Master's Thesis

Eben Rogers

May 2, 2024

Contents

1 Introduction

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

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

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

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

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

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 read- and write-ability. Can you write a high-performance, single-traversal, version of the following function (Harper, 2011)?

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

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

$$\begin{aligned}&f' :: (\mathit{Int}, \mathit{Int}) \rightarrow \mathit{Int}\\&f'\ (x, y) = \mathit{loop}\ x\\&\quad \mathbf{where}\ \mathit{loop}\ x \mid x > y = 0\\&\quad\quad\quad\quad\quad\quad \mid \mathit{otherwise} = \mathbf{if}\ \mathit{odd}\ x\\&\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{then}\ (x+1) + \mathit{loop}\ (x+1)\\&\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{else}\ \mathit{loop}\ (x+1)\end{aligned}$$

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

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

Initial work on this was done my Wadler (1984, 1986, 1990), and was dubbed 'deforestation', referring to the removal of intermediate trees (or lists). The details of the original deforestation work are not relevant to this thesis, but, the weaknesses of the work are described and a different technique are proposed by Gill et al. (1993). Gill et al. (1993) describe a technique nowadays called `foldr/build` fusion, which, when employed, can eliminate most intermediate lists. This technique is described 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, leveras the coinductive list datatype, streams. This technique ended up being called *stream fusion* introduced by Coutts et al. (2007).

(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 of (Co)Church encodings are described and, pragmatically, how to implement them in Haskell. My thesis is centered on Harper (2011)'s work and makes two crucial contributions:

1. The Church and Cochurch encodings described are formalized, 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). This is discussed in detail in Section 3.

2. The Church and Cochurch encodings' implementation in Haskell, as described by Harper (2011) are replicated and investigated further as to their performance characteristics. In this process, a bug was found in Haskell's optimizer, and further practical insights were gleaned as to how to get these encodings to properly fuse as well (especially for Cochurch encodings) and what optimizations enable shortcut fusion to do its work. This is discussed in detail in Section 4.

Fusion, Category theory, Libfusion paper, church encodings, formalization of it, Haskell's suite of optimizations that enable fusion, (theorems for free?).

2 Background

2.1 Foldr/build fusion (on lists)

Starting with the basics of fusion. In Gill et al. (1993)'s paper the original 'schortcut deforestation' technique was described. The core idea is described here as follows:

In functional programming lists are (often) used to store the output of one function such that it can then be consumed by another function. To co-opt Gill et al. (1993)'s example:

$$\mathit{all}\ p\ \mathit{xs} = \mathit{and}\ (\mathit{map}\ p\ \mathit{xs})$$

`map p xs` applies `p` to all of the elements, producing a boolean list, and `and` takes that new list and "ands" all of them together to produce a resulting boolean value. "The intermediate list is discarded, and eventually recovered by the garbage collector" (Gill et al., 1993).

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

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

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

Gill et al. (1993)'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`:

$$\mathit{build}\ g = g\ (:)\ [\,]$$

Such that:

$$\mathit{foldr}\ k\ z\ (\mathit{build}\ g) = g\ k\ z$$

Gill et al. (1993)."

Gill et al. (1993) dubbed this the `foldr/build` rule. For its validity `g` needs to be of type:

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

Which can be proved to be true through the use of g's free theorem à la Wadler (1989). For more information on free theorems see Section 2.4

2.1.1 An example

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

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

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

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

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

$$\mathit{from}\ a\ b = \mathit{build}\ (\mathit{from}'\ a\ b)$$

With some inlining and β reduction, one can see that this definition is identical to the original `from` definition. Now for the killer feature (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` rule here to prevent an intermediate list being produced. Any adjacent `foldr/build` pair "cancel away". This is an example of shortcut fusion.

One can rewrite many functions in terms of `foldr` and `build` such that this fusion can be applied. This can be seen in Figure 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 an as efficcient minimum as can be expected.

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

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

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

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

2.1.2 Generalization to any datastructure

This is all well and good, when working with lists, that can be written in terms of `foldr`'s and/or `build`'s (which covers a lot of common functions already), but what if we want to do this for any data structure? Is there a way of generalizing this? The answer is yes*. *So long as the datatype we are working with is an initial algebra or terminal coalgebra, and the functions we are working with are instances of cata- or anamorphisms.

What does that even mean?

2.2 The category theory

In order to explain what an initial/terminal (co) algebra is, I'll first need to explain what a functor is and, more pressingly, what a category is. The concept of cata- and anamorphisms will follow suit. If you're familiar with category theory and these concepts, you can skip this section.

2.2.1 A Category

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

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

These pieces of data should satisfy the following three properties:

1. (**Left unit law**) For any morphism $f \in \texttt{hom}_{\mathcal{C}}(X, Y)$:

$$f \circ \texttt{Id}_X = f$$

2. (**Right unit law**) For any morphism $f \in \texttt{hom}_{\mathcal{C}}(X, Y)$:

$$\texttt{Id}_Y \circ f = f$$

3. (**Associative law**) For any morphisms $f \in \texttt{hom}_{\mathcal{C}}(X, Y), g \in \texttt{hom}_{\mathcal{C}}(Y, Z)$, and $h \in \texttt{hom}_{\mathcal{C}}(Z, W)$:

$$h \circ (g \circ f) = (h \circ g) \circ f$$

2.2.2 Initial/Terminal Objects

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

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

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

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

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

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

2.2.3 Functors

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

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

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

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

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

These pieces of data should satisfy these two properties:

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

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

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

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

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

2.2.4 (Category of) F-(Co)Algebras

Given an endofunctor $F : \mathcal{C} \to \mathcal{C}$:

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

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

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

$$\begin{tikzcd}
FC \arrow[r, "\phi"] \arrow[d, "Ff"'] & C \arrow[d, "f"] \\
FD \arrow[r, "\psi"] & D
\end{tikzcd}$$

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

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

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

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

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

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

2.2.5 Cata- and Anamorphisms

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

$$\begin{tikzcd}
F\mu^F \arrow[r, "in"] \arrow[d, "F(\!|\phi|\!)"'] & \mu^F \arrow[d, "(\!|\phi|\!)"] \\
FC \arrow[r, "\phi"] & C
\end{tikzcd}$$

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

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

2.2.6 Fusion property

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

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

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

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

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

2.3 Library Writer's Guide to Shortcut Fusion

Gill et al. (1993)'s work has been built upon in several ways:

-

One work that attempts to clearly explain a generalized form of Gill et al. (1993)'s work is "A Library Writer's Guide to Shortcut Fusion" by Harper (2011).

In the work, Harper (2011) explain the concept of Church and CoChurch encodings in three steps:

1. Explaining the mathematical background of Category theory, including F-Algebras, Fusion, and

2.4 Theorems for Free

2.5 Containers

3 Formalization

In Harper (2011)'s work "A Library Writer's Guide to Shortcut Fusion", the practice of implementing Church and CoChurch encodings is described, as well a paper proof necessary to show that the encodings optimizations employed are correct.

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

The code can be neatly presented in roughly 2 parts:

- The proofs of the category theory truths described by Harper (2011).
- The proofs about the (Co)Church encodings, again as described by Harper (2011).

A note on imports: Imports are omitted in the agda code except when an import renames a construct it is importing, this is most prevalent for `Category`, `Data.W`, and `Container`.

3.1 Category Theory Formalization

3.1.1 funct

This module contains some simple definition, utilized in both complimentary structures (cata-/anamorphisms, church/cochurch).

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

```
module agda.funct.funext where
open import Axiom.Extensionality.Propositional
postulate funext : ∀{a b} → Extensionality a b
funexti : ∀{a b} → ExtensionalityImplicit a b
funexti = implicit-extensionality funext
```

Endofunctors An endofunctor is defined across the category of agda sets, where the functors are interpretations of containers. There is a little bit of unwieldyness as `Sets` defines equality through extensionality, but using an implicit parameter. In order to combine it with `funext` a little bit of unpacking and repacking of the definitions needs to be done.

```
module agda.funct.endo where
F[_] : (F : Container 0ℓ 0ℓ) → Endofunctor (Sets 0ℓ)
F[ F ] = record { F₀ = ⟦ F ⟧ ; F₁ = map
                ; identity = refl ; homomorphism = refl
                ; F-resp-≈ = λ p → cong₂ map (funext (λ x → p {x})) refl }
```

3.1.2 init

This module defines F-Algebras, a candidate initial object μ, and catamorphisms, and proves initiality of μ, the fusion properties, and the catamorphism laws.

Initial algebras and catamorphisms This module defines a function and shows it to be a catamorphism in the category of F-Agebras. Specifically, it is shown that `(μ F, in')` is initial.

```
module agda.init.initalg where
open import Categories.Category renaming (Category to Cat)
open import Data.W using () renaming (sup to in')
```

A shorthand for the Category of F-Algebras.

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

A shorthand for an F-Algebra homomorphism:

```
_Alghom[_,_] : {X Y : Set}(F : Container 0ℓ 0ℓ)(x : ⟦ F ⟧ X → X)(Y : ⟦ F ⟧ Y → Y) → Set
F Alghom[ x , y ] = C[ F ]Alg [ to-Algebra x , to-Algebra y ]
```

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

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

It is shown that any ⦇_⦈ is a valid F-Algebra homomorphism from `in'` to any other object `a`. This constitutes a proof of existence:

```
valid-falghom : {F : Container 0ℓ 0ℓ}{X : Set}(a : ⟦ F ⟧ X → X) → F Alghom[ in' , a ]
valid-falghom {X} a = record { f = ⦇ a ⦈ ; commutes = refl }
```

It is shown that any other valid F-Algebra homomorphism from `in'` to `a` is equal to the ⦇_⦈ function defined. This constitutes a proof of uniqueness:

```
isunique : {F : Container 0ℓ 0ℓ}{X : Set}{a : ⟦ F ⟧ X → X}(fhom : F Alghom[ in' , a ])(x : μ F) →
             ⦇ a ⦈ x ≡ fhom .f x
isunique {_}{_}{a} fhom (in' (op , ar)) = begin
        ⦇ a ⦈ (in' (op , ar))
  ≡⟨⟩ -- Dfn of ⦇_⦈
        a (op , ⦇ a ⦈ ∘ ar)
  ≡⟨ cong (λ h → a (op , h)) (funext $ isunique fhom ∘ ar) ⟩ -- induction
        a (op , (fhom .f) ∘ ar)
  ≡⟨⟩ -- Dfn of map
        (a ∘ map (fhom .f)) (op , ar)
  ≡⟨ sym $ fhom .commutes ⟩
        (fhom .f ∘ in') (op , ar)
  □
```

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

```
initial-in : {F : Container 0ℓ 0ℓ} → IsInitial C[ F ]Alg (to-Algebra in')
initial-in = record { ! = λ {A} → valid-falghom (A .α)
                    ; !-unique = λ fhom {x} → isunique fhom x }
```

Initial F-Algebra fusion This module proves the categorical fusion property (see Section 2.2.6). From it, it extracts the 'fusion law' as it was declared by Harper (2011); which is easier to work with. This shows that the fusion law does follow from the fusion property.

```
module agda.init.fusion where
open import Categories.Category renaming (Category to Cat)
```

The categorical fusion property:

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

The 'fusion law':

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

Universal properties of catamorphisms This module proves some properties of catamorphisms.

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

The forward direction of the *universal property of folds* (Harper, 2011):

```
universal-prop : {F : Container 0ℓ 0ℓ}{X : Set}(a : ⟦ F ⟧ X → X)(h : μ F → X) →
                 h ≡ ⦇ a ⦈ → h ∘ in' ≡ a ∘ map h
universal-prop a h eq rewrite eq = refl
```

The *computation law* (Harper, 2011) (this is exactly how ⦇_⦈ is defined in the first place):

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

The *reflection law* (Harper, 2011):

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

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

3.1.3 term

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

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

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

A shorthand for the Category of F-Coalgebras:

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

A shorthand for an F-Coalgebra homomorphism:

```
_CoAlghom[_,_] : {X Y : Set}(F : Container 0ℓ 0ℓ)(x : X → ⟦ F ⟧ X)(Y : Y → ⟦ F ⟧ Y) → Set
F CoAlghom[ x , y ] = C[ F ]CoAlg [ to-Coalgebra x , to-Coalgebra y ]
```

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

```
record ν (F : Container 0ℓ 0ℓ) : Set where
  coinductive
  field out : ⟦ F ⟧ (ν F)
open ν
A⟦_⟧ : {F : Container 0ℓ 0ℓ}{X : Set} → (X → ⟦ F ⟧ X) → X → ν F
out (A⟦ c ⟧ x) = (λ (op , ar) → op , A⟦ c ⟧ ∘ ar) (c x)
```

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

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

It is shown that any ⟦_⟧ is a valid F-Coalgebra homomorphism from `out` to any other object `a`. This constitutes a proof of existence:

```
valid-fcoalghom : {F : Container 0ℓ 0ℓ}{X : Set}(a : X → ⟦ F ⟧ X) → F CoAlghom[ a , out ]
valid-fcoalghom {X} a = record { f = A⟦ a ⟧ ; commutes = refl }
```

It is shown that any other valid F-Coalgebra homomorphism from `out` to `a` is equal to the ⟦_⟧ defined. 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, a modification to ν might be needed in order ensure proper termination checking for this proof:

```
{-# NON_TERMINATING #-}
isunique : {F : Container 0ℓ 0ℓ}{X : Set}{c : X → ⟦ F ⟧ X}(fhom : F CoAlghom[ c , out ])
                (x : X) → A⟦ c ⟧ x ≡ fhom .f x
isunique {_}{_}{c} fhom x = out-injective (begin
            (out ∘ A⟦ c ⟧) x
   ≡⟨⟩ -- Definition of ⟦_⟧
            map A⟦ c ⟧ (c x)
   ≡⟨⟩
            (λ(op , ar) → (op , A⟦ c ⟧ ∘ ar)) (c x)
   -- Same issue as with the proof of reflection it seems...
   ≡⟨ cong (λ f → op , f) (funext $ isunique fhom ∘ ar) ⟩ -- induction
            (op , fhom .f ∘ ar)
   ≡⟨⟩
            map (fhom .f) (c x)
   ≡⟨⟩ -- Definition of composition
            (map (fhom .f) ∘ c) x
   ≡⟨ sym $ fhom .commutes ⟩
            (out ∘ fhom .f) x
```

```
  □)
  where op = Σ.proj₁ (c x)
        ar = Σ.proj₂ (c x)
```

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

```
terminal-out : {F : Container 0ℓ 0ℓ} → IsTerminal C[ F ]CoAlg (to-Coalgebra out)
terminal-out = record { ! = λ {A} → valid-fcoalghom (A .α)
                      ; !-unique = λ fhom {x} → isunique fhom x }
```

Terminal F-Coalgebra fusion This module proves the categorical fusion property. From it, it extracts a 'fusion law' as it was defined by Harper (2011); which is easier to work with. This shows that the fusion law does follow from the fusion property.

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

The categorical fusion property:

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

The 'fusion law':

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

Universal property of anamorphisms This module proves some property of anamorphisms.

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

The forward direction of the *universal property of unfolds* Harper (2011):

```
universal-prop : {F : Container 0ℓ 0ℓ}{C : Set}(c : C → ⟦ F ⟧ C)(h : C → ν F) →
                 h ≡ A⟦ c ⟧ → out ∘ h ≡ map h ∘ c
universal-prop c h eq rewrite eq = refl
```

The *computation law* Harper (2011):

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

The *reflection law* Harper (2011): SOMETHING ABOUT TERMINATION.

```
{-# NON_TERMINATING #-}
reflection : {F : Container 0ℓ 0ℓ}(x : ν F) → A⟦ out ⟧ x ≡ x
reflection x = out-injective (begin
    out (A⟦ out ⟧ x)
  ≡⟨⟩
    map A⟦ out ⟧ (out x)
  ≡⟨⟩
    op , A⟦ out ⟧ ∘ ar
  ≡⟨ cong (λ f → op , f) (funext $ reflection ∘ ar) ⟩
```

```
    op , id ∘ ar
  ≡⟨⟩
    map id (out x)
  ≡⟨⟩
    out x
  □)
  where op = Σ.proj₁ (out x)
        ar = Σ.proj₂ (out x)
```

3.2 Short cut fusion

3.2.1 Church encodings

Definition of Church encodings This module defines Church encodings and the two conversions `con` and `abs`, called `toCh` and `fromCh` here, respectively. It also defines the generalized producing, transformation, and consuming functions, as described by Harper (2011).

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

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 0ℓ 0ℓ} → 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)
prod : {ℓ : Level}{F : Container _ _}{Y : Set ℓ}
         (g : {X : Set} → (⟦ F ⟧ X → X) → Y → X)(y : Y) → μ F
prod g = fromCh ∘ prodCh g
eqProd : {F : Container _ _}{Y : Set}
         {g : {X : Set} → (⟦ F ⟧ X → X) → Y → X} → prod g ≡ g in'
eqProd = refl
```

Second, the natural transformation function:

```
natTransCh : {F G : Container _ _}
             (nat : {X : Set} → ⟦ F ⟧ X → ⟦ G ⟧ X) → Church F → Church G
natTransCh nat (Ch g) = Ch (λ a → g (a ∘ nat))
natTrans : {F G : Container _ _}
             (nat : {X : Set} → ⟦ F ⟧ X → ⟦ G ⟧ X) → μ F → μ G
natTrans nat = fromCh ∘ natTransCh nat ∘ toCh
eqNatTrans : {F G : Container _ _}
             {nat : {X : Set} → ⟦ F ⟧ X → ⟦ G ⟧ X} →
             natTrans nat ≡ ⦇ in' ∘ nat ⦈
eqNatTrans = refl
```

Third, the consuming function, note that this is a generalized version of 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
cons : {F : Container _ _}{X : Set}
         (c : ⟦ F ⟧ X → X) → μ F → X
cons c = consCh c ∘ toCh
eqCons : {F : Container _ _}{X : Set}
         {c : ⟦ F ⟧ X → X} → cons c ≡ ⦇ c ⦈
eqCons = refl
```

Proof obligations In Harper (2011)'s work, five proofs proofs are given for Church encodings. These are formalized in this module.

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

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

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

The second proof is similar to the first, but it proves the composition in the other direction `toCh ∘ fromCh = id`. This proofs leverages parametricity as described by Wadler (1989). It postulates the free theorem of the function `g :` *forall* `A . (F A -> A) -> A`, to prove that "applying `g` to `b` and then passing the result to `h`, is the same as just folding `c` over the datatype" (Harper, 2011):

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

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

The third proof shows church-encoded functions constitute an implementation for the consumer functions being replaced. The proof is proved via reflexivity, but Harper (2011)'s original proof steps are included here for completeness:

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

The fourth proof shows that church-encoded functions constitute an implementation for the producing functions being replaced. The proof is proved via reflexivity, but Harper (2011)'s original proof steps are included here for completeness:

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

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

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

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

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

```
    ≡⟨ funext (λ where (Ch g) → refl) ⟩
      natTransCh (nat2 ∘ nat1)
    □
```

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

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

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

```
module agda.church.inst.list where
open import Data.W renaming (sup to in')
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
eq : {f : ℕ → ℕ} → sum1 ∘ map1 f ∘ between1 ≡ sum2 ∘ map2 f ∘ between2
eq {f} = begin
      ⦇ s ⦈ ∘ ⦇ in' ∘ m f ⦈ ∘ b in'
   ≡⟨ cong (λ g → ⦇ s ⦈ ∘ ⦇ in' ∘ m f ⦈ ∘ g) (prod-pres b) ⟩ -- reflexive
      ⦇ s ⦈ ∘ ⦇ in' ∘ m f ⦈ ∘ fromCh ∘ prodCh b
   ≡⟨ cong (λ f → ⦇ s ⦈ ∘ f ∘ prodCh b) (sym $ trans-pres (m f)) ⟩
      ⦇ s ⦈ ∘ fromCh ∘ natTransCh (m f) ∘ prodCh b
   ≡⟨ cong (λ g → g ∘ fromCh ∘ natTransCh (m f) ∘ prodCh b) (cons-pres s) ⟩ -- reflexive
      consCh s ∘ toCh ∘ fromCh ∘ natTransCh (m f) ∘ prodCh b
   ≡⟨ cong (λ g → consCh s ∘ toCh ∘ fromCh ∘ natTransCh (m f) ∘ g ∘ prodCh b) (sym to-from-id) ⟩
      consCh s ∘ toCh ∘ fromCh ∘ natTransCh (m f) ∘ toCh ∘ fromCh ∘ prodCh b
   ≡⟨⟩
      consu s ∘ natTrans (m f) ∘ prod b
   □
```

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

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

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

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


open import Level hiding (zero; suc)
open import Data.Product hiding (map)
--open import Data.Nat
open import Data.Sum as S
open import Data.Fin hiding (_+_; _¿_; _-_)
open import Data.Empty
open import Data.Unit
--open import Function.Base
open import Data.Bool
open import Agda.Builtin.Nat
open import agda.church.defs
open import agda.church.proofs
open import agda.funct.funext
open import agda.init.initalg
open import Relation.Binary.PropositionalEquality as Eq
open ≡-Reasoning


module agda.church.inst.free where
open import Data.Container using (Container; ⟦_⟧; μ; map; _▷_)
open import Data.Container.Combinator as C using (const; to-⊎; _⊎_)
open import Data.W renaming (sup to in')

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

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

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

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

3.2.2 Cochurch encodings

Definition of Cochurch encodings This module defines Cochurch encodings and the two conversion functions `con` and `abs`, called `toCoCh` and `fromCoCh` here, respectively. It also defines the generalized producing, transformation, and consuming functions, as described by Harper (2011). The definition of the CoChurch datatypes is defined slightly 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. The original definition is included as `CoChurch'`.

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

The Cochurch encoding, agian leveraging containers:

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

The conversion functions:

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

The generalized encoded producing, transformation, and consuming functions, alongside the proof that they are equal to the functions they are encoding. First, the producing function, note that this is a generalized version of 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
prod : {F : Container 0ℓ 0ℓ}{Y : Set} → (g : Y → ⟦ F ⟧ Y) → Y → ν F
prod g = fromCoCh ∘ prodCoCh g
eqprod : {F : Container 0ℓ 0ℓ}{Y : Set}{g : (Y → ⟦ F ⟧ Y)} →
         prod g ≡ A⟦ g ⟧
eqprod = refl
```

Second the transformation function:

```
natTransCoCh : {F G : Container 0ℓ 0ℓ}(nat : {X : Set} → ⟦ F ⟧ X → ⟦ G ⟧ X) → CoChurch F → CoChurch G
natTransCoCh n (CoCh h s) = CoCh (n ∘ h) s
natTrans : {F G : Container 0ℓ 0ℓ}(nat : {X : Set} → ⟦ F ⟧ X → ⟦ G ⟧ X) → ν F → ν G
natTrans nat = fromCoCh ∘ natTransCoCh nat ∘ toCoCh
eqNatTrans : {F G : Container 0ℓ 0ℓ}{nat : {X : Set} → ⟦ F ⟧ X → ⟦ G ⟧ X} →
  natTrans nat ≡ A⟦ nat ∘ out ⟧
eqNatTrans = refl
```

Third the consuming function, note that this a is a generalized version of 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
cons : {F : Container 0ℓ 0ℓ}{Y : Set} → (c : {S : Set} → (S → ⟦ F ⟧ S) → S → Y) → ν F → Y
cons c = consCoCh c ∘ toCoCh
eqcons : {F : Container 0ℓ 0ℓ}{X : Set}{c : {S : Set} → (S → ⟦ F ⟧ S) → S → X} →
         cons c ≡ c out
eqcons = refl
```

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

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

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

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

Proof obligations As with Church encodings, in Harper (2011)'s work, five proof obligations needed to be satisfied. These are formalized in this module.

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

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

```
from-to-id : {F : Container 0ℓ 0ℓ} → fromCoCh ∘ toCoCh ≡ id
from-to-id {F} = funext (λ (x : ν F) → begin
    fromCoCh (toCoCh x)
  ≡⟨⟩ -- Definition of toCh
```

```
      fromCoCh (CoCh out x)
    ≡⟨⟩ -- Definition of fromCh
      A⟦ out ⟧ x
    ≡⟨ reflection x ⟩
      x
    ≡⟨⟩
      id x
    ∎)
```

The second proof proof is similar to the first, but it proves the composition in the other direction `toCoCh ∘ fromCoCh = id`. This proof leverages the parametricity as described by 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):

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

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

The third proof shows that cochurch-encoded functions constitute an implementation for the producing functions being replaced. The proof is proved via reflexivity, but Harper (2011)'s original proof steps are included here for completeness:

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

The fourth proof shows that cochurch-encoded functions constitute an implementation for the consuming functions being replaced. The proof is proved via reflexivity, but Harper (2011)'s original proof steps are included here for completeness:

```
cons-pres : {F : Container 0ℓ 0ℓ}{X : Set} → (f : {Y : Set} → (Y → ⟦ F ⟧ Y) → Y → X) →
            consCoCh f ∘ toCoCh ≡ f out
cons-pres f = funext λ x → begin
```

```
    consCoCh f (toCoCh x)
  ≡⟨⟩ -- definition of toCoCh
    consCoCh f (CoCh out x)
  ≡⟨⟩ -- function application
    f out x
  □
```

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

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

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

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

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

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

```
pipefuse : {F G : Container 0ℓ 0ℓ}{X : Set}(c : X → ⟦ F ⟧ X)
           (nat : {X : Set} → ⟦ F ⟧ X → ⟦ G ⟧ X) →
           (f : {Y : Set} → (Y → ⟦ G ⟧ Y) → Y → X) →
           cons f ∘ natTrans nat ∘ prod c ≡ f (nat ∘ c)
pipefuse c nat f = begin
```

```
  consCoCh f ∘ toCoCh ∘ fromCoCh ∘ natTransCoCh nat ∘ toCoCh ∘ fromCoCh ∘ prodCoCh c
≡⟨ cong (λ g → consCoCh f ∘ g ∘ natTransCoCh nat ∘ toCoCh ∘ fromCoCh ∘ prodCoCh c) to-from-id ⟩
  consCoCh f ∘ natTransCoCh nat ∘ toCoCh ∘ fromCoCh ∘ prodCoCh c
≡⟨ cong (λ g → consCoCh f ∘ natTransCoCh nat ∘ g ∘ prodCoCh c) to-from-id ⟩
  consCoCh f ∘ natTransCoCh nat ∘ prodCoCh c
≡⟨⟩
  f (nat ∘ c)
□
```

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

```
module agda.cochurch.inst.list where
open import agda.cochurch.defs renaming (cons to consu)
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

equality The below proof shows the equality between the non-cochurch-endcoded pipeline and the cochurch-encoded pipeline. Note how it is different from the proof for church-encoded pipelines. This is because Harper (2011)'s proof for the proof obligation of natural transformations is different for cochurch encodings than for church encodings. Because of this the first and second proof step for `eq` in the church-encoded lists is done in one step here:

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

4 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? etc.

In this section I'll describe my work replicating the fused Haskell code of the Harper (2011)'s work and further optimization opportunities that were discovered along the way.

I'll start off with the existing working code, followed by a discussion of the discoveries made throughout the process of writing, replicating, and further optimizing Harper (2011)'s example code.

4.1 Replicated Code

4.1.1 Leaf Trees

In this section, the replication of Harper (2011)'s code is described. We start with his motivating example at the beggining of the paper, followed by the 'fused' version that we want the pipeline to become, once compiled:

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

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

Datatypes In his paper Harper (2011) implemented his example functions using leaf trees, this is defined as `Tree` below. Furthermore, the base functor of `Tree` was defined, as `Tree_`, with the recursive positions of the functor turned into a paramater of the datatype:

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

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

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

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

```
toCh :: Tree a → TreeCh a
toCh t = TreeCh (λa → fold a t)
fold :: (Tree_ a b → b) → Tree a → b
fold a Empty = a Empty_
fold a (Leaf x) = a (Leaf_ x)
fold a (Fork l r) = a (Fork_ (fold a l)
                             (fold a r))
fromCh :: TreeCh a → Tree a
fromCh (TreeCh fold) = fold in′
in′ :: Tree_ a (Tree a) → Tree a
in′ Empty_ = Empty
in′ (Leaf_ x) = Leaf x
in′ (Fork_ l r) = Fork l r
```

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

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

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

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

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

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

```
toCoCh :: Tree a → TreeCoCh a
toCoCh = TreeCoCh out
out Empty = Empty_
out (Leaf a) = Leaf_ a
out (Fork l r) = Fork_ l r
fromCoCh :: TreeCoCh a → Tree a
fromCoCh (TreeCoCh h s) = unfold h s
unfold h s = case h s of
  Empty_ → Empty
  Leaf_ a → Leaf a
  Fork_ sl sr → Fork (unfold h sl) (unfold h sr)
```

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

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

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

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

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

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

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

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

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

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

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

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

```
filt :: (a → Bool) → Tree_ a c → Tree_ a c
filt p Empty_ = Empty_
filt p (Leaf_ x) = if p x then Leaf_ x else Empty_
filt p (Fork_ l r) = Fork_ l r
    -- Why can’t I generalize this function???
filterCh :: (a → Bool) → TreeCh a → TreeCh a
filterCh p (TreeCh g) = TreeCh (λa → g (a ∘ filt p))
filter2 :: (a → Bool) → Tree a → Tree a
filter2 p = fromCh ∘ filterCh p ∘ toCh
 {-# INLINE filter2 #-}
filterCoCh :: (a → Bool) → TreeCoCh a → TreeCoCh a
filterCoCh p (TreeCoCh h s) = TreeCoCh (filt p ∘ h) s
filter3 :: (a → Bool) → Tree a → Tree a
filter3 p = fromCoCh ∘ filterCoCh 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
mapCh :: (a → b) → TreeCh a → TreeCh b
mapCh f (TreeCh g) = TreeCh (λa → g (a ∘ m f))
map2 :: (a → b) → Tree a → Tree b
map2 f = fromCh ∘ mapCh f ∘ toCh
 {-# INLINE map2 #-}
mapCoCh :: (a → b) → TreeCoCh a → TreeCoCh b
mapCoCh f (TreeCoCh h s) = TreeCoCh (m f ∘ h) s
map3 :: (a → b) → Tree a → Tree b
map3 f = fromCoCh ∘ mapCoCh f ∘ toCoCh
 {-# INLINE map3 #-}
```

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

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

The church encoded version leverages an alagebra `s`:

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

The cochurch encoding is defined using a coinduction principle. Note that it is possible to implement this function using an accumulator of a list datatype (used like a queue), but it currently does not seem to provide a fused Core AST, for a more expansive discussion on tail-recursive cochurch-encoded pipelines, see 4.2.1:

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

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

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

sumApp1 (x, y) = sum1 (append1 (between1 (x, y)) (between1 (x, y)))
sumApp2 (x, y) = sum2 (append2 (between2 (x, y)) (between2 (x, y)))
sumApp3 (x, y) = sum3 (append3 (between3 (x, y)) (between3 (x, y)))

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

4.1.2 Lists

In this section further replication of Harper (2011)'s work is described, but instead of implementing Leaf trees, Lists are implemented.

This was done to see how the descriptions in Harper (2011)'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:

```
data List′ a = Nil | Cons a (List′ a)
data List_ a b = Nil_ | Cons_ a b
```

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

```
data ListCh a = ListCh (∀ b ∘ (List_ a b → b) → b)
toCh :: List′ a → ListCh a
toCh t = ListCh (λa → fold a t)
fold :: (List_ a b → b) → List′ a → b
fold a Nil = a Nil_
fold a (Cons x xs) = a (Cons_ x (fold a xs))
fromCh :: ListCh a → List′ a
fromCh (ListCh fold′) = fold′ in′
in′ :: List_ a (List′ a) → List′ a
in′ Nil_ = Nil
in′ (Cons_ x xs) = Cons x xs
 {-# RULES "toCh/fromCh fusion" forall x. toCh (fromCh x) = x #-}

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

The cochurch encodings are defined similarly:

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

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

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

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

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

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

For the Cochurch-encoded version we define a coalgebra:

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

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

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

The church-encoded version does **not** leverage an algebra, as is normally done for natural transformations, but instead something else. I.e. the function `a` below is only selectively applied to the resultant subterms (see the `else` case specifically):

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

For the cochurch-encoding, a natural transformation can be defined, but it is not a simple algebra, instead it is a recursive function. There is existing work, called joint-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:

```
filt p h s = go s
   where go s = case h s of
            Nil_ → Nil_
```

```
          Cons_ x xs → if p x then Cons_ x xs else go xs
filterCoCh :: (a → Bool) → ListCoCh a → ListCoCh a
filterCoCh p (ListCoCh h s) = ListCoCh (filt p h) s
filter3 :: (a → Bool) → List′ a → List′ a
filter3 p = fromCoCh ∘ filterCoCh p ∘ toCoCh
 {-# INLINE filter3 #-}
```

It is possible to implement filter using a natural transformation, but this requires us to modify the type of the base functor, so we can communicate 'skip' to the datatype, which our corecursion principle can handle accordingly. This technique is called *stream fusion* and is described by Coutts et al. (2007).

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

```
map1 :: (a → b) → List′ a → List′ b
map1 _ Nil = Nil
map1 f (Cons x xs) = Cons (f x) (map1 f xs)
 {-# INLINE map1 #-}
m :: (a → b) → List_ a c → List_ b c
m f (Cons_ x xs) = Cons_ (f x) xs
m _ Nil_ = Nil_
mapCh :: (a → b) → ListCh a → ListCh b
mapCh f (ListCh g) = ListCh (λa → g (a ∘ m f))
map2 :: (a → b) → List′ a → List′ b
map2 f = fromCh ∘ mapCh f ∘ toCh
 {-# INLINE map2 #-}
m′ :: (a → b) → List_ a c → List_ b c
m′ f (Cons_ x xs) = Cons_ (f x) xs
m′ _ (Nil_) = Nil_
mapCoCh :: (a → b) → ListCoCh a → ListCoCh b
mapCoCh f (ListCoCh h s) = ListCoCh (m′ f ∘ h) s
map3 :: (a → b) → List′ a → List′ b
map3 f = fromCoCh ∘ mapCoCh f ∘ toCoCh
 {-# INLINE map3 #-}
```

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

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

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

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


 {- TAIL RECURSION!!! -}
 {-# INLINE sum3 #-}
```

Note that two subfunctions are provided to `su'`, the `loop` and the `loopt` function. The former function is implement as one would naively expect. The latter, interestingly, is implemented using tail-recursion. Because this `loopt` function constitutes a corecursion principle, all the algebras (or natural transformations) applied to it, will be inlined in such a way that the resultant function is also tail recursive, in some cases providing a significant speedup! For more details, see the discussion in Section 4.2.1.

Pipelines and GHC list fusion

```
trodd :: Int → Bool
trodd n = n ‘rem‘ 2 ≡ 0
 {-# INLINE trodd #-}


pipeline1 = sum1 ∘ map1 (+2) ∘ filter1 trodd ∘ between1
pipeline2 = sum2 ∘ map2 (+2) ∘ filter2 trodd ∘ between2
pipeline3 = sum3 ∘ map3 (+2) ∘ filter3 trodd ∘ between3
pipeline4 (x, y) = loop x y 0
  where loop z y sum = case z > y of
                         True → sum
                         False → if trodd z
                                   then loop (z + 1) y (sum + z + 2)
                                   else loop (z + 1) y sum

    -- Time to implement these function use Haskell list for performance comparison


between5 :: (Int, Int) → [Int]
between5 (x, y) = [x . . y]
 {-# INLINE between5 #-}
filter5 :: (Int → Bool) → [Int] → [Int]
filter5 f xs = build (λc n → foldr (λa b → if f a then c a b else b) n xs)
 {-# INLINE filter5 #-}
map5 :: ∀ a b ∘ (a → b) → [a] → [b]
map5 f xs = build (λc n → foldr (λa b → c (f a) b) n xs)
 {-# INLINE map5 #-}
sum5 :: [Int] → Int
sum5 = foldl′ (λa b → a + b) 0
 {-# INLINE sum5 #-}
pipeline5 = sum5 ∘ map5 (+2) ∘ filter5 trodd ∘ between5

    -- sumApp1 (x, y) = sum1 (append1 (between1 (x, y)) (between1 (x, y)))
    -- sumApp2 (x, y) = sum2 (append2 (between2 (x, y)) (between2 (x, y)))
    -- sumApp3 (x, y) = sum3 (append3 (between3 (x, y)) (between3 (x, y)))

input :: (Int, Int)
input = (1, 10000)
main :: IO ()
main = print (pipeline5 input)
```

4.2 Discussion of code

4.2.1 Limitations of Church-fusion and perks of stream-fusion

Lack of explanation on Harper (2011)'s part about choice of leaf trees, how in lists, it is not possible to implement filter as a natural transformation.

4.2.2 The strength of cochurch encodings: tail recursion

4.2.3 Join point optimization

5 Conclusion and Future Work

5.1 Future Work

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

References

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

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

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

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