Master's Thesis

Eben Rogers

April 10, 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):

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

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 function programming so attractive, but it comes at the cost of computational overhead. 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, traverser 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, one could arrive at the following solution:

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

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, 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:

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

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`:

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

Such that:

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

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:

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

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

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

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

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

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

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

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

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

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

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

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

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 $\texttt{hom}_{\mathcal{C}}(X, Y)$, which is called a *hom-set*.
3. For each object $X \in \mathcal{C}_0$, a morphism $\texttt{Id}_X \in \texttt{hom}_{\mathcal{C}}(X, X)$, called the *identity morphism* on X.
4. A binary operation: $(\circ)_{X,Y,Z} : \texttt{hom}_{\mathcal{C}}(Y, Z) \to \texttt{hom}_{\mathcal{C}}(X, Y) \to \texttt{hom}_{\mathcal{C}}(X, Z)$, 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 $B \in \mathcal{C}_0$:
$$\forall A, B \in \mathcal{C}_0 : \exists! \texttt{hom}_{\mathcal{C}}(A, B) \Longrightarrow \textbf{initial}(A)$$

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

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 $\mathcal{D}$:

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

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. $F : \mathcal{C} \to \mathcal{C}$

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

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

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

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 $\mathtt{Id}_C$ 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 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 $(\!|\phi|\!) \in \mathtt{hom}_{\mathcal{C}}(\mu^F, C)$ 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 $f \in \mathtt{hom}_{\mathcal{C}}(C, D)$ we have a **fusion property**:

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

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

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

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-propᵣ : {F : Container 0ℓ 0ℓ}{X : Set}(a : ⟦ F ⟧ X → X)(h : μ F → X) →
                  h ≡ ⦇ a ⦈ → h ∘ in' ≡ a ∘ map h
universal-propᵣ a h eq = begin
      h ∘ in'
    ≡⟨ cong (_∘ in') eq ⟩
      ⦇ a ⦈ ∘ in'
    ≡⟨⟩
      a ∘ map ⦇ a ⦈
```

```
  ≡⟨ cong (λ x → a ∘ map x) (sym eq) ⟩
    a ∘ map h
  □
```

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
open import Categories.Category renaming (Category to Cat)
open import Data.Container using (Container; map) renaming (⟦_⟧ to I⟦_⟧)
```

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 → I⟦ F ⟧ X)(Y : Y → I⟦ 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 : I⟦ F ⟧ (ν F)
open ν
⟦_⟧ : {F : Container 0ℓ 0ℓ}{X : Set} → (X → I⟦ F ⟧ X) → X → ν F
out (⟦ c ⟧ x) = (λ (op , ar) → op , ⟦ 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 → I⟦ F ⟧ X) → F CoAlghom[ a , out ]
valid-fcoalghom {X} a = record { f = ⟦ 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. SOMETHING ABOUT TERMINATION CHECKING.

```
{-# NON_TERMINATING #-}
isunique : {F : Container 0ℓ 0ℓ}{X : Set}{c : X → I⟦ F ⟧ X}(fhom : F CoAlghom[ c , out ])
           (x : X) → ⟦ c ⟧ x ≡ fhom .f x
isunique {_}{_}{c} fhom x = out-injective (begin
          (out ∘ ⟦ c ⟧) x
  ≡⟨⟩ -- Definition of ⟦_⟧
          map ⟦ c ⟧ (c x)
  ≡⟨⟩
          (λ(op , ar) → (op , ⟦ 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
open import Data.Container using (Container; map) renaming (⟦_⟧ to I⟦_⟧)
```

The categorical fusion property:

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

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

```
{-# OPTIONS --guardedness #-}
module agda.term.terminal where
open import Data.Container using (Container; map) renaming (⟦_⟧ to I⟦_⟧)
```

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

```
universal-propᵣ : {F : Container 0ℓ 0ℓ}{C : Set}(c : C → I⟦ F ⟧ C)(h : C → ν F) →
                  h ≡ ⟦ c ⟧ → out ∘ h ≡ map h ∘ c
universal-propᵣ c h eq = begin
    out ∘ h
  ≡⟨ cong (_∘_ out) eq ⟩
    out ∘ ⟦ c ⟧
  ≡⟨⟩
    map ⟦ c ⟧ ∘ c
  ≡⟨ cong (_∘ c) (cong map (sym eq)) ⟩
    map h ∘ c
  □
```

The *computation law* Harper (2011):

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

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

```
{-# NON_TERMINATING #-}
reflection : {F : Container 0ℓ 0ℓ}(x : ν F) → ⟦ out ⟧ x ≡ x
reflection x = out-injective (begin
    out (⟦ out ⟧ x)
  ≡⟨⟩
    map ⟦ out ⟧ (out x)
  ≡⟨⟩
    op , ⟦ 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 encoded producing, transformation, and consuming function, alongside proofs that they are equal to the functions they are encoding:

```
-- Generalized producer and consuming functions.
prodCh : {F : Container _ _}{X : Set}(g : {Y : Set} → (⟦ F ⟧ Y → Y) → X → Y)(x : X) → Church F
prodCh g x = Ch (λ a → g a x)
eqprod : {F : Container _ _}{X : Set}{g : {Y : Set} → (⟦ F ⟧ Y → Y) → X → Y} →
         fromCh ∘ prodCh g ≡ g in'
eqprod = refl
transCh : {F G : Container _ _}(nat : {X : Set} → ⟦ F ⟧ X → ⟦ G ⟧ X) → Church F → Church G
transCh n (Ch g) = Ch (λ a → g (a ∘ n))
eqtrans : {F G : Container _ _}{nat : {X : Set} → ⟦ F ⟧ X → ⟦ G ⟧ X} →
          fromCh ∘ transCh nat ∘ toCh ≡ ⦇ in' ∘ nat ⦈
eqtrans = refl
consCh : {F : Container _ _}{X : Set} → (c : (⟦ F ⟧ X → X)) → Church F → X
consCh c (Ch g) = g c
eqcons : {F : Container _ _}{X : Set}{c : (⟦ F ⟧ X → X)} →
         consCh c ∘ toCh ≡ ⦇ c ⦈
eqcons = refl
```

Proof obligations In **?**'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 theo ther direction `toCh ∘ fromCh = id`. This proofs leverages parametricity as described by Wadler (1989). It postulates the free theorem of the function `g`, 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 freetheorem-initial : {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 = freetheorem-initial ⦇ a ⦈ g refl
to-from-id : {F : Container 0ℓ 0ℓ}{g : {X : Set} → (⟦ F ⟧ X → X) → X} →
             toCh (fromCh (Ch g)) ≡ Ch g
to-from-id {F}{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
  ∎
to-from-id' : {F : Container 0ℓ 0ℓ} → toCh ∘ fromCh ≡ id
to-from-id' {F} = funext (λ where (Ch g) → to-from-id {F}{g})
```

The third proof shows that encoding functions constitute an implementation for the consumer functions being replaced:

```
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 producing functions constitute an implementation for the producing functions being replaced:

```
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
  ∎

-- This last proofs could all use a rewrite, now that I've generalized the three different types
-- PAGE 51 - Proof 5
-- New function constitutes an implementation for the transformation function being replaced
chTrans : {F G : Container 0ℓ 0ℓ}(f : {X : Set} → ⟦ F ⟧ X → ⟦ G ⟧ X) → Church F → Church G
chTrans f (Ch g) = Ch (λ a → g (a ∘ f))
trans-pred : {F G : Container 0ℓ 0ℓ}(g : {X : Set} → (⟦ F ⟧ X → X) → X )
             (f : {X : Set} → ⟦ F ⟧ X → ⟦ G ⟧ X) →
             fromCh (chTrans f (Ch g)) ≡ ⦇ in' ∘ f ⦈ (fromCh (Ch g))
trans-pred g f = begin
    fromCh (chTrans 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))
  □


module agda.church.inst.list where
open import Data.Container using (Container; ⟦_⟧; μ; map; _▷_)
open import Data.W renaming (sup to in')
open import Level hiding (zero; suc)
open import Data.Product hiding (map)
open import Data.Nat
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

data ListOp (A : Set) : Set where
  nil : ListOp A
  cons : A → ListOp A

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


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

[] : {A : Set} → μ (F A)
[] = in' (nil , λ())


_::_ : {A : Set} → A → List A → List A
_::_ x xs = in' (cons x , λ tt → xs)
infixr 20 _::_


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

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)
map1 : {A B : Set}(f : A → B) → List A → List B
```

```
map1 f = ⦇ in' ∘ m f ⦈
mapCh : {A B : Set}(f : A → B) → Church (F A) → Church (F B)
mapCh f (Ch g) = Ch (λ a → g (a ∘ m f))
map2 : {A B : Set}(f : A → B) → List A → List B
map2 f = fromCh ∘ mapCh f ∘ toCh


l1 : μ (F N)
l1 = 5 :: 8 :: []
l2 : μ (F N)
l2 = 3 :: 6 :: []
proof : (map1 (_+_ 2) l2) ≡ l1
proof = refl


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

sum1 : List N → N
sum1 = ⦇ su ⦈
sumCh : Church (F N) → N
sumCh (Ch g) = g su
sum2 : List N → N
sum2 = sumCh ∘ toCh

sumworks : sum1 (5 :: 6 :: 7 :: []) ≡ 18
sumworks = refl


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 , λ tt → (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))

between1 : N × N → List N
between1 xy = b in' xy
betweenCh : N × N → Church (F N)
betweenCh xy = Ch (λ a → b a xy)
between2 : N × N → List N
between2 = fromCh ∘ betweenCh


check : 2 :: 3 :: 4 :: 5 :: 6 :: [] ≡ between2 (2 , 6)
check = refl

eq1 : {xy : N × N}{f : N → N} → (sum2 ∘ map2 f ∘ between2) ≡ (sumCh ∘ mapCh f ∘ betweenCh)
eq1 {xy}{f} = begin
    sumCh ∘ toCh ∘ fromCh ∘ mapCh f ∘ toCh ∘ fromCh ∘ betweenCh
  ≡⟨ cong (λ g → sumCh ∘ g ∘ mapCh f ∘ g ∘ betweenCh) to-from-id' ⟩
    sumCh ∘ mapCh f ∘ betweenCh
  □

eq2 : {xy : N × N}{f : N → N} → (sumCh ∘ mapCh f) (betweenCh xy) ≡ (sum1 ∘ map1 f) (between1 xy)
eq2 {xy}{f} = begin
    (sumCh ∘ mapCh f) (betweenCh xy)
  ≡⟨⟩
    (sumCh (Ch (λ a → b (a ∘ m f) xy)))
  ≡⟨⟩
    b (su ∘ m f) xy
```

```
  ≡⟨⟩
    unCh su (Ch (λ a → b (a ∘ m f) xy))
  ≡⟨ cong (unCh su) (sym $ cong-app to-from-id' (Ch (λ a → b (a ∘ m f) xy))) ⟩
    unCh su (toCh (fromCh (Ch (λ a → b (a ∘ m f) xy))))
  ≡⟨ cong-app (cons-pres su) (fromCh (Ch (λ a → b (a ∘ m f) xy))) ⟩
    ⦇ su ⦈ (fromCh (Ch (λ a → b (a ∘ m f) xy)))
  ≡⟨ cong ⦇ su ⦈ (trans-pred (flip b xy) (m f)) ⟩
    ⦇ su ⦈ (⦇ in' ∘ m f ⦈ (fromCh (Ch (λ a → b a xy))))
  ≡⟨ cong (⦇ su ⦈ ∘ ⦇ in' ∘ m f ⦈) (prod-pres b xy) ⟩
    (⦇ su ⦈ ∘ ⦇ in' ∘ m f ⦈) (b in' xy)
  ≡⟨⟩
    (sum1 ∘ map1 f) (between1 xy)
  □

-- Proofs for each of the above functions
eqsum : sum1 ≡ sum2
eqsum = refl
eqmap : {f : N → N} → map1 f ≡ map2 f
eqmap = refl
eqbetween : between1 ≡ between2
eqbetween = refl


-- Generalization of the above proofs for any container
-- MOVED TO DEFS.


transfuse : {F G H : Container 0ℓ 0ℓ}(nat1 : {X : Set} → ⟦ F ⟧ X → ⟦ G ⟧ X) →
            (nat2 : {X : Set} → ⟦ G ⟧ X → ⟦ H ⟧ X) →
            transCh nat2 ∘ toCh ∘ fromCh ∘ transCh nat1 ≡ transCh (nat2 ∘ nat1)
transfuse nat1 nat2 = begin
            transCh nat2 ∘ toCh ∘ fromCh ∘ transCh nat1
          ≡⟨ cong (λ f → transCh nat2 ∘ f ∘ transCh nat1) to-from-id' ⟩
            transCh nat2 ∘ transCh nat1
          ≡⟨ funext (λ where (Ch g) → refl) ⟩
            transCh (nat2 ∘ nat1)
          □
pipfuse : {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)} →
          consCh c ∘ transCh nat ∘ prodCh g ≡ g (c ∘ nat)
pipfuse = refl

-- Using the generalizations, we now get our encoding proofs and shortcut fusion for free :)
between3 : N × N → List N
between3 = fromCh ∘ prodCh b
map3 : {A B : Set}(f : A → B) → List A → List B
map3 f = fromCh ∘ transCh (m f) ∘ toCh
sum3 : List N → N
sum3 = consCh su ∘ toCh




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
```

3.2.2 Cochurch encodings

```
{-# OPTIONS --guardedness #-}
open import agda.term.termcoalg
open ν
open import Data.Product
open import Level
open import Function
open import Relation.Binary.PropositionalEquality as Eq
open ≡-Reasoning
module agda.cochurch.defs where
open import Data.Container using (Container) renaming (⟦_⟧ to I⟦_⟧)

data CoChurch (F : Container 0ℓ 0ℓ) : Set₁ where
  CoCh : {X : Set} → (X → I⟦ F ⟧ X) → X → CoChurch F
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) = ⟦ h ⟧ x


prodCoCh : {F : Container 0ℓ 0ℓ}{Y : Set} → (g : Y → I⟦ F ⟧ Y) → Y → CoChurch F
prodCoCh g x = CoCh g x
eqprod : {F : Container 0ℓ 0ℓ}{Y : Set}{g : (Y → I⟦ F ⟧ Y)} →
         fromCoCh ∘ prodCoCh g ≡ ⟦ g ⟧
eqprod = refl
transCoCh : {F G : Container 0ℓ 0ℓ}(nat : {X : Set} → I⟦ F ⟧ X → I⟦ G ⟧ X) → CoChurch F → CoChurch G
transCoCh n (CoCh h s) = CoCh (n ∘ h) s
eqtrans : {F G : Container 0ℓ 0ℓ}{nat : {X : Set} → I⟦ F ⟧ X → I⟦ G ⟧ X} →
          fromCoCh ∘ transCoCh nat ∘ toCoCh ≡ ⟦ nat ∘ out ⟧
eqtrans = refl
consCoCh : {F : Container 0ℓ 0ℓ}{Y : Set} → (c : {S : Set} → (S → I⟦ F ⟧ S) → S → Y) → CoChurch F → Y
consCoCh c (CoCh h s) = c h s
eqcons : {F : Container 0ℓ 0ℓ}{X : Set}{c : {S : Set} → (S → I⟦ F ⟧ S) → S → X} →
         consCoCh c ∘ toCoCh ≡ c out
eqcons = refl


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


{-# OPTIONS --guardedness #-}
open import Data.Container using (Container; map) renaming (⟦_⟧ to I⟦_⟧)
open import Level
module agda.cochurch.proofs where
open import Function.Base using (id; _∘_; flip; _$_)
open import Relation.Binary.PropositionalEquality as Eq
```

```
open ≡-Reasoning
open import Data.Product using (_,_)
open import agda.term.termcoalg
open ν
open import agda.term.terminal
open import agda.term.cofusion
open import agda.funct.funext
open import agda.cochurch.defs

-- PAGE 52 - Proof 1
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
    ⟦ out ⟧ x
  ≡⟨ reflection x ⟩
    x
  ≡⟨⟩
    id x
  ∎)

-- PAGE 52 - Proof 2
postulate freetheorem-terminal : {F : Container 0ℓ 0ℓ}
                                 {C D : Set}{Y : Set₁}{c : C → I⟦ F ⟧ C}{d : D → I⟦ F ⟧ D}
                                 (h : C → D)(f : {X : Set} → (X → I⟦ 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.
to-from-id : {F : Container 0ℓ 0ℓ}{X : Set}(c : X → I⟦ F ⟧ X)(x : X) →
             toCoCh (fromCoCh (CoCh c x)) ≡ CoCh c x
to-from-id c x = begin
    toCoCh (fromCoCh (CoCh c x))
  ≡⟨⟩ -- definition of fromCh
    toCoCh (⟦ c ⟧ x)
  ≡⟨⟩ -- definition of toCh
    CoCh out (⟦ c ⟧ x)
  ≡⟨⟩ -- composition
    (CoCh out ∘ ⟦ c ⟧) x
  ≡⟨ flip cong-app x ∘ sym $ freetheorem-terminal ⟦ c ⟧ CoCh refl ⟩ -- I made some use of this: https://www-
    CoCh c x
  ∎

to-from-id' : {F : Container 0ℓ 0ℓ} → toCoCh ∘ fromCoCh ≡ id
to-from-id' {F} = funext (λ where (CoCh c x) → to-from-id {F} c x)

-- PAGE 52 - Proof 3
-- New function constitutes an implementation for the produces function being replaced
prod-pres : {F : Container 0ℓ 0ℓ}{X : Set} (c : X → I⟦ F ⟧ X) (x : X) →
            fromCoCh ((λ s → CoCh c s) x) ≡ ⟦ c ⟧ x
prod-pres c x = begin
    fromCoCh ((λ s → CoCh c s) x)
  ≡⟨⟩ -- function application
    fromCoCh (CoCh c x)
  ≡⟨⟩ -- definition of toCh
    ⟦ c ⟧ x
  ∎

-- PAGE 52 - Proof 4
-- New function constitutes an implementation for the produces function being replaced
```

```
unCoCh : {F : Container 0ℓ 0ℓ}(f : {Y : Set} → (Y → I⟦ F ⟧ Y) → Y → ν F) (c : CoChurch F) → ν F
unCoCh f (CoCh c s) = f c s
cons-pres : {F : Container 0ℓ 0ℓ}{X : Set} → (f : {Y : Set} → (Y → I⟦ F ⟧ Y) → Y → ν F) → (x : ν F) →
            unCoCh f (toCoCh x) ≡ f out x
cons-pres f x = begin
    unCoCh f (toCoCh x)
  ≡⟨⟩ -- definition of toCoCh
    unCoCh f (CoCh out x)
  ≡⟨⟩ -- function application
    f out x
  ∎

-- PAGE 52 - Proof 5
-- New function constitutes an implementation for the transformation function being replaced
--(nat f)
record nat {F G : Container 0ℓ 0ℓ}(f : {X : Set} → I⟦ F ⟧ X → I⟦ G ⟧ X): Set₁ where
  field
    coherence : {A B : Set}(h : A → B) → map h ∘ f ≡ f ∘ map h
open nat { ... }

valid-hom : {F G : Container 0ℓ 0ℓ}{X : Set}(h : X → I⟦ F ⟧ X)(f : {X : Set} → I⟦ F ⟧ X → I⟦ G ⟧ X){ _ : nat
            map ⟦ h ⟧ ∘ f ∘ h ≡ f ∘ out ∘ ⟦ h ⟧
valid-hom h f = begin
    (map ⟦ h ⟧ ∘ f) ∘ h
  ≡⟨ cong (_∘ h) (coherence ⟦ h ⟧) ⟩
    (f ∘ map ⟦ h ⟧) ∘ h
  ≡⟨⟩
    f ∘ out ∘ ⟦ h ⟧
  ∎

chTrans : {F G : Container 0ℓ 0ℓ}(f : {X : Set} → I⟦ F ⟧ X → I⟦ G ⟧ X) → CoChurch F → CoChurch G
chTrans f (CoCh c s) = CoCh (f ∘ c) s
trans-pred : {F G : Container 0ℓ 0ℓ}{X : Set} (h : X → I⟦ F ⟧ X) (f : {X : Set} → I⟦ F ⟧ X → I⟦ G ⟧ X)(x : X)
             fromCoCh (chTrans f (CoCh h x)) ≡ (⟦ f ∘ out ⟧ ∘ ⟦ h ⟧) x
trans-pred h f x = begin
    fromCoCh (chTrans f (CoCh h x))
  ≡⟨⟩ -- Function application
    fromCoCh (CoCh (f ∘ h) x)
  ≡⟨⟩ -- Definition of fromCh
    ⟦ f ∘ h ⟧ x
  ≡⟨ flip cong-app x $ fusion ⟦ h ⟧ (sym (valid-hom h f)) ⟩
    (⟦ f ∘ out ⟧ ∘ ⟦ h ⟧) x
  ∎


{-# OPTIONS --guardedness #-}
module agda.cochurch.inst.list where
open import agda.cochurch.defs
open import agda.cochurch.proofs
open import Data.Container using (Container; map; _▷_) renaming (⟦_⟧ to I⟦_⟧)
open import Level hiding (suc)
open import Data.Empty
open import Data.Unit
open import agda.term.termcoalg
open ν
open import Data.Product
open import Data.Sum
open import Function
open import Data.Nat
open import Agda.Builtin.Nat
```

```
open import Relation.Binary.PropositionalEquality as Eq
open ≡-Reasoning
open import agda.funct.funext

data ListOp (A : Set) : Set where
  nil : ListOp A
  cons : A → ListOp A

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


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

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


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 , λ tt → x')
unfold' : {F : Container 0ℓ 0ℓ}{A X : Set}(f : X → ⊤ ⊎ (A × X)) → X → List A
unfold' {A}{X} f = ⟦ mapping f ⟧

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)
map1 : {A B : Set}(f : A → B) → List A → List B
map1 f = ⟦ m f ∘ out ⟧
mapCoCh : {A B : Set}(f : A → B) → CoChurch (F A) → CoChurch (F B)
mapCoCh f (CoCh h s) = CoCh (m f ∘ h) s
map2 : {A B : Set}(f : A → B) → List A → List B
map2 f = fromCoCh ∘ mapCoCh f ∘ toCoCh

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

sum1 : List N → N
sum1 = su' out
sumCoCh : CoChurch (F N) → N
sumCoCh (CoCh h s) = su' h s
sum2 : List N → N
sum2 = sumCoCh ∘ toCoCh
--s2works : sum2 (1 :: 2 :: 3 :: []) ≡ 6
--s2works = refl

b' : N × N → List' N (N × N)
```

```
b' (x , zero) = (nil , λ())
b' (x , suc n) = (cons x , λ tt → (suc x , n))

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

between1 : N × N → List N
between1 xy = ⟦ b ⟧ xy
betweenCoCh : (N × N → List' N (N × N)) → (N × N) → CoChurch (F N)
betweenCoCh b = CoCh b
between2 : N × N → List N
between2 = fromCoCh ∘ CoCh b

-- Proofs for each of the above functions
eqsum : sum1 ≡ sum2
eqsum = refl
eqmap : {f : N → N} → map1 f ≡ map2 f
eqmap = refl
eqbetween : between1 ≡ between2
eqbetween = refl


-- Generalization of the above proofs for any container
-- MOVED TO DEFS


transfuse : {F G H : Container 0ℓ 0ℓ}(nat1 : {X : Set} → I⟦ F ⟧ X → I⟦ G ⟧ X) →
            (nat2 : {X : Set} → I⟦ G ⟧ X → I⟦ H ⟧ X) →
            transCoCh nat2 ∘ toCoCh ∘ fromCoCh ∘ transCoCh nat1 ≡ transCoCh (nat2 ∘ nat1)
transfuse nat1 nat2 = begin
            transCoCh nat2 ∘ toCoCh ∘ fromCoCh ∘ transCoCh nat1
          ≡⟨ cong (λ f → transCoCh nat2 ∘ f ∘ transCoCh nat1) to-from-id' ⟩
            transCoCh nat2 ∘ transCoCh nat1
          ≡⟨ funext (λ where (CoCh h s) → refl) ⟩
            transCoCh (nat2 ∘ nat1)
          □
pipfuse : {F G : Container 0ℓ 0ℓ}{Y : Set}{g : Y → I⟦ F ⟧ Y}
          {nat : {X : Set} → I⟦ F ⟧ X → I⟦ G ⟧ X}{c : {S : Set} → (S → I⟦ G ⟧ S) → S → Y} →
          consCoCh c ∘ transCoCh nat ∘ prodCoCh g ≡ c (nat ∘ g)
pipfuse = refl

---- Using the generalizations, we now get our encoding proofs and shortcut fusion for free :)
between3 : N × N → List N
between3 = fromCoCh ∘ prodCoCh b
map3 : {A B : Set}(f : A → B) → List A → List B
map3 f = fromCoCh ∘ transCoCh (m f) ∘ toCoCh
sum3 : List N → N
sum3 = consCoCh su' ∘ toCoCh
fused : {f : N → N} → sum3 ∘ map3 f ∘ between3 ≡ su' (m f ∘ b)
fused {f} = begin
    consCoCh su' ∘ toCoCh ∘ fromCoCh ∘ transCoCh (m f) ∘ toCoCh ∘ fromCoCh ∘ prodCoCh b
  ≡⟨ cong (λ g → consCoCh su' ∘ g ∘ transCoCh (m f) ∘ g ∘ prodCoCh b) to-from-id' ⟩
    consCoCh su' ∘ transCoCh (m f) ∘ prodCoCh b
  ≡⟨⟩
    su' (m f ∘ 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.

4.1 Church encodings

4.2 Cochurch encodings

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

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

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