

```
module agda.init.fusion where
```

The categorical fusion property:

```
fusionprop : {F : Container 0ℓ 0ℓ}{A B μ : Set}{φ : ⟦ F ⟧ A → A}{ψ : ⟦ F ⟧ B → B}
             {init : ⟦ F ⟧ μ → μ}(i : IsInitial C[ F ]Alg (to-Algebra init)) →
             (f : C[ F ]Alg [ to-Algebra φ , to-Algebra ψ ]) → C[ F ]Alg [ i .! ≈ C[ F ]Alg [ f ∘ i .! ] ]
fusionprop {F} i f = i .!-unique (C[ F ]Alg [ f ∘ i .! ])
```

The 'fusion law':

```
fusion : {F : Container 0ℓ 0ℓ}{A B : Set}{a : ⟦ F ⟧ A → A}{b : ⟦ F ⟧ B → B}
         (h : A → B) → h ∘ a ≡ b ∘ map h → (| b |) ≡ h ∘ (| a |)
fusion h p = funext λ x → fusionprop initial-in (record { f = h ; commutes = λ {y} → cong-app p y }) {x}
```

4.2 Category Theory: Terminality

This module defines the category of F-CoAlgebras, a candidate terminal object ν, anamorphisms, proves terminality of ν, the universal properties of unfolds, and the fusion properties. This module is the compliment of `init`.

4.2.1 Terminal coalgebras and anamorphisms

This module defines a datatype and shows it to be initial; and a function and shows it to be an anamorphism in the category of F-Coalgebras. Specifically, it is shown that `(ν, out)` is terminal.

```
{-# OPTIONS -guardedness #-}
module agda.term.terminal where
open import Agda.Builtin.Sigma public
open import Level using (0ℓ; Level) renaming (suc to lsuc) public
open import Data.Container using (Container; ⟦_⟧; map; _▷_) public
open import Function using (_∘_; _$_; id; const) public
open import Relation.Binary.PropositionalEquality as Eq using (_≡_; refl; sym; cong; cong-app; subst) public
open Eq.≡-Reasoning public
```

A shorthand for the Category of F-Coalgebras:

```
C[_]CoAlg : (F : Container 0ℓ 0ℓ) → Cat (lsuc 0ℓ) 0ℓ 0ℓ
C[ F ]CoAlg = F-Coalgebras F[ F ]
```

A candidate terminal datatype and anamorphism function are defined, they will be proved to be so later on this module:

```
out : {F : Container 0ℓ 0ℓ} → ν F → ⟦ F ⟧ (ν F)
out nu = head nu , tail nu
-A⟦_⟧ : {F : Container 0ℓ 0ℓ}{X : Set} → (X → ⟦ F ⟧ X) → X → ν F
-out (A⟦ c ⟧ x) = let (op , ar) = c x in
-                 (op , A⟦ c ⟧ ∘ ar)
```

It is shown that any ⟦_⟧ is a valid F-Coalgebra homomorphism from `out` to any other object `a`; i.e. the forward direction of the *universal property of unfolds* **?**. This constitutes a proof of existence:

```
univ-to : {F : Container 0ℓ 0ℓ}{C : Set}(h : C → ν F){c : C → ⟦ F ⟧ C} →
            h ≡ A⟦ c ⟧ → out ∘ h ≡ map h ∘ c
univ-to _ refl = refl
```

Injectivity of the `out` constructor is postulated, I have not found a way to prove this, yet.

```
postulate out-injective : {F : Container 0ℓ 0ℓ}{x y : ν F} → out x ≡ out y → x ≡ y
-out-injective eq = funext ?
```

It is shown that any other valid F-Coalgebra homomorphism from `out` to `a` is equal to the ⟦_⟧ defined; i.e. the backward direction of the *universal property of unfolds* **?**. This constitutes a proof of uniqueness. This uses `out` injectivity. Currently, Agda's termination checker does not seem to notice that the proof in question terminates:

```
{-# NON_TERMINATING #-}
univ-from : {F : Container _ _}{C : Set}(h : C → ν F){c : C → ⟦ F ⟧ C} →
                          out ∘ h ≡ map h ∘ c → (x : C) → h x ≡ A⟦ c ⟧ x
univ-from h {c} eq x = let (op , ar) = c x in
  out-injective (begin
      (out ∘ h) x
    ≡⟨ cong (λ f → f x) eq ⟩
      (map h ∘ c) x
    ≡⟨⟩
      map h (op , ar)
    ≡⟨⟩
      (op , h ∘ ar)
    ≡⟨ cong (λ f → op , f) (funext $ univ-from h eq ∘ ar) ⟩ - induction
      (op , A⟦ c ⟧ ∘ ar)
    ≡⟨⟩ - Definition of ⟦_⟧
      (out ∘ A⟦ c ⟧) x
    ■)
```

The two previous proofs, constituting a proof of existence and uniqueness, are combined to show that `(ν F, out)` is terminal:

```
terminal-out : {F : Container 0ℓ 0ℓ} → IsTerminal C[ F ]CoAlg (to-Coalgebra out)
terminal-out = record { ! = λ {A} → record {
                                      f = A⟦ α A ⟧
                                      ; commutes = λ {x} → cong-app (univ-to A⟦ α A ⟧ {α A} refl) x }
                      ; !-unique = λ {A} fhom {x} → sym (univ-from (f fhom) {α A} (funext (λ y → commutes fho
```

The *computation law* **?**:

```
computation-law : {F : Container 0ℓ 0ℓ}{C : Set}(c : C → ⟦ F ⟧ C) → out ∘ A⟦ c ⟧ ≡ map A⟦ c ⟧ ∘ c
computation-law c = refl
```

The *reflection law* **?**: SOMETHING ABOUT TERMINATION.

```
{-# NON_TERMINATING #-}
reflection : {F : Container 0ℓ 0ℓ}(x : ν F) → A⟦ out ⟧ x ≡ x
```

```
reflection x = let (op , ar) = out x in
  out-injective (begin
    out (A⟦ out ⟧ x)
  ≡⟨⟩
    op , A⟦ out ⟧ ∘ ar
  ≡⟨ cong (λ f → op , f) (funext $ reflection ∘ ar) ⟩
    out x
  ■)
```

4.2.2 Terminal F-Coalgebra fusion

This module proves the categorical fusion property. From it, it extracts a 'fusion law' as it was defined by **?**; which is easier to work with. This shows that the fusion law does follow from the fusion property.

```
{-# OPTIONS -guardedness #-}
module agda.term.cofusion where
```

The categorical fusion property:

```
fusionprop : {F : Container 0ℓ 0ℓ}{C D ν : Set}
             {φ : C → ⟦ F ⟧ C}{ψ : D → ⟦ F ⟧ D}{term : ν → ⟦ F ⟧ ν}
             (i : IsTerminal C[ F ]CoAlg (to-Coalgebra term))(f : C[ F ]CoAlg [ to-Coalgebra ψ , to-Coalgebra φ ]) →
             C[ F ]CoAlg [ i .! ≈ C[ F ]CoAlg [ i .! ∘ f ] ]
fusionprop {F} i f = i .!-unique (C[ F ]CoAlg [ i .! ∘ f ])
```

The 'fusion law':

```
fusion : {F : Container 0ℓ 0ℓ}{C D : Set}
         {c : C → ⟦ F ⟧ C}{d : D → ⟦ F ⟧ D}(h : C → D) →
         d ∘ h ≡ map h ∘ c → A⟦ c ⟧ ≡ A⟦ d ⟧ ∘ h
fusion h comm = funext λ x → fusionprop terminal-out (record {f = h ; commutes = λ {y} → cong-app comm y}) {
```

4.3 Fusion: Church encodings

This section focuses on the fusion of Church encodings, leveraging parametricity (free theorems).

4.3.1 Definition of Church encodings

This module defines Church encodings and the two conversions `con` and `abs`, called `toCh` and `fromCh` here, respectively. It also defines the generalized producing, transformation, and consuming functions, as described by **?**.

```
module agda.church.defs where
```

The church encoding, leveraging containers:

```
data Church (F : Container 0ℓ 0ℓ) : Set₁ where
  Ch : ({X : Set} → (⟦ F ⟧ X → X) → X) → Church F
```

The conversion functions:

```
toCh : {F : Container _ _} → μ F → Church F
toCh {F} x = Ch (λ {X : Set} → λ (a : ⟦ F ⟧ X → X) → ⦇ a ⦈ x)

fromCh : {F : Container _ _} → Church F → μ F
fromCh (Ch g) = g in'
```

The generalized and encoded producing, transformation, and consuming functions, alongside proofs that they are equal to the functions they are encoding. First the producing function, this is a generalized version of **?**'s `build` function:

```
prodCh : {ℓ : Level}{F : Container _ _}{Y : Set ℓ}
         (g : {X : Set} → (⟦ F ⟧ X → X) → Y → X)(y : Y) → Church F
prodCh g x = Ch (λ a → g a x)

prod : {ℓ : Level}{F : Container _ _}{Y : Set ℓ}
       (g : {X : Set} → (⟦ F ⟧ X → X) → Y → X)(y : Y) → μ F
prod g = fromCh ∘ prodCh g

eqProd : {F : Container _ _}{Y : Set}
         {g : {X : Set} → (⟦ F ⟧ X → X) → Y → X} → prod g ≡ g in'
eqProd = refl
```

Second, the natural transformation function:

```
natTransCh : {F G : Container _ _}
             (nat : {X : Set} → ⟦ F ⟧ X → ⟦ G ⟧ X) → Church F → Church G
natTransCh nat (Ch g) = Ch (λ a → g (a ∘ nat))

natTrans : {F G : Container _ _}
           (nat : {X : Set} → ⟦ F ⟧ X → ⟦ G ⟧ X) → μ F → μ G
natTrans nat = fromCh ∘ natTransCh nat ∘ toCh

eqNatTrans : {F G : Container _ _}
             {nat : {X : Set} → ⟦ F ⟧ X → ⟦ G ⟧ X} →
             natTrans nat ≡ ⦇ in' ∘ nat ⦈
eqNatTrans = refl
```

Third, the consuming function, note that this is a generalized version of **?**'s `foldr` function.

```
consCh : {F : Container _ _}{X : Set}
         (c : ⟦ F ⟧ X → X) → Church F → X
consCh c (Ch g) = g c

cons : {F : Container _ _}{X : Set}
       (c : ⟦ F ⟧ X → X) → μ F → X
cons c = consCh c ∘ toCh

eqCons : {F : Container _ _}{X : Set}
         {c : ⟦ F ⟧ X → X} → cons c ≡ ⦇ c ⦈
eqCons = refl
```

4.3.2 Proof obligations

In **?**'s work, five proofs proofs are given for Church encodings. These are formalized in this module.

```
module agda.church.proofs where
```

The first proof proves that `fromCh ∘ toCh = id`, using the reflection law:

```
from-to-id : {F : Container 0ℓ 0ℓ} → fromCh ∘ toCh {F} ≡ id
from-to-id {F} = funext λ x → begin
    fromCh (toCh x)
  ≡⟨⟩ - Definition of toCh
    fromCh (Ch (λ {X}a → ⦇ a ⦈ x))
  ≡⟨⟩ - Definition of fromCh
    (λ a → ⦇ a ⦈ x) in'
  ≡⟨⟩ - function application
    ⦇ in' ⦈ x
  ≡⟨ reflection x ⟩
    x
  ■
```

The second proof is similar to the first, but it proves the composition in the other direction `toCh ∘ fromCh = id`. This proofs leverages parametricity as described by **?**. It postulates the free theorem of the function `g : forall A . (F A -> A) -> A`, to prove that "applying `g` to `b` and then passing the result to `h`, is the same as just folding `c` over the datatype" (**?**):

```
postulate free : {F : Container 0ℓ 0ℓ}{B C : Set}{b : ⟦ F ⟧ B → B} {c : ⟦ F ⟧ C → C}
                 (h : B → C)(g : {X : Set} → (⟦ F ⟧ X → X) → X) →
                 h ∘ b ≡ c ∘ map h → h (g b) ≡ g c
fold-invariance : {F : Container 0ℓ 0ℓ}{Y : Set}
                  (g : {X : Set} → (⟦ F ⟧ X → X) → X)(a : ⟦ F ⟧ Y → Y) →
                  ⦇ a ⦈ (g in') ≡ g a
fold-invariance g a = free ⦇ a ⦈ g refl

to-from-id : {F : Container 0ℓ 0ℓ} → toCh ∘ fromCh {F} ≡ id
to-from-id {F} = funext λ where
  (Ch g) → begin
      toCh (fromCh (Ch g))
    ≡⟨⟩ - definition of fromCh
      toCh (g in')
    ≡⟨⟩ - definition of toCh
      Ch (λ{X}a → ⦇ a ⦈ (g in'))
    ≡⟨ cong Ch (funexti λ{Y} → funext (fold-invariance g)) ⟩
      Ch g
    ■
```

The third proof shows church-encoded functions constitute an implementation for the consumer functions being replaced. The proof is proved via reflexivity, but **?**'s original proof steps are included here for completeness:

```
cons-pres : {F : Container 0ℓ 0ℓ}{X : Set}(b : ⟦ F ⟧ X → X) →
            consCh b ∘ toCh ≡ ⦇ b ⦈
cons-pres {F} b = funext λ (x : μ F) → begin
    consCh b (toCh x)
  ≡⟨⟩ - definition of toCh
    consCh b (Ch (λ a → ⦇ a ⦈ x))
  ≡⟨⟩ - function application
    (λ a → ⦇ a ⦈ x) b
  ≡⟨⟩ - function application
    ⦇ b ⦈ x
  ■
```

The fourth proof shows that church-encoded functions constitute an implementation for the producing functions being replaced. The proof is proved via reflexivity, but **?**'s original proof steps are included here for completeness:

```
prod-pres : {F : Container 0ℓ 0ℓ}{X : Set}(f : {Y : Set} → (⟦ F ⟧ Y → Y) → X → Y) →
            fromCh ∘ prodCh f ≡ f in'
prod-pres {F}{X} f = funext λ (s : X) → begin
    fromCh ((λ (x : X) → Ch (λ a → f a x)) s)
  ≡⟨⟩ - function application
    fromCh (Ch (λ a → f a s))
  ≡⟨⟩ - definition of fromCh
    (λ {Y : Set} (a : ⟦ F ⟧ Y → Y) → f a s) in'
  ≡⟨⟩ - function application
    f in' s
  ■
```

The fifth, and final proof shows that church-encoded functions constitute an implementation for the conversion functions being replaced. The proof again leverages the free theorem defined earlier:

```
trans-pres : {F G : Container 0ℓ 0ℓ} (f : {X : Set} → ⟦ F ⟧ X → ⟦ G ⟧ X) →
             fromCh ∘ natTransCh f ≡ (| in' ∘ f |) ∘ fromCh
trans-pres f = funext λ where
  (Ch g) → begin
      fromCh (natTransCh f (Ch g))
    ≡⟨⟩ - Function application
      fromCh (Ch (λ a → g (a ∘ f)))
    ≡⟨⟩ - Definition of fromCh
      (λ a → g (a ∘ f)) in'
    ≡⟨⟩ - Function application
      g (in' ∘ f)
    ≡⟨ sym (fold-invariance g (in' ∘ f)) ⟩
      (| in' ∘ f |) (g in')
    ≡⟨⟩ - Definition of fromCh
      (| in' ∘ f |) (fromCh (Ch g))
    ■
```

Finally two additional proofs were made to clearly show that any pipeline made using church encodings will fuse down to a simple function application. The first of these two proofs shows that any two composed natural transformation fuse down to one single natural transformation:

```
natfuse : {F G H : Container 0ℓ 0ℓ}
          (nat1 : {X : Set} → ⟦ F ⟧ X → ⟦ G ⟧ X) →
          (nat2 : {X : Set} → ⟦ G ⟧ X → ⟦ H ⟧ X) →
          natTransCh nat2 ∘ toCh ∘ fromCh ∘ natTransCh nat1 ≡ natTransCh (nat2 ∘ nat1)
natfuse nat1 nat2 = begin
            natTransCh nat2 ∘ toCh ∘ fromCh ∘ natTransCh nat1
          ≡⟨ cong (λ f → natTransCh nat2 ∘ f ∘ natTransCh nat1) to-from-id ⟩
            natTransCh nat2 ∘ natTransCh nat1
          ≡⟨ funext (λ where (Ch g) → refl) ⟩
            natTransCh (nat2 ∘ nat1)
          ■
```

The second of these two proofs shows that any pipeline, consisting of a producer, transformer, and consumer function, fuse down to a single function application:

```
pipefuse : {F G : Container 0ℓ 0ℓ}{X : Set}(g : {Y : Set} → (⟦ F ⟧ Y → Y) → X → Y)
           (nat : {X : Set} → ⟦ F ⟧ X → ⟦ G ⟧ X)(c : (⟦ G ⟧ X → X)) →
           cons c ∘ natTrans nat ∘ prod g ≡ g (c ∘ nat)
pipefuse g nat c = begin
    consCh c ∘ toCh ∘ fromCh ∘ natTransCh nat ∘ toCh ∘ fromCh ∘ prodCh g
```

```
  ≡⟨ cong (λ f → consCh c ∘ f ∘ natTransCh nat ∘ toCh ∘ fromCh ∘ prodCh g) to-from-id ⟩
    consCh c ∘ natTransCh nat ∘ toCh ∘ fromCh ∘ prodCh g
  ≡⟨ cong (λ f → consCh c ∘ natTransCh nat ∘ f ∘ prodCh g) to-from-id ⟩
    consCh c ∘ natTransCh nat ∘ prodCh g
  ≡⟨⟩
    g (c ∘ nat)
  ■
```

4.3.3 Example: List fusion

In order to clearly see how the Church encodings allows functions to fuse, a datatype was selected such the abstracted function, which have so far been used to prove the needed properties, can be instantiated to demonstrate how the fusion works for functions across a cocrete datatype. In this module is defined: the container, whose interpretation represents the base functor for lists, some convenience functions to make type annotations more readable, a producer function `between`, a transformation function `map`, a consumer function `sum`, and a proof that non-church and church-encoded implementations are equal.

```
module agda.church.inst.list where
```

Datatypes The index set for the container, as well as the container whose interpretation represents the base funtor for list. Note how ListOp is isomorphis to the datatype `1 + A`, I use ListOp instead to make the code more readable:

```
data ListOp (A : Set) : Set where
  nil : ListOp A
  cons : A → ListOp A
F : (A : Set) → Container _ _
F A = ListOp A ▷ λ { nil → Fin 0 ; (cons n) → Fin 1 }
```

Functions representing the run-of-the-mill list datatype and the base functor for list:

```
List : (A : Set) → Set
List A = μ (F A)
List' : (A B : Set) → Set
List' A B = ⟦ F A ⟧ B
```

Helper functions to assist in cleanly writing out instances of lists:

```
[] : {A : Set} → μ (F A)
[] = in' (nil , λ())
_::_ : {A : Set} → A → List A → List A
_::_ x xs = in' (cons x , const xs)
infixr 20 _::_
```

The fold funtion as it would normally be encountered for lists, defined in terms of ⦇_⦈:

```
fold' : {A X : Set}(n : X)(c : A → X → X) → List A → X
fold' {A}{X} n c = ⦇ (λ{(nil , _) → n; (cons n , g) → c n (g zero)}) ⦈
```

between The recursion principle `b`, which when used, represents the between function. It uses `b'` to assist termination checking:

```
b' : {B : Set} → (a : List' ℕ B → B) → ℕ → ℕ → B
b' a x zero = a (nil , λ())
b' a x (suc n) = a (cons x , const (b' a (suc x) n))
b : {B : Set} → (a : List' ℕ B → B) → ℕ × ℕ → B
b a (x , y) = b' a x (suc (y - x))
```

The functions `between1` and `between2`. The former is defined without a church-encoding, the latter with. A reflexive proof and sanity check is included to show equality:

```
between1 : ℕ × ℕ → List ℕ
between1 xy = b in' xy
between2 : ℕ × ℕ → List ℕ
between2 = prod b
eqbetween : between1 ≡ between2
eqbetween = refl
checkbetween : 2 :: 3 :: 4 :: 5 :: 6 :: [] ≡ between2 (2 , 6)
checkbetween = refl
```

map The algebra `m`, which when used in an algebra, represents the map function:

```
m : {A B C : Set}(f : A → B) → List' A C → List' B C
m f (nil , _) = (nil , λ())
m f (cons n , l) = (cons (f n) , l)
```

The functions `map1` and `map2`. The former is defined without a church-encoding, the latter with. A reflexive proof and sanity check is included to show equality:

```
map1 : {A B : Set}(f : A → B) → List A → List B
map1 f = ⟪ in' ∘ m f ⟫
map2 : {A B : Set}(f : A → B) → List A → List B
map2 f = natTrans (m f)
eqmap : {f : ℕ → ℕ} → map1 f ≡ map2 f
eqmap = refl
checkmap : (map1 (_+_ 2) (3 :: 6 :: [])) ≡ 5 :: 8 :: []
checkmap = refl
```

sum The algebra `s`, which when used in an algebra, represents the sum function:

```
s : List' ℕ ℕ → ℕ
s (nil , _) = 0
s (cons n , f) = n + f zero
```

The functions `sum1` and `sum2`. The former is defined without a church-encoding, the latter with. A reflexive proof and sanity check is included to show equality:

```
sum1 : List ℕ → ℕ
sum1 = ⟪ s ⟫
sum2 : List ℕ → ℕ
sum2 = consu s
eqsum : sum1 ≡ sum2
eqsum = refl
checksum : sum1 (5 :: 6 :: 7 :: []) ≡ 18
checksum = refl
```

equality The below proof shows the equality between the non-church-endcoded pipeline and the church-encoded pipeline:

```
eq : {f : ℕ → ℕ} → sum1 ∘ map1 f ∘ between1 ≡ sum2 ∘ map2 f ∘ between2
eq {f} = begin
    ⟪ s ⟫ ∘ ⟪ in' ∘ m f ⟫ ∘ b in'
  ≡⟨ cong (λ g → ⟪ s ⟫ ∘ ⟪ in' ∘ m f ⟫ ∘ g) (prod-pres b) ⟩ - reflexive
    ⟪ s ⟫ ∘ ⟪ in' ∘ m f ⟫ ∘ fromCh ∘ prodCh b
  ≡⟨ cong (λ f → ⟪ s ⟫ ∘ f ∘ prodCh b) (sym $ trans-pres (m f)) ⟩
```

```
    (| s |) ∘ fromCh ∘ natTransCh (m f) ∘ prodCh b
  ≡⟨ cong (λ g → g ∘ fromCh ∘ natTransCh (m f) ∘ prodCh b) (cons-pres s) ⟩ - reflexive
    consCh s ∘ toCh ∘ fromCh ∘ natTransCh (m f) ∘ prodCh b
  ≡⟨ cong (λ g → consCh s ∘ toCh ∘ fromCh ∘ natTransCh (m f) ∘ g ∘ prodCh b) (sym to-from-id) ⟩
    consCh s ∘ toCh ∘ fromCh ∘ natTransCh (m f) ∘ toCh ∘ fromCh ∘ prodCh b
  ≡⟨⟩
    consu s ∘ natTrans (m f) ∘ prod b
  ■


- Bonus functions
count : (ℕ → Bool) → μ (F ℕ) → ℕ
count p = (| (λ where
                (nil , _) → 0
                (cons true , f) → 1 + f zero
                (cons false , f) → f zero) |) ∘ map1 p

even : ℕ → Bool
even 0 = true
even (suc n) = not (even n)
odd : ℕ → Bool
odd = not ∘ even

countworks : count even (5 :: 6 :: 7 :: 8 :: []) ≡ 2
countworks = refl

- Investigation related to filter, the following lines are tangentially related to list
build : {F : Container _ _}{X : Set} → ({Y : Set} → (⟦ F ⟧ Y → Y) → X → Y) → (x : X) → μ F
build g = fromCh ∘ prodCh g
foldr' : {F : Container _ _}{X : Set} → (⟦ F ⟧ X → X) → μ F → X
foldr' c = consCh c ∘ toCh
filter : {A : Set} → (A → Bool) → List A → List A
filter p = fromCh ∘ prodCh (λ f → consCh (λ where
  (nil , l) → f (nil , l)
  (cons a , l) → if (p a) then f (cons a , l) else l zero)) ∘ toCh


open import Data.Sum as S
open import Data.Fin hiding (_+_; _>_; _-_)
open import Data.Empty
open import Data.Unit
open import Data.Bool
open import Agda.Builtin.Nat
open import agda.church.defs
open import agda.church.proofs
open import agda.funct.funext
open import agda.init.initial hiding (const)


module agda.church.inst.free where
open import Data.Container.Combinator as C using (const; to-⊎; _⊎_)

-Below definition retrieved from Agda stdlib
Fr : Container 0ℓ 0ℓ → Set → Container 0ℓ 0ℓ
Fr f a = const a C.⊎ f

Free : Container 0ℓ 0ℓ → Set → Set
Free f a = μ (Fr f a)
Free' : Container 0ℓ 0ℓ → Set → Set → Set
Free' f a X = ⟦ Fr f a ⟧ X
```

```
record Handler (f f' : Container 0ℓ 0ℓ)(a b : Set) : Set where
  field
    hdlr : Free' f a (Free f' b) → Free f' b

- Handle is a consumer! This might mean that we cannot fuse it! :(
handle : {f f' : Container _ _}{a b : Set} →
         (Free' f a (Free f' b) → Free f' b) →
         Free (f C.⊎ f') a → Free f' b
handle h = ⦇ (λ where
                (inj₁ a , l) → h        (inj₁ a , l)
                (inj₂ (inj₁ x) , l) → h (inj₂ x , l)
                (inj₂ (inj₂ y) , l) → in' (inj₂ y , l)) ⦈
```

4.4 Fusion: Cochurch encodings

This section focuses on the fusion of Cochurch encodings, leveraging parametricity (free theorems) and the fusion property.

4.4.1 Definition of Cochurch encodings

This module defines Cochurch encodings and the two conversion functions `con` and `abs`, called `toCoCh` and `fromCoCh` here, respectively. It also defines the generalized producing, transformation, and consuming functions, as described by **?**. The definition of the CoChurch datatypes is defined slightly differently to how it is initially defined by **?**. Instead an Isomorphic definition is used, whose type is described later on on the same page. The original definition is included as `CoChurch'`.

```
{-# OPTIONS -guardedness #-}
module agda.cochurch.defs where
```

The Cochurch encoding, agian leveraging containers:

```
data CoChurch (F : Container 0ℓ 0ℓ) : Set₁ where
  CoCh : {X : Set} → (X → ⟦ F ⟧ X) → X → CoChurch F
```

The conversion functions:

```
toCoCh : {F : Container 0ℓ 0ℓ} → ν F → CoChurch F
toCoCh x = CoCh out x

fromCoCh : {F : Container 0ℓ 0ℓ} → CoChurch F → ν F
fromCoCh (CoCh h x) = A⟦ h ⟧ x
```

The generalized encoded producing, transformation, and consuming functions, alongside the proof that they are equal to the functions they are encoding. First, the producing function, note that this is a generalized version of **?**'s `unfoldr` function:

```
prodCoCh : {F : Container 0ℓ 0ℓ}{Y : Set} → (g : Y → ⟦ F ⟧ Y) → Y → CoChurch F
prodCoCh g x = CoCh g x

prod : {F : Container 0ℓ 0ℓ}{Y : Set} → (g : Y → ⟦ F ⟧ Y) → Y → ν F
prod g = fromCoCh ∘ prodCoCh g

eqprod : {F : Container 0ℓ 0ℓ}{Y : Set}{g : (Y → ⟦ F ⟧ Y)} →
         prod g ≡ A⟦ g ⟧
eqprod = refl
```

Second the transformation function:

```
natTransCoCh : {F G : Container 0ℓ 0ℓ}(nat : {X : Set} → ⟦ F ⟧ X → ⟦ G ⟧ X) → CoChurch F → CoChurch G
natTransCoCh n (CoCh h s) = CoCh (n ∘ h) s

natTrans : {F G : Container 0ℓ 0ℓ}(nat : {X : Set} → ⟦ F ⟧ X → ⟦ G ⟧ X) → ν F → ν G
natTrans nat = fromCoCh ∘ natTransCoCh nat ∘ toCoCh

eqNatTrans : {F G : Container 0ℓ 0ℓ}{nat : {X : Set} → ⟦ F ⟧ X → ⟦ G ⟧ X} →
  natTrans nat ≡ A⟦ nat ∘ out ⟧
eqNatTrans = refl
```

Third the consuming function, note that this a is a generalized version of **?**'s `destroy` function:

```
consCoCh : {F : Container 0ℓ 0ℓ}{Y : Set} → (c : {S : Set} → (S → ⟦ F ⟧ S) → S → Y) → CoChurch F → Y
consCoCh c (CoCh h s) = c h s

cons : {F : Container 0ℓ 0ℓ}{Y : Set} → (c : {S : Set} → (S → ⟦ F ⟧ S) → S → Y) → ν F → Y
cons c = consCoCh c ∘ toCoCh

eqcons : {F : Container 0ℓ 0ℓ}{X : Set}{c : {S : Set} → (S → ⟦ F ⟧ S) → S → X} →
         cons c ≡ c out
eqcons = refl
```

The original CoChurch definition is included here for completeness' sake, but it is note used elsewhere in the code.

```
data CoChurch' (F : Container 0ℓ 0ℓ) : Set₁ where
  cochurch : (∃ λ S → (S → ⟦ F ⟧ S) × S) → CoChurch' F
```

A mapping from `CoChurch'` to `CoChurch` and back is provided as well as a proof that their compositions are equal to the identity function, thereby proving isomorphism:

```
toConv : {F : Container _ _} → CoChurch' F → CoChurch F
toConv (cochurch (S , (h , x))) = CoCh {_}{S} h x

fromConv : {F : Container _ _} → CoChurch F → CoChurch' F
fromConv (CoCh {X} h x) = cochurch ((X , h , x))

to-from-conv-id : {F : Container 0ℓ 0ℓ} → toConv ∘ fromConv {F} ≡ id
to-from-conv-id = funext λ where
  (CoCh {X} h x) → refl

from-to-conv-id : {F : Container 0ℓ 0ℓ} → fromConv ∘ toConv {F} ≡ id
from-to-conv-id = funext λ where
  (cochurch (S , (h , x))) → refl
```

4.4.2 Proof obligations

As with Church encodings, in **?**'s work, five proof obligations needed to be satisfied. These are formalized in this module.

```
module agda.cochurch.proofs where
```

The first proof proves that `fromCoCh ∘ toCh = id`, using the reflection law:

```
from-to-id : {F : Container 0ℓ 0ℓ} → fromCoCh ∘ toCoCh {F} ≡ id
from-to-id {F} = funext λ x → begin
    fromCoCh (toCoCh x)
  ≡⟨⟩ - Definition of toCh
    fromCoCh (CoCh out x)
  ≡⟨⟩ - Definition of fromCh
    A⟦ out ⟧ x
  ≡⟨ reflection x ⟩
```

```
    x
  ≡⟨⟩ - Definition of identity
    id x
  ■
```

The second proof proof is similar to the first, but it proves the composition in the other direction `toCoCh ∘ fromCoCh = id`. This proof leverages the parametricity as described by **?**. It postulates the free theorem of the function g for a fixed Y `f : ∀ X → (X → F X) → X → Y`, to prove that "unfolding a Cochurch-encoded structure and then re-encoding it yields an equivalent structure" **?**:

```
postulate free : {F : Container 0ℓ 0ℓ}
                 {C D : Set}{Y : Set₁}{c : C → ⟦ F ⟧ C}{d : D → ⟦ F ⟧ D}
                 (h : C → D)(f : {X : Set} → (X → ⟦ F ⟧ X) → X → Y) →
                 map h ∘ c ≡ d ∘ h → f c ≡ f d ∘ h
                 - TODO: Do D and Y need to be the same thing? This may be a cop-out...
unfold-invariance : {F : Container 0ℓ 0ℓ}{Y : Set}
                    (c : Y → ⟦ F ⟧ Y) →
                    CoCh c ≡ (CoCh out) ∘ A⟦ c ⟧
unfold-invariance c = free A⟦ c ⟧ CoCh refl

to-from-id : {F : Container 0ℓ 0ℓ} → toCoCh ∘ fromCoCh {F} ≡ id
to-from-id = funext λ where
  (CoCh c x) → begin
      toCoCh (fromCoCh (CoCh c x))
    ≡⟨⟩ - definition of fromCh
      toCoCh (A⟦ c ⟧ x)
    ≡⟨⟩ - definition of toCh
      CoCh out (A⟦ c ⟧ x)
    ≡⟨⟩ - composition
      (CoCh out ∘ A⟦ c ⟧) x
    ≡⟨ cong (λ f → f x) (sym $ unfold-invariance c) ⟩
      CoCh c x
    ■
```

The third proof shows that cochurch-encoded functions constitute an implementation for the producing functions being replaced. The proof is proved via reflexivity, but **?**'s original proof steps are included here for completeness:

```
prod-pres : {F : Container 0ℓ 0ℓ}{X : Set}(c : X → ⟦ F ⟧ X) →
            fromCoCh ∘ prodCoCh c ≡ A⟦ c ⟧
prod-pres c = funext λ x → begin
    fromCoCh ((λ s → CoCh c s) x)
  ≡⟨⟩ - function application
    fromCoCh (CoCh c x)
  ≡⟨⟩ - definition of toCh
    A⟦ c ⟧ x
  ■
```

The fourth proof shows that cochurch-encoded functions constitute an implementation for the consuming functions being replaced. The proof is proved via reflexivity, but **?**'s original proof steps are included here for completeness:

```
cons-pres : {F : Container 0ℓ 0ℓ}{X : Set} → (f : {Y : Set} → (Y → ⟦ F ⟧ Y) → Y → X) →
            consCoCh f ∘ toCoCh ≡ f out
cons-pres f = funext λ x → begin
    consCoCh f (toCoCh x)
  ≡⟨⟩ - definition of toCoCh
    consCoCh f (CoCh out x)
  ≡⟨⟩ - function application
    f out x
  ■
```

The fifth, and final proof shows that cochurch-encoded functions constitute an implementation for the consuming functions being replaced. The proof leverages the categorical fusion property and the naturality of `f`:

```
- PAGE 52 - Proof 5
valid-hom : {F G : Container 0ℓ 0ℓ}{X : Set}(h : X → ⟦ F ⟧ X)
            (f : {X : Set} → ⟦ F ⟧ X → ⟦ G ⟧ X)(nat : ∀ {X : Set}(g : X → ν F) → map g ∘ f ≡ f ∘ map g) →
            map A⟦ h ⟧ ∘ f ∘ h ≡ f ∘ out ∘ A⟦ h ⟧
valid-hom h f nat = begin
    (map A⟦ h ⟧ ∘ f) ∘ h
  ≡⟨ cong (λ f → f ∘ h) (nat A⟦ h ⟧) ⟩
    (f ∘ map A⟦ h ⟧) ∘ h
  ≡⟨⟩ - dfn of A⟦_⟧
    f ∘ out ∘ A⟦ h ⟧
  ■

trans-pres : {F G : Container 0ℓ 0ℓ}{X : Set}(h : X → ⟦ F ⟧ X) (f : {X : Set} → ⟦ F ⟧ X → ⟦ G ⟧ X)
             (nat : {X : Set}(g : X → ν F) → map g ∘ f ≡ f ∘ map g) →
             fromCoCh ∘ natTransCoCh f ≡ A⟦ f ∘ out ⟧ ∘ fromCoCh
trans-pres h f nat = funext λ where
  (CoCh h x) → begin
      fromCoCh (natTransCoCh f (CoCh h x))
    ≡⟨⟩ - Function application
      fromCoCh (CoCh (f ∘ h) x)
    ≡⟨⟩ - Definition of fromCh
      A⟦ f ∘ h ⟧ x
    ≡⟨ cong-app (fusion A⟦ h ⟧ (sym (valid-hom h f nat))) x ⟩ - Can I remove the fusion prop?
      A⟦ f ∘ out ⟧ (A⟦ h ⟧ x)
    ≡⟨⟩ - This step is missing from the paper, but mirrors the step taken on the Church-side.
      A⟦ f ∘ out ⟧ (fromCoCh (CoCh h x))
    ■
```

Finally two additional proofs were made to clearly show that any pipeline made using cochurch encodings will fuse down to a simple function application. The first of these two proofs shows that any two composed natural transformation fuse down to one single natural transformation:

```
natfuse : {F G H : Container 0ℓ 0ℓ}
          (nat1 : {X : Set} → ⟦ F ⟧ X → ⟦ G ⟧ X) →
          (nat2 : {X : Set} → ⟦ G ⟧ X → ⟦ H ⟧ X) →
          natTransCoCh nat2 ∘ toCoCh ∘ fromCoCh ∘ natTransCoCh nat1 ≡ natTransCoCh (nat2 ∘ nat1)
natfuse nat1 nat2 = begin
            natTransCoCh nat2 ∘ toCoCh ∘ fromCoCh ∘ natTransCoCh nat1
          ≡⟨ cong (λ f → natTransCoCh nat2 ∘ f ∘ natTransCoCh nat1) to-from-id ⟩
            natTransCoCh nat2 ∘ natTransCoCh nat1
          ≡⟨ funext (λ where (CoCh g s) → refl) ⟩
            natTransCoCh (nat2 ∘ nat1)
          ■
```

The second of these two proofs shows that any pipeline, consisting of a producer, transformer, and consumer function, fuse down to a single function application:

```
pipefuse : {F G : Container 0ℓ 0ℓ}{X : Set}(c : X → ⟦ F ⟧ X)
           (nat : {X : Set} → ⟦ F ⟧ X → ⟦ G ⟧ X) →
           (f : {Y : Set} → (Y → ⟦ G ⟧ Y) → Y → X) →
           cons f ∘ natTrans nat ∘ prod c ≡ f (nat ∘ c)
pipefuse c nat f = begin
    consCoCh f ∘ toCoCh ∘ fromCoCh ∘ natTransCoCh nat ∘ toCoCh ∘ fromCoCh ∘ prodCoCh c
  ≡⟨ cong (λ g → consCoCh f ∘ g ∘ natTransCoCh nat ∘ toCoCh ∘ fromCoCh ∘ prodCoCh c) to-from-id ⟩
    consCoCh f ∘ natTransCoCh nat ∘ toCoCh ∘ fromCoCh ∘ prodCoCh c
  ≡⟨ cong (λ g → consCoCh f ∘ natTransCoCh nat ∘ g ∘ prodCoCh c) to-from-id ⟩
```

```
  consCoCh f ∘ natTransCoCh nat ∘ prodCoCh c
≡⟨⟩
  f (nat ∘ c)
■
```

4.4.3 Example: List fusion

In order to clearly see how the Cochurch encodings allows functions to fuse, a datatype was selected such the abstracted function, which have so far been used to prove the needed properties, can be instantiated to demonstrate how the fusion works for functions across a cocrete datatype. In this module is defined: the container, whose interpretation represents the base functor for lists, some convenience functions to make type annotations more readable, a producer function `between`, a transformation function `map`, a consumer function `sum`, and a proof that non-cochurch and cochurch-encoded implementations are equal.

Datatypes The index set for the container, as well as the container whose interpretation represents the base funtor for list. Note how ListOp is isomorphis to the datatype `1 + A`, I use ListOp instead to make the code more readable:

```
data ListOp (A : Set) : Set where
  nil : ListOp A
  cons : A → ListOp A
F : (A : Set) → Container 0ℓ 0ℓ
F A = ListOp A ▷ λ { nil → Fin 0 ; (cons n) → Fin 1 }
```

Functions representing the run-of-the-mill (potentially infinite) list datatype and the base functor for list:

```
List : (A : Set) → Set
List A = ν (F A)
List' : (A B : Set) → Set
List' A B = ⟦ F A ⟧ B
```

Helper functions to assist in cleanly writing out instances of lists:

```
[] : {A : Set} → List A
head [] = nil
tail [] = λ()
_::_ : {A : Set} → A → List A → List A
head (x :: xs) = cons x
tail (x :: xs) = const xs
infixr 20 _::_
```

The unfold funtion as it would normally be encountered for lists, defined in terms of ⟦_⟧:

```
mapping : {A X : Set} → (f : X → ⊤ ⊎ (A × X)) → (X → List' A X)
mapping f x with f x
mapping f x | (inj₁ tt) = (nil , λ())
mapping f x | (inj₂ (a , x')) = (cons a , const x')
unfold' : {F : Container 0ℓ 0ℓ}{A X : Set}(f : X → ⊤ ⊎ (A × X)) → X → List A
unfold' {A}{X} f = A⟦ mapping f ⟧
```

between The recursion principle `b`, which when used, represents the between function. It uses `b'` to assist termination checking:

```
b' : ℕ × ℕ → List' ℕ (ℕ × ℕ)
b' (x , zero) = (nil , λ())
b' (x , suc n) = (cons x , const (suc x , n))
```

```
b : ℕ × ℕ → List' ℕ (ℕ × ℕ)
b (x , y) = b' (x , (suc (y - x)))
```

The functions `between1` and `between2`. The former is defined without a cochurch-encoding, the latter with. A reflexive proof and sanity check (not working currently) is included to show equality:

```
between1 : ℕ × ℕ → List ℕ
between1 = A⟦ b ⟧
between2 : ℕ × ℕ → List ℕ
between2 = prod b
eqbetween : between1 ≡ between2
eqbetween = refl
-checkbetween : out (2 :: 3 :: 4 :: 5 :: 6 :: []) ≡ out (between2 (2 , 6))
-checkbetween = refl
```

map The coalgebra `m`, which when used in an algebra, represents the map function:

```
m : {A B C : Set}(f : A → B) → List' A C → List' B C
m f (nil , l) = (nil , l)
m f (cons n , l) = (cons (f n) , l)
```

The functions `map1` and `map2`. The former is defined without a cochurch-encoding, the latter with. A reflexive proof and sanity check (not currently working) is included to show equality:

```
map1 : {A B : Set}(f : A → B) → List A → List B
map1 f = A⟦ m f ∘ out ⟧
map2 : {A B : Set}(f : A → B) → List A → List B
map2 f = natTrans (m f)
eqmap : {f : ℕ → ℕ} → map1 f ≡ map2 f
eqmap = refl
-checkmap : map1 (_+_ 2) (3 :: 6 :: []) ≡ 5 :: 8 :: []
-checkmap = refl
```

sum The coalgebra `s`, which when used in an algebra, represents the sum function. Note that it is currently set to be non-terminating. A modification to ν is likely needed to enable usage of size type for the termination checker to accept this:

```
{-# NON_TERMINATING #-}
s : {S : Set} → (S → List' ℕ S) → S → ℕ
s h s' with h s'
s h s' | (nil , f) = 0
s h s' | (cons x , f) = x + s h (f zero)
```

The functions `sum1` and `sum2`. The former is defined without a cochurch-encoding, the latter with. A reflexive proof and sanity check (currently not working) is included to show equality:

```
sum1 : List ℕ → ℕ
sum1 = s out
sum2 : List ℕ → ℕ
sum2 = consu s
eqsum : sum1 ≡ sum2
eqsum = refl
-checksum : sum1 (5 :: 6 :: 7 :: []) ≡ 18
-checksum = refl
```

equality The below proof shows the equality between the non-cochurch-endcoded pipeline and the cochurch-encoded pipeline. Note how it is different from the proof for church-encoded pipelines. This is because **?**'s proof for the proof obligation of natural transformations is different for cochurch encodings than for church encodings. Because of this the first and second proof step for `eq` in the church-encoded lists is done in one step here:

```
eq : {f : ℕ → ℕ} → sum1 ∘ map1 f ∘ between1 ≡ sum2 ∘ map2 f ∘ between2
eq {f} = begin
    s out ∘ A⟦ m f ∘ out ⟧ ∘ A⟦ b ⟧
  ≡⟨ cong (λ g → s out ∘ A⟦ m f ∘ out ⟧ ∘ g) (prod-pres b) ⟩
    s out ∘ A⟦ m f ∘ out ⟧ ∘ fromCoCh ∘ prodCoCh b
  ≡⟨ cong (λ g → s out ∘ g ∘ prodCoCh b) (sym (trans-pres b (m f) λ _ → funext (λ {(nil , l) → refl; (cons n , l) -
    s out ∘ fromCoCh ∘ natTransCoCh (m f) ∘ prodCoCh b
  ≡⟨ cong (λ g → g ∘ fromCoCh ∘ natTransCoCh (m f) ∘ prodCoCh b) (cons-pres s) ⟩
    consCoCh s ∘ toCoCh ∘ fromCoCh ∘ natTransCoCh (m f) ∘ prodCoCh b
  ≡⟨ cong (λ g → consCoCh s ∘ toCoCh ∘ fromCoCh ∘ natTransCoCh (m f) ∘ g ∘ prodCoCh b) (sym to-from-id) ⟩
    consCoCh s ∘ toCoCh ∘ fromCoCh ∘ natTransCoCh (m f) ∘ toCoCh ∘ fromCoCh ∘ prodCoCh b
  ≡⟨⟩
    consu s ∘ natTrans (m f) ∘ prod b
  ■
```

5 Related Works

5.1 Fusion

Initial work on fusion was done my **???**, and was dubbed 'deforestation', referring to the removal of intermediate trees (or lists). The details of the original deforestation work are not relevant to this thesis, but, the weaknesses of the work are described and different techniques proposed by **?**. **?** describe a technique nowadays called `foldr/build` fusion, which, when employed, can eliminate most intermediate lists. This technique is described further in **??**.

A converse approach, aptly named the `destroy/unfoldr` rule, is described by **?**, which describes the converse technique to **?**'s. A further generalization of this technique, leverages the coinductive list datatype, stream. This technique is called *stream fusion* introduced by **?**.

(Co)Church encodings Finally, **?** combined all of these concepts into one paper, called "The Library Writer's Guide to Shortcut Fusion". In it the concept of (Co)Church encodings are described and, pragmatically, how to implement them in Haskell.

6 Conclusion and Future Work

6.1 Future Work

- Strengthen Agda's typechecker wrt implicit parameters
- Strengthen Agda's termination checker wrt corecursive datastructures
- Implement (co)church-fused versions of Haskell's library functions.
- Investigate if creating a language that has this fusion built-in natively can be compiled more efficiently
- Look into leveraging parametricity with agda, so no `posulate`'s are needed.

References

Abbott, M., Altenkirch, T., & Ghani, N. (2005, September). Containers: Constructing strictly positive types. *Theoretical Computer Science*, *342*(1), 3–27. Retrieved from http://dx.doi.org/10.1016/j.tcs.2005.06.002 doi: 10.1016/j.tcs.2005.06.002

Ahrens, B., & Wullaert, K. (2022). *Category theory for programming.* arXiv. Retrieved from https://arxiv.org/abs/2209.01259 doi: 10.48550/ARXIV.2209.01259

Breitner, J. (2018, June). Call arity. *Computer Languages, Systems & Structures*, *52*, 65–91. Retrieved from http://dx.doi.org/10.1016/j.cl.2017.03.001 doi: 10.1016/j.cl.2017.03.001

Coutts, D., Leshchinskiy, R., & Stewart, D. (2007, October). Stream fusion: from lists to streams to nothing at all. In *Proceedings of the 12th acm sigplan international conference on functional programming.* ACM. Retrieved from http://dx.doi.org/10.1145/1291151.1291199 doi: 10.1145/1291151.1291199

Gill, A., Launchbury, J., & Peyton Jones, S. L. (1993, July). A short cut to deforestation. In *Proceedings of the conference on functional programming languages and computer architecture.* ACM. Retrieved from http://dx.doi.org/10.1145/165180.165214 doi: 10.1145/165180.165214

Harper, T. (2011, September). A library writer's guide to shortcut fusion. *ACM SIGPLAN Notices*, *46*(12), 47–58. Retrieved from http://dx.doi.org/10.1145/2096148.2034682 doi: 10.1145/2096148.2034682

Jones, S. L. P. (1996). Compiling haskell by program transformation: A report from the trenches. In *Lecture notes in computer science* (p. 18–44). Springer Berlin Heidelberg. Retrieved from http://dx.doi.org/10.1007/3-540-61055-3_27 doi: 10.1007/3-540-61055-3_27

Svenningsson, J. (2002, September). Shortcut fusion for accumulating parameters & zip-like functions. *ACM SIGPLAN Notices*, *37*(9), 124–132. Retrieved from http://dx.doi.org/10.1145/583852.581491 doi: 10.1145/583852.581491

Van Muylder, A., Nuyts, A., & Devriese, D. (2023). *Agda –bridges vm.* Zenodo. Retrieved from https://zenodo.org/doi/10.5281/zenodo.10009365 doi: 10.5281/ZENODO.10009365

Wadler, P. (1984). Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-time. In *Proceedings of the 1984 acm symposium on lisp and functional programming - lfp '84.* ACM Press. Retrieved from http://dx.doi.org/10.1145/800055.802020 doi: 10.1145/800055.802020

Wadler, P. (1986). Listlessness is better than laziness ii: Composing listless functions. In *Lecture notes in computer science* (p. 282–305). Springer Berlin Heidelberg. Retrieved from http://dx.doi.org/10.1007/3-540-16446-4_16 doi: 10.1007/3-540-16446-4_16

Wadler, P. (1989). Theorems for free! In *Proceedings of the fourth international conference on functional programming languages and computer architecture - fpca '89.* ACM Press. Retrieved from http://dx.doi.org/10.1145/99370.99404 doi: 10.1145/99370.99404

Wadler, P. (1990, June). Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, *73*(2), 231–248. Retrieved from http://dx.doi.org/10.1016/0304-3975(90)90147-A doi: 10.1016/0304-3975(90)90147-a

A Derivations

A.1 Cochurch Stream-fused encoding derivation

Here I will provide an example derivation of a Church encoded function pipeline. We start with the definitions:

$$\begin{aligned}
&\textbf{data}\ \mathit{List}\ _\ a\ b = \mathit{Nil}\ _ \mid \mathit{Cons}\ _\ a\ b\ \textbf{deriving}\ \mathit{Functor}\\
&\textbf{data}\ \mathit{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)\ \textbf{deriving}\ (\mathit{Functor}, \mathit{Show})\\
&\textbf{data}\ \mathit{ListCh}\ a = \mathit{ListCh}\ (\forall\ b\ .\ (\mathit{List}\ _\ a\ b \to b) \to b)
\end{aligned}$$

$$\begin{aligned}
&\mathit{toCh} :: \mathit{List}\ a \to \mathit{ListCh}\ a\\
&\mathit{toCh}\ t = \mathit{ListCh}\ (\lambda a \to \mathit{fold}\ a\ t)\\
&\mathit{fold} :: (\mathit{List}\ _\ a\ b \to b) \to \mathit{List}\ a \to b\\
&\mathit{fold}\ a\ \mathit{Nil} = a\ \mathit{Nil}\ _\\
&\mathit{fold}\ a\ (\mathit{Cons}\ x\ xs) = a\ (\mathit{Cons}\ _\ x\ (\mathit{fold}\ a\ xs))
\end{aligned}$$

```
fromCh :: ListCh a → List a
fromCh (ListCh fold) = fold in'
in' :: List_ a (List a) → List a
in' Nil_ = Nil
in' (Cons_ x xs) = Cons x xs
```

`toCh` takes an input datastructure and puts it into a thunked fold that is still waiting for an input function. `fromCh` takes the fold, and executes it, replacing our Tree_ datastructure with the normal Tree. Church encoded versions of sum, map (+1), filter odd, and between look like the following:

```
b :: (List_ Int b → b) → (Int, Int) → b
b a (x, y) = loop (x, y)
  where loop (x, y) = case x > y of
    True → a Nil_
    False → a (Cons_ x (loop (x + 1, y)))
betweenCh :: (Int, Int) → ListCh Int
betweenCh (x, y) = ListCh (λa → b a (x, y))

m :: (a → b) → List_ a c → List_ b c
m f Nil_ = Nil_
m f (Cons_ x xs) = Cons_ (f x) xs
mapCh :: (a → b) → ListCh a → ListCh b
mapCh f (ListCh g) = ListCh (λa → g (a . m f))

filterCh :: (a → Bool) → ListCh a → ListCh a
filterCh p (ListCh g) = ListCh (λa → g (λcase
    Nil_ → a Nil_
    Cons_ x xs → if (p x) then a (Cons_ x xs) else xs
 ))

s :: List_ Int Int → Int
s Nil_ = 0
s (Cons_ x y) = x + y
sumCh :: ListCh Int → Int
sumCh (ListCh g) = g s
```

Next, the actual functions:

```
sum :: List Int → Int
sum = sumCh . toCh

map :: (a → b) → List a → List b
map f = fromCh . mapCh f . toCh

filter :: (a → Bool) → List a → List a
filter p = fromCh . filterCh p . toCh

between :: (Int, Int) → List Int
between = fromCh . betweenCh
```

I will be providing an example fusion of this pipeline:

```
f = sum . map (+1) . filter odd . between

f = sumCh . toCh .
fromCh . mapCh (+1) . toCh .
fromCh . filterCh odd . toCh .
fromCh . betweenCh
```

When 'fused' (`toCh . fromCh` removed) it looks like this:

```
sumCh . mapCh (+1) . filterCh odd . betweenCh
```

For some input (x, y), we derive:

```
sumCh . mapCh (+1) . filterCh odd . betweenCh (x, y)
    -- Inlining of betweenCh
sumCh . mapCh (+1) . filterCh odd . ListCh (λa → b a (x, y))
    -- Dfn of filterCh + beta reduction
sumCh . mapCh (+1) .
  ListCh (λa' →
    (λa → b a (x, y))
      (λx → case x of
        Nil_ → a' Nil_
        Cons_ x xs → if (p x) then a' (Cons_ x xs) else xs
    )
  )
    -- Beta reduction
sumCh . mapCh (+1) .
  ListCh (λa' →
    b (λx → case x of
      Nil_ → a' Nil_
      Cons_ x xs → if (p x) then a' (Cons_ x xs) else xs
    )
      (x, y))
    -- Dfn of mapCh + beta reduction
sumCh . ListCh (λa →
  (λa' →
    b (λx → case x of
      Nil_ → a' Nil_
      Cons_ x xs → if (p x) then a' (Cons_ x xs) else xs
    )
      (x, y)
  )
  (a . m (+1)))
    -- Substitution
sumCh . ListCh (λa →
  b (λx → case x of
    Nil_ → (a . m (+1)) Nil_
    Cons_ x xs → if (p x) then (a . m (+1)) (Cons_ x xs) else xs
  )
  (x, y))
    -- Dfn of sumCh
(λa →
  b (λx → case x of
    Nil_ → (a . m (+1)) Nil_
    Cons_ x xs → if (p x) then (a . m (+1)) (Cons_ x xs) else xs
  )
  (x, y)
) s
    -- Beta reduction
b (λx → case x of
  Nil_ → s (m (+1) Nil_)
  Cons_ x xs → if (p x) then s (m (+1) (Cons_ x xs)) else xs
) (x, y)
    -- Inlining m + beta reduction
b (λx → case x of
  Nil_ → s Nil_
  Cons_ x xs → if (p x) then s (Cons_ ((+1) x) xs) else xs
) (x, y)
```

```
    -- Inlining s + beta reduction
b (λx → case x of
  Nil _ → 0
  Cons _ x xs → if (p x) then (((+1) x) + xs) else xs
) (x, y)
    -- Inlining of b + beta reduction
loop (x, y) = case x > y of
  True → case Nil _ of
    Nil _ → 0
    Cons _ x xs → if (p x) then (((+1) x) + xs) else xs
  False → case (Cons _ x (loop (x + 1, y))) of
    Nil _ → 0
    Cons _ x xs → if (p x) then (((+1) x) + xs) else xs
loop (x, y)
    -- case-of-known-case optimization
loop (x, y) = case x > y of
  True → 0
  False → if (p x) then ((+1) x + loop (x + 1, y)) else loop (x + 1, y)
loop (x, y)
    -- Cleaning it up:
loop (x, y) = if x > y
              then 0
              else if (p x)
                   then x + 1 + loop (x + 1, y)
                              else loop (x + 1, y)
loop (x, y)
```

This concludes the example derivation for Church fusion.

A.2 Cochurch Stream-fused encoding derivation

Here I will provide an example derivation of a Cochurch encoded function pipeline, using stream fusion techniques. We start with the definitions:

```
data List' _ a b = Nil' _ | NilT' _ b | Cons' _ a b deriving Functor
data List a = Nil | Cons a (List a) deriving (Functor, Show)
data ListCoCh a = ∀ s . ListCoCh (s → List' _ a s) s

toCoCh :: List a → ListCoCh a
toCoCh = ListCoCh out
out :: List a → List' _ a (List a)
out Nil = Nil' _
out (Cons x xs) = Cons' _ x xs

fromCoCh :: ListCoCh a → List a
fromCoCh (ListCoCh h s) = unfold h s
unfold :: (b → List' _ a b) → b → List a
unfold h s = case h s of
  Nil' _ → Nil
  NilT' _ xs → unfold h xs
  Cons' _ x xs → Cons x (unfold h xs)
```

CoChurch encoded versions of sum, map (+2), filter odd, and between look like the following:

```
su' :: (s → List' _ Int s) → s → Int
su' h s = loop s
  where loop s' = case h s' of
    Nil' _ → 0
    NilT' _ xs → loop xs
    Cons' _ x xs → x + loop xs
```

```
sumCoCh :: ListCoCh Int → Int
sumCoCh (ListCoCh h s) = su' h s

m' :: (a → b) → List'_ a c → List'_ b c
m' f (Cons'_ x xs) = Cons'_ (f x) xs
m' _ (NilT'_ xs) = NilT'_ xs
m' _ (Nil'_) = Nil'_
mapCoCh :: (a → b) → ListCoCh a → ListCoCh b
mapCoCh f (ListCoCh h s) = ListCoCh (m' f . h) s

filt p h s = case h s of
                Nil'_ → Nil'_
                NilT'_ xs → NilT'_ xs
                Cons'_ x xs → if p x then Cons'_ x xs else NilT'_ xs
filterCoCh :: (a → Bool) → ListCoCh a → ListCoCh a
filterCoCh p (ListCoCh h s) = ListCoCh (filt p h) s

betweenCoCh :: (Int, Int) → List'_ Int (Int, Int)
betweenCoCh (x, y) = case x > y of
  case True → Nil'_
  case False → Cons'_ x (x + 1, y)
```

Next, the actual functions:

```
sum :: List Int → Int
sum = sumCoCh . toCoCh

map :: (a → b) → List a → List b
map f = fromCoCh . mapCoCh f . toCoCh

filter :: (a → Bool) → List a → List a
filter p = fromCoCh . filterCoCh p . toCoCh

between :: (Int, Int) → List Int
between = fromCoCh . ListCoCh betweenCoCh
```

We again will fuse the following pipeline:

```
f = sum . map (+2) . filter odd . between

f = sumCoCh . toCoCh .
fromCoCh . mapCoCh (+2) . toCoCh .
fromCoCh . filterCoCh odd . toCoCh .
fromCoCh . ListCoCh betweenCoCh
```

When 'fused' it looks like this:

```
sumCoCh . mapCoCh (+2) . filterCoCh odd . ListCoCh betweenCoCh
```

For some input (x, y), we derive:

```
sumCoCh . mapCoCh (+2) . filterCoCh odd . ListCoCh betweenCoCh (x, y)
    -- Inlining of filterCoCh + beta reduction
sumCoCh . mapCoCh (+2) . ListCoCh (filt odd betweenCoCh) (x, y)
    -- Inlining of mapCoCh + beta reduction
sumCoCh . ListCoCh (m' (+2) . filt odd betweenCoCh) (x, y)
    -- Inlining of sumCoCh + beta reduction
su' (m' (+2) . filt odd betweenCoCh) (x, y)
    -- Inlining of su' + beta reduction
loop (x, y) = case ((m' (+2) . filt odd betweenCoCh) (x, y)) of
  Nil'_ → 0
```

```
  NilT'_ s → loop s
  Cons'_ x s → x + loop s
loop (x, y)
    -- Inlining of filt + beta reduction
loop (x, y) = case (m' (+2) . (
                  case betweenCoCh (x, y) of
                      Nil'_ → Nil'_
                      NilT'_ xs → NilT'_ xs
                      Cons'_ x xs → if odd x then Cons'_ x xs else NilT'_ xs
  )) of
  Nil'_ → 0
  NilT'_ s → loop s
  Cons'_ x s → x + loop s
loop (x, y)
    -- Inlining of betweenCoCh + beta reduction
loop (x, y) = case (m' (+2) . (
                  case (
                    case (x > y) of
                      True → Nil'_
                      False → Cons'_ x (x + 1, y)
                    ) of
                    Nil'_ → Nil'_
                    NilT'_ xs → NilT'_ xs
                    Cons'_ x xs → if odd x then Cons'_ x xs else NilT'_ xs
                  )) of
  Nil'_ → 0
  NilT'_ s → loop s
  Cons'_ x s → x + loop s
loop (x, y)
    -- Case-of-case optimization
loop (x, y) = case (m' (+2) . (
                  case (x > y) of
                    True → case (Nil'_) of
                      Nil'_ → Nil'_
                      NilT'_ xs → NilT'_ xs
                      Cons'_ x xs → if odd x then Cons'_ x xs else NilT'_ xs
                    False → case (Cons'_ x (x + 1, y)) of
                      Nil'_ → Nil'_
                      NilT'_ xs → NilT'_ xs
                      Cons'_ x xs → if odd x then Cons'_ x xs else NilT'_ xs
                  )) of
  Nil'_ → 0
  NilT'_ s → loop s
  Cons'_ x s → x + loop s
loop (x, y)
    -- Case-of-known-case optimization
loop (x, y) = case (m' (+2) (case (x > y) of
    True → Nil'_
    False → if odd x then Cons'_ x (x + 1, y) else NilT'_ (x + 1, y)
  )) of
  Nil'_ → 0
  NilT'_ s → loop s
  Cons'_ x s → x + loop s
loop (x, y)
    -- Inlining of m' + beta reduction
loop (x, y) = case (
                  case (
                    case (x > y) of
```

```
                    True → Nil'_
                    False → if odd x then Cons'_ x (x + 1, y) else NilT'_ (x + 1, y)
                ) of
                  Cons'_ x xs → Cons'_ ((+2) x) xs
                  NilT'_ xs → NilT'_ xs
                  Nil'_ ⇒ Nil'_
            ) of
              Nil'_ → 0
              NilT'_ s → loop s
              Cons'_ x s → x + loop s
loop (x, y)
    -- Case-of-case optimization
loop (x, y) = case (
                case (x > y) of
                  True → case (Nil'_) of
                    Cons'_ x xs → Cons'_ ((+2) x) xs
                    NilT'_ xs → NilT'_ xs
                    Nil'_ ⇒ Nil'_
                  False → case (if odd x then Cons'_ x (x + 1, y) else NilT'_ (x + 1, y)) of
                    Cons'_ x xs → Cons'_ ((+2) x) xs
                    NilT'_ xs → NilT'_ xs
                    Nil'_ ⇒ Nil'_
            ) of
              Nil'_ → 0
              NilT'_ s → loop s
              Cons'_ x s → x + loop s
loop (x, y)
    -- Case-of-known-case optimization
loop (x, y) = case (
                case (x > y) of
                  True → Nil'_
                  False → case (if odd x then Cons'_ x (x + 1, y) else NilT'_ (x + 1, y)) of
                    Cons'_ x xs → Cons'_ ((+2) x) xs
                    NilT'_ xs → NilT'_ xs
                    Nil'_ ⇒ Nil'_
            ) of
              Nil'_ → 0
              NilT'_ s → loop s
              Cons'_ x s → x + loop s
loop (x, y)
    -- Inlining of if + beta reduction
loop (x, y) = case (
                case (x > y) of
                  True → Nil'_
                  False → case (
                    case (odd x) of
                      True → Cons'_ x (x + 1, y)
                      False → NilT'_ (x + 1, y)
                  ) of
                    Cons'_ x xs → Cons'_ ((+2) x) xs
                    NilT'_ xs → NilT'_ xs
                    Nil'_ ⇒ Nil'_
            ) of
              Nil'_ → 0
              NilT'_ s → loop s
              Cons'_ x s → x + loop s
loop (x, y)
    -- case-of-case optimization
```

```
loop (x, y) = case (
                    case (x > y) of
                      True → Nil'_
                      False → case (odd x) of
                        True → case (Cons'_ x (x + 1, y)) of
                          Cons'_ x xs → Cons'_ ((+2) x) xs
                          NilT'_ xs → NilT'_ xs
                          Nil'_ ⇒ Nil'_
                        False → case (NilT'_ (x + 1, y)) of
                          Cons'_ x xs → Cons'_ ((+2) x) xs
                          NilT'_ xs → NilT'_ xs
                          Nil'_ ⇒ Nil'_
              ) of
                Nil'_ → 0
                NilT'_ s → loop s
                Cons'_ x s → x + loop s
loop (x, y)
    -- Case-of-known-case optimization
loop (x, y) = case (
                    case (x > y) of
                      True → Nil'_
                      False → case (odd x) of
                        True → Cons'_ ((+2) x) (x + 1, y)
                        False → NilT'_ (x + 1, y)
              ) of
                Nil'_ → 0
                NilT'_ s → loop s
                Cons'_ x s → x + loop s
loop (x, y)
    -- case-of-case optimization
loop (x, y) = case (x > y) of
                    True → case (Nil'_) of
                      Nil'_ → 0
                      NilT'_ s → loop s
                      Cons'_ x s → x + loop s
                    False → case (
                      case (odd x) of
                        True → Cons'_ ((+2) x) (x + 1, y)
                        False → NilT'_ (x + 1, y)
                    ) of
                      Nil'_ → 0
                      NilT'_ s → loop s
                      Cons'_ x s → x + loop s
loop (x, y)
    -- Case-of-known-case optimization
loop (x, y) = case (x > y) of
                    True → 0
                    False → case (
                      case (odd x) of
                        True → Cons'_ ((+2) x) (x + 1, y)
                        False → NilT'_ (x + 1, y)
                    ) of
                      Nil'_ → 0
                      NilT'_ s → loop s
                      Cons'_ x s → x + loop s
loop (x, y)
    -- case-of-case optimization
loop (x, y) = case (x > y) of
```

```
                    True → 0
                    False → case (odd x) of
                      True → case (Cons'_ ((+2) x) (x + 1, y)) of
                        Nil'_ → 0
                        NilT'_ s → loop s
                        Cons'_ x s → x + loop s
                      False → case (NilT'_ (x + 1, y)) of
                        Nil'_ → 0
                        NilT'_ s → loop s
                        Cons'_ x s → x + loop s
loop (x, y)
    -- Case-of-known-case optimization
loop (x, y) = case (x > y) of
                    True → 0
                    False → case (odd x) of
                      True → ((+2) x) + loop (x + 1, y)
                      False → loop (x + 1, y)
loop (x, y)
    -- Boom! Finally a sane path to solution
loop (x, y) = case (x > y) of
                    True → 0
                    False → case (odd x) of
                      True → (x + 2) + loop (x + 1, y)
                      False → loop (x + 1, y)
loop (x, y)
    -- With some nicer syntax, compiles to same case of case tree:
loop (x, y) = if (x > y)
              then 0
              else if (odd x)
                   then (x + 2) + loop (x + 1, y)
                   else → loop (x + 1, y)
loop (x, y)
```

This concludes the derivation for Cochurch stream fusion. For completeness, however, here is the demostration that toCoCh and fromCoCh are mutually inverse:

```
fromCoCh . toCoCh l
    -- Inlining of toCoCh + beta reduction
fromCoCh . ListCoCh out l
    -- Inlining of fromCoCh + beta reduction
unfold out l
    -- Inlining of unfold + beta reduction
case out l of
  Nil'_ → Nil
  NilT'_ xs → unfold out xs
  Cons'_ x xs → Cons x (unfold out xs)
    -- Inlining of out + beta reduction
case (
  case l of
    Nil → Nil'_
    Cons x xs → Cons'_ x xs
  ) of
  Nil'_ → Nil
  NilT'_ xs → unfold out xs
  Cons'_ x xs → Cons x (unfold out xs)
    -- case-of-case
case l of
  Nil → case Nil'_ of
    Nil'_ → Nil
```

```
      NilT' _ xs → unfold out xs
      Cons' _ x xs → Cons x (unfold out xs)
    Cons x xs → case Cons' _ x xs
      Nil' _ → Nil
      NilT' _ xs → unfold out xs
      Cons' _ x xs → Cons x (unfold out xs)
      -- case-of-known-case
case l of
  Nil → Nil
  Cons x xs → Cons x (unfold out xs)
    -- Function is same as id through induction.


toCoCh . fromCoCh (ListCoCh h s)
    -- Unfold fromCoCh
toCoCh . unfold h s
    -- Inlining of toCoCh
ListCoCh out (unfold h s)
    -- This is true, so long as parametricity holds, see second proof of page 51 of Harper
```

A.3 Cochurch Stream-fused tail-recursive encoding derivation

Here I will provide an example derivation of a Cochurch encoded function pipeline, using stream fusion techniques, making sure that the coinduction principle is tail-recursive. We start with the definitions:

```
data List' _ a b = Nil' _ | NilT' _ b | Cons' _ a b deriving Functor
data List a = Nil | Cons a (List a) deriving (Functor, Show)
data ListCoCh a = ∀ s . ListCoCh (s → List' _ a s) s

toCoCh :: List a → ListCoCh a
toCoCh = ListCoCh out
out :: List a → List' _ a (List a)
out Nil = Nil' _
out (Cons x xs) = Cons' _ x xs

fromCoCh :: ListCoCh a → List a
fromCoCh (ListCoCh h s) = unfold h s
unfold :: (b → List' _ a b) → b → List a
unfold h s = case h s of
  Nil' _ → Nil
  NilT' _ xs → unfold h xs
  Cons' _ x xs → Cons x (unfold h xs)
```

CoChurch encoded versions of sum, map (+2), filter odd, and between look like the following:

```
su' :: (s → List' _ Int s) → s → Int
su' h s = loop s 0
  where loop s' acc = case h s' of
    Nil' _ → acc
    NilT' _ xs → loop xs acc
    Cons' _ x xs → loop xs (x + acc)
sumCoCh :: ListCoCh Int → Int
sumCoCh (ListCoCh h s) = su' h s

m' :: (a → b) → List' _ a c → List' _ b c
m' f (Cons' _ x xs) = Cons' _ (f x) xs
m' _ (NilT' _ xs) = NilT' _ xs
m' _ (Nil' _) = Nil' _
mapCoCh :: (a → b) → ListCoCh a → ListCoCh b
mapCoCh f (ListCoCh h s) = ListCoCh (m' f . h) s
```

```
filt p h s = case h s of
                Nil' _ → Nil' _
                NilT' _ xs → NilT' _ xs
                Cons' _ x xs → if p x then Cons' _ x xs else NilT' _ xs
filterCoCh :: (a → Bool) → ListCoCh a → ListCoCh a
filterCoCh p (ListCoCh h s) = ListCoCh (filt p h) s

betweenCoCh :: (Int, Int) → List' _ Int (Int, Int)
betweenCoCh (x, y) = case x > y of
  True → Nil' _
  False → Cons' _ x (x + 1, y)
```

Next, the actual functions:

```
sum :: List Int → Int
sum = sumCoCh . toCoCh

map :: (a → b) → List a → List b
map f = fromCoCh . mapCoCh f . toCoCh

filter :: (a → Bool) → List a → List a
filter p = fromCoCh . filterCoCh p . toCoCh

between :: (Int, Int) → List Int
between = fromCoCh . ListCoCh betweenCoCh
```

Here is the example piepline:

```
f = sum . map (+2) . filter odd . between

f = sumCoCh . toCoCh .
fromCoCh . mapCoCh (+2) . toCoCh .
fromCoCh . filterCoCh odd . toCoCh .
fromCoCh . ListCoCh betweenCoCh
```

When 'fused' it looks like this:

```
sumCoCh . mapCoCh (+2) . filterCoCh odd . ListCoCh betweenCoCh
```

For some input (x, y), we derive:

```
sumCoCh . mapCoCh (+2) . filterCoCh odd . ListCoCh betweenCoCh (x, y)
    -- Inlining of filterCoCh + beta reduction
sumCoCh . mapCoCh (+2) . ListCoCh (filt odd betweenCoCh) (x, y)
    -- Inlining of mapCoCh + beta reduction
sumCoCh . ListCoCh (m' (+2) . filt odd betweenCoCh) (x, y)
    -- Inlining of sumCoCh + beta reduction
su' (m' (+2) . filt odd betweenCoCh) (x, y)
    -- Inlining of su' + beta reduction
loop (x, y) acc = case ((m' (+2) . filt odd betweenCoCh) (x, y)) of
  Nil' _ → acc
  NilT' _ s → loop s acc
  Cons' _ x s → loop s (x + acc)
loop (x, y) 0
    -- Inlining of filt + beta reduction + beta reduction
loop (x, y) acc = case (m' (+2) . (
  case betweenCoCh (x, y) of
      Nil' _ → Nil' _
      NilT' _ xs → NilT' _ xs
      Cons' _ x xs → if odd x then Cons' _ x xs else NilT' _ xs
```

```
)) of
  Nil'_ → acc
  NilT'_ s → loop s acc
  Cons'_ x s → loop s (x + acc)
loop (x, y) 0
    -- Inlining of betweenCoCh + beta reduction
loop (x, y) acc = case (m' (+2) . (
  case (
    case (x > y) of
      True → Nil'_
      False → Cons'_ x (x + 1, y)
    ) of
    Nil'_ → Nil'_
    NilT'_ xs → NilT'_ xs
    Cons'_ x xs → if odd x then Cons'_ x xs else NilT'_ xs
)) of
  Nil'_ → acc
  NilT'_ s → loop s acc
  Cons'_ x s → loop s (x + acc)
loop (x, y) 0
    -- Case-of-case optimization
loop (x, y) acc = case (m' (+2) . (
  case (x > y) of
    True → case (Nil'_) of
      Nil'_ → Nil'_
      NilT'_ xs → NilT'_ xs
      Cons'_ x xs → if odd x then Cons'_ x xs else NilT'_ xs
    False → case (Cons'_ x (x + 1, y)) of
      Nil'_ → Nil'_
      NilT'_ xs → NilT'_ xs
      Cons'_ x xs → if odd x then Cons'_ x xs else NilT'_ xs
)) of
  Nil'_ → acc
  NilT'_ s → loop s acc
  Cons'_ x s → loop s (x + acc)
loop (x, y) 0
    -- Case-of-known-case optimization
loop (x, y) acc = case (m' (+2) (
  case (x > y) of
    True → Nil'_
    False → if odd x then Cons'_ x (x + 1, y) else NilT'_ (x + 1, y)
)) of
  Nil'_ → 0
  NilT'_ s → loop s
  Cons'_ x s → x + loop s
loop (x, y)
    -- Inlining of m'
loop (x, y) = case (
  case (
    case (x > y) of
      True → Nil'_
      False → if odd x then Cons'_ x (x + 1, y) else NilT'_ (x + 1, y)
  ) of
    Cons'_ x xs → Cons'_ ((+2) x) xs
    NilT'_ xs → NilT'_ xs
    Nil'_ ⇒ Nil'_
) of
  Nil'_ → acc
```

```
  NilT' _ s → loop s acc
  Cons' _ x s → loop s (x + acc)
loop (x, y) 0
    -- Case-of-case optimization
loop (x, y) acc = case (
  case (x > y) of
    True → case (Nil' _) of
      Cons' _ x xs → Cons' _ ((+2) x) xs
      NilT' _ xs → NilT' _ xs
      Nil' _ ⇒ Nil' _
    False → case (if odd x then Cons' _ x (x + 1, y) else NilT' _ (x + 1, y)) of
      Cons' _ x xs → Cons' _ ((+2) x) xs
      NilT' _ xs → NilT' _ xs
      Nil' _ ⇒ Nil' _
) of
  Nil' _ → acc
  NilT' _ s → loop s acc
  Cons' _ x s → loop s (x + acc)
loop (x, y) 0
    -- Case-of-known-case optimization
loop (x, y) acc = case (
  case (x > y) of
    True → Nil' _
    False → case (if odd x then Cons' _ x (x + 1, y) else NilT' _ (x + 1, y)) of
      Cons' _ x xs → Cons' _ ((+2) x) xs
      NilT' _ xs → NilT' _ xs
      Nil' _ ⇒ Nil' _
) of
  Nil' _ → 0
  NilT' _ s → loop s
  Cons' _ x s → x + loop s
loop (x, y) 0
    -- Inlining of if + beta reduction
loop (x, y) acc = case (
  case (x > y) of
    True → Nil' _
    False → case (
      case (odd x) of
        True → Cons' _ x (x + 1, y)
        False → NilT' _ (x + 1, y)
    ) of
      Cons' _ x xs → Cons' _ ((+2) x) xs
      NilT' _ xs → NilT' _ xs
      Nil' _ ⇒ Nil' _
) of
  Nil' _ → acc
  NilT' _ s → loop s acc
  Cons' _ x s → loop s (x + acc)
loop (x, y) 0
    -- Case-of-case optimization
loop (x, y) acc = case (
  case (x > y) of
    True → Nil' _
    False → case (odd x) of
      True → case (Cons' _ x (x + 1, y)) of
    Cons' _ x xs → Cons' _ ((+2) x) xs
    NilT' _ xs → NilT' _ xs
    Nil' _ ⇒ Nil' _
```

```
        False → case (NilT'_ (x + 1, y)) of
          Cons'_ x xs → Cons'_ ((+2) x) xs
          NilT'_ xs → NilT'_ xs
          Nil'_ ⇒ Nil'_
) of
  Nil'_ → acc
  NilT'_ s → loop s acc
  Cons'_ x s → loop s (x + acc)
loop (x, y) 0
    -- Case-of-known-case optimization
loop (x, y) acc = case (
  case (x > y) of
    True → Nil'_
    False → case (odd x) of
      True → Cons'_ ((+2) x) (x + 1, y)
      False → NilT'_ (x + 1, y)
  ) of
    Nil'_ → acc
    NilT'_ s → loop s acc
    Cons'_ x s → loop s (x + acc)
loop (x, y) 0
    -- Case-of-case optimization
loop (x, y) acc = case (x > y) of
  True → case (Nil'_) of
    Nil'_ → acc
    NilT'_ s → loop s acc
    Cons'_ x s → loop s (x + acc)
  False → case (
    case (odd x) of
      True → Cons'_ ((+2) x) (x + 1, y)
      False → NilT'_ (x + 1, y)
  ) of
    Nil'_ → acc
    NilT'_ s → loop s acc
    Cons'_ x s → loop s (x + acc)
loop (x, y) 0
    -- Case-of-known-case optimization
loop (x, y) acc = case (x > y) of
  True → acc
  False → case (
    case (odd x) of
      True → Cons'_ ((+2) x) (x + 1, y)
      False → NilT'_ (x + 1, y)
  ) of
    Nil'_ → acc
    NilT'_ s → loop s acc
    Cons'_ x s → loop s (x + acc)
loop (x, y) 0
    -- Case-of-case optimization
loop (x, y) acc = case (x > y) of
  True → acc
  False → case (odd x) of
    True → case (Cons'_ ((+2) x) (x + 1, y)) of
      Nil'_ → acc
      NilT'_ s → loop s acc
      Cons'_ x s → loop s (x + acc)
    False → case (NilT'_ (x + 1, y)) of
      Nil'_ → acc
```

```
        NilT' _ s → loop s acc
        Cons' _ x s → loop s (x + acc)
  loop (x, y) 0
      -- Case-of-known-case optimization
  loop (x, y) acc = case (x > y) of
    True → acc
    False → case (odd x) of
      True → loop (x + 1, y) (((+2) x) + acc)
      False → loop (x + 1, y) acc
  loop (x, y) 0
      -- Boom! Finally a sane path to solution
  loop (x, y) acc = case (x > y) of
    True → acc
    False → case (odd x) of
      True → loop (x + 1, y) ((x + 2) + acc)
      False → loop (x + 1, y) acc
  loop (x, y) 0
      -- With some nicer syntax, compiles to same case tree
  loop (x, y) acc = if (x > y)
                    then acc
                    else if (odd x)
                         then loop (x + 1, y) ((x + 2) + acc)
                         else loop (x + 1, y)
  loop (x, y) 0
      -- Notice how the final result, like the original su', is tail-recursive
```

■ This concludes the example derivation for tail-recursive Cochurch stream fusion.