Master's Thesis

Replication and formalization of (Co)Church encoded shortcut fusion.

Eben Rogers

Submitted in partial fulfillment of the requirements for the degree of:

MASTER OF SCIENCE
in
COMPUTER SCIENCE

With thanks to:

Casper Bach Poulsen
Jesper Cockx
Benedikt Ahrens

And with special thanks to:
Jaro Reinders, for the weekly help.

Programming Language Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands

June 17, 2024

Master's Thesis

Replication and formalization of (Co)Church encoded shortcut fusion.

Eben Rogers

Abstract

When writing functional code that composes multiple recursive functions that operate on a datastrcuture, we often incur a lot of computational overhead allocating memory, only to later read, use, and discard this information. This can be alleviated using fusion, a technique that combines these multiple recursive datastructure traversals into one. This thesis explores shortcut fusion using (Co)Church encodings based on the work of Harper (2011), focusing on two questions: What is needed to reliably achieve fusion in Haskell, and the correctness of these transformations through a formalization in Agda.

The first contribution replicates and extends Harper's (Co)Church encodings in Haskell, uncovering optimizer weaknesses and providing practical insights for achieving fusion within Haskell. The second contribution formalizes these encodings in Agda, leveraging parametricity and the category theory described by Harper. The formalization proves the equivalence of these encoded functions to the unencoded ones, showing that the encodings are in fact isomorphisms, as long as parametricity (Wadler, 1989) is assumed.

These findings highlight the effectiveness and correctness of shortcut fusion techniques and show the promise of shortcut fusion: Reduce the computational overhead associated with functional programming while retaining its nice, compositional properties.

Thesis committee:

Chair:	Jesper Cockx	Programming Languages	Assistant Professor
Core Member 2:	Casper Poulsen	Programming Languages	Assistant Professor
Core Member 3:	Rihan Hai	Web Information Systems	Assistant Professor
Daily Supervisor:	Jaro Reinders	Programming Languages	PhD Student

Contents

1 Introduction

When writing code there are two main paradigms: Imperative and Functional. Imperative programming has the benefit of being performant at the cost of weaker type systems, lack of compositionality, and memory consistency issues. Functional programming, in contrast, offers a stronger type system with more guarantees and more compositionality, but at the cost of computational overhead. I briefly describe one of the sources of computational overhead, followed by my work on automating the alleviation of this overhead, also known as fusion:

When writing functional code, we often use lists (or other data structures) to 'glue' multiple pieces of data together. For example, the following function in the programming language Haskell, as introduced by Gill et al. (1993):

$$
\begin{aligned}
&\mathit{all} :: (a \to \mathit{Bool}) \to [a] \to \mathit{Bool}\\
&\mathit{all}\ p\ \mathit{xs} = \mathit{and}\ (\mathit{map}\ p\ \mathit{xs})
\end{aligned}
$$

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 a list cell is allocated, only for the following function to subsequently deallocate it once the value has been read. We could instead rewrite `all` in the following fashion:

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

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 readability, writability, and composability. Can you write a high-performance, single-traversal, version of the following function (Harper, 2011)[1]?

$$
\begin{aligned}
&f :: (\mathit{Int}, \mathit{Int}) \to \mathit{Int}\\
&f = \mathit{sum}\ .\ \mathit{map}\ (+1)\ .\ \mathit{filter}\ \mathit{odd}\ .\ \mathit{between}
\end{aligned}
$$

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

$$
\begin{aligned}
&f :: (\mathit{Int}, \mathit{Int}) \to \mathit{Int}\\
&f\ (x, y) = \mathit{loop}\ x\\
&\quad \textbf{where}\ \mathit{loop}\ x = \textbf{if}\ x > y\\
&\qquad\qquad\qquad\qquad \textbf{then}\ 0\\
&\qquad\qquad\qquad\qquad \textbf{else if}\ \mathit{odd}\ x\\
&\qquad\qquad\qquad\qquad\quad \textbf{then}\ (x+1) + \mathit{loop}\ (x+1)\\
&\qquad\qquad\qquad\qquad\quad \textbf{else}\ \mathit{loop}\ (x+1)
\end{aligned}
$$

Doing this by hand every time, to get from the nice, elegant, compositional style of programming to the higher-performance, single-traversal style, is repetitive and error-prone. Especially if this needs to be done, by hand, every single time any two functions are composed. Is there some way to automate this process?

The answer is yes*, but it comes with an asterisk attached, namely: *The functions that are being fused need to be 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 one that traverses the datastructure just once.

Question Related work is discussed in detail in Section 5. My thesis focuses on a specific form of fusion called shortcut fusion through the use of (Co)Church encodings as described by Harper (2011) 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:

[1] The `between` is usually called `enumFromTo`, we keep it as `between` to remain consistent with Harper's naming scheme.

- What optimizations are present in Haskell that enable fusion to work?
- What tools and techniques are available 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 language?
   - If the mathematics and the encodings are implemented in a dependently typed language, is it possible to prove them to be correct?

Contributions My thesis centers on formalizing, replicating, and expanding upon Harper (2011)'s work and makes two crucial contributions, answering the two questions above:

1. I replicate the Church and Cochurch encodings' implementation in Haskell, as described by Harper and investigate further as to their performance characteristics. In this process, I find a weakness in Haskell's optimizer, glean further practical insights as to how to get these encodings to properly fuse, especially for Cochurch encodings, and what optimizations enable shortcut fusion to do its work.

   This is important as Harper gave a good pragmatic explanation of how to implement the (Co)Church encodings in Haskell, gave an example implementation, and benchmarked that implementation. He did not, however, provide 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." (Harper, 2011). This is what my research has set out to look into. This is discussed in detail in Section 3.

2. (Co)Church encodings are formalized and implemented, including the relevant category theory, in Agda, in as a general fashion as possible, leveraging containers (Abbott et al., 2005) 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). Furthermore, I apply the general proofs to an example `List` instance.

   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 Harper, aiding in understanding in how the different pieces of mathematics relate. This is discussed in detail in Section 4.

There are multiple future avenues that could be worked on to build on my work: The discussion and implementation of the Haskell code could help future readers understand how (Co)Church encodings work, hopefully aiding in the wider adoption and implementation of fusible functions at the library level.

From the current Agda implementation, it should be relatively simple to merge the Church and Cochurch encodings into Agda's stdlib. This would also make easier future work building on (Co)Church encodings. The work could be extended further to propertly implement a bisumulation or to leverage Agda `-bridges`'s internalized parametricity to be able to prove the free theorems currently postulated.

2 Background

Before discussing my work, it is important to describe the necessary background. For the reader I assume familiarity with Agda and Haskell. I also assume familiarity with the work of Harper (2011), it is recommended to have his work close at hand when reading mine; I do summarize parts of it, but I will not embed his work in mine.

My work builds on a body of existing work, namely foldr/build fusion and variants (Gill et al., 1993; Svenningsson, 2002; Coutts et al., 2007), some category theory (Ahrens & Wullaert, 2022), (Co)Church encodings (Harper, 2011), Containers (Abbott et al., 2005), parametricity (free theorems) (Wadler, 1989), and optimizations in Haskell's optimization pipeline that are relevant for fusion (Jones, 1996). We will be describing some of these works briefly, others will be described when needed.

2.1 Foldr/build fusion (on lists)

Starting with the basics of fusion. Gill et al. (1993) describes the original 'shortcut fusion' technique. The core idea is 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 Gill's example (and repeat a part of my introduction):

$$
\begin{aligned}
&\mathit{all} :: (a \to \mathit{Bool}) \to [\,a\,] \to \mathit{Bool} \\
&\mathit{all}\ p\ \mathit{xs} = \mathit{and}\ (\mathit{map}\ p\ \mathit{xs})
\end{aligned}
$$

`map p xs` applies `p` to all 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" (Gill et al., 1993).

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

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

In this case 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 functional 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?

Gill'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`:
>
> $$
> \begin{aligned}
> &\mathit{build} :: (\forall\ b\ .\ (a \to b \to b) \to b \to b) \to [\,a\,] \\
> &\mathit{build}\ g = g\ (:)\ [\,]
> \end{aligned}
> $$
>
> Such that:
>
> $$\mathit{foldr}\ k\ z\ (\mathit{build}\ g) = g\ k\ z$$
>
> Gill et al. (1993).

Gill 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 is true by `g`'s free theorem à la Wadler (1989). For more information on free theorems see Section 2.2.

2.1.1 An example

Take as an example the function `from`, that takes two numbers and produces a list of all integers between the:

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

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

$$
\begin{aligned}
&\mathit{from}' :: \mathit{Int} \to \mathit{Int} \to (\forall\ b\ .\ (\mathit{Int} \to b \to b) \to b \to b) \to [\,\mathit{Int}\,] \\
&\mathit{from}'\ a\ b = \lambda c\ n \to \textbf{if}\ a > b \\
&\qquad\qquad\qquad\qquad \textbf{then}\ n \\
&\qquad\qquad\qquad\qquad \textbf{else}\ c\ a\ (\mathit{from}\ (a+1)\ b\ c\ n)
\end{aligned}
$$

This is obviously a different function, we now redefine `from` in terms of `build` (Gill et al., 1993):

$$\begin{aligned}&\mathit{from} :: \mathit{Int} \to \mathit{Int} \to [\mathit{Int}]\\&\mathit{from}\ a\ b = \mathit{build}\ (\mathit{from}'\ a\ b)\end{aligned}$$

With some inlining and β reduction, one can see that this definition is identical to the original `from` definition. Now for the actual fusion (Gill et al., 1993):

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

Notice how we can apply the `foldr/build` from step two to three to prevent the generation of an intermediate list. Any adjacent `foldr/build` pair 'cancels 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 1. See Gill et al. (1993)'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 to as an efficient minimum as can be expected.

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

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

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

Figure 1: Examples of functions rewritten in terms of `foldr/build`. (Gill et al., 1993)

2.1.2 Generalization to recursive datastructures

This foldr/build fusion works for lists, but it has several limitations. One is that it only works on lists, which can be alleviated using Church encodings and is described by Harper (2011). Secondly, the functions that we are writing need to be expressible in terms of compositions of foldr's and builds. What if we want to implement the converse approach using an unfoldr? This exists and is destroy/unfoldr fusion and is described by Coutts et al. (2007). This work, generalized by Cochurch encodings, is also described by Harper (2011).

The generalization by Harper leverages (Co)Church encodings, which uses definitions from category theory such as F-algebras and initiality. Read on to Section 2.3, where we discuss these category theory definitions, after first having discussed free theorems.

2.2 Theorems for Free

Wadler (1989) in his work 'Theorems for Free', which builds on the abstraction theorem of Reynolds (1983), 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 state truths about polymorphic functions. This is also done in Harper (2011)'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 (Harper, 2011):

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

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 internalized parametricity, as initially described by Bernardy & Moulin (2012). Recent work by Van Muylder et al. (2024) 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 on the two needed locations.

2.3 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 catamorphisms and anamorphisms (folds and unfolds) will follow suit. The mathematics described here are based on the lecture notes by Ahrens & Wullaert (2022).

2.3.1 A Category

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

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 $h \in \mathtt{hom}_{\mathcal{C}}(Z, W)$:
$$h \circ (g \circ f) = (h \circ g) \circ f$$

2.3.2 Initial/Terminal Objects

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

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 $B \in \mathcal{C}_0$:

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

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:

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

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 $\mathtt{hom}_{\mathcal{C}}(A, B)$ (for the initial case) exists and showing that it must be equal to the one constructed.

2.3.3 Functors

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

1. A function F 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 $\mathcal{D}$:

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

These pieces of data should satisfy these two properties:

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

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

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

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

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

2.3.4 (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 $\phi \in \mathtt{hom}_{\mathcal{C}}(F(C), C)$

An **F-Algebra Homomorphism** is, given by two F-Algebras $(C, \phi), (D, \psi)$, and a morphism $f \in \mathtt{hom}_{\mathcal{C}}(C, D)$, 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 $\mathtt{Id}_C$ 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 $\phi \in \mathtt{hom}_{\mathcal{C}}(C, F(C))$

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

2.3.5 Catamorphisms 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 $\llparenthesis \phi \rrparenthesis \in \mathtt{hom}_{\mathcal{C}}(\mu^F, C)$ such that the following diagram commutes i.e., $\llparenthesis \phi \rrparenthesis \circ in = \phi \circ F\llparenthesis \phi \rrparenthesis$:

$$\begin{tikzcd}
F\mu^F \arrow[r, "in"] \arrow[d, "F\llparenthesis \phi \rrparenthesis"'] & \mu^F \arrow[d, "\llparenthesis \phi \rrparenthesis"] \\
FC \arrow[r, "\phi"] & C
\end{tikzcd}$$

A morphism of the form $\llparenthesis \phi \rrparenthesis$ is called a **catamorphism**.

A converse definition of catamorphisms exists, for terminal objects in $\mathcal{C}o\mathcal{A}lg(F)$ exists, called **anamorphisms**, denoted by $\llbracket \phi \rrbracket$

2.3.6 Fusion property

Now for the definition we've been building to, **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 $f \in \mathtt{hom}_{\mathcal{C}}(C, D)$ we have a **fusion property**:

$$f \circ \phi = \psi \circ F(f) \implies f \circ \llparenthesis \phi \rrparenthesis = \llparenthesis \psi \rrparenthesis$$

In English, if f is an F-Algebra homomorphism, we know that the composition of f and the catamorphism of ϕ equals the catamorphism of ψ ($f \circ \llparenthesis \phi \rrparenthesis = \llparenthesis \psi \rrparenthesis$). We can fuse two functions into one! This is summarized in the following diagram:

$$\begin{tikzcd}
F\mu^F \arrow[r, "in"] \arrow[d, "F\llparenthesis \phi \rrparenthesis"'] \arrow[dd, bend right=60, "F\llparenthesis \psi \rrparenthesis"'] & \mu^F \arrow[d, "\llparenthesis \phi \rrparenthesis"] \arrow[dd, bend left=60, "\llparenthesis \psi \rrparenthesis"] \\
FC \arrow[r, "\phi"] \arrow[d, "Ff"'] & C \arrow[d, "f"] \\
FD \arrow[r, "\psi"] & D
\end{tikzcd}$$

The top square commutes by initiality of `(μF, in)`. The bottom one is the precondition, and the right triangle is the fusion.

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

Having described all of this category theory, you might have a natural question: How does this relate to foldr/build fusion? To tie all this together, we will describe Harper (2011)'s work in Section 2.4, who discusses a more generalized form of foldr/build list fusion, allowing for a much broader class of datastructures through Church encodings. As well as a generalized form of destroy/unfoldr fusion through Cochurch encodings.

Before describing Harper's work, it is prudent to clearly show the correspondence between category theory terms and functional programming terms that we use interchangeably. This can be seen in Table 1

category theory	**functional programming**
catamorphism	fold
anamorphism	unfold
F-algebra	algebra
F-coalgebra	coalgebra
F-algebra initiality	universal property of folds
F-coalgebra terminality	universal property of unfolds

Table 1: The above tables matches category theoretical terms to functional programming terms.

2.4 Library Writer's Guide to Shortcut Fusion

Now that the sufficient category theory has been explained, it is possible to describe the work of Harper (2011), which my thesis centers on, called "A Library Writer's Guide to Shortcut Fusion".

In his work, Harper explains 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. We will now go through each step briefly.

2.4.1 Category Theory

For the full overview of the category theory, see Section 2.3. The main concepts that Harper 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 described earlier.

The universal property of folds is as follows:

$$h = \llparenthesis a \rrparenthesis \iff h \circ in = a \circ Fh$$

The fusion law as:

$$h \circ \llparenthesis a \rrparenthesis = \llparenthesis b \rrparenthesis \impliedby h \circ a \ = b \circ Fh$$

And the reflection law as:

$$\llparenthesis in \rrparenthesis = id$$

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, and can be found in the formalization in Section 4.3.1.

2.4.2 Encodings

Harper, before describing Church and Cochurch encodings, first discusses what merits a correct encoding of a datatype. His reason for creating an encoding is to encode recursive functions, which are not inlined by Haskell's optimizer, into nonrecursive 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" (Harper, 2011).

Now, instead of writing functions over a normal datatype μ`F`, we write functions over an encoded datatype `C`, along with two conversion functions `con:` μ`F` $\to$ `C` (concrete) and `abs :` `C` $\to$ μ`F` (abstract), which will enable us to convert from one datatype to another. 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.

This requirement above is a proof obligation, Harper states three additional ones which are summarized in the following three commutative diagrams:

$$\begin{tikzcd} \mu F \arrow[d, "f"'] & C \arrow[l, "abs"'] \arrow[d, "f_C"] \\ \mu F & C \arrow[l, "abs"] \end{tikzcd}$$

$$f \circ abs = abs \circ f_C$$

$$\begin{tikzcd} S \arrow[d, "p"'] \arrow[dr, "p_C"] & \\ \mu F & C \arrow[l, "abs"] \end{tikzcd}$$

$$p = abs \circ p_C$$

$$\begin{tikzcd} \mu F \arrow[r, "con"] \arrow[d, "c"'] & C \arrow[dl, "c_C"] \\ T & \end{tikzcd}$$

$$c = c_C \circ con$$

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

Harper also describes a fifth lemma: $cons \circ abs = id_C$, but he initially mentions that this is too strong of a condition to require from an encoding, requiring it to be an isomorphism. However, he did end up proving it later on in his proofs, once for Church encodings and once for Cochurch encodings. In fact, he uses the fifth proof as the basis for the fusion pragma in Haskell. It is the basis for correctness for the (Co)Church encodings he later ends up presenting in Haskell.

He did end up proving this fifth proof using the free theorems, pulled from the type of the polymorphic functions that the (Co)Church encodings contain. That he first discourages this fifth proof, only to subsequently prove it seems a bit inconsistent, but the fact that he did end up proving it and using it for the basis of the fusion he implemented in Haskell indicates that proving this fifth proof *is* important.

2.4.3 (Co)Church Encodings

Next, Harper (2011) proposes two encodings, `Church` and `CoChurch`.

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

$$\textbf{data}\ \mathit{Church}\ F = \mathit{Ch}\ (\forall\ A\ .\ (F\ A \to A) \to A)$$

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

$$\begin{aligned}
&\mathit{toCh} :: \mathit{mu}\ F \to \mathit{Church}\ F\\
&\mathit{toCh}\ x = \mathit{Ch}\ (\lambda a \to \mathit{fold}\ a\ x)\\
&\mathit{fromCh} :: \mathit{Church}\ F \to \mathit{mu}\ F\\
&\mathit{fromCh}\ (\mathit{Ch}\ g) = g\ \textbf{in}
\end{aligned}$$

Where `in` is the initial algebra $\mathit{in} : F(\mu F) \to \mu F$. From these definitions, Harper proves the four proof obligations, showing Church encodings to be a faithful encoding; along with a fifth proof, 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:

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

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

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

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

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

Where `out` is the terminal coalgebra $\mathit{out} : \nu F \to F(\nu F)$. 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 uses the free theorem and, in addition, the fusion property for unfolds.

2.4.4 Example implementation

After describing (Co)Church encodings, Harper goes on to demonstrate how they are used by implementing an example (Co)Church encoding of 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 functions that are typically used in Haskell, to Church and Cochurch encoded versions. To conclude he compares the performance of different compositions of functions to show the performance benefits and differences between the three different variants of functions.

We are omitting details for the description of Harper's example implementation because my work replicates this example implementation and is therefore described in detail in Section 3

3 Haskell Optimizations

In Harper (2011)'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?

So I pose the following research question(s): To implement (Co)Church encodings, what is necessary to make the code reliably fuse? This leads to the following sub-questions:

- What optimizations are present in Haskell that enable fusion to work?
- What tools and techniques are available to get Haskell's compiler to cooperate and trigger fusion?

In this section we will discuss my work replicating the fused Haskell code of Harper's work and further optimization opportunities that were discovered along the way.

We will 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 Harper's example code, starting in Section 3.2.1. We then discuss the performance impact of implementing fusion on an example function pipeline. The section is finished by a discussion of the results.

3.1 Leaf Trees

In this section, we discuss my replication of Harper (2011)'s code.

Datatypes In his paper Harper 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 parameter of the datatype:

$$\begin{array}{l} \mathbf{data}\ \mathit{Tree}\ a = \mathit{Empty} \mid \mathit{Leaf}\ a \mid \mathit{Fork}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a) \\ \mathbf{data}\ \mathit{Tree\_}\ a\ b = \mathit{Empty\_} \mid \mathit{Leaf\_}\ a \mid \mathit{Fork\_}\ b\ b \end{array}$$

Church encoding We encode our function using Church encodings using non-recusive algebras combined with induction principles rather than solely recursive functions. This way, the algebras can first be composed by Haskell's optimizer and then passed to the recursion principle. This ensures that the fused result only traverses the datastructure once.

The algebra being encoded is in this case $(\mathit{Tree\_}\ a\ b \rightarrow b)$. The Church encoding of the `Tree` datatype is defined using the base functor for `Tree`:

$$\mathbf{data}\ \mathit{TreeCh}\ a = \mathit{TreeCh}\ (\forall\ b\ .\ (\mathit{Tree\_}\ a\ b \rightarrow b) \rightarrow b)$$

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

$$\begin{array}{l} \mathit{toCh} :: \mathit{Tree}\ a \rightarrow \mathit{TreeCh}\ a \\ \mathit{toCh}\ t = \mathit{TreeCh}\ (\lambda a \rightarrow \mathit{fold}\ a\ t) \\ \quad \mathbf{where}\ \mathit{fold} :: (\mathit{Tree\_}\ a\ b \rightarrow b) \rightarrow \mathit{Tree}\ a \rightarrow b \\ \qquad\quad \mathit{fold}\ a\ \mathit{Empty} = a\ \mathit{Empty\_} \\ \qquad\quad \mathit{fold}\ a\ (\mathit{Leaf}\ x) = a\ (\mathit{Leaf\_}\ x) \\ \qquad\quad \mathit{fold}\ a\ (\mathit{Fork}\ l\ r) = a\ (\mathit{Fork\_}\ (\mathit{fold}\ a\ l) \\ \qquad\qquad\qquad\qquad\qquad\qquad (\mathit{fold}\ a\ r)) \\ \mathit{fromCh} :: \mathit{TreeCh}\ a \rightarrow \mathit{Tree}\ a \\ \mathit{fromCh}\ (\mathit{TreeCh}\ \mathit{fold}) = \mathit{fold}\ \mathit{in}' \\ \quad \mathbf{where}\ \mathit{in}' :: \mathit{Tree\_}\ a\ (\mathit{Tree}\ a) \rightarrow \mathit{Tree}\ a \\ \qquad\quad \mathit{in}'\ \mathit{Empty\_} = \mathit{Empty} \\ \qquad\quad \mathit{in}'\ (\mathit{Leaf\_}\ x) = \mathit{Leaf}\ x \\ \qquad\quad \mathit{in}'\ (\mathit{Fork\_}\ l\ r) = \mathit{Fork}\ l\ r \end{array}$$

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. This way the `toCh/fromCh` functions exist for as long as possible, maximizing the opportunity for the `RULES` pragma to identify adjacent `toCh/fromCh` pairs and fuse them away throughout the compilation process:

```
{-# RULES "toCh/fromCh fusion" forall x. toCh (fromCh x) = x #-}
{-# INLINE [0] toCh #-}
{-# INLINE [0] fromCh #-}
```

A generalized transformation function is defined to standardize and ease later implementations of transformation functions:

$$\begin{array}{l} \mathit{natCh} :: (\forall\ c\ .\ \mathit{Tree\_}\ a\ c \rightarrow \mathit{Tree\_}\ b\ c) \rightarrow \mathit{TreeCh}\ a \rightarrow \mathit{TreeCh}\ b \\ \mathit{natCh}\ f\ (\mathit{TreeCh}\ g) = \mathit{TreeCh}\ (\lambda a \rightarrow g\ (a\ .\ f)) \end{array}$$

Cochurch encoding Conversely to Church encodings, for Cochurch encodings we encode our function using non-recusive coalgebras combined with coinduction principles rather than solely recursive functions. Similarly to Church encodings, the coalgebras can first be composed by Haskell's optimizer after which they are passed to the corecursion principle. This again ensures that the fused result only traverses the datastructure once.

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

$$\textbf{data}\ \mathit{TreeCoCh}\ a = \forall\ s\ .\ \mathit{TreeCoCh}\ (s \to \mathit{Tree\_}\ a\ s)\ s$$

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

```
toCoCh :: Tree a → TreeCoCh a
toCoCh = TreeCoCh out
  where 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
  where 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. These work in the same fashion within Haskell as they do for Church encodings:

```
{-# RULES "toCh/fromCh fusion" forall x. toCoCh (fromCoCh x) = x #-}
{-# INLINE [0] toCoCh #-}
{-# INLINE [0] fromCoCh #-}
```

A generalized transformation function is defined again to standardize and ease later implementations of transformation functions:

```
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 non-encoded function 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:

```
between2 :: (Int, Int) → Tree Int
between2 = fromCh . betweenCh
  where betweenCh :: (Int, Int) → TreeCh Int
        betweenCh (x, y) = TreeCh (λa → b a (x, y))
        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
  {-# INLINE between2 #-}
```

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

```
between3 :: (Int, Int) → Tree Int
between3 = fromCoCh . TreeCoCh h
  where 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
  {-# 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 constructed. 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:

$$
\begin{aligned}
&\mathit{sum1} :: \mathit{Tree}\ \mathit{Int} \to \mathit{Int} \\
&\mathit{sum1}\ \mathit{Empty} = 0 \\
&\mathit{sum1}\ (\mathit{Leaf}\ x) = x \\
&\mathit{sum1}\ (\mathit{Fork}\ x\ y) = \mathit{sum1}\ x + \mathit{sum1}\ y
\end{aligned}
$$

The Church encoded version leverages an algebra `s`:

$$
\begin{aligned}
&\mathit{sum2} :: \mathit{Tree}\ \mathit{Int} \to \mathit{Int} \\
&\mathit{sum2} = \mathit{sumCh}\ .\ \mathit{toCh} \\
&\quad \textbf{where}\ \mathit{sumCh} :: \mathit{TreeCh}\ \mathit{Int} \to \mathit{Int} \\
&\qquad\qquad \mathit{sumCh}\ (\mathit{TreeCh}\ g) = g\ s \\
&\qquad\qquad s :: \mathit{Tree\_}\ \mathit{Int}\ \mathit{Int} \to \mathit{Int} \\
&\qquad\qquad s\ \mathit{Empty\_} = 0 \\
&\qquad\qquad s\ (\mathit{Leaf\_}\ x) = x \\
&\qquad\qquad s\ (\mathit{Fork\_}\ x\ y) = x + y \\
&\quad \{\text{-\# INLINE sum2 \#-}\}
\end{aligned}
$$

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 Section 3.3.4:

$$
\begin{aligned}
&\mathit{sum3} :: \mathit{Tree}\ \mathit{Int} \to \mathit{Int} \\
&\mathit{sum3} = \mathit{sumCoCh}\ .\ \mathit{toCoCh} \\
&\quad \textbf{where}\ \mathit{sumCoCh} :: \mathit{TreeCoCh}\ \mathit{Int} \to \mathit{Int} \\
&\qquad\qquad \mathit{sumCoCh}\ (\mathit{TreeCoCh}\ h\ s') = \mathit{loop}\ s' \\
&\qquad\qquad\quad \textbf{where}\ \mathit{loop}\ s = \textbf{case}\ h\ s\ \textbf{of} \\
&\qquad\qquad\qquad\qquad \mathit{Empty\_} \to 0 \\
&\qquad\qquad\qquad\qquad \mathit{Leaf\_}\ x \to x \\
&\qquad\qquad\qquad\qquad \mathit{Fork\_}\ l\ r \to \mathit{loop}\ l + \mathit{loop}\ r \\
&\quad \{\text{-\# INLINE sum3 \#-}\}
\end{aligned}
$$

Pipelines Finally, an example pipeline, whose performance can be measure or Core representation inspected[2], is defined:

$$
\begin{aligned}
&\mathit{pipeline1} :: (\mathit{Int}, \mathit{Int}) \to \mathit{Int} \\
&\mathit{pipeline1} = \mathit{sum1}\ .\ \mathit{map1}\ (+2)\ .\ \mathit{filter1}\ \mathit{odd}\ .\ \mathit{between1}
\end{aligned}
$$

3.2 Lists

In this section we discuss my further replication of Harper (2011)'s work. We implement some of the functions and pipelines that Harper described, such as `between`, `filter`, and `sum`, but using the List datatype instead of Leaf Trees. This was done to see how the descriptions in Harper'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:

$$
\begin{aligned}
&\textbf{import}\ \mathit{GHC.List} \\
&\textbf{data}\ \mathit{List'}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List'}\ a) \\
&\textbf{data}\ \mathit{List\_}\ a\ b = \mathit{Nil\_} \mid \mathit{Cons\_}\ a\ b
\end{aligned}
$$

[2]For the documentation for which flags to use on GHC, see https://downloads.haskell.org/ghc/latest/docs/users_guide/debugging.html#core-representation-and-simplification

Church encodings We define the Church encoding and proper encoding and decoding functions:

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

We omit the pragmas defined for `toCh` and `fromCh` as well as the `natCh`, as their definition is identical to the ones defined for Leaf Trees.

Cochurch encodings We defined the Cochurch encodings conversely:

```
data ListCoCh a = ∀ s . ListCoCh (s → List_ a s) s
toCoCh :: List' a → ListCoCh a
toCoCh = ListCoCh out
  where 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
  where 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)
```

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:

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

For the Cochurch encoded version we define a coalgebra:

```
between3 :: (Int, Int) → List′ Int
between3 = fromCoCh . ListCoCh betweenCoCh
  where betweenCoCh :: (Int, Int) → List_ Int (Int, Int)
        betweenCoCh (x, y) = if x > y
                             then Nil_
                             else Cons_ x (x + 1, y)
        {-# INLINE betweenCoCh #-}
 {-# INLINE between3 #-}
```

Map It is possible to implement the `map` function using 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: unencoded, 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 to the existing recursion principle:

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

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

A second recursion principle is also implemented that modifies the type of the recursion element in the base functor. Leveraging call arity techniques as described and made possible by Breitner (2018) to obtain a tail recursive implementation of sum for Church encodings.

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

```haskell
sum3 :: List' Int → Int
sum3 = sumCoCh . toCoCh
  where sumCoCh :: ListCoCh Int → Int
        sumCoCh (ListCoCh h s) = su h s
        su :: (s → List_ Int s) → s → Int
        su h s = loopt s 0
          where loopt s' sum = case h s' of
                  Nil_ → sum
                  Cons_ x xs → loopt xs (x + sum)
{-# INLINE sum3 #-}

sum8 :: List' Int → Int
sum8 = sumCoCh . toCoCh
  where sumCoCh :: ListCoCh Int → Int
        sumCoCh (ListCoCh h s) = su h s
        su :: (s → List_ Int s) → s → Int
        su h s = loop s
          where loop s' = case h s' of
                  Nil_ → 0
                  Cons_ x xs → x + loop xs
{-# INLINE sum8 #-}
```

Note that two subfunctions are provided to `su'`, the `loop` and the `loopt` function. The former function is implemented using typical recursion. 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 Section 3.3.4.

Pipelines and GHC list fusion Now we can make a pipeline in the following fashion:

```haskell
pipeline1 :: (Int, Int) → Int
pipeline1 = sum1 . map1 (+2) . filter1 trodd . between1
```

You may notice I have not yet discussed the filter function, this is for a good reason, which I will discuss now.

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:

```haskell
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 #-}
```

However, the (Co)Church encoded version, contrary to `map`, can not be implemented using a natural transformation.

The following section will start to answer the following research question: What tools and techniques are available to get Haskell's compiler to cooperate and trigger fusion?

3.2.1 The Filter Problem

There are multiple ways of implementing Church and Cochurch encoded filter functions, none of them immediately obvious from Harper (2011)'s description of how it should be implemented as a natural transformation.

When replicating Harper's code for lists, there is one major limitation on natural 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:

$$
\begin{array}{l}
\mathit{filt} :: (a \to \mathit{Bool}) \to \mathit{Tree\_}\ a\ c \to \mathit{Tree\_}\ a\ c \\
\mathit{filt}\ p\ \mathit{Empty\_} = \mathit{Empty\_} \\
\mathit{filt}\ p\ (\mathit{Leaf\_}\ x) = \textbf{if}\ p\ x\ \textbf{then}\ \mathit{Leaf\_}\ x\ \textbf{else}\ \mathit{Empty\_} \\
\mathit{filt}\ p\ (\mathit{Fork\_}\ l\ r) = \mathit{Fork\_}\ l\ r \\
\mathit{filter2} :: (a \to \mathit{Bool}) \to \mathit{Tree}\ a \to \mathit{Tree}\ a \\
\mathit{filter2}\ p = \mathit{fromCh}\ .\ \mathit{natCh}\ (\mathit{filt}\ p)\ .\ \mathit{toCh} \\
\mathit{filter3} :: (a \to \mathit{Bool}) \to \mathit{Tree}\ a \to \mathit{Tree}\ a \\
\mathit{filter3}\ p = \mathit{fromCoCh}\ .\ \mathit{natCoCh}\ (\mathit{filt}\ p)\ .\ \mathit{toCoCh}
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{filt} :: (a \to \mathit{Bool}) \to \mathit{List\_}\ a\ c \to \mathit{List\_}\ a\ c \\
\mathit{filt}\ p\ \mathit{Nil\_} = \mathit{Nil\_} \\
\mathit{filt}\ p\ (\mathit{Cons\_}\ x\ xs) = \textbf{if}\ p\ x\ \textbf{then}\ \mathit{Cons\_}\ x\ xs\ \textbf{else}?
\end{array}
$$

The question is, what should be in the place of the `?` above? Initially one 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. We could try to modify the type to allow this change, but if we did that we wouldn't have the type of a natural 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 natural transformations, instead creating a new algebra from an existing one. 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 Before we wanted to implement our `filter` function in the following manner:

$$
\begin{array}{l}
\mathit{filterCh} :: (\forall\ c\ .\ \mathit{List\_}\ a\ c \to \mathit{List\_}\ b\ c) \to \mathit{ListCh}\ a \to \mathit{ListCh}\ b \\
\mathit{filterCh}\ p\ (\mathit{ListCh}\ g) = \mathit{ListCh}\ (\lambda a \to g\ (a\ .\ (\mathit{filt}\ p)))
\end{array}
$$

$$
\begin{array}{l}
\mathit{filter2} :: (a \to \mathit{Bool}) \to \mathit{List}'\ a \to \mathit{List}'\ a \\
\mathit{filter2}\ p = \mathit{fromCh}\ .\ \mathit{filterCh}\ p\ .\ \mathit{toCh} \\
\ \{\text{-\# INLINE filter2 \#-}\}
\end{array}
$$

We now need to modify the `filterCh` function such that we can still express a filter function *without* using a natural transformation:

$$
\begin{array}{l}
\mathit{filterCh} :: (\forall\ c\ .\ \mathit{List\_}\ a\ c \to \mathit{List\_}\ b\ c) \to \mathit{ListCh}\ a \to \mathit{ListCh}\ b \\
\mathit{filterCh}\ p\ (\mathit{ListCh}\ g) = \mathit{ListCh}\ (\lambda a \to g\ (?))
\end{array}
$$

Replacing the hole `?` in the expression `g (?)` above such that we apply the `a` selectively we can yield:

$$
\begin{array}{l}
\mathit{filterCh} :: (a \to \mathit{Bool}) \to \mathit{ListCh}\ a \to \mathit{ListCh}\ a \\
\mathit{filterCh}\ p\ (\mathit{ListCh}\ g) = \mathit{ListCh}\ (\lambda a \to g\ (\lambda\textbf{case} \\
\qquad \mathit{Nil\_} \to a\ \mathit{Nil\_} \\
\qquad \mathit{Cons\_}\ x\ xs \to \textbf{if}\ (p\ x)\ \textbf{then}\ a\ (\mathit{Cons\_}\ x\ xs)\ \textbf{else}\ xs \\
\quad))
\end{array}
$$

We create a new algebra from an existing one, `a`, that selectively postcomposes `a`, this is not a natural transformation anymore in the way that `f` below is.

$$
\begin{array}{l}
\mathit{natCh} :: (\forall\ c\ .\ \mathit{List\_}\ a\ c \to \mathit{List\_}\ b\ c) \to \mathit{ListCh}\ a \to \mathit{ListCh}\ b \\
\mathit{natCh}\ f\ (\mathit{ListCh}\ g) = \mathit{ListCh}\ (\lambda a \to g\ (a\ .\ f))
\end{array}
$$

In the new solution we do not apply `a` to `xs`, and, in doing so, can put `xs` in the place where we wanted to earlier. Before we were limited because the `natCh` function forced a postcomposition of `a` in all cases, which is now lifted by abandoning the `natCh` function.

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

$$\begin{aligned}&\mathit{filter3} :: (a \to \mathit{Bool}) \to \mathit{List}\ a \to \mathit{List}\ a\\ &\mathit{filter3}\ p = \mathit{fromCoCh}\ .\ \mathit{natCoCh}\ (\mathit{filt}\ p)\ .\ \mathit{toCoCh}\end{aligned}$$

For the Cochurch encoding, a natural transformation can be defined, but it is not a simple coalgebra, instead it is a recursive function.[3] The core idea is: we combine the natural transformation and post-composition again, but this time we make the function recursively grab elements from the seed until it finds one that satisfies the predicate.

$$\begin{aligned}&\mathit{filt} :: (a \to \mathit{Bool}) \to (s \to \mathit{List\_}\ a\ s) \to s \to \mathit{List\_}\ a\ s\\ &\mathit{filt}\ p\ h\ s = \mathit{go}\ s\\ &\quad \mathbf{where}\ \mathit{go}\ s = \mathbf{case}\ h\ s\ \mathbf{of}\\ &\qquad\qquad \mathit{Nil\_} \to \mathit{Nil\_}\\ &\qquad\qquad \mathit{Cons\_}\ x\ \mathit{xs} \to \mathbf{if}\ p\ x\ \mathbf{then}\ \mathit{Cons\_}\ x\ \mathit{xs}\ \mathbf{else}\ \mathit{go}\ \mathit{xs}\\ &\mathit{filterCoCh} :: (a \to \mathit{Bool}) \to \mathit{ListCoCh}\ a \to \mathit{ListCoCh}\ a\\ &\mathit{filterCoCh}\ p\ (\mathit{ListCoCh}\ h\ s) = \mathit{ListCoCh}\ (\mathit{filt}\ p\ h)\ s\\ &\mathit{filter3} :: (a \to \mathit{Bool}) \to \mathit{List'}\ a \to \mathit{List'}\ a\\ &\mathit{filter3}\ p = \mathit{fromCoCh}\ .\ \mathit{filterCoCh}\ p\ .\ \mathit{toCoCh}\\ &\ \{\text{-\# INLINE filter3 \#-}\}\end{aligned}$$

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.[4]

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:

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

Now we do need to modify all of our already defined functions to take into account this modified datatype. Readers familiar with the work might notice that this technique is in fact *stream fusion* as described by Coutts et al. (2007). 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 changing 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 changing the list datatype such that this nullability is also possible, we can now 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:

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

The `Leaf` constructor of `BinTree` is made nullable. We will leave the following question to future work: Is this generalizable to any datastructure (perhaps containers)?

[3] And not necessarily guaranteed to terminate, the seed could generate an infinite structure.
[4] https://gitlab.haskell.org/ghc/ghc/-/issues/22227

3.3 Haskell's optimization pipeline

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

Repeating my research question, the first sub-point should be answered here: What optimizations are present in Haskell that enable fusion to work?

3.3.1 Beta reduction

Beta reduction is 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`.

3.3.2 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`.

3.3.3 Case of case, and known-case elimination

As discussed by Jones (1996), case of case optimization is the transformation of the following pattern:

case (
  case C **of**
    $B1 \rightarrow F1$
    $B2 \rightarrow F2$
) **of**
  $A1 \rightarrow E1$
  $A2 \rightarrow E2$

To the following[5]:

case C **of**
  $B1 \rightarrow$ **case** $F1$ **of**
    $A1 \rightarrow E1$
    $A2 \rightarrow E2$
  $B2 \rightarrow$ **case** $F2$ **of**
    $A1 \rightarrow E1$
    $A2 \rightarrow E2$

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

case V **of**
  $V \rightarrow Expr$
  ...

Can be simplified by case-of-known-constructor optimization to:

$Expr$

Together, these optimizations can often lead to the removal of unnecessary computations. Take as an example (Jones, 1996):

if $(\neg\ x)$ **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" (Jones, 1996):

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

$$
\begin{aligned}
&\textbf{case}\ (\textbf{case}\ x\ \textbf{of}\\
&\quad \mathit{True} \rightarrow \mathit{False}\\
&\quad \mathit{False} \rightarrow \mathit{True}\\
&)\ \textbf{of}\\
&\quad \mathit{True} \rightarrow \mathit{E1}\\
&\quad \mathit{False} \rightarrow \mathit{E2}
\end{aligned}
$$

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

$$
\begin{aligned}
&\textbf{case}\ x\ \textbf{of}\\
&\quad \mathit{True} \rightarrow \textbf{case}\ \mathit{False}\ \textbf{of}\\
&\qquad \mathit{True} \rightarrow \mathit{El}\\
&\qquad \mathit{False} \rightarrow \mathit{E2}\\
&\quad \mathit{False} \rightarrow \textbf{case}\ \mathit{True}\ \textbf{of}\\
&\qquad \mathit{True} \rightarrow \mathit{El}\\
&\qquad \mathit{False} \rightarrow \mathit{E2}
\end{aligned}
$$

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

$$
\begin{aligned}
&\textbf{case}\ x\ \textbf{of}\\
&\quad \mathit{True} \rightarrow \mathit{E2}\\
&\quad \mathit{False} \rightarrow \mathit{E1}
\end{aligned}
$$

No more runtime evaluation of not!

3.3.4 Tail Recursion

Definition We call a recursive function tail-recursive, if all its recursive calls return 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, only needing to execute make a jump each time a recursive 'call' is made. Haskell is able to identify tail-recursive functions and optimize the compiled byte code accordingly[6].

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 Section C.2 for derivation):

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

This definition is not tail recursive as the `then (x + 2) + loop (x+1, y)` line includes some calculations 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 Section C.3 for derivation):

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

Reduces to:

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

[6] See Simon Peyton Jones' excellent talk on tail recursion and join points here: https://www.youtube.com/watch?v=LMTr8yw0Gk4

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 Breitner (2018)'s work and Section 3.4.

3.3.5 Performance Considerations

We discuss many different considerations and details when optimizing the fusible code. This discussion is summarized here. In order to make sure a pipeline of functions fuses in Haskell, there are several things that need to be taken into consideration:

- Make sure you only pass through parameters that change between recursive calls. Instead of writing:

$$
\begin{aligned}
&b\ a\ (x, y) = loop\ x\ y \\
&\quad \textbf{where}\ loop\ x\ y = \textbf{if}\ x > y \\
&\qquad\qquad\qquad\qquad \textbf{then}\ a\ Nil\_ \\
&\qquad\qquad\qquad\qquad \textbf{else}\ a\ (Cons\_\ x\ (loop\ (x+1)\ y))
\end{aligned}
$$

  Where the `y` doesn't change between calls of `loop`, modify `loop` such that it doesn't pass through the `y`:

$$
\begin{aligned}
&b\ a\ (x, y) = loop\ x \\
&\quad \textbf{where}\ loop\ x = \textbf{if}\ x > y \\
&\qquad\qquad\qquad\quad \textbf{then}\ a\ Nil\_ \\
&\qquad\qquad\qquad\quad \textbf{else}\ a\ (Cons\_\ x\ (loop\ (x+1)))
\end{aligned}
$$

  This way, data needs to be pushed around in memory for each (recursive) function call.

- Ensure that functions are inlined properly. So for the second example above add a pragma that inlines the function. This ensures that other pragmas, that do the actual fusion, can fire during the compilation process:

  {-# INLINE betweenCh #-}

- Ensure that the fused result is tail recursive. For consumer functions, it is often possible to make the function tail recursive. For the corecursion principle of sum `su`:

$$
\begin{aligned}
&su :: (s \to List\_\ Int\ s) \to s \to Int \\
&su\ h\ s = loop\ s \\
&\quad \textbf{where}\ loop\ s' = \textbf{case}\ h\ s'\ \textbf{of} \\
&\qquad\qquad Nil\_ \to 0 \\
&\qquad\qquad Cons\_\ x\ xs \to x + loop\ xs
\end{aligned}
$$

  It is possible to modify the definition of the corecursion `loop` such that it is tail-recursive:

$$
\begin{aligned}
&su :: (s \to List\_\ Int\ s) \to s \to Int \\
&su\ h\ s = loopt\ s\ 0 \\
&\quad \textbf{where}\ loopt\ s'\ sum = \textbf{case}\ h\ s'\ \textbf{of} \\
&\qquad\qquad Nil\_ \to sum \\
&\qquad\qquad Cons\_\ x\ xs \to loopt\ xs\ (x + sum)
\end{aligned}
$$

  For Church encodings, it is a little more tricky to get the resultant function to be tail-recursive, it is possible, however. Taking the algebra for sum again:

$$
\begin{aligned}
&sum2 :: List'\ Int \to Int \\
&sum2 = sumCh\ .\ toCh \\
&\quad \textbf{where}\ sumCh :: ListCh\ Int \to Int \\
&\qquad\quad sumCh\ (ListCh\ g) = g\ su \\
&\qquad\quad su :: List\_\ Int\ Int \to Int \\
&\qquad\quad su\ Nil\_ = 0 \\
&\qquad\quad su\ (Cons\_\ x\ y) = x + y
\end{aligned}
$$

  We can modify the type of the recursive part of list and the return type to be a function instead of just a simple datatype (`Int -> Int` instead of `Int`):

$$
\begin{aligned}
&\mathit{sum7} :: \mathit{List}'\ \mathit{Int} \to \mathit{Int}\\
&\mathit{sum7} = \mathit{flip}\ \mathit{sumCh}\ 0\ .\ \mathit{toCh}\\
&\quad \textbf{where}\ \mathit{sumCh} :: \mathit{ListCh}\ \mathit{Int} \to (\mathit{Int} \to \mathit{Int})\\
&\qquad\qquad \mathit{sumCh}\ (\mathit{ListCh}\ g) = g\ \mathit{su}\\
&\qquad\qquad \mathit{su} :: \mathit{List\_}\ \mathit{Int}\ (\mathit{Int} \to \mathit{Int}) \to (\mathit{Int} \to \mathit{Int})\\
&\qquad\qquad \mathit{su}\ \mathit{Nil\_}\ s = s\\
&\qquad\qquad \mathit{su}\ (\mathit{Cons\_}\ x\ y)\ s = y\ (s + x)
\end{aligned}
$$

Breitner (2018) introduced and subsequently make possible this optimization in Haskell, called call arity.

- Ensure that the fused result is a single recursive function (so no helper functions such as `go`). This was a problem when writing the filter function in Cochurch encoded fashion. This is only possible if a recursive natural transformation function is used, but it unfortunately does not fuse. This is due to `go` being a recursive function, which Haskell does not inline on its own.
  The current workaround for this is Stream fusion as described by Coutts et al. (2007). Adding a Skip constructor makes a big difference to enabling the avoidance of recursive functions. This workaround is part of my performance analysis.

3.4 Performance Comparison

We tested the performance of the following pipeline:

$$f = \mathit{sum}\ .\ \mathit{map}\ (+2)\ .\ \mathit{filter}\ \mathit{odd}\ .\ \mathit{between}$$

We define 11 different variants of the above functions which can be categorized into the following five groups:

- Unfused
- Hand fused
- GHC List fused
- Church fused (normally, with tail recursion[7], with skip constructor, and both)
- Cochurch fused (normally, with tail recursion, with skip constructor, and both)

For the implementation of all the functions, see the source code in the artifacts.

For the testing I ensured that the first two of the points in 3.3.5 were satisfied, partially through analysis of the GHC[8] generated Core representation. The latter two points became part of the testing setup. I measured the performance using tasty-bench[9]. I tested all of the piplines with an input going from `(1, 100)` to `(1, 10000000)`, running tastybench five times for each input, setting a maximum standard deviation of 2% of the mean result. For the presentation of the data I took the mean of these five runs. Tastybench keeps running tests until the standard deviation becomes small enough; each time running doubling the amount of runs before checking the new standard deviation. Tastybench measured time using CPU time.

3.4.1 Performance differences

There are two main results figures, which can be seen at Figure 4 and Figure 5. However, their y axes are logarithmic, due to the nature of the input sizes provided from `(1,100)` to `(1,10000000)` in powers of 10. It is more illustrative to look at a linear scale, and that is easiest when zooming in on one specific input. For the illustration, I will choose `(1,10000)` as input, as it is relatively representative. There are specific variations when changing scale, but those will be discussed in Section 3.4.2.

In Figure 3 you can see how implementing fusion can bring quite a large speedup to a function pipeline. With the following things of note:

- Tail recursive Church, stream Church, and stream Cochurch implementations were the fastest, and as fast as each other. A speedup of 25x over the unfused pipeline for this input.
- Stream fusion does not offer a speedup for Church encodings.
- Adding tail recursion speeds up the encoding in all cases, except:
- Church-encoded non-stream pipelines are not faster, this is due to a recursive natural transformation for filter (the function `go`).

[7] oneShot needed to be used in order to get this to work.

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

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

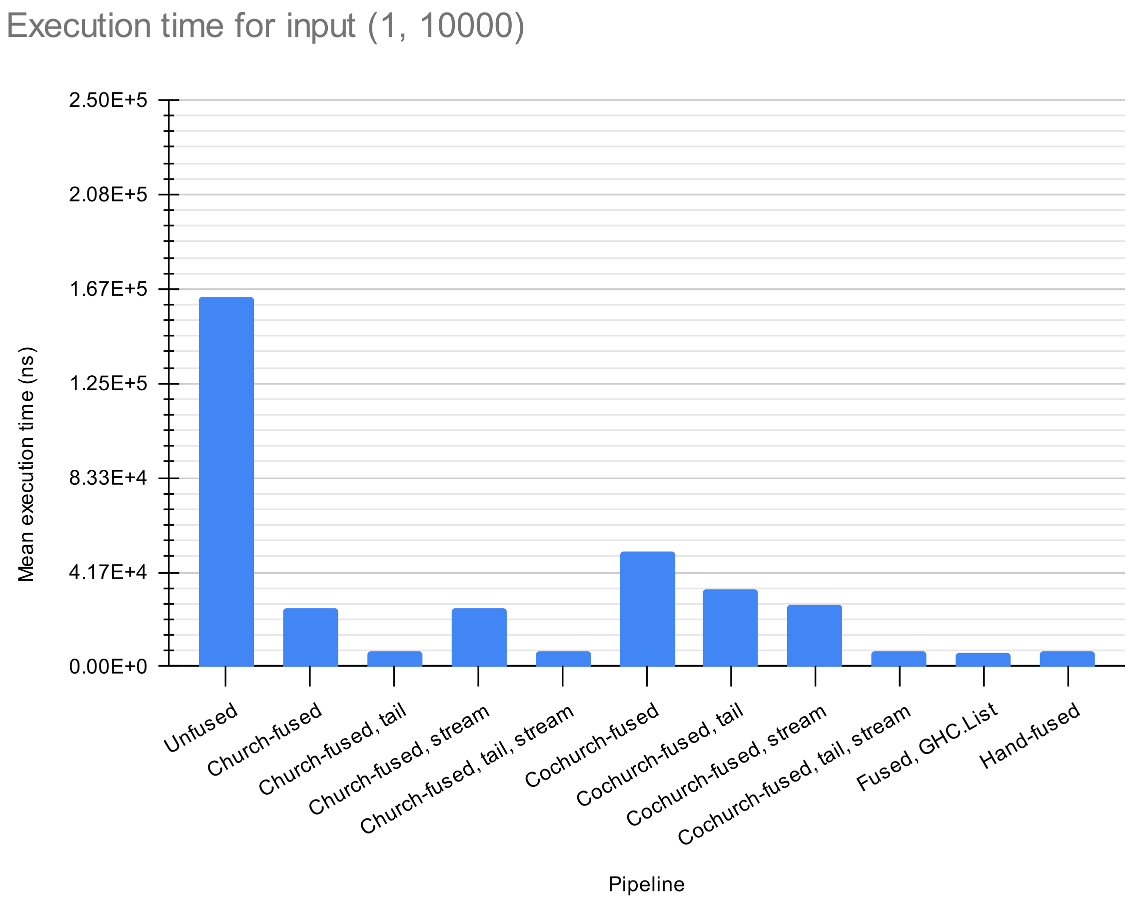

Figure 3: Comparison of execution times between the different pipelines.

3.4.2 Scale Variations

In general, as the scale increases (see Figure 12 and Figure 13 for all graphs):

- The factor speedup between the unfused and three fully fused pipelines increases, most notably between `(1, 10000)` and `(1,100000)`, going from 25.4x to 31.5x. This likely has to do with the increased volume of data that needs to be stored in random access memory.
- All the non tail-recursive encoding implementations get slower relative to the tail-recursive implementations. This is likely due to extra time spent allocating to and retrieving from memory.
- Most importantly for all fully fused pipelines: The speedup that fusion offers only seems to increase as the order of magnitude of the calculation grows.

These findings highlight how the fusion can provide a significant speedup to compiled Haskell code. The replication of Harper's code show that achieving fused, tail-recursive compiled code requires taking into consideration many parts of Haskell's optimizer. I believe that there exists big potential for Haskell's library writers to implement many datastructures using in a (Co)Church encodings; especially the foldr/build and destroy/unfoldr functions, from which many standard library functions and other functionalities can be derived.

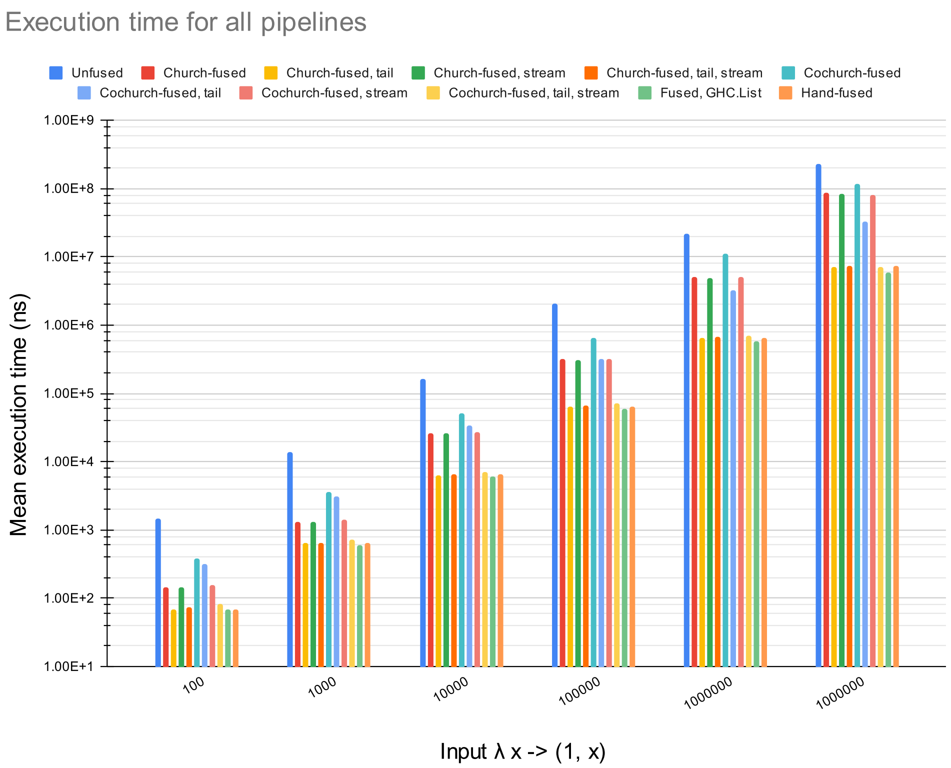

Figure 4: Comparison of executions times between the different pipelines and input sizes, bar chart

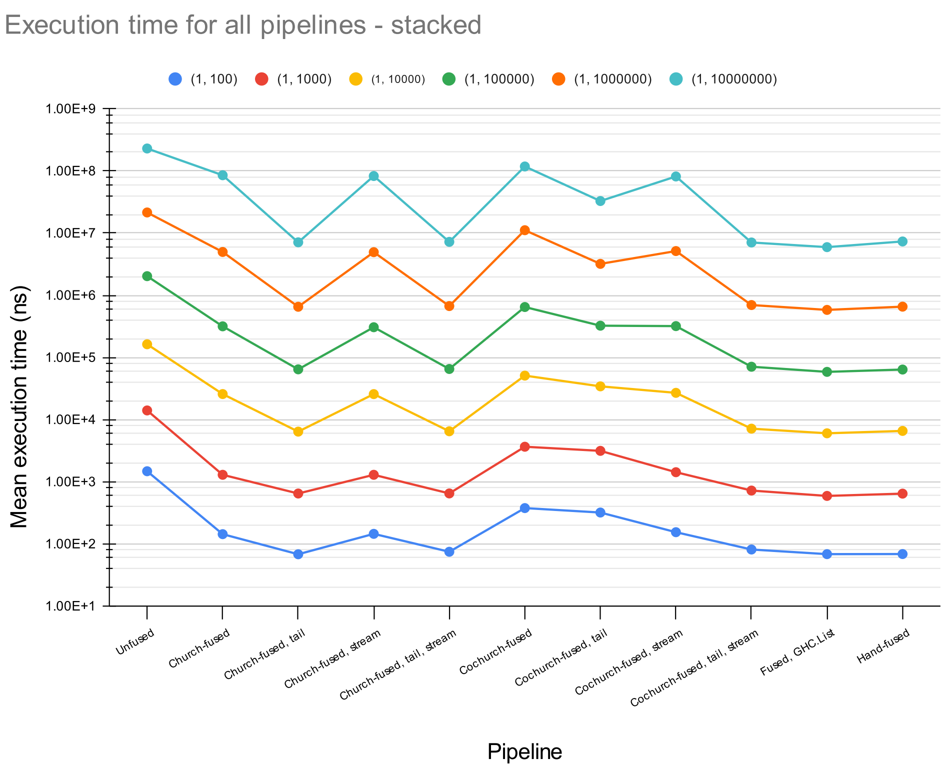

Figure 5: Comparison of executions times between the different pipelines and input sizes, line chart. This view makes it slightly easier to compare the differences between pipelines across different inputs.

4 Agda Formalization of the Optimization

In Harper (2011)'s work "A Library Writer's Guide to Shortcut Fusion", he describes the practice of implementing Church and Cochurch encodings, as well as a paper proof necessary to show that the encodings employed are correct. I pose and answer the following question in this section: Are the transformations used to enable fusion safe? Meaning:

- Do the transformations in Haskell preserve the semantics of the language?
- If the mathematics and the encodings are implemented in a dependently typed language, is it possible to prove them to be correct?

In this section we discuss the work that was done to formalize Harper's proofs in the programming language Agda, as well as additional proofs to support the claims made in the paper. My code is represented in roughly 3 parts, once for Church and once for Cochurch encodings, each part builds on the previous one:

- The proofs of the category theory properties, such as initiality/terminality of datatypes and the reflection property.
- The definition and proofs about the (Co)Church encodings, again as described by Harper.
- An example implementation of the list datatype, using containers.

The Agda code makes use of two libraries: agda-stdlib[10] v2.0 and agda-categories[11] v0.2.0.
The discussion of my implementation can be found in Section 4.4

4.1 Common definitions

Both the Church (initial) and Cochurch (terminal) halves of the formalization use these definitions.

Functional Extensionality I postulate functional extensionality. This is done through Agda's builtin Extensionality module:

```
postulate funext : ∀{a b} → Extensionality a b
funexti : ∀{a b} → ExtensionalityImplicit a b
funexti = implicit-extensionality funext
```

Containers In the Agda formalization we need to represent functors. While a `RawFunctor` datatype does exist in Agda's stdlib, it does not provide the necessary data such that proofs can easily be done over it, such as the functor laws.

Instead, we opt to use Containers to represent strictly positive functors as described by Abbott et al. (2005). 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 it maintains positivity as well as that the functor laws are true by definition. We will discuss the process for deriving a container from a given (polynomial) later on when we need to derive it for lists, in Section 4.2.4.

[10] https://github.com/agda/agda-stdlib
[11] https://github.com/agda/agda-categories

4.2 Church Encodings

4.2.1 Category Theory: Initiality

This section is about my formalization of Harper (2011)'s work that proves the needed category theory that is to be used later on in the (Co)Church part of the formalization. This section specifically defines the category of F-Algebras and proves initiality of (μF, in') (the universal properties of folds) and the fusion property.

Universal properties of catamorphisms and initiality This section proves the universal property of folds. It takes the definition of `W` types and shows that the ⦇_⦈ function defined for it is a catamorphism. This is done by proving that the fold is a unique F-algebra homomorphism to any datatype through a proof of existence and uniqueness:

$$\forall (C, \phi) \in \mathcal{A}lg(F)_0 : \exists ! (\!|\phi|\!) \in \mathtt{hom}_{\mathtt{Alg}(\mathcal{F})}(\mu F, C), s.t. (\!|\phi|\!) \circ in = \phi \circ F(\!|\phi|\!)$$

```
module agda.church.initial where
open import Data.W using () renaming (sup to in') public
```

I define a function below which turns out to be a catamorphism. This fact is proved in this section through a proof of existence `univ-to` and a proof of uniqueness `univ-from`. The `fold` function for containers in Agda's stdlib is defined identically and could be imported, omitting this definition. However, for clarity I'm including the definition here instead of importing it:

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

We show 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* (Harper, 2011):

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

This constitutes a proof of existence; there exists a function (in this case called ⦇_⦈), that is a valid F-Algebra homomorphism:

```
univ-to : {F : Container 0ℓ 0ℓ}{X : Set}{a : ⟦ F ⟧ X → X}{h : μ F → X} →
            ({x : μ F} → h x ≡ ⦇ a ⦈ x) → {x : ⟦ F ⟧ (μ F)} → (h ∘ in') x ≡ (a ∘ map h) x
univ-to {_}{_}{a}{h} eq {x@(op , ar)} = begin
    h (in' (op , ar))
  ≡⟨ eq ⟩
    ⦇ a ⦈ (in' (op , ar))
  ≡⟨⟩
    a (op , ⦇ a ⦈ ∘ ar)
  ≡⟨ cong (λ f → a (op , f)) (funext λ _ → sym eq) ⟩
    a (op , h ∘ ar)
  ≡⟨⟩
    a (map h x)
  ∎
```

We show 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* (Harper, 2011).

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

This constitutes a proof of uniqueness; for any function that is a valid F-Algebra homomorphism (in this case called h), it is equal to ⦇_⦈:

```
univ-from : {F : Container 0ℓ 0ℓ}{X : Set}{a : ⟦ F ⟧ X → X}(h : μ F → X) →
              ({x : ⟦ F ⟧ (μ F)} → (h ∘ in') x ≡ (a ∘ map h) x) → {x : μ F} → h x ≡ ⦇ a ⦈ x
univ-from {_}{_}{a} h eq {in' x@(op , ar)} = begin
    (h ∘ in') x
```

```
  ≡⟨ eq ⟩
    a (op , h ∘ ar)
  ≡⟨ cong (λ f → a (op , f)) (funext λ _ → univ-from h eq) ⟩
    a (op , ⦇ a ⦈ ∘ ar)
  ≡⟨⟩
    (⦇ a ⦈ ∘ in') x
  ∎
```

The two previous proofs, constituting a proof of existence and uniqueness, together prove initiality of `(μ F, in')`. The *reflection law* (Harper, 2011):

```
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 f → in' (op , f)) (funext λ _ → reflection) ⟩
    in' (op , ar)
  ≡⟨⟩ - Dfn of y
    y
  ∎
```

The fusion property, which follows from the backwards direction of the *universal property of folds*:

```
fusion : {F : Container 0ℓ 0ℓ}{A B : Set}{a : ⟦ F ⟧ A → A}{b : ⟦ F ⟧ B → B}{h : A → B} →
         ({x : ⟦ F ⟧ A} → (h ∘ a) x ≡ (b ∘ map h) x) → (x : μ F) → (h ∘ ⦇ a ⦈) x ≡ ⦇ b ⦈ x
fusion {_}{_}{_}{a}{b}{h} eq x = univ-from (h ∘ ⦇ a ⦈) eq {x}
```

4.2.2 Fusion: Church encodings

This section focuses on the fusion of Church encodings, leveraging parametricity (free theorems) (Wadler, 1989).

Definition of Church encodings This section 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 Harper (2011). 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 Gill et al. (1993)'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)

build : {ℓ : Level}{F : Container _ _}{Y : Set ℓ}
        (g : {X : Set} → (⟦ F ⟧ X → X) → Y → X)(y : Y) → μ F
```

```
build g = fromCh ∘ prodCh g

eqProd : {F : Container _ _}{Y : Set}{g : {X : Set} → (⟦ F ⟧ X → X) → Y → X} →
         build 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 Gill et al. (1993)'s `foldr` function.

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

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

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

4.2.3 Proof obligations

In Harper (2011)'s work, five proofs are given for Church encodings. These are formalized here. The first proof shows that `fromCh ∘ toCh = id`, using the reflection law. This corresponds to the first proof obligation mentioned in Section 2.4.2:

```
from-to-id : {F : Container 0ℓ 0ℓ}(x : μ F) →
             (fromCh ∘ toCh) x ≡ id x
from-to-id 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
  ∎
```

The second proof is similar to the first, but it proves the composition in the other direction `toCh ∘ fromCh = id`. This proof leverages parametricity as described by Wadler (1989). It postulates the free theorem of the function `g :∀ 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" (Harper, 2011). This together with the first proof shows that Church encodings are isomorphic to the datatypes they are encoding:

```
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}(x : Church F) →
             (toCh ∘ fromCh) x ≡ id x
to-from-id (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 Harper (2011)'s original proof steps are included here for completeness. This corresponds to the third proof obligation (second diagram) mentioned in Section 2.4.2:

```
cons-pres : {F : Container 0ℓ 0ℓ}{X : Set}(b : ⟦ F ⟧ X → X) →
            (x : μ F) → (consCh b ∘ toCh) x ≡ ⦇ b ⦈ x
cons-pres b x = 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
  ∎
```

$$\begin{tikzcd} S \arrow[d, "(\!|b|\!)"'] \arrow[dr, "consChb"] & \\ \mu F & C \arrow[l, "toCh"] \end{tikzcd}$$

$(\!|b|\!) = toCh \circ consChb$

The fourth proof shows that Church encoded functions constitute an implementation for the producing functions being replaced. The proof is proved via reflexivity, but Harper (2011)'s original proof steps are included here for completeness. This corresponds to the fourth proof obligation (third diagram) mentioned in Section 2.4.2:

```
prod-pres : {F : Container 0ℓ 0ℓ}{X : Set}(f : {Y : Set} → (⟦ F ⟧ Y → Y) → X → Y) →
            (s : X) → (fromCh ∘ prodCh f) s ≡ f in' s
prod-pres {F}{X} f s = 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
  ∎
```

$$\begin{tikzcd} \mu F \arrow[r, "fromCh"] \arrow[d, "f\ in"'] & C \arrow[dl, "prodCh\ f"] \\ T & \end{tikzcd}$$

$f\ in = prodCh\ f \circ fromCh$

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. This corresponds to the second proof obligation (first diagram) mentioned in Section 2.4.2:

```
trans-pres : {F G : Container 0ℓ 0ℓ}(f : {X : Set} → ⟦ F ⟧ X → ⟦ G ⟧ X) →
             (x : Church F) → (fromCh ∘ natTransCh f) x ≡ (⦇ in' ∘ f ⦈ ∘ fromCh) x
trans-pres f (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))
  ∎
```

$$\begin{tikzcd} \mu F \arrow[d, "(\!|in \circ f|\!)"'] & C \arrow[l, "fromCh"] \arrow[d, "natTransCh\ f"] \\ \mu F & C \arrow[l, "fromCh"] \end{tikzcd}$$

$$(\!|in \circ f|\!) \circ fromCh = fromCh \circ natTransCh\ f$$

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) → (x : Church F) →
          (natTransCh nat2 ∘ toCh ∘ fromCh ∘ natTransCh nat1) x ≡ natTransCh (nat2 ∘ nat1) x
natfuse {F}{G}{H} nat1 nat2 x@(Ch g) = begin
            (natTransCh nat2 ∘ toCh ∘ fromCh ∘ natTransCh nat1) x
          ≡⟨ cong (natTransCh nat2) (to-from-id (natTransCh nat1 x)) ⟩
            (natTransCh nat2 ∘ natTransCh nat1) x
          ≡⟨ refl ⟩
            natTransCh (nat2 ∘ nat1) x
          ∎
```

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. This also shows the foldr/build fusion if the `nat` given is `id`:

```
pipefuse : {F G : Container 0ℓ 0ℓ}{X : Set}(g : {Y : Set} → (⟦ F ⟧ Y → Y) → X → Y)
           (nat : {Y : Set} → ⟦ F ⟧ Y → ⟦ G ⟧ Y){Y : Set}(c : ⟦ G ⟧ Y → Y) →
           (x : X) → (foldr c ∘ natTrans nat ∘ build g) x ≡ g (c ∘ nat) x
pipefuse {F}{G} g nat c x = begin
    (consCh c ∘ toCh ∘ fromCh ∘ natTransCh nat ∘ toCh ∘ fromCh ∘ prodCh g) x
  ≡⟨ cong (consCh c ∘ toCh ∘ fromCh ∘ natTransCh nat) (to-from-id (prodCh g x)) ⟩
     (consCh c ∘ toCh ∘ fromCh ∘ natTransCh nat ∘ prodCh g) x
  ≡⟨ cong (consCh c) (to-from-id (natTransCh nat (prodCh g x))) ⟩
    (consCh c ∘ natTransCh nat ∘ prodCh g) x
  ≡⟨⟩
    g (c ∘ nat) x
  ∎
```

4.2.4 Example: Church Encoded List fusion

Deriving a container from a functor Deriving the container from a given (polynomial) functor can be done in a few 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`.

We summarize the above Table 2. For a concrete example of how a datatype is implemented, see Section 4.2.4. An example of the implementation for Lists is discussed in Section 4.2.4

	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`

Table 2: This table shows two examples of deriving the implementation of a container from a base functor.

Example: List Fusion In order to clearly see how the Church encodings allows functions to fuse, a datatype was selected such that 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. This section defines: 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.

Datatypes The index set for the container, as well as the container whose interpretation represents the base funtor for list. Note how ListOp is isomorphic to the datatype $\top$ `+ const 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 $(\!|\_|\!)$:

$$\mathsf{fold'} : \{A\ X : \mathsf{Set}\}(n : X)(c : A \to X \to X) \to \mathsf{List}\ A \to X$$
$$\mathsf{fold'}\ \{A\}\{X\}\ n\ c = (\!|\ (\lambda\{(\mathsf{nil}\ ,\ \_) \to n;\ (\mathsf{cons}\ n\ ,\ g) \to c\ n\ (g\ \mathsf{zero})\})\ |\!)$$

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

$$\mathsf{b'} : \{B : \mathsf{Set}\} \to (a : \mathsf{List'}\ \mathbb{N}\ B \to B) \to \mathbb{N} \to \mathbb{N} \to B$$
$$\mathsf{b'}\ a\ x\ \mathsf{zero} = a\ (\mathsf{nil}\ ,\ \lambda())$$
$$\mathsf{b'}\ a\ x\ (\mathsf{suc}\ n) = a\ (\mathsf{cons}\ x\ ,\ \mathsf{const}\ (\mathsf{b'}\ a\ (\mathsf{suc}\ x)\ n))$$
$$\mathsf{b} : \{B : \mathsf{Set}\} \to (a : \mathsf{List'}\ \mathbb{N}\ B \to B) \to \mathbb{N} \times \mathbb{N} \to B$$
$$\mathsf{b}\ a\ (x\ ,\ y) = \mathsf{b'}\ a\ x\ (\mathsf{suc}\ (y\ \text{-}\ x))$$

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

$$\mathsf{between1} : \mathbb{N} \times \mathbb{N} \to \mathsf{List}\ \mathbb{N}$$
$$\mathsf{between1}\ xy = \mathsf{b}\ \mathsf{in'}\ xy$$
$$\mathsf{between2} : \mathbb{N} \times \mathbb{N} \to \mathsf{List}\ \mathbb{N}$$
$$\mathsf{between2} = \mathsf{build}\ \mathsf{b}$$
$$\mathsf{eqbetween} : \mathsf{between1} \equiv \mathsf{between2}$$
$$\mathsf{eqbetween} = \mathsf{refl}$$
$$\mathsf{checkbetween} : 2 :: 3 :: 4 :: 5 :: 6 :: [] \equiv \mathsf{between2}\ (2\ ,\ 6)$$
$$\mathsf{checkbetween} = \mathsf{refl}$$

Map The natural transformation `m`, which when used in a transformation function, represents the map function:

$$\mathsf{m} : \{A\ B\ C : \mathsf{Set}\}(f : A \to B) \to \mathsf{List'}\ A\ C \to \mathsf{List'}\ B\ C$$
$$\mathsf{m}\ f\ (\mathsf{nil}\ ,\ \_) = (\mathsf{nil}\ ,\ \lambda())$$
$$\mathsf{m}\ f\ (\mathsf{cons}\ n\ ,\ l) = (\mathsf{cons}\ (f\ n)\ ,\ l)$$

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

$$\mathsf{map1} : \{A\ B : \mathsf{Set}\}(f : A \to B) \to \mathsf{List}\ A \to \mathsf{List}\ B$$
$$\mathsf{map1}\ f = (\!|\ \mathsf{in'} \circ \mathsf{m}\ f\ |\!)$$
$$\mathsf{map2} : \{A\ B : \mathsf{Set}\}(f : A \to B) \to \mathsf{List}\ A \to \mathsf{List}\ B$$
$$\mathsf{map2}\ f = \mathsf{natTrans}\ (\mathsf{m}\ f)$$
$$\mathsf{eqmap} : \{f : \mathbb{N} \to \mathbb{N}\} \to \mathsf{map1}\ f \equiv \mathsf{map2}\ f$$
$$\mathsf{eqmap} = \mathsf{refl}$$
$$\mathsf{checkmap} : (\mathsf{map1}\ (\_+\_\ 2)\ (3 :: 6 :: [])) \equiv 5 :: 8 :: []$$
$$\mathsf{checkmap} = \mathsf{refl}$$

Sum The algebra `s`, which when used in a consumer function, represents the sum function:

$$\mathsf{s'} : \mathsf{List'}\ \mathbb{N}\ (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$$
$$\mathsf{s'}\ (\mathsf{nil}\ ,\ fn)\ s = s$$
$$\mathsf{s'}\ (\mathsf{cons}\ n\ ,\ fn)\ s = fn\ \mathsf{zero}\ (n + s)$$
$$\mathsf{s} : \mathsf{List'}\ \mathbb{N}\ \mathbb{N} \to \mathbb{N}$$
$$\mathsf{s}\ (\mathsf{nil}\ ,\ \_) = 0$$
$$\mathsf{s}\ (\mathsf{cons}\ n\ ,\ f) = n + f\ \mathsf{zero}$$

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

$$\mathsf{sum1} : \mathsf{List}\ \mathbb{N} \to \mathbb{N}$$
$$\mathsf{sum1} = (\!|\ \mathsf{s}\ |\!)$$

```
sum2 : List ℕ → ℕ
sum2 = foldr s
sum2' : List ℕ → ℕ
sum2' l = foldr s' l 0
checksum : sum2 (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}{x : ℕ × ℕ} → (sum1 ∘ map1 f ∘ between1) x ≡ (sum2 ∘ map2 f ∘ between2) x
eq {f}{x} = begin
    (⦇ s ⦈ ∘ ⦇ in' ∘ m f ⦈ ∘ b in') x
  ≡⟨ cong (⦇ s ⦈ ∘ ⦇ in' ∘ m f ⦈) (prod-pres b x) ⟩ - refl
    (⦇ s ⦈ ∘ ⦇ in' ∘ m f ⦈ ∘ fromCh ∘ prodCh b) x
  ≡⟨ cong ⦇ s ⦈ (sym $ trans-pres (m f) (prodCh b x)) ⟩
    (⦇ s ⦈ ∘ fromCh ∘ natTransCh (m f) ∘ prodCh b) x
  ≡⟨ cons-pres s ((fromCh ∘ natTransCh (m f) ∘ prodCh b) x) ⟩ - refl
    (consCh s ∘ toCh ∘ fromCh ∘ natTransCh (m f) ∘ prodCh b) x
  ≡⟨ cong (consCh s ∘ toCh ∘ fromCh ∘ natTransCh (m f))
        (sym $ to-from-id (prodCh b x)) ⟩
    (consCh s ∘ toCh ∘ fromCh ∘ natTransCh (m f) ∘ toCh ∘ fromCh ∘ prodCh b) x
  ≡⟨⟩
    (foldr s ∘ natTrans (m f) ∘ build b) x
  ∎
```

Fusing the functions down to a pipeline I present the equality between two functions: One is the `pipeline` function and the other is the composition of the three functions presented so far, along with the `filter2` function.

The `pipeline` function has been implemented with the aid of a `pipeline'` function. This is to aid in termination checking and the same technique used for `b` and `b'` above.

The `filt'` function is a function that creates a new algebra from an existing one. The `filter2` function takes this partial algebra composition and encodes it using a build/foldr pair.

```
filt' : {A X : Set} → (A → Bool) → (List' A X → X) → (List' A X → X)
filt' {A}{X} p f (nil , l) = f (nil , l)
filt' {A}{X} p f (cons a , l) = if (p a) then f (cons a , l) else l zero
filter2 : {A : Set} → (A → Bool) → List A → List A
filter2 {A} p = build (foldr ∘ filt' p)

pipeline' : (ℕ → Bool) → ℕ → ℕ → ℕ
pipeline' p x zero = zero
pipeline' p x (suc n) = if p x
                        then (1 + x) + pipeline' p (1 + x) n
                        else pipeline' p (1 + x) n
pipeline : (ℕ → Bool) → (ℕ × ℕ) → ℕ
pipeline p (x , y) = pipeline' p x (suc (y - x))
```

The `eqPips` lemma proves that the fused pipelines are the same for all inputs using induction and pattern matching, while the `eqPipelines` lemma proves that the fusion is possible, even with this build/foldr pair. One crucial insight for this latter proof is that `prodCh` is associative for functions postcomposed to it. This is stated formally in lemma `prodAssoc` and proved via reflexivity.

These lemmas show, in as clear as a fashion as possible, that the composition of the Church encoded functions are equal to the hand-fused pipeline written above.

```
eqPips : (p : ℕ → Bool)(x y : ℕ) → b' (filt' p (s ∘ m (_+_ 1))) x y ≡ pipeline' p x y
eqPips p _ zero = refl
eqPips p zero (suc y) with p 0
```

```
... | true      = cong suc (eqPips p 1 y)
... | false = eqPips p 1 y
eqPips p (suc x) (suc y) with p (suc x)
... | true      = cong 2+ (cong (_+_ x) (eqPips p (2+ x) y))
... | false = eqPips p (2+ x) y

prodAssoc : {F : Container _ _}{Y : Set₁}{Z : Set}(f : Z → Y)
            (g : {X : Set} → (⟦ F ⟧ X → X) → Y → X)(z : Z) →
            (prodCh g ∘ f) z ≡ prodCh (λ a → g a ∘ f) z
prodAssoc _ _ _ = refl

eqPipelines : {p : ℕ → Bool}{xy : ℕ × ℕ} →
              (sum2 ∘ map2 (_+_ 1) ∘ filter2 p ∘ between2) xy ≡ pipeline p xy
eqPipelines {p}{xy@(x , y)} = begin
      (foldr s ∘ natTrans (m (_+_ 1)) ∘
        (fromCh ∘ prodCh (consCh ∘ filt' p) ∘ toCh) ∘ build b) xy
  ≡⟨ cong (foldr s ∘ natTrans (m (_+_ 1)) ∘ fromCh)
          (prodAssoc (toCh ∘ build b) (consCh ∘ filt' p) xy) ⟩
      (foldr s ∘ natTrans (m (_+_ 1)) ∘ fromCh ∘
        prodCh (λ a → consCh (filt' p a) ∘ toCh ∘ build b)) xy
  ≡⟨ pipefuse (λ a → consCh (filt' p a) ∘ toCh ∘ build b) (m (_+_ 1)) s xy ⟩
      (λ a → consCh (filt' p a) ∘ toCh ∘ build b) (s ∘ m (_+_ 1)) xy
  ≡⟨⟩ - beta reduction
      (consCh (filt' p (s ∘ m (_+_ 1))) ∘ toCh ∘ build b) xy
  ≡⟨⟩ - inlining of build
      (consCh (filt' p (s ∘ m (_+_ 1))) ∘ toCh ∘ fromCh ∘ prodCh b) xy
  ≡⟨ cong (consCh (filt' p (s ∘ m (_+_ 1)))) (to-from-id (prodCh b xy)) ⟩
      (consCh (filt' p (s ∘ m (_+_ 1))) ∘ prodCh b) xy
  ≡⟨⟩ - inlining of consCh and prodCh
      b (filt' p (s ∘ m (_+_ 1))) xy
  ≡⟨⟩ - inlining of b
      b' (filt' p (s ∘ m (_+_ 1))) x (suc (y - x))
  ≡⟨ eqPips p x (suc (y - x)) ⟩
      pipeline' p x (suc (y - x))
    ≡⟨⟩ - inlining of pipeline
      pipeline p xy
    ∎
```

4.3 Cochurch Encodings

4.3.1 Category Theory: Terminality

This section specifically defines the category of F-Coalgebras and proves terminality of ν`F, out` (the universal properties of unfolds) and the fusion property. This section is the complement of Section 4.2.1.

Terminal coalgebras and anamorphisms This section defines a datatype and shows it to be terminal; and a function and shows it to be an anamorphism in the category of F-Coalgebras. It takes the definition of `M` types and shows that the A⟦_⟧ function defined for it is an anamorphism. This is done by proving that the A⟦_⟧ is a unique F-coalgebra homomorphism from any datatype through a proof of existence and uniqueness:

$$\forall (C, \phi) \in \mathcal{C}o\mathcal{A}lg(F)_0 : \exists! (\!|\phi|\!) \in \mathtt{hom}_{\mathtt{CoAlg}(\mathcal{F})}(C, \nu F), s.t. out \circ A[\![\phi]\!] = FA[\![\phi]\!] \circ \phi$$

Specifically, it is shown that (ν`F, out`) is terminal.

```
{-# OPTIONS --guardedness #-}
module agda.cochurch.terminal where
open import Codata.Guarded.M public using (head; tail) renaming (M to ν)
```

I define a function below which turns out to be a catamorphism. This fact is proved in this section through a proof of existence `univ-to` and a proof of uniqueness `univ-from`. The `unfold` function for containers in Agda's stdlib is defined identically and could be imported, omitting this definition. However, for clarity I'm including the definition here instead of importing it:

```
A⟦_⟧ : {F : Container 0ℓ 0ℓ}{X : Set} → (X → ⟦ F ⟧ X) → X → ν F
head (A⟦ c ⟧ x) = fst (c x)
tail (A⟦ c ⟧ x) = A⟦ c ⟧ ∘ (snd (c x))
out : {F : Container 0ℓ 0ℓ} → ν F → ⟦ F ⟧ (ν F)
out nu = head nu , tail nu
```

We show that any A⟦_⟧ is a valid F-CoAlgebra homomorphism from any other object to `out`; i.e. the forward direction of the *universal property of unfolds* (Harper, 2011):

$$h = A[\![a]\!] \Longrightarrow out \circ h = Fh \circ c$$

This constitutes a proof of existence; there exists a function (in this case called A⟦_⟧), that is a valid F-Algebra homomorphism:

```
univ-to : {F : Container 0ℓ 0ℓ}{C : Set}{h : C → ν F}{c : C → ⟦ F ⟧ C} →
            ({x : C} → h x ≡ A⟦ c ⟧ x) → {x : C} → (out ∘ h) x ≡ (map h ∘ c) x
univ-to {_}{_}{h}{c} eq {x} = let (op , ar) = c x in begin
    out (h x)
  ≡⟨ cong out eq ⟩
    out (A⟦ c ⟧ x)
  ≡⟨⟩
    (op , A⟦ c ⟧ ∘ ar)
  ≡⟨ cong (λ f → op , f) (funext (λ x → sym eq)) ⟩
    (op , h ∘ ar)
  ≡⟨⟩
    map h (c x)
  ∎
```

Injectivity of the `out` constructor is postulated. To prove equality between two terminal datatypes a bisimulation relation is needed. I made an attempt to prove the `univ-from`, `univ-to`, and `reflection` lemmas through the use of a bisimilation relation, but due to time constraings there was too much work remaining to warrant continuing it. The final state of this code can be found in Appendix B and is summarized as follows:

- The reflection law was proven (as a bisimilarity)
- The termination of the 'proof of uniqueness' part of the proof of terminality (also as a bisimilarity)
- The plan and execution of restructuring the further code that rests on the above proofs. Most likely the use of propositional equalities throughout the following proofs need to be modified to instead use some combination of the bisimilarity and propositional equality in Agda.

Instead, we postulate injectivity of the `out` constructor and use propositional equality.

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

It is shown that any other valid F-Coalgebra homomorphism from `out` to `a` is equal to the A⟦_⟧ defined; i.e. the backward direction of the *universal property of unfolds* Harper (2011).

$$h = A[\![a]\!] \Longleftarrow out \circ h = Fh \circ c$$

This constitutes a proof of uniqueness; for any function that is a valid F-Algebra homomorphism (in this case called h), it is equal to $A[\![\_]\!]$. This uses `out` injectivity. Currently, Agda's termination checker does notice that the proof in question terminates. The proof needs to be rewritten to properly use guardedness through the use of a bisimilarity:

```
{-# NON_TERMINATING #-}
univ-from : {F : Container _ _}{C : Set}(h : C → ν F){c : C → ⟦ F ⟧ C} →
```

```
                ({x : C} → (out ∘ h) x ≡ (map h ∘ c) x) → {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
    ≡⟨ eq ⟩
      (op , h ∘ ar)
    ≡⟨ cong (λ f → op , f) (funext $ λ x → univ-from h eq {ar x}) ⟩ - induction
      (op , A⟦ c ⟧ ∘ ar)
    ≡⟨⟩ - Definition of ⟦_⟧
      (out ∘ A⟦ c ⟧) x
    ∎)
```

The two previous proofs, constituting a proof of existence and uniqueness, together prove terminailty of `(ν F, out)`. The *reflection law* Harper (2011):

```
{-# 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 λ x → reflection {_}{ar x}) ⟩
    out x
  ∎)
```

The fusion property, which follows from the backwards direction of the *universal property of unfolds*:

```
fusion : {F : Container 0ℓ 0ℓ}{C D : Set}{c : C → ⟦ F ⟧ C}{d : D → ⟦ F ⟧ D}(h : C → D) →
         ({x : C} → (d ∘ h) x ≡ (map h ∘ c) x) → (x : C) → (A⟦ d ⟧ ∘ h) x ≡ A⟦ c ⟧ x
fusion {_}{C}{_}{c}{d} h eq x = univ-from (A⟦ d ⟧ ∘ h) (cong (map A⟦ d ⟧) eq) {x}
```

4.3.2 Fusion: Cochurch encodings

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

Definition of Cochurch encodings This section 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 Harper (2011). The definition of the CoChurch datatypes is defined differently to how it is initially defined by Harper (2011). Instead an Isomorphic definition is used, whose type is described later on on the same page. This was done by Harper such that the free theorem about the datatype being encoded, is easier to work with. It is also a datatype that lends itself to better to theorem proving, as otherwise a coproduct datatype would need to be involved. See the bottom of page 52 of Harper's work for his discussion on the isomorphism. The original definition is included as `CoChurch'`. 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 Svenningsson (2002)'s `unfoldr` function:

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

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

eqprod : {F : Container 0ℓ 0ℓ}{Y : Set}{g : (Y → ⟦ F ⟧ Y)} →
         unfoldr 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 Svenningsson (2002)'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

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

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

The original CoChurch definition is included here for completeness' sake, but it is not 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 constructing an 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}(x : CoChurch F) → (toConv ∘ fromConv) x ≡ x
to-from-conv-id (CoCh h x) = refl

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

4.3.3 Proof obligations

As with Church encodings, in Harper (2011)'s work, five proof obligations needed to be satisfied. These are formalized here. The first proof proves that `fromCoCh ∘ toCh = id`, using the reflection law. This corresponds to the first proof obligation mentioned in Section 2.4.2:

```
from-to-id : {F : Container 0ℓ 0ℓ}(x : ν F) → (fromCoCh ∘ toCoCh) x ≡ id x
from-to-id {F} x = begin
    fromCoCh (toCoCh x)
  ≡⟨⟩ - Definition of toCh
    fromCoCh (CoCh out x)
  ≡⟨⟩ - Definition of fromCh
    A⟦ out ⟧ x
  ≡⟨ reflection ⟩
    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 parametricity as described by Wadler (1989). 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" (Harper, 2011). This together with the first proof shows that Cochurch encodings are isomorphic to the datatypes they are encoding:

```
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
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}(x : CoChurch F) → (toCoCh ∘ fromCoCh) x ≡ id x
to-from-id (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 Harper (2011)'s original proof steps are included here for completeness. This corresponds to the third proof obligation (second diagram) mentioned in Section 2.4.2:

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

$$\begin{tikzcd} S \arrow[d, "A[\![c]\!]"'] \arrow[dr, "prodCoCh\ c"] & \\ \mu F & C \arrow[l, "fromCoCh"] \end{tikzcd}$$

$$A[\![c]\!] = fromCoCh \circ prodCoCh\ c$$

The fourth proof shows that cochurch encoded functions constitute an implementation for the consuming functions being replaced. The proof is proved via reflexivity, but Harper (2011)'s original proof steps are included here for completeness. This corresponds to the fourth proof obligation (third diagram) mentioned in Section 2.4.2:

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

$$\begin{tikzcd} \mu F \arrow[r, "toCoCh"] \arrow[d, "fout"'] & C \arrow[dl, "consCoCh\ f"] \\ T & \end{tikzcd}$$

$$fout = consCoCh\ f \circ toCoCh$$

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`. This corresponds to the second proof obligation (first diagram) mentioned in Section 2.4.2:

```
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)(x : ⟦ F ⟧ X) → (map g ∘ f) x ≡ (f ∘ map g) x) →
            {x : X} → (map A⟦ h ⟧ ∘ f ∘ h) x ≡ (f ∘ out ∘ A⟦ h ⟧) x
valid-hom h f nat {x} = begin
    (map A⟦ h ⟧ ∘ f ∘ h) x
  ≡⟨ nat A⟦ h ⟧ (h x) ⟩
    (f ∘ map A⟦ h ⟧ ∘ h) x
  ≡⟨⟩ - dfn of A⟦_⟧
    (f ∘ out ∘ A⟦ h ⟧) x
  ■
```

$$\begin{tikzcd} \mu F \arrow[d, "A[\![fo out]\!]"'] & C \arrow[l, "fromCoCh"'] \arrow[d, "natTransCoCh\ f"] \\ \mu F & C \arrow[l, "fromCoCh"] \end{tikzcd}$$

$$A[\![f \circ out]\!] \circ fromCoCh = fromCoCh \circ natTransCoCh\ f$$

```
trans-pres : {F G : Container 0ℓ 0ℓ}(f : {X : Set} → ⟦ F ⟧ X → ⟦ G ⟧ X)
             (nat : {X : Set}(g : X → ν F)(x : ⟦ F ⟧ X) → (map g ∘ f) x ≡ (f ∘ map g) x)
             (x : CoChurch F) → (fromCoCh ∘ natTransCoCh f) x ≡ (A⟦ f ∘ out ⟧ ∘ fromCoCh) x
trans-pres f nat (CoCh h x) = begin
    fromCoCh (natTransCoCh f (CoCh h x))
  ≡⟨⟩ - Function application
    fromCoCh (CoCh (f ∘ h) x)
  ≡⟨⟩ - Definition of fromCh
    A⟦ f ∘ h ⟧ x
  ≡⟨ sym $ fusion A⟦ h ⟧ (sym $ valid-hom h f nat) x ⟩
    A⟦ f ∘ out ⟧ (A⟦ h ⟧ x)
```

```
≡⟨⟩ - This step is not in the paper, but mirrors the one on the Church-side.
  A⟦ f ∘ out ⟧ (fromCoCh (CoCh h x))
■
```

Two additional proofs, `natfuse` and `pipefuse`, were made in a similary fashion to how they were done for Church encodings they are omitted here for brevity and can be found in the artifacts.

4.3.4 Example: Cochurch Encoded 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 concrete datatype. In this section 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 encoded 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 isomorphic to the datatype ⊤ `+ const 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 corecursion principle `b`, which when used, represents the between function. It uses `b'` to assist in 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 is included to show equality:

```
between1 : ℕ × ℕ → List ℕ
between1 = A⟦ b ⟧
between2 : ℕ × ℕ → List ℕ
between2 = unfoldr b
eqbetween : between1 ≡ between2
eqbetween = refl
```

Map The natural transformation `m`, which when used in a natrual transformation function, 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 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
```

Sum The coalgebra `s`, which when used in a consumer function, represents the sum function. Note that it is currently set to be non-terminating.

```
{-# 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 is included to show equality:

```
sum1 : List ℕ → ℕ
sum1 = s out
sum2 : List ℕ → ℕ
sum2 = destroy s
eqsum : sum1 ≡ sum2
eqsum = refl
```

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

```
eq : {f : ℕ → ℕ}(x : ℕ × ℕ) → (sum1 ∘ map1 f ∘ between1) x ≡ (sum2 ∘ map2 f ∘ between2) x
eq {f} x = begin
    (s out ∘ A⟦ m f ∘ out ⟧ ∘ A⟦ b ⟧) x
  ≡⟨ cong (s out ∘ A⟦ m f ∘ out ⟧) (prod-pres b x) ⟩ - refl
    (s out ∘ A⟦ m f ∘ out ⟧ ∘ fromCoCh ∘ prodCoCh b) x
  ≡⟨ cong (s out) (sym $ trans-pres (m f)
```

```
      (λ _ → (λ { (nil , l) → refl ; (cons n , l) → refl })) (prodCoCh b x)) ⟩
    (s out ∘ fromCoCh ∘ natTransCoCh (m f) ∘ prodCoCh b) x
  ≡⟨ (cons-pres s ((fromCoCh ∘ natTransCoCh (m f) ∘ prodCoCh b) x)) ⟩ - refl
    (consCoCh s ∘ toCoCh ∘ fromCoCh ∘ natTransCoCh (m f) ∘ prodCoCh b) x
  ≡⟨ cong (consCoCh s ∘ toCoCh ∘ fromCoCh ∘ natTransCoCh (m f))
         (sym $ to-from-id (prodCoCh b x)) ⟩
  (consCoCh s ∘ toCoCh ∘ fromCoCh ∘ natTransCoCh (m f) ∘ toCoCh ∘ fromCoCh ∘ prodCoCh b) x
  ≡⟨⟩
    (destroy s ∘ natTrans (m f) ∘ unfoldr b) x
  ■
```

4.4 Discussion of Agda Formalization

I formalized that, given parametricity, the fusion of (Co)Church encodings are the same as their non-encoded counterpart as proved by Harper (2011).

This was done through multiple steps that build on each other, each step leaning on proofs and definitions from the previos one: I did this by first proving initality of `W` (μ) types and terminality of `M` (ν) types. Then I formalized the categorical fusion property, which only ended up being used in the proofs for fusion of Cochurch encodings. Then I defined the Church and Cochurch encodings, along with their associated conversion functions. After defining all of this, I formalized Harper's proof that shortcut fusion is possible for both Church and Cochurch encodings.

Building on this, I implemented the `List` datastructure using containers. Across this datastructure I implemented normal and (Co)Church encoded functions across these lists: `between`, `map`, `filter`, and `sum`.

Repeating my qustion: Are the transformations used to enable fusion safe? Meaning:

- Do the transformations in Haskell preserve the semantics of the language?
- If the mathematics and the encodings are implemented in a dependently typed language, is it possible to prove them to be correct?

The first question can be answered as a conditional: Yes, as long as Haskell's type system contains parametricity. The second question can be answered as: Yes it is possible, with some limitations, which are discussed below. The question as a whole can be answered as a tentative 'yes', keeping in mind the weaknesses discussed below.

Remaining Weaknesses There are two main remaining weaknesses in my current work: First, the proof of terminality of terminal coalgebras is currently not terminating. Second, the free theorems are currently postulated to be true instead of being proven to be true.

Termination Checking I made an attempt to construct a terminating proof of terminality of `M` types in agda through the use of a bisimulation, but due to time constraints I reverted to a version of the code that type checked, but did not terminate for a few proofs. The functions are currently proved using a postulate called `out-injective` that postulates that the constructor of the coinductive datatype is injective. The above three functions are now non-terminating in the final state of the code. Furthermore, the implementation of the Cochurch encoded list `sum` function also was set to be non-terminating. The state of the code before reverting can be seen in Appendix B.

Postulates There are currently four postulates in the codebase. I'll go through them in increasing order of noteworthiness:

- Functional extensionality. I used functional extensionality extensively throughout the repository. Its use is well-understood to be consistent and is provable from within cubical Agda.
- `out`-injectivity. Injectivity of coinductive datatypes is not supported out-of-the-box in Agda for good reason. However, it is needed for my type checking of the proofs of terminality, without the use of a bisimilarity. It exists to patch over the larger problem of termination checking above. If a bisimilarity relation were to be introduced, it can be removed.
- Two free theorems. The postulating of the free theorems was needed as it is currently not possible to prove the correctness of free theorems from within Agda. New research does exist by Van Muylder et

al. (2024) that would enable the proving of the two free theorems, using internalized parametricity as originally described by Bernardy & Moulin (2012). Doing so falls outside the scope of this research and is left to future work.

My Agda formalization has shown that Harper's work is correct, with some limitation, namely with respect to the proof of terminality of M. It can be clearly seen, through the proof of some of the lemmas, that the fusion does not destroy the semantics of the functions being fused; that the fusion is correct. There are multiple of future avenues that this research can take, this is discussed more extensively in Section 6.

5 Related Works

Initial work, done by Wadler (1984, 1986, 1990) 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 Gill et al. (1993) described the weaknesses of the work and proposed an alternative technique. This so-called foldr/build fusion technique can, when employed, eliminate the runtime generation of intermediate lists. I describe this technique further in Section 2.1.

A converse approach, aptly named the `destroy/unfoldr` rule, is described by Svenningsson (2002), which describes the converse technique to Gill et al. (1993)'s. A further generalization of this technique, dubbed *stream fusion* by Coutts et al. (2007), further strengthened the work by Svenningsson (2002).

(Co)Church encodings Finally, Harper (2011) combined all of these concepts into one paper, called "The Library Writer's Guide to Shortcut Fusion". In it the concept he describes (Co)Church encodings and, pragmatically, how to implement them in Haskell.

Other approaches Other approaches exist such as 'Warm fusion' by Launchbury & Sheard (1995), who attempt to derive fold and build combinators for a data type and automatically rewrite explicitly recursive functions. A calculational appcoach to fusion, as opposed to a search-based one, is discussed by Onoue et al. (1997) in their system HYLO.

Before Gill et al. (1993) published his work on shortcut fusion, there was existing work by Meijer et al. (1991), describing the fusion properties of catamorphisms and anamorphisms, called "Functional programming with bananas, lenses, envelopes and barbed wire".

6 Conclusion and Future Work

I have presented my work on implementing and formalizing shortcut fusion of (Co)Church encodings as described by Harper (2011). I have replicated Harper's work of Church and Cochurch encoded functions operating on leaf trees: `between`, `map`, `filter`, and `sum`; and shown the generalizability of his example by also implementing the functions on lists. In doing so I discovered that in Haskell full fusion is not currently possible for the Cochurch encoded filter function. GHC needs either proper loopification using join points[12], or additional encoding techniques such as those described by Coutts et al. (2007).

I benchmarked the performance of multiple different variants of the same pipeline: unencoded, hand-fused, Church fused, Cochurch fused, and GHC.List fused; where the (Co)Church fused pipelines had four variants: tail recursive, stream fused, neither, and both. I discovered that changing the underlying datatype for Church encodings from List to Stream datatypes gave no performance improvement, for both tail and non-tail recursive implementations. Implementing tail recursion however did offer a speedup, for Cochurch encodings. It was also faster to implement tail recrusion in addition to modifying the underlying type from List to Stream. This was likely due to the improper loopification of the recursive coalgebra `go`. The fully fused (fastest) pipelines of both Church and Cochurch encodings were about as fast as the hand-fused and GHC.List fused pipelines; for some inputs the (Co)Church fusion was faster, for others the hand-fused/GHC.List fused.

I implemented Harper's description of Church and Cochurch encodings using Agda's dependent type system, using containers to represent strictly positive functors. Before formalizing the proof of the shortcut fusion property, I first formalized all of the needed underlying category theory: the universal property of folds (i.e., initiality of initial algebras), the computation law, the reflection law, and the fusion property. Using these, I formalized Harper's proofs of the Church and Cochurch encodings being faithful,

[12]See `https://gitlab.haskell.org/ghc/ghc/-/issues/22227#note_551000`

showing that they are isomorphic to the datatype that they are encoding. This came with one major caveat: The reliance on the free theorems of parametric functions, which was not provable in Agda. There is recent work on this *internalized parametricity* by Van Muylder et al. (2024), which would make the free theorems provable from within Agda, dubbed Agda –bridges. Finally, I implemented (Co)Church encoded versions of the following four functions `between`, `map`, `filter`, and `sum` and showed that their composition as (Co)Church encodings was equal to the hand fused function also presented.

These findings highlight the effectiveness and correctness of shortcut fusion techniques and show the promise of shortcut fusion: Reduce the computational overhead associated with functional programming while retaining its nice, compositional properties.

Future Work

There are many future avenues that could be taken to continue my research:

- Tool (Language) improvements:
  - See if it is possible to implement warm fusion in Haskell or some other language as described by Launchbury & Sheard (1995).
  - Strengthen Agda's typechecker with respect to implicit parameters. Currently two variants of functional extensionality had to be defined to work around this.
  - Investigate if creating a new programming language that has this fusion as a first-class feature can enable fusion to be compiled more efficiently and consistently.
- Extensions of my work
  - Implement a bisimilarity relation for the coinductive `M`/ν type in Agda to prove its terminality. After which modifying all the code resting on top of this proof to properly use this new relation.
  - Investige whether it is to generalize the work of Coutts et al. (2007) to more datastrcutures, with a motivating example being Leaf Trees.
  - Use Agda –bridges to see if it is possible to prove the free theorems currently postulated in my work.
- Applications of my work:
  - Implement (Co)Church fused versions of Haskell's library functions.
  - Merge into Agda the Church and Cochurch encodings, as well as the bisimilarity across the guarged `M` type.

7 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

Bernardy, J.-P., & Moulin, G. (2012, June). A computational interpretation of parametricity. In *2012 27th annual ieee symposium on logic in computer science.* IEEE. Retrieved from http://dx.doi.org/10.1109/LICS.2012.25 doi: 10.1109/lics.2012.25

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

Launchbury, J., & Sheard, T. (1995). Warm fusion: deriving build-catas from recursive definitions. In *Proceedings of the seventh international conference on functional programming languages and computer architecture - fpca '95*. ACM Press. Retrieved from http://dx.doi.org/10.1145/224164.224223 doi: 10.1145/224164.224223

Meijer, E., Fokkinga, M., & Paterson, R. (1991). Functional programming with bananas, lenses, envelopes and barbed wire. In *Lecture notes in computer science* (p. 124–144). Springer Berlin Heidelberg. Retrieved from http://dx.doi.org/10.1007/3540543961_7 doi: 10.1007/3540543961_7

Onoue, Y., Hu, Z., Iwasaki, H., & Takeichi, M. (1997). A calculational fusion system hylo. In *Ifip advances in information and communication technology* (p. 76–106). Springer US. Retrieved from http://dx.doi.org/10.1007/978-0-387-35264-0_4 doi: 10.1007/978-0-387-35264-0_4

Reynolds, J. C. (1983). Types, abstraction and parametric polymorphism. In *Information processing 83, proceedings of the ifip 9th world computer congres* (pp. 513–523).

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. (2024, January). Internal and observational parametricity for cubical agda. *Proceedings of the ACM on Programming Languages*, *8*(POPL), 209–240. Retrieved from http://dx.doi.org/10.1145/3632850 doi: 10.1145/3632850

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 Figures

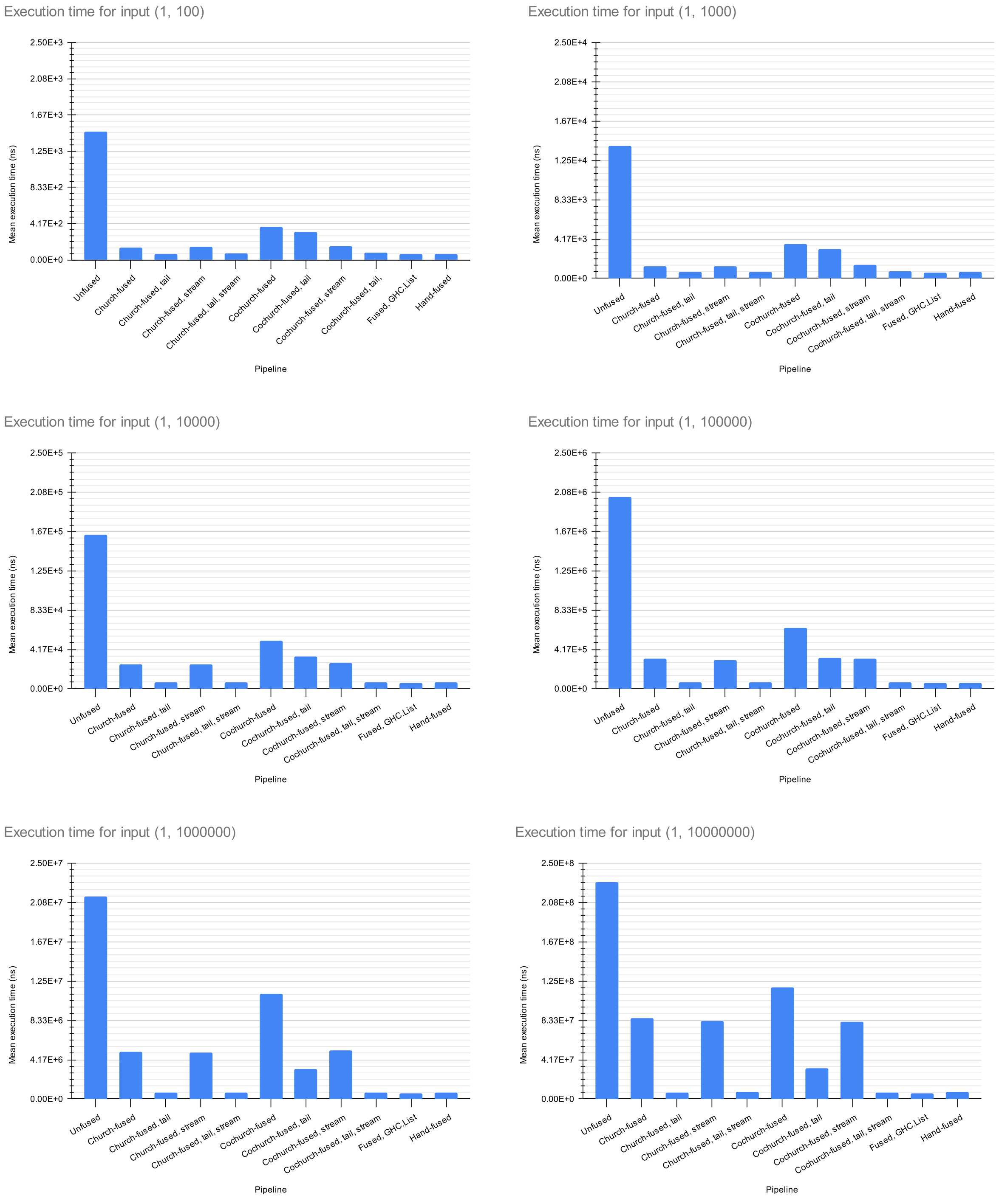

Figure 12: All execution times

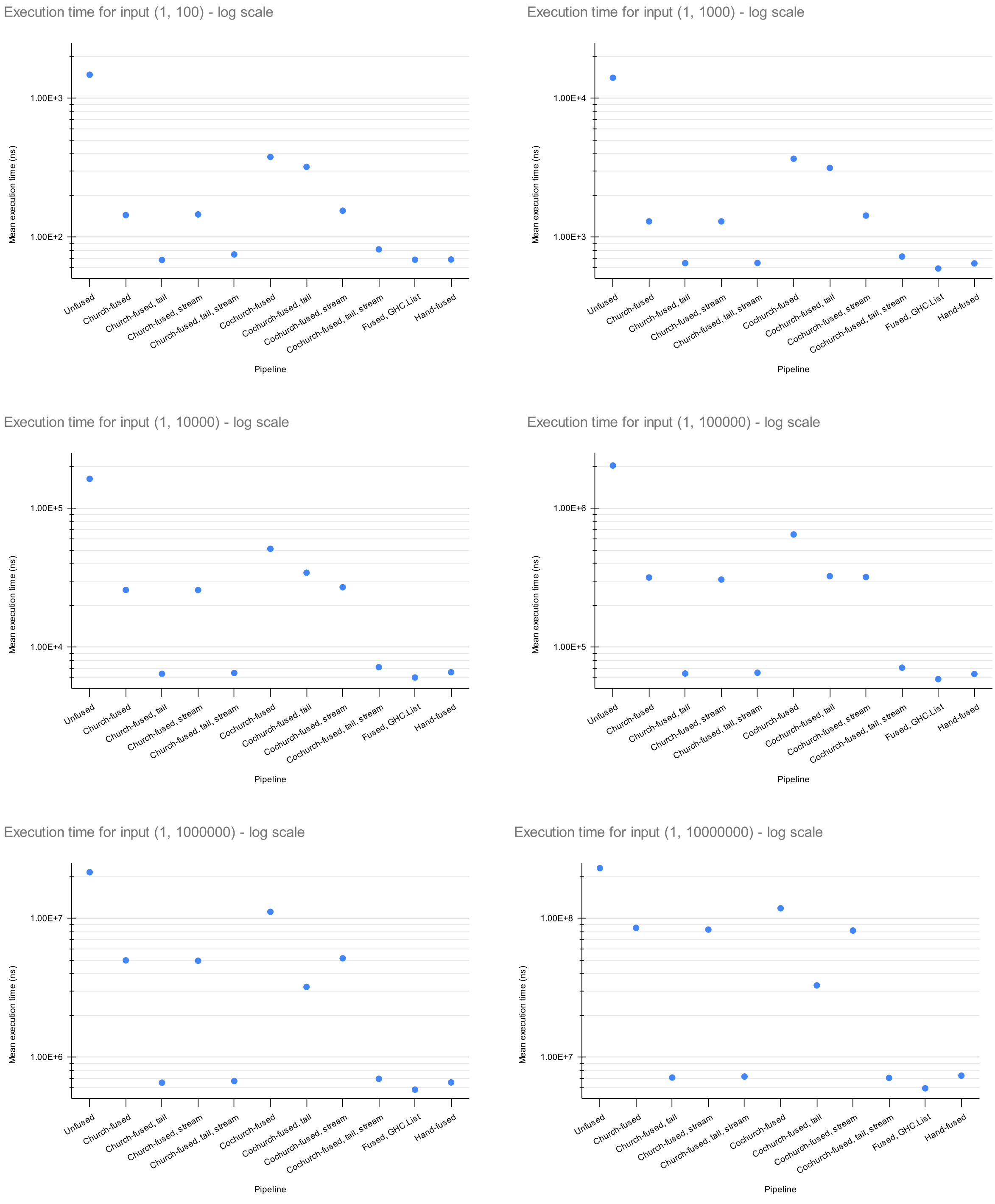

Figure 13: All execution times — log scale.

B Terminal Bismulation Code

The state of the code at the cutoff moment for proving terminality of ν can be seen here:

Terminal coalgebras and anamorphisms: bisimulation 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 -with-K -allow-unsolved-metas #-}
module agda.cochurch.terminalbisim where
open import Codata.Guarded.M using (head; tail) renaming (M to ν) public
```

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

```
A⟦_⟧ : {F : Container 0ℓ 0ℓ}{X : Set} → (X → ⟦ F ⟧ X) → X → ν F
head (A⟦ c ⟧ x) = proj₁ (c x)
tail (A⟦ c ⟧ x) = A⟦ c ⟧ ∘ snd (c x)
```

It is shown that any A⟦_⟧ is a valid F-Coalgebra homomorphism from `out` to any other object `a`; i.e. the forward direction of the *universal property of unfolds* Harper (2011). This constitutes a proof of existence:

```
univ-to : {F : Container 0ℓ 0ℓ}{C : Set}(h : C → ν F)
          {c : C → ⟦ F ⟧ C}(eq : ∀ {x} → h x ≈≈ A⟦ c ⟧ x)(x : C) →
          out (h x) ≡ map h (c x)
univ-to {F}{C} h {c} eq x = {!!}
```

It is shown that any other valid F-Coalgebra homomorphism from `out` to `a` is equal to the A⟦_⟧ defined; i.e. the backward direction of the *universal property of unfolds* Harper (2011). 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:

```
univ-from : {F : Container _ _}{C : Set}(h : C → ν F)(c : C → ⟦ F ⟧ C) →
            (eq : ∀{x} → out (h x) ≡ map h (c x)) →
            {x : C} → h x ≈≈ A⟦ c ⟧ x
outfst (univ-from h c eq) = ,-injectiveˡ eq
outsnd (univ-from {F} h c eq1 {x}) {y} = {!!}
  where open ≡-Reasoning
```

The two previous proofs, constituting a proof of existence and uniqueness, together show terminality of `(ν F, out)`. The *computation law* Harper (2011):

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

The *reflection law* Harper (2011):

```
reflection' : {F : Container 0ℓ 0ℓ}{x : ν F} → A⟦ out ⟧ x ≈≈ x
outfst (reflection') = Eq.refl
outsnd (reflection') {y} = reflection'
reflection : {F : Container 0ℓ 0ℓ}{x : ν F} → A⟦ out ⟧ x ≡ x
reflection = nueq reflection'
```

C Derivations

C.1 Cochurch Stream-fused encoding derivation

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

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

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

$$
\begin{array}{l}
\mathit{fromCh} :: \mathit{ListCh}\ a \to \mathit{List}\ a \\
\mathit{fromCh}\ (\mathit{ListCh}\ \mathit{fold}) = \mathit{fold}\ \mathit{in'} \\
\mathit{in'} :: \mathit{List\_}\ a\ (\mathit{List}\ a) \to \mathit{List}\ a \\
\mathit{in'}\ \mathit{Nil\_} = \mathit{Nil} \\
\mathit{in'}\ (\mathit{Cons\_}\ x\ xs) = \mathit{Cons}\ x\ xs
\end{array}
$$

`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:

$$
\begin{array}{l}
b :: (\mathit{List\_}\ \mathit{Int}\ b \to b) \to (\mathit{Int}, \mathit{Int}) \to b \\
b\ a\ (x, y) = \mathit{loop}\ (x, y) \\
\quad \textbf{where}\ \mathit{loop}\ (x, y) = \textbf{case}\ x > y\ \textbf{of} \\
\qquad \mathit{True} \to a\ \mathit{Nil\_} \\
\qquad \mathit{False} \to a\ (\mathit{Cons\_}\ x\ (\mathit{loop}\ (x + 1, y))) \\
\mathit{betweenCh} :: (\mathit{Int}, \mathit{Int}) \to \mathit{ListCh}\ \mathit{Int} \\
\mathit{betweenCh}\ (x, y) = \mathit{ListCh}\ (\lambda a \to b\ a\ (x, y))
\end{array}
$$

$$
\begin{array}{l}
m :: (a \to b) \to \mathit{List\_}\ a\ c \to \mathit{List\_}\ b\ c \\
m\ f\ \mathit{Nil\_} = \mathit{Nil\_} \\
m\ f\ (\mathit{Cons\_}\ x\ xs) = \mathit{Cons\_}\ (f\ x)\ xs \\
\mathit{mapCh} :: (a \to b) \to \mathit{ListCh}\ a \to \mathit{ListCh}\ b \\
\mathit{mapCh}\ f\ (\mathit{ListCh}\ g) = \mathit{ListCh}\ (\lambda a \to g\ (a\ .\ m\ f))
\end{array}
$$

$$
\begin{array}{l}
\mathit{filterCh} :: (a \to \mathit{Bool}) \to \mathit{ListCh}\ a \to \mathit{ListCh}\ a \\
\mathit{filterCh}\ p\ (\mathit{ListCh}\ g) = \mathit{ListCh}\ (\lambda a \to g\ (\lambda\textbf{case} \\
\qquad \mathit{Nil\_} \to a\ \mathit{Nil\_} \\
\qquad \mathit{Cons\_}\ x\ xs \to \textbf{if}\ (p\ x)\ \textbf{then}\ a\ (\mathit{Cons\_}\ x\ xs)\ \textbf{else}\ xs \\
\quad))
\end{array}
$$

$$
\begin{array}{l}
s :: \mathit{List\_}\ \mathit{Int}\ \mathit{Int} \to \mathit{Int} \\
s\ \mathit{Nil\_} = 0 \\
s\ (\mathit{Cons\_}\ x\ y) = x + y \\
\mathit{sumCh} :: \mathit{ListCh}\ \mathit{Int} \to \mathit{Int} \\
\mathit{sumCh}\ (\mathit{ListCh}\ g) = g\ s
\end{array}
$$

Next, the actual functions:

$$
\begin{array}{l}
\mathit{sum} :: \mathit{List}\ \mathit{Int} \to \mathit{Int} \\
\mathit{sum} = \mathit{sumCh}\ .\ \mathit{toCh}
\end{array}
$$

$$
\begin{array}{l}
\mathit{map} :: (a \to b) \to \mathit{List}\ a \to \mathit{List}\ b \\
\mathit{map}\ f = \mathit{fromCh}\ .\ \mathit{mapCh}\ f\ .\ \mathit{toCh}
\end{array}
$$

$filter :: (a \rightarrow Bool) \rightarrow List\ a \rightarrow List\ a$
$filter\ p = fromCh\ .\ filterCh\ p\ .\ toCh$

$between :: (Int, Int) \rightarrow List\ Int$
$between = fromCh\ .\ betweenCh$

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
$sumCh\ .\ mapCh\ (+1)\ .\ filterCh\ odd\ .\ ListCh\ (\lambda a \rightarrow b\ a\ (x, y))$
-- Dfn of filterCh + beta reduction
$sumCh\ .\ mapCh\ (+1)\ .$
$ListCh\ (\lambda a' \rightarrow$
$(\lambda a \rightarrow b\ a\ (x, y))$
$(\lambda x \rightarrow \mathbf{case}\ x\ \mathbf{of}$
$Nil\_ \rightarrow a'\ Nil\_$
$Cons\_\ x\ xs \rightarrow \mathbf{if}\ (p\ x)\ \mathbf{then}\ a'\ (Cons\_\ x\ xs)\ \mathbf{else}\ xs$
$)$
$)$
-- Beta reduction
$sumCh\ .\ mapCh\ (+1)\ .$
$ListCh\ (\lambda a' \rightarrow$
$b\ (\lambda x \rightarrow \mathbf{case}\ x\ \mathbf{of}$
$Nil\_ \rightarrow a'\ Nil\_$
$Cons\_\ x\ xs \rightarrow \mathbf{if}\ (p\ x)\ \mathbf{then}\ a'\ (Cons\_\ x\ xs)\ \mathbf{else}\ xs$
$)$
$(x, y))$
-- Dfn of mapCh + beta reduction
$sumCh\ .\ ListCh\ (\lambda a \rightarrow$
$(\lambda a' \rightarrow$
$b\ (\lambda x \rightarrow \mathbf{case}\ x\ \mathbf{of}$
$Nil\_ \rightarrow a'\ Nil\_$
$Cons\_\ x\ xs \rightarrow \mathbf{if}\ (p\ x)\ \mathbf{then}\ a'\ (Cons\_\ x\ xs)\ \mathbf{else}\ xs$
$)$
(x, y)
$)$
$(a\ .\ m\ (+1)))$
-- Substitution
$sumCh\ .\ ListCh\ (\lambda a \rightarrow$
$b\ (\lambda x \rightarrow \mathbf{case}\ x\ \mathbf{of}$
$Nil\_ \rightarrow (a\ .\ m\ (+1))\ Nil\_$
$Cons\_\ x\ xs \rightarrow \mathbf{if}\ (p\ x)\ \mathbf{then}\ (a\ .\ m\ (+1))\ (Cons\_\ x\ xs)\ \mathbf{else}\ xs$
$)$
$(x, y))$
-- Dfn of sumCh
$(\lambda a \rightarrow$
$b\ (\lambda x \rightarrow \mathbf{case}\ x\ \mathbf{of}$

```
    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.

C.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
```

$$
\begin{aligned}
&\mathit{fromCoCh} :: \mathit{ListCoCh}\ a \to \mathit{List}\ a\\
&\mathit{fromCoCh}\ (\mathit{ListCoCh}\ h\ s) = \mathit{unfold}\ h\ s\\
&\mathit{unfold} :: (b \to \mathit{List'\_}\ a\ b) \to b \to \mathit{List}\ a\\
&\mathit{unfold}\ h\ s = \mathbf{case}\ h\ s\ \mathbf{of}\\
&\quad \mathit{Nil'\_} \to \mathit{Nil}\\
&\quad \mathit{NilT'\_}\ \mathit{xs} \to \mathit{unfold}\ h\ \mathit{xs}\\
&\quad \mathit{Cons'\_}\ x\ \mathit{xs} \to \mathit{Cons}\ x\ (\mathit{unfold}\ h\ \mathit{xs})
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{su'} :: (s \to \mathit{List'\_}\ \mathit{Int}\ s) \to s \to \mathit{Int}\\
&\mathit{su'}\ h\ s = \mathit{loop}\ s\\
&\quad \mathbf{where}\ \mathit{loop}\ s' = \mathbf{case}\ h\ s'\ \mathbf{of}\\
&\qquad \mathit{Nil'\_} \to 0\\
&\qquad \mathit{NilT'\_}\ \mathit{xs} \to \mathit{loop}\ \mathit{xs}\\
&\qquad \mathit{Cons'\_}\ x\ \mathit{xs} \to x + \mathit{loop}\ \mathit{xs}\\
&\mathit{sumCoCh} :: \mathit{ListCoCh}\ \mathit{Int} \to \mathit{Int}\\
&\mathit{sumCoCh}\ (\mathit{ListCoCh}\ h\ s) = \mathit{su'}\ h\ s
\end{aligned}
$$

$$
\begin{aligned}
&m' :: (a \to b) \to \mathit{List'\_}\ a\ c \to \mathit{List'\_}\ b\ c\\
&m'\ f\ (\mathit{Cons'\_}\ x\ \mathit{xs}) = \mathit{Cons'\_}\ (f\ x)\ \mathit{xs}\\
&m'\ \_\ (\mathit{NilT'\_}\ \mathit{xs}) = \mathit{NilT'\_}\ \mathit{xs}\\
&m'\ \_\ (\mathit{Nil'\_}) = \mathit{Nil'\_}\\
&\mathit{mapCoCh} :: (a \to b) \to \mathit{ListCoCh}\ a \to \mathit{ListCoCh}\ b\\
&\mathit{mapCoCh}\ f\ (\mathit{ListCoCh}\ h\ s) = \mathit{ListCoCh}\ (m'\ f\ .\ h)\ s
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{filt}\ p\ h\ s = \mathbf{case}\ h\ s\ \mathbf{of}\\
&\qquad\qquad\qquad \mathit{Nil'\_} \to \mathit{Nil'\_}\\
&\qquad\qquad\qquad \mathit{NilT'\_}\ \mathit{xs} \to \mathit{NilT'\_}\ \mathit{xs}\\
&\qquad\qquad\qquad \mathit{Cons'\_}\ x\ \mathit{xs} \to \mathbf{if}\ p\ x\ \mathbf{then}\ \mathit{Cons'\_}\ x\ \mathit{xs}\ \mathbf{else}\ \mathit{NilT'\_}\ \mathit{xs}\\
&\mathit{filterCoCh} :: (a \to \mathit{Bool}) \to \mathit{ListCoCh}\ a \to \mathit{ListCoCh}\ a\\
&\mathit{filterCoCh}\ p\ (\mathit{ListCoCh}\ h\ s) = \mathit{ListCoCh}\ (\mathit{filt}\ p\ h)\ s
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{betweenCoCh} :: (\mathit{Int}, \mathit{Int}) \to \mathit{List'\_}\ \mathit{Int}\ (\mathit{Int}, \mathit{Int})\\
&\mathit{betweenCoCh}\ (x, y) = \mathbf{case}\ x > y\ \mathbf{of}\\
&\quad \mathbf{case}\ \mathit{True} \to \mathit{Nil'\_}\\
&\quad \mathbf{case}\ \mathit{False} \to \mathit{Cons'\_}\ x\ (x + 1, y)
\end{aligned}
$$

Next, the actual functions:

$$
\begin{aligned}
&\mathit{sum} :: \mathit{List}\ \mathit{Int} \to \mathit{Int}\\
&\mathit{sum} = \mathit{sumCoCh}\ .\ \mathit{toCoCh}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{map} :: (a \to b) \to \mathit{List}\ a \to \mathit{List}\ b\\
&\mathit{map}\ f = \mathit{fromCoCh}\ .\ \mathit{mapCoCh}\ f\ .\ \mathit{toCoCh}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{filter} :: (a \to \mathit{Bool}) \to \mathit{List}\ a \to \mathit{List}\ a\\
&\mathit{filter}\ p = \mathit{fromCoCh}\ .\ \mathit{filterCoCh}\ p\ .\ \mathit{toCoCh}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{between} :: (\mathit{Int}, \mathit{Int}) \to \mathit{List}\ \mathit{Int}\\
&\mathit{between} = \mathit{fromCoCh}\ .\ \mathit{ListCoCh}\ \mathit{betweenCoCh}
\end{aligned}
$$

We again will fuse the following pipeline:

$$
f = \mathit{sum}\ .\ \mathit{map}\ (+2)\ .\ \mathit{filter}\ \mathit{odd}\ .\ \mathit{between}
$$

$$
\begin{aligned}
&f = \mathit{sumCoCh}\ .\ \mathit{toCoCh}\ .\\
&\mathit{fromCoCh}\ .\ \mathit{mapCoCh}\ (+2)\ .\ \mathit{toCoCh}\ .\\
&\mathit{fromCoCh}\ .\ \mathit{filterCoCh}\ \mathit{odd}\ .\ \mathit{toCoCh}\ .\\
&\mathit{fromCoCh}\ .\ \mathit{ListCoCh}\ \mathit{betweenCoCh}
\end{aligned}
$$

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 (
```

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{case}\ (odd\ x)\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad True \to Cons'\_\ x\ (x+1, y) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad False \to NilT'\_\ (x+1, y) \\
\qquad\qquad\qquad\qquad\qquad\qquad)\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad Cons'\_\ x\ xs \to Cons'\_\ ((+2)\ x)\ xs \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad NilT'\_\ xs \to NilT'\_\ xs \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad Nil'\_ \Rightarrow Nil'\_ \\
\qquad\qquad\qquad\qquad)\ \textbf{of} \\
\qquad\qquad\qquad\qquad\quad Nil'\_ \to 0 \\
\qquad\qquad\qquad\qquad\quad NilT'\_\ s \to loop\ s \\
\qquad\qquad\qquad\qquad\quad Cons'\_\ x\ s \to x + loop\ s \\
loop\ (x, y)
\end{array}
$$

-- case-of-case optimization

$$
\begin{array}{l}
loop\ (x, y) = \textbf{case}\ (\\
\qquad\qquad\qquad\quad \textbf{case}\ (x > y)\ \textbf{of} \\
\qquad\qquad\qquad\qquad True \to Nil'\_ \\
\qquad\qquad\qquad\qquad False \to \textbf{case}\ (odd\ x)\ \textbf{of} \\
\qquad\qquad\qquad\qquad\quad True \to \textbf{case}\ (Cons'\_\ x\ (x+1, y))\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad Cons'\_\ x\ xs \to Cons'\_\ ((+2)\ x)\ xs \\
\qquad\qquad\qquad\qquad\qquad NilT'\_\ xs \to NilT'\_\ xs \\
\qquad\qquad\qquad\qquad\qquad Nil'\_ \Rightarrow Nil'\_ \\
\qquad\qquad\qquad\qquad\quad False \to \textbf{case}\ (NilT'\_\ (x+1, y))\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad Cons'\_\ x\ xs \to Cons'\_\ ((+2)\ x)\ xs \\
\qquad\qquad\qquad\qquad\qquad NilT'\_\ xs \to NilT'\_\ xs \\
\qquad\qquad\qquad\qquad\qquad Nil'\_ \Rightarrow Nil'\_ \\
\qquad\qquad\quad)\ \textbf{of} \\
\qquad\qquad\qquad Nil'\_ \to 0 \\
\qquad\qquad\qquad NilT'\_\ s \to loop\ s \\
\qquad\qquad\qquad Cons'\_\ x\ s \to x + loop\ s \\
loop\ (x, y)
\end{array}
$$

-- Case-of-known-case optimization

$$
\begin{array}{l}
loop\ (x, y) = \textbf{case}\ (\\
\qquad\qquad\qquad\quad \textbf{case}\ (x > y)\ \textbf{of} \\
\qquad\qquad\qquad\qquad True \to Nil'\_ \\
\qquad\qquad\qquad\qquad False \to \textbf{case}\ (odd\ x)\ \textbf{of} \\
\qquad\qquad\qquad\qquad\quad True \to Cons'\_\ ((+2)\ x)\ (x+1, y) \\
\qquad\qquad\qquad\qquad\quad False \to NilT'\_\ (x+1, y) \\
\qquad\qquad\quad)\ \textbf{of} \\
\qquad\qquad\qquad Nil'\_ \to 0 \\
\qquad\qquad\qquad NilT'\_\ s \to loop\ s \\
\qquad\qquad\qquad Cons'\_\ x\ s \to x + loop\ s \\
loop\ (x, y)
\end{array}
$$

-- case-of-case optimization

$$
\begin{array}{l}
loop\ (x, y) = \textbf{case}\ (x > y)\ \textbf{of} \\
\qquad\qquad\qquad\quad True \to \textbf{case}\ (Nil'\_)\ \textbf{of} \\
\qquad\qquad\qquad\qquad Nil'\_ \to 0 \\
\qquad\qquad\qquad\qquad NilT'\_\ s \to loop\ s \\
\qquad\qquad\qquad\qquad Cons'\_\ x\ s \to x + loop\ s \\
\qquad\qquad\qquad\quad False \to \textbf{case}\ (\\
\qquad\qquad\qquad\qquad \textbf{case}\ (odd\ x)\ \textbf{of} \\
\qquad\qquad\qquad\qquad\quad True \to Cons'\_\ ((+2)\ x)\ (x+1, y) \\
\qquad\qquad\qquad\qquad\quad False \to NilT'\_\ (x+1, y) \\
\qquad\qquad\qquad\quad)\ \textbf{of} \\
\qquad\qquad\qquad\qquad Nil'\_ \to 0 \\
\qquad\qquad\qquad\qquad NilT'\_\ s \to loop\ s \\
\qquad\qquad\qquad\qquad Cons'\_\ x\ s \to x + loop\ s \\
loop\ (x, y)
\end{array}
$$

-- 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
```

C.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
```

$$
\begin{aligned}
&\quad \textbf{where } loop\ s'\ acc = \textbf{case}\ h\ s'\ \textbf{of}\\
&\qquad Nil'\_ \rightarrow acc\\
&\qquad NilT'\_\ xs \rightarrow loop\ xs\ acc\\
&\qquad Cons'\_\ x\ xs \rightarrow loop\ xs\ (x + acc)\\
&sumCoCh :: ListCoCh\ Int \rightarrow Int\\
&sumCoCh\ (ListCoCh\ h\ s) = su'\ h\ s
\end{aligned}
$$

$$
\begin{aligned}
&m' :: (a \rightarrow b) \rightarrow List'\_\ a\ c \rightarrow List'\_\ b\ c\\
&m'\ f\ (Cons'\_\ x\ xs) = Cons'\_\ (f\ x)\ xs\\
&m'\ \_\ (NilT'\_\ xs) = NilT'\_\ xs\\
&m'\ \_\ (Nil'\_) = Nil'\_\\
&mapCoCh :: (a \rightarrow b) \rightarrow ListCoCh\ a \rightarrow ListCoCh\ b\\
&mapCoCh\ f\ (ListCoCh\ h\ s) = ListCoCh\ (m'\ f\ .\ h)\ s
\end{aligned}
$$

$$
\begin{aligned}
&filt\ p\ h\ s = \textbf{case}\ h\ s\ \textbf{of}\\
&\qquad\qquad\qquad Nil'\_ \rightarrow Nil'\_\\
&\qquad\qquad\qquad NilT'\_\ xs \rightarrow NilT'\_\ xs\\
&\qquad\qquad\qquad Cons'\_\ x\ xs \rightarrow \textbf{if}\ p\ x\ \textbf{then}\ Cons'\_\ x\ xs\ \textbf{else}\ NilT'\_\ xs\\
&filterCoCh :: (a \rightarrow Bool) \rightarrow ListCoCh\ a \rightarrow ListCoCh\ a\\
&filterCoCh\ p\ (ListCoCh\ h\ s) = ListCoCh\ (filt\ p\ h)\ s
\end{aligned}
$$

$$
\begin{aligned}
&betweenCoCh :: (Int, Int) \rightarrow List'\_\ Int\ (Int, Int)\\
&betweenCoCh\ (x, y) = \textbf{case}\ x > y\ \textbf{of}\\
&\quad True \rightarrow Nil'\_\\
&\quad False \rightarrow Cons'\_\ x\ (x + 1, y)
\end{aligned}
$$

Next, the actual functions:

$$
\begin{aligned}
&sum :: List\ Int \rightarrow Int\\
&sum = sumCoCh\ .\ toCoCh
\end{aligned}
$$

$$
\begin{aligned}
&map :: (a \rightarrow b) \rightarrow List\ a \rightarrow List\ b\\
&map\ f = fromCoCh\ .\ mapCoCh\ f\ .\ toCoCh
\end{aligned}
$$

$$
\begin{aligned}
&filter :: (a \rightarrow Bool) \rightarrow List\ a \rightarrow List\ a\\
&filter\ p = fromCoCh\ .\ filterCoCh\ p\ .\ toCoCh
\end{aligned}
$$

$$
\begin{aligned}
&between :: (Int, Int) \rightarrow List\ Int\\
&between = fromCoCh\ .\ ListCoCh\ betweenCoCh
\end{aligned}
$$

Here is the example piepline:

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

$$
\begin{aligned}
&f = sumCoCh\ .\ toCoCh\ .\\
&fromCoCh\ .\ mapCoCh\ (+2)\ .\ toCoCh\ .\\
&fromCoCh\ .\ filterCoCh\ odd\ .\ toCoCh\ .\\
&fromCoCh\ .\ ListCoCh\ betweenCoCh
\end{aligned}
$$

When 'fused' it looks like this:

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

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

$$
\begin{aligned}
&sumCoCh\ .\ mapCoCh\ (+2)\ .\ filterCoCh\ odd\ .\ ListCoCh\ betweenCoCh\ (x, y)\\
&\qquad \text{-- Inlining of filterCoCh + beta reduction}\\
&sumCoCh\ .\ mapCoCh\ (+2)\ .\ ListCoCh\ (filt\ odd\ betweenCoCh)\ (x, y)\\
&\qquad \text{-- Inlining of mapCoCh + beta reduction}\\
&sumCoCh\ .\ ListCoCh\ (m'\ (+2)\ .\ filt\ odd\ betweenCoCh)\ (x, y)\\
&\qquad \text{-- Inlining of sumCoCh + beta reduction}
\end{aligned}
$$

```
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.