Proving the Unique Fixed-Point Principle Correct

An Adventure with Category Theory

Ralf Hinze Daniel W. H. James

Department of Computer Science, University of Oxford, Wolfson Building, Parks Road, Oxford, OX1 3QD, England

{ralf.hinze, daniel.james}@cs.ox.ac.uk

Abstract

Say you want to prove something about an infinite data-structure, such as a stream or an infinite tree, but you would rather not subject yourself to coinduction. The unique fixed-point principle is an easy-to-use, calculational alternative. The proof technique rests on the fact that certain recursion equations have unique solutions; if two elements of a coinductive type satisfy the same equation of this kind, then they are equal. In this paper we precisely characterize the conditions that guarantee a unique solution. Significantly, we do so not with a syntactic criterion, but with a semantic one that stems from the categorical notion of naturality. Our development is based on distributive laws and bialgebras, and draws heavily on Turi and Plotkin's pioneering work on mathematical operational semantics. Along the way, we break down the design space in two dimensions, leading to a total of nine points. Each gives rise to varying degrees of expressiveness, and we will discuss three in depth. Furthermore, our development is generic in the syntax of equations and in the behaviour they encode—we are not caged in the world of streams.

Categories and Subject Descriptors D.2.4 [*Software/ Program Verification*]: correctness proofs, formal methods; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—specification techniques

General Terms Languages, Theory, Verification

Keywords unique fixed-points, bialgebras, distributive laws

1. Introduction

"Whence cometh this?" Our aim is to provide an elegant proof of correctness for an elegant proof principle. Elegance comes, in large part, through simplicity, and specifically we value the simplicity afforded by the notion of naturality and initial/final algebra/coalgebra semantics. The key component for correctness of the unique fixed-point principle is a sound characterization of what gives a recursion equation a unique solution.

"Why does uniqueness matter?" Uniqueness has two complementary perspectives: programs and proofs. When read as a program, the unique solution implies that it is well-defined. When the unicity is utilized in a proof, we are able to show that two given solutions are equal—the unique fixed-point principle (UFP).

ICFP'11 September 19–21, 2011, Tokyo, Japan.

Reprinted from ICFP'11, Proceedings of the ACM SIGPLAN International Conference on Functional Programming, September 19–21, 2011, Tokyo, Japan., pp. 359–371.

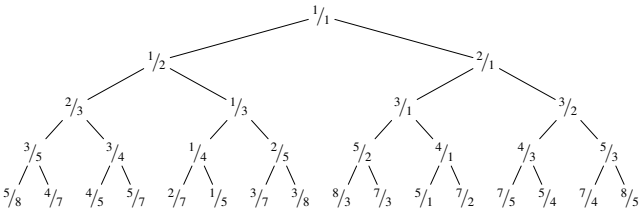


Figure 1. The Bird tree

"Why not a syntactic criterion?" In prior work [10] a simple syntactic criterion, specific to stream equations, was proffered, but this is unsatisfactory. A criterion must exclude all bad things and accept as many good things as possible. As criteria are complicated to accept more good things, so is the understanding and trust. Their syntactic nature makes them intrinsically fragile. Just as it is often easy to satisfy a criterion by a program transformation, it is just as easy to lose the satisfaction.

"Category complex" Our theoretical underpinnings, distributive laws and bialgebras, draw on Turi and Plotkin's mathematical operational semantics [21], and just as in theirs, the following pages contain plenty of category theory. However, our concern throughout is making the theory accessible. We do so by grounding it in an application, and targeting the categorical parlance to readers who are familiar with the Algebra of Programming [5]. This is the holy trinity of categories, functors and natural transformations, along with algebras and coalgebras, which we will summarize. For the categorically enlightened, we will note the more direct reasoning.

"Interpretation of categorical structures" We will introduce a selection of categorical structures and discuss their interpretation in this domain. It is well known that free monads can been seen as terms with variables and cofree comonads as labelled trees, but we will see how these and two other structures influence the expressivity of recursion equations, by the 'language features' they induce. These features are positioned in two dimensions. We will focus on just one, and we give a fuller account in the extended paper [11]. As a small exception, we will consider what Niqui and Rutten [16] calls *sampling* functions, and what we would more loosely describe as stream operators that consume more than they produce.

"Beauty and Elegance" We hope to showcase the beauty of the unique fixed-point principle. Its elegance is due, in no small part, to its calculational style, a style that proofs will follow throughout.

2. The Unique Fixed-Point Principle

2.1 Infinite Trees

In Figure 1 we can see the first five levels of the *Bird Tree* [9], an infinite tree in which you can find every positive rational number exactly once. It has several remarkable properties that come from

its nature as a *fractal* object—its subtrees are similar to the whole tree. The tree can be transformed into its left subtree by incrementing and then taking the reciprocal of every element; the right subtree is obtained by swapping these operations. This description can be nicely captured by a *corecursive* definition (we will use Haskell as a meta-language for the category of sets and total functions):

$$bird = Node\ 1\ (1\ /\ (bird + 1))\ ((1\ /\ bird) + 1)\ .$$

The picture suggests that taking the reciprocal of each element is the same as mirroring the tree, $mirror\,bird = 1\ /\ bird$, and this is indeed the case. We shall see that we can prove this effortlessly using the unique fixed-point principle.

Before we get to the proof, we must introduce some definitions.

$$\textbf{data}\ \mathsf{Tree}\ x = Node\ \{root :: x, left :: \mathsf{Tree}\ x, right :: \mathsf{Tree}\ x\}$$

The type $\mathsf{Tree}\,x$ is a so-called *coinductive datatype*. Its definition is similar to the textbook definition of binary trees, except that there are no leaves, so we cannot build a finite tree. And without leaves, *mirror* is a one-liner:

$$mirror\ (Node\ x\ l\ r) = Node\ x\ (mirror\ r)\ (mirror\ l)\ .$$

The definition of *bird* uses $+$ and $/$ lifted to trees. We obtain these liftings for free as Tree is a so-called *idiom* [14]:

$$\begin{array}{l}\textbf{class}\ Idiom\ f\ \textbf{where}\\ \quad pure :: x \to f\,x\\ \quad (\diamond)\ \ :: f\,(x \to y) \to (f\,x \to f\,y)\\ \textbf{instance}\ Idiom\ \mathsf{Tree}\ \textbf{where}\\ \quad pure\ x = t\ \textbf{where}\ t = Node\ x\ t\ t\\ \quad t \diamond u\ \ = Node\ (root\ t\ \$\ root\ u)\\ \qquad\qquad\qquad (left\ t \diamond left\ u)\ (right\ t \diamond right\ u)\ .\end{array}$$

The call $pure\,x$ constructs an infinite tree of xs; idiomatic application $\diamond$ takes a tree of functions and a tree of arguments to a tree of results. Using *pure* and $\diamond$, we can lift arithmetic operations *generically* to idioms.

$$\begin{array}{l}\textbf{instance}\ (Idiom\,f, Num\,x) \Rightarrow Num\ (f\,x)\ \textbf{where}\\ \quad fromInteger\ n = pure\ (fromInteger\ n)\\ \quad negate\ u\ \ \ \ \ \ = pure\ negate \diamond u\\ \quad u + v\ \ \ \ \ \ \ \ = pure\ (+) \diamond u \diamond v\ \ \ \ \ldots\end{array}$$

Since the operations are defined pointwise, the familiar arithmetic laws also hold for trees. Mirroring a tree preserves the idiomatic structure, the function *mirror* is an *idiom homomorphism*: $mirror\,(pure\,x) = pure\,x$ and $mirror\,(x \diamond y) = mirror\,x \diamond mirror\,y$. This implies, for instance, that $mirror\,(u+v) = mirror\,u + mirror\,v$.

Let us return to the promised proof and the unique fixed-point principle. Consider the recursion equation $x = Node\,y\,l\,r$, where l and r possibly contain the variable x, but *not* the expressions $root\,x$, $left\,x$ or $right\,x$. Equations in this syntactic form possess a *unique solution*. Uniqueness can be exploited to prove that two infinite trees are equal: if they satisfy the same equation, then they are.

$$\begin{array}{ll}& mirror\,bird\\ =& \quad \{\ \text{definitions of } mirror \text{ and } bird\ \}\\ & Node\ 1\ (mirror\ ((1\ /\ bird) + 1))\ (mirror\ (1\ /\ (bird + 1)))\\ =& \quad \{\ mirror \text{ is an idiom homomorphism}\ \}\\ & Node\ 1\ ((1\ /\ mirror\,bird) + 1)\ (1\ /\ (mirror\,bird + 1))\\ \propto& \quad \{\ x = Node\ 1\ ((1\ /\ x) + 1)\ (1\ /\ (x + 1)) \text{ has a unique sol.}\ \}\\ & Node\ 1\ ((1\ /\ (1\ /\ bird)) + 1)\ (1\ /\ ((1\ /\ bird) + 1))\\ =& \quad \{\ \text{arithmetic}\ \}\\ & 1\ /\ Node\ 1\ (1\ /\ (bird + 1))\ ((1\ /\ bird) + 1)\\ =& \quad \{\ \text{definition of } bird\ \}\\ & 1\ /\ bird\end{array}$$

The link $\propto$ indicates that the proof rests on the *unique fixed-point principle*; the recursion equation is given within the curly braces. The upper part shows that *mirror bird* satisfies the equation $x = Node\ 1\ ((1\ /\ x) + 1)\ (1\ /\ (x + 1))$; the lower part establishes that $1\ /\ bird$ satisfies the same equation. The symbol $\propto$ links the two parts, effectively proving the equality of both expressions.

We mentioned that the Bird Tree contains every positive rational exactly once. A proof that exclusively builds on the unique fixed-point principle can be found in Hinze [9].

2.2 Streams

Let us consider a second coinductive type, one that will accompany us for the rest of the paper: the type of streams, infinite sequences of elements.

$$\begin{array}{l}\textbf{data}\ \mathsf{Stream}\ x = Cons\ \{head :: x, tail :: \mathsf{Stream}\ x\}\\ (\prec)\ \ :: x \to \mathsf{Stream}\ x \to \mathsf{Stream}\ x\\ x \prec s = Cons\ x\ s\end{array}$$

Like the type of infinite trees, Stream is an idiom.

$$\begin{array}{l}\textbf{instance}\ Idiom\ \mathsf{Stream}\ \textbf{where}\\ \quad pure\ x = s\ \textbf{where}\ s = x \prec s\\ \quad s \diamond t\ \ = (head\ s\ \$\ head\ t) \prec (tail\ s \diamond tail\ t)\end{array}$$

Using this vocabulary, we can define, for instance, the stream of Fibonacci numbers.

$$\begin{array}{l}fib\ \ = 0 \prec fib'\\ fib' = 1 \prec fib + fib'\end{array}$$

The Fibonacci numbers satisfy a myriad of properties. For example, if we form the stream of their partial sums and increment the result, we obtain *fib'*. Again, we shall see that the UFP allows for a concise proof. But first, we have to capture summation as a stream operator.

$$\Sigma\ s = 0 \prec s + \Sigma\ s$$

Turning to the proof of $\Sigma\,fib + 1 = fib'$, we can either show that $\Sigma\,fib + 1$ satisfies the defining equation of *fib'*, or that $fib' - 1$ satisfies the recursion equation of $\Sigma\,fib$. Both approaches work, the calculations are left as really (!) easy exercises to the reader.

A related property is the following: if we sum the Fibonacci numbers at odd positions, we obtain the Fibonacci numbers at even positions. The so-called *sampling functions* [16] *even* and *odd* enjoy simple corecursive definitions.

$$\begin{array}{lll}even\ s = head\ s \prec odd & (tail\ s)\\ odd\ \ s = & even\ (tail\ s)\end{array}$$

Turning to the proof of $\Sigma\,(odd\,fib) = even\,fib$, we reason:

$$\begin{array}{ll}& \Sigma\,(odd\,fib)\\ =& \quad \{\ \text{definition of } \Sigma\ \}\\ & 0 \prec odd\,fib + \Sigma\,(odd\,fib)\\ =& \quad \{\ \text{definition of } odd\ \}\\ & 0 \prec even\,fib' + \Sigma\,(odd\,fib)\\ \propto& \quad \{\ x = 0 \prec even\,fib' + x\ \}\\ & 0 \prec even\,fib' + even\,fib\\ =& \quad \{\ even \text{ is an idiom homomorphism and arithmetic}\ \}\\ & 0 \prec even\ (fib + fib')\\ =& \quad \{\ \text{definitions of } fib' \text{ and } odd\ \}\\ & 0 \prec odd\,fib'\\ =& \quad \{\ \text{definitions of } fib \text{ and } even\ \}\\ & even\,fib\ .\end{array}$$

This completes our short survey. The UFP is not only easy-to-use, but also surprisingly powerful: in prior work [10] we have

shown how to redevelop the theory of recurrences, finite calculus and generating functions using streams and stream operators, building solely on the cornerstone of unique solutions.

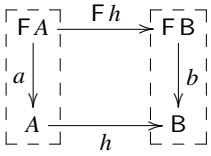
What remains to be done? We have been somewhat vague about the syntactic conditions that guarantee uniqueness. We shall see that systems of recursion equations can be classified along two dimensions, leading to a total of nine different points of interest. The system for *fib* falls into one ("consume one element, produce one"), the system for *even* into another ("consume many, but don't nest calls"). When defining streams we cannot mix styles. For instance, the equation $x = 0 \prec \mathit{even}\, x$ has infinitely many solutions. We shall see that we can capture the conditions that guarantee unicity semantically, using the categorical concept of naturality.

Furthermore, we abstract away from the type of infinite trees and streams. The development is generic both in the syntax and in the behaviour—which operations are defined and over which coinductive type. An appropriate setting is provided by the categorical notion of algebras and coalgebras which we introduce next. The resulting proofs are not only more general, they are also shorter than specific instances that have appeared in the literature [17, 19].

3. Initial Algebras and Final Coalgebras

We hope the reader has encountered the material of this section before, but we will reiterate it here as it serves as a simple demonstration of the power of *duality*. We will invoke the power to construct 'the opposite thing' time and time again.

Let $\mathsf{F} : \mathscr{C} \to \mathscr{C}$ be an endofunctor. An F-algebra is a pair $\langle A, a\rangle$ consisting of an object $A : \mathscr{C}$ and an arrow $a : \mathsf{F}\,A \to A : \mathscr{C}$. We say that A is the carrier and a is the action of the algebra; however, we often refer to the algebra simply by a as it determines the carrier. An F-homomorphism between algebras $\langle A, a\rangle$ and $\langle \mathsf{B}, b\rangle$ is an arrow $h : A \to \mathsf{B} : \mathscr{C}$ such that $h \cdot a = b \cdot \mathsf{F}\,h$.



A characteristic of functors is that they preserve identity and composition; this entails that F-homomorphisms compose and have an identity. Consequently, F-algebras and F-homomorphisms form a category, called $\mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C})$. If this category has an initial object, it is called the *initial* F-*algebra* $\langle \mu\mathsf{F}, \mathit{in}\rangle$. Initiality implies that it has a unique F-homomorphism to any F-algebra $\langle A, a\rangle$, which is written $(\!|a|\!)$ and called *fold*. It satisfies the *uniqueness property*

$$h = (\!|a|\!) \iff h \cdot \mathit{in} = a \cdot \mathsf{F}\,h \ . \tag{3.1}$$

We will now seize the opportunity to dualize these constructions to the opposite things: F-*coalgebras* and *unfolds*. An F-coalgebra is a pair $\langle C, c\rangle$ of an object $C : \mathscr{C}$ and an arrow $c : C \to \mathsf{F}\,C : \mathscr{C}$. An F-homomorphism between coalgebras $\langle C, c\rangle$ and $\langle D, d\rangle$ is an arrow $h : C \to D : \mathscr{C}$ such that $\mathsf{F}\,h \cdot c = d \cdot h$. In the same way, we can form a category $\mathsf{F}\text{-}\mathbf{Coalg}(\mathscr{C})$. If this category has a final object, it is called the *final* F-*coalgebra* $\langle \nu\mathsf{F}, \mathit{out}\rangle$. Being the final object, it has a unique F-homomorphism to it from any F-coalgebra $\langle C, c\rangle$, which is written $[\![c]\!]$ and called *unfold*. It satisfies the following *uniqueness property*.

$$h = [\![c]\!] \iff \mathsf{F}\,h \cdot c = \mathit{out} \cdot h \tag{3.2}$$

In case you were wondering, *final algebras* and *initial coalgebras* are unexciting, although we will find a use for them. The final algebra is $\langle 1, !_{\mathsf{F}\,1}\rangle$, and the initial coalgebra is $\langle 0, ¡_{\mathsf{F}\,0}\rangle$.

Let $\mathsf{F}, \mathsf{G} : \mathscr{C} \to \mathscr{C}$ be endofunctors, and $\alpha : \mathsf{F} \leftarrow \mathsf{G}$ a natural transformation. We can turn α into a functor $\alpha\text{-}\mathbf{Alg} : \mathsf{F}\text{-}\mathbf{Alg} \to \mathsf{G}\text{-}\mathbf{Alg}$ between the categories of F- and G-algebras. (We use $\dot{\leftarrow}$ to highlight the contravariance between α and $\alpha\text{-}\mathbf{Alg}$.)

$$\alpha\text{-}\mathbf{Alg}\,\langle X, a : \mathsf{F}\,X \to X\rangle = \langle X, a \cdot \alpha\,X : \mathsf{G}\,X \to X\rangle \quad \alpha\text{-}\mathbf{Alg}\,h = h$$

That $\alpha\text{-}\mathbf{Alg}$ is a functor follows from a more general construction given in Appendix B. We will see various instantiations of $\alpha\text{-}\mathbf{Alg}$ later on, where its functor properties will come in handy.

4. Meet Iniga and Finn

Once upon a time a teacher had a pair of bright and capable students, who, for better or worse, were hooked on category theory. The first, Iniga, was a go-getting student with plenty of initiative. Interestingly, this was in stark contrast to Finn, a reserved character who perceived the world with a sense of finality.

The teacher posed them the problem of demonstrating that a system of stream equations has a unique solution. Owing to their polar opposite outlooks, Iniga and Finn took divergent approaches to tackling the problem, but as we will discover, their approaches turned out to be two sides of the same coin.

The teacher started with a minimalistic example, asking them to consider the following stream equations.

$$\begin{aligned} \mathit{one} &= 1 \prec \mathit{one} \\ \mathit{plus}\ (\mathit{Cons}\ m\ s, \mathit{Cons}\ n\ t) &= (m+n) \prec \mathit{plus}\ (s,t) \end{aligned}$$

Streams of natural numbers are the resultant behaviour of these equations, so the teacher provided the functor $\mathsf{B}\,X = \mathbb{N} \times X$ as the *behaviour functor*. We can give this a Haskell rendering:

$$\textbf{data}\ \mathsf{B}\ x = \mathit{Cons}\ (\mathbb{N}, x)\ .$$

For simplicity, the teacher fixed the element type of streams. An element of $\nu\mathsf{B}$, the carrier of the final coalgebra of the behaviour functor, is a stream of natural numbers: $\nu\mathsf{B} \cong \mathsf{Stream}\,\mathbb{N}$.

The stream constant *one* and the stream operator *plus* in the example stream equations are also modelled categorically with the functor $\mathsf{S}\,X = 1 + X \times X$ as the *syntax functor*.

$$\textbf{data}\ \mathsf{S}\ x = \mathit{One} \mid \mathit{Plus}\ (x, x)$$

An element of $\mu\mathsf{S}$, the initial algebra carrier of the syntax functor, is a finite, closed term, built from the syntax constructors of S.

Iniga (taking the initiative): Ok, given these definitions we can model the stream equations by a simple function.

$$\begin{aligned} \lambda\ (\mathit{One}) &= \mathit{Cons}\ (1, \mathit{One}) \\ \lambda\ (\mathit{Plus}\ (\mathit{Cons}\ (m,s), \mathit{Cons}\ (n,t))) &= \mathit{Cons}\ (m+n, \mathit{Plus}\ (s,t)) \end{aligned}$$

Teacher: Observe that λ is really a natural transformation of type $\mathsf{S}\circ\mathsf{B} \dot{\to} \mathsf{B}\circ\mathsf{S}$. This is crucial: the syntactic requirements on stream equations to ensure uniqueness of solutions are captured by the naturality requirement on λ. Its type can be seen as a promise that only the head of the incoming stream will be inspected and that an element of the outgoing stream will be constructed. Can you see how the slogan "consume one, produce one" translates?

Iniga: Yes! An interpretation of the syntax is then given by an S-algebra $a : \mathsf{S}\,(\nu\mathsf{B}) \to \nu\mathsf{B}$ whose carrier is the final B-coalgebra $\langle \nu\mathsf{B}, \mathit{out}\rangle$. The algebra a takes a level of syntax over a stream and turns it into a stream.

Teacher: How do we model that a respects the stream equations captured by λ? Your algebra a has to satisfy the following law:

$$\mathit{out} \cdot a = \mathsf{B}\,a \cdot \lambda\,(\nu\mathsf{B}) \cdot \mathsf{S}\,\mathit{out}\ :\ \mathsf{S}\,(\nu\mathsf{B}) \to \mathsf{B}\,(\nu\mathsf{B})\ . \tag{4.1}$$

The law states that unrolling the result of a is the same as unrolling the arguments of the syntax, $\mathsf{S}\,\mathit{out}$, applying the stream equations $\lambda\,(\nu\mathsf{B})$, and then interpreting the tail, $\mathsf{B}\,a$.

Iniga: Great, for our example I will rearrange the law to observe the Haskell convention of definition by pattern matching,

$a \cdot \mathsf{S}\,out^\circ = out^\circ \cdot \mathsf{B}\,a \cdot \lambda\,(\nu\mathsf{B})$. If I instantiate this law to our running example, I obtain a definition of the algebra a:

$$\begin{array}{l} a\ One \\ \quad = Out^\circ\ (Cons\ (1, a\ One)) \\ a\ (Plus\ (Out^\circ\ (Cons\ (m,s)), Out^\circ\ (Cons\ (n,t)))) \\ \quad = Out^\circ\ (Cons\ (m+n, a\ (Plus\ (s,t))))\ . \end{array}$$

With a, I can now define the semantic counterparts of *One* and *Plus*, the stream constant *one* and the stream operator *plus*, underlined to emphasize that they are semantic entities:

$$\begin{array}{ll} \underline{one} & = a\ One \\ \underline{plus}\ (s,t) & = a\ (Plus\ (s,t))\ . \end{array}$$

Teacher (interrupting): You are actually building upon the isomorphism $\mathsf{S}\,X \to X \cong (1 \to X) \times (X \times X \to X)$ here: the pair of functions, *one* and *plus*, is just another way of writing the algebra a.

Iniga: Using $a \prec s$ as a shorthand for $Out^\circ\ (Cons\ (a,s))$, the definition of a is the same as,

$$\begin{array}{ll} \underline{one} & = 1 \prec \underline{one} \\ \underline{plus}\ (Cons\ m\ s, Cons\ n\ t) & = m+n \prec \underline{plus}\ (s,t)\ , \end{array}$$

that is *one* and *plus* satisfy the original stream equations. Again, the notation makes clear that we have to read the stream operators semantically—*one* and *plus* are the entities defined by the system.

Teacher: We can wrap this up by showing that the law (4.1) *uniquely* determines a:

$$\begin{array}{ll} & out \cdot a = \mathsf{B}\,a \cdot \lambda\,(\nu\mathsf{B}) \cdot \mathsf{S}\,out \\ \iff & \{\ \text{uniqueness of unfold (3.2)}\ \} \\ & a = [\![\lambda\,(\nu\mathsf{B}) \cdot \mathsf{S}\,out]\!] \end{array}$$

So $[\![\lambda\,(\nu\mathsf{B}) \cdot \mathsf{S}\,out]\!]$ is the unique solution of the stream equations. Furthermore, the fold $(\!|a|\!) : \mu\mathsf{S} \to \nu\mathsf{B}$ takes syntax to behaviour by evaluating a term. Finn, what are your thoughts?

Finn: To start with, I would write the stream equations differently. I find them too Haskell-like, and I prefer what Jan Rutten calls "behavioural differential equations" [17].

$$\begin{array}{ll} head\ one & = 1 \\ tail\ \ one & = one \\ head\ (plus\ (s,t)) & = head\ s + head\ t \\ tail\ \ (plus\ (s,t)) & = plus\ (tail\ s, tail\ t) \end{array}$$

A semantics is given by a B-coalgebra $c : \mu\mathsf{S} \to \mathsf{B}\,(\mu\mathsf{S})$ whose carrier is the initial S-algebra $\langle \mu\mathsf{S}, in \rangle$. The coalgebra c takes a term and produces the first number of the defined stream, and a term to generate the rest of the stream.

Teacher: Just as for Iniga, your coalgebra c has to satisfy the following law:

$$c \cdot in = \mathsf{B}\,in \cdot \lambda\,(\mu\mathsf{S}) \cdot \mathsf{S}\,c\ :\ \mathsf{S}\,(\mu\mathsf{S}) \to \mathsf{B}\,(\mu\mathsf{S})\ . \tag{4.2}$$

The law states that building a term and applying c is the same as giving a semantics to the subterms, $\mathsf{S}\,c$, applying the stream equations $\lambda\,(\mu\mathsf{S})$, and building a term in the tail of a stream, $\mathsf{B}\,in$.

Finn: I will follow Iniga's lead and specialize the law to our example, obtaining a definition of c:

$$\begin{array}{l} c\ (In\ One) \\ \quad = Cons\ (1, In\ One) \\ c\ (In\ (Plus\ (s,t))) \\ \quad = Cons\ (head\ (c\ s) + head\ (c\ t), \\ \qquad\qquad In\ (Plus\ (tail\ (c\ s), tail\ (c\ t))))\ , \end{array}$$

where $head\,(Cons\,(a,s)) = a$ and $tail\,(Cons\,(a,s)) = s$. Given a *stream program*, my c gives the head of the stream and a stream program for the tail of the stream. I can now define the semantic counterparts of *head* and *tail*:

$$\begin{array}{ll} \underline{head}\ s & = head\ (c\ s) \\ \underline{tail}\ \ s & = tail\ \ (c\ s)\ . \end{array}$$

Teacher: You are building upon the isomorphism $X \to \mathsf{B}\,X \cong (X \to \mathbb{N}) \times (X \to X)$ here: *head* and *tail* is just another way of writing c.

Finn: Using *one* as a shorthand for $In\,One$ and $plus\,(s,t)$ for $In\,(Plus\,(s,t))$, the definition of c is the same as,

$$\begin{array}{ll} \underline{head}\ one & = 1 \\ \underline{tail}\ \ one & = one \\ \underline{head}\ (plus\ (s,t)) & = \underline{head}\ s + \underline{head}\ t \\ \underline{tail}\ \ (plus\ (s,t)) & = plus\ (\underline{tail}\ s, \underline{tail}\ t)\ , \end{array}$$

that is, *head* and *tail* satisfy the original stream equations. The notation emphasizes that we have to read the stream selectors semantically—*head* and *tail* are the entities defined by the system.

Teacher: Again, we can show that the law (4.2) determines c:

$$\begin{array}{ll} & c \cdot in = \mathsf{B}\,in \cdot \lambda\,(\mu\mathsf{S}) \cdot \mathsf{S}\,c \\ \iff & \{\ \text{uniqueness of fold (3.1)}\ \} \\ & c = (\!|\mathsf{B}\,in \cdot \lambda\,(\mu\mathsf{S})|\!) \end{array}$$

So $(\!|\mathsf{B}\,in \cdot \lambda\,(\mu\mathsf{S})|\!)$ is the *unique* solution of your stream equations. And the unfold $[\![c]\!] : \mu\mathsf{S} \to \nu\mathsf{B}$ takes syntax to behaviour by unrolling a complete stream.

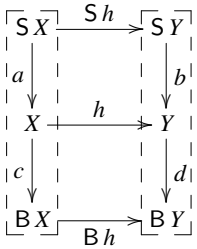
Iniga and Finn, you should reconcile your two viewpoints. Your semantic functions are of type $\mu\mathsf{S} \to \nu\mathsf{B}$, so is the fold of Iniga's algebra the same as the unfold of Finn's coalgebra: $(\!|a|\!) = [\![c]\!]$? Did you notice that we made use of the naturality of λ: Iniga used λ at type $\nu\mathsf{B}$, while Finn required the $\mu\mathsf{S}$ instance? For now, we have only discussed a minimalistic example, and we are not immediately able to model stream equations such as the ones that define the Fibonacci stream: there is more to this story.

Epilogue

Now that we have met Iniga and Finn and got a taste for the problem that their teacher posed to them, we will move on to introduce the infrastructure that is needed for the reconciliation.

5. Bialgebras

Let $\mathsf{S}, \mathsf{B} : \mathscr{C} \to \mathscr{C}$ be functors. A bialgebra is a triple $\langle X, a, c \rangle$ consisting of an object $X : \mathscr{C}$, an arrow $a : \mathsf{S}\,X \to X : \mathscr{C}$, and an arrow $c : X \to \mathsf{B}\,X : \mathscr{C}$. It is an S-algebra and a B-coalgebra with a common carrier. Let $\langle X, a, c \rangle$ and $\langle Y, b, d \rangle$ be bialgebras and $h : X \to Y : \mathscr{C}$ an arrow. Then h is a bialgebra homomorphism if it is both an S-algebra homomorphism and a B-coalgebra homomorphism.

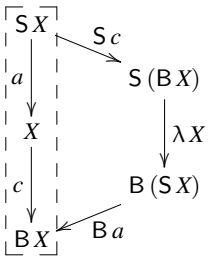


Identity is a bialgebra homomorphism and homomorphisms compose. Consequently, bialgebras and their homomorphisms form a category, called $\mathbf{Bialg}(\mathscr{C})$.

We are concerned with λ-bialgebras, which are bialgebras equipped with a so-called *distributive law* $\lambda : \mathsf{S} \circ \mathsf{B} \stackrel{\cdot}{\to} \mathsf{B} \circ \mathsf{S}$. This extra structure imposes a coherence condition on bialgebras.

$$c \cdot a = \mathsf{B}\,a \cdot \lambda X \cdot \mathsf{S}\,c \tag{5.1}$$

The condition is also called the *pentagonal law*.



(5.2)

The category of bialgebras that satisfy the pentagonal law (5.2) is denoted λ-$\mathbf{Bialg}(\mathscr{C})$. It is a full subcategory of $\mathbf{Bialg}(\mathscr{C})$.

6. Iniga and Finn with Bialgebraic-tinted Glasses

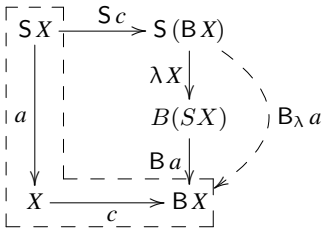
We will now use λ-bialgebras to explicate Iniga and Finn's conversation with their teacher, and begin to reconcile their solutions.

Let $\mathsf{S}, \mathsf{B} : \mathscr{C} \to \mathscr{C}$ be functors, and $\lambda : \mathsf{S}\circ\mathsf{B} \dot{\to} \mathsf{B}\circ\mathsf{S}$ be a natural transformation. We will read these to imply syntax and behaviour functors, and a distributive law modelling a set of equations. Using λ-bialgebras, we will characterize the semantic function from syntax to behaviour—the arrow from the least fixed-point of S to the greatest fixed-point of B.

The algebra $\langle \mu\mathsf{F}, in\rangle$ is the initial object in F-$\mathbf{Alg}$. We will now show that from this we can form the initial object in λ-$\mathbf{Bialg}$. If the carrier has been fixed as $\mu\mathsf{S}$, then the coalgebra will have type $\mu\mathsf{S} \to \mathsf{B}\,(\mu\mathsf{S})$. This is exactly Finn's coalgebra, and his teacher has derived it: $(\!|\mathsf{B}\,in \cdot \lambda\,(\mu\mathsf{S})|\!)$. As one might guess, the laws the teacher provided came from λ-bialgebras. Let us take a step back to re-examine λ and the pentagonal law.

6.1 Lifting Endofunctors to Algebras

The pentagonal law confers a useful property both on the algebra and the coalgebra component of a λ-bialgebra. Let us illustrate this first for the coalgebra component by redrawing diagram (5.2).



Here we can see that c is not only a B-coalgebra, but also an S-algebra homomorphism from $\langle X, a\rangle$ to $\langle \mathsf{B}\,X, \mathsf{B}\,a \cdot \lambda\,X\rangle$.

We can characterize this situation as lifting the endofunctor $\mathsf{B} : \mathscr{C} \to \mathscr{C}$ to a functor on S-algebras; we will give it the name $\mathsf{B}_\lambda : \mathsf{S}$-$\mathbf{Alg}(\mathscr{C}) \to \mathsf{S}$-$\mathbf{Alg}(\mathscr{C})$, and define it as,

$$\mathsf{B}_\lambda\,\langle X, a : \mathsf{S}\,X \to X\rangle = \langle \mathsf{B}\,X, \mathsf{B}\,a \cdot \lambda\,X : \mathsf{S}\,(\mathsf{B}\,X) \to \mathsf{B}\,X\rangle \ , \quad (6.1)$$

$$\mathsf{B}_\lambda\,h = \mathsf{B}\,h \ . \quad (6.2)$$

For notational simplicity, we shall employ lifted functors synecdochically: by $\mathsf{B}_\lambda\,a$ we mean $\mathsf{B}_\lambda\,\langle X, a\rangle$, a is used *pars pro toto*, and in certain contexts, $\mathsf{B}_\lambda\,a$ is used *totum pro parte* for the arrow of the resultant algebra, $\mathsf{B}\,a \cdot \lambda\,X$. That B_λ is a functor follows from a more general construction given in Appendix B. For reference, we record that it preserves S-algebra homomorphisms.

$$\mathsf{B}\,h : \mathsf{B}_\lambda\,a \to \mathsf{B}_\lambda\,b : \mathsf{S}\text{-}\mathbf{Alg} \iff h : a \to b : \mathsf{S}\text{-}\mathbf{Alg} \quad (6.3)$$

Therefore, we can give c, viewed as an algebra homomorphism, the more succinct type $c : a \to \mathsf{B}_\lambda\,a : \mathsf{S}$-$\mathbf{Alg}$.

Dually, a is both an S-algebra and a B-coalgebra homomorphism, with the type $a : \mathsf{S}^\lambda\,c \to c : \mathsf{B}$-$\mathbf{Coalg}$, where the lifted functor $\mathsf{S}^\lambda : \mathsf{B}$-$\mathbf{Coalg}(\mathscr{C}) \to \mathsf{B}$-$\mathbf{Coalg}(\mathscr{C})$ is defined as,

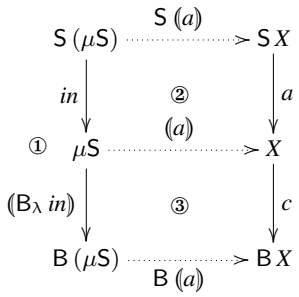
$$\mathsf{S}^\lambda\,\langle X, c : X \to \mathsf{B}\,X\rangle = \langle \mathsf{S}\,X, \lambda\,X \cdot \mathsf{S}\,c : \mathsf{S}\,X \to \mathsf{B}\,(\mathsf{S}\,X)\rangle \ , \quad (6.4)$$

$$\mathsf{S}^\lambda\,h = \mathsf{S}\,h \ . \quad (6.5)$$

By duality, S^λ is functorial, as well.

6.2 Initial and Final Objects

Our initial λ-bialgebra will be $\langle \mu\mathsf{S}, in, (\!|\mathsf{B}_\lambda\,in|\!)\rangle$, as depicted below.



We have three proof obligations. First we must show that the triple $\langle \mu\mathsf{S}, in, (\!|\mathsf{B}_\lambda\,in|\!)\rangle$ is indeed a λ-bialgebra (①)—it has the right types, but it must also satisfy (5.1).

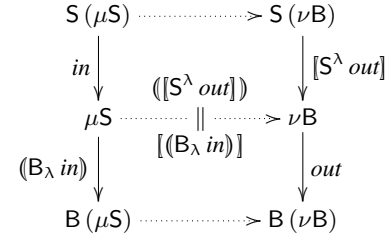
$$\begin{aligned}
& (\!|\mathsf{B}_\lambda\,in|\!) \cdot in \\
= \ & \{ \text{ fold computation (§A) } \} \\
& \mathsf{B}_\lambda\,in \cdot \mathsf{S}\,(\!|\mathsf{B}_\lambda\,in|\!) \\
= \ & \{ \text{ definition of } \mathsf{B}_\lambda \text{ (6.1) } \} \\
& \mathsf{B}\,in \cdot \lambda\,(\mu\mathsf{S}) \cdot \mathsf{S}\,(\!|\mathsf{B}_\lambda\,in|\!)
\end{aligned}$$

The second obligation, that $(\!|a|\!)$ is an S-algebra homomorphism is by construction—the top half of the diagram commutes (②). Moreover, the uniqueness of this arrow comes for free. Finally, we must show that $(\!|a|\!)$ is also a B-coalgebra homomorphism—that the bottom half of the diagram commutes (③).

$$\begin{aligned}
& c \cdot (\!|a|\!) = \mathsf{B}\,(\!|a|\!) \cdot (\!|\mathsf{B}_\lambda\,in|\!) \\
\iff \ & \{ \text{ fold fusion (§A) with } c : a \to \mathsf{B}_\lambda\,a : \mathsf{S}\text{-}\mathbf{Alg} \ \} \\
& (\!|\mathsf{B}_\lambda\,a|\!) = \mathsf{B}\,(\!|a|\!) \cdot (\!|\mathsf{B}_\lambda\,in|\!) \\
\impliedby \ & \{ \text{ fold fusion (§A) } \} \\
& \mathsf{B}\,(\!|a|\!) : \mathsf{B}_\lambda\,in \to \mathsf{B}_\lambda\,a : \mathsf{S}\text{-}\mathbf{Alg} \\
\impliedby \ & \{ \ \mathsf{B}_\lambda \text{ functor (6.3) } \} \\
& (\!|a|\!) : in \to a : \mathsf{S}\text{-}\mathbf{Alg}
\end{aligned}$$

We can dualize the results above. We have just used Finn's coalgebra to construct the initial λ-bialgebra, so naturally we will use Iniga's algebra to construct the final λ-bialgebra. Indeed, $\langle \nu\mathsf{B}, [\![\mathsf{S}^\lambda\,out]\!], out\rangle$ is the final λ-bialgebra; and $[\![c]\!]$ is the unique homomorphism from any λ-bialgebra $\langle X, a, c\rangle$ to the final λ-bialgebra. The duality extends to the satisfaction of the proof obligations.

We have defined the initial and final λ-bialgebras, and we are now in a position to state the homomorphism between them—the semantic function from syntax to behaviour $\mu\mathsf{S} \to \nu\mathsf{B}$.



The semantic arrow is unique, and we can give two justifications for it being so: namely that it is the unique homomorphism both from the initial λ-bialgebra and to the final λ-bialgebra. For the same two reasons, we can give two definitions of this arrow, and by uniqueness they are equal. And just like that, we have the basic resolution of Iniga and Finn's seemingly opposing points of view.

In a manner of speaking, Iniga and Finn's personalities would appear to be entwined. Iniga thought in terms of initial algebras and folds, but ended up constructing the final λ-bialgebra, and vice versa for Finn. This is not a coincidence as the category of λ-bialgebras is isomorphic to a category of algebras over coalgebras.

$$
\begin{aligned}
& \langle X, a, c\rangle : \lambda\text{-}\mathbf{Bialg}(\mathscr{C}) \\
\iff & \quad \{ \text{ definition of } \lambda\text{-bialgebra } \} \\
& c \cdot a = \mathsf{B}\, a \cdot \lambda X \cdot \mathsf{S}\, c \\
\iff & \quad \{ \text{ definition of } \mathsf{S}^{\lambda} \text{ (6.4) } \} \\
& c \cdot a = \mathsf{B}\, a \cdot \mathsf{S}^{\lambda}\, c \\
\iff & \quad \{ \text{ definition of B-coalgebra homomorphism } \} \\
& a : \mathsf{S}^{\lambda}\, \langle X, c\rangle \to \langle X, c\rangle : \mathsf{B}\text{-}\mathbf{Coalg}(\mathscr{C}) \\
\iff & \quad \{ \text{ definition of an } \mathsf{S}^{\lambda}\text{-algebra } \} \\
& \langle \langle X, c\rangle, a\rangle : \mathsf{S}^{\lambda}\text{-}\mathbf{Alg}(\mathsf{B}\text{-}\mathbf{Coalg}(\mathscr{C}))
\end{aligned}
$$

The proof shows that the objects are in one-to-one correspondence. A similar calculation establishes a bijection between arrows.

$$
\begin{aligned}
& h : \langle X_1, a_1, c_1\rangle \to \langle X_2, a_2, c_2\rangle \\
\iff & \quad \{ \text{ definition of } \lambda\text{-bialgebra homomorphism } \} \\
& h \cdot a_1 = a_2 \cdot \mathsf{S}\, h \ \ \wedge \ \ \mathsf{B}\, h \cdot c_1 = c_2 \cdot h \\
\iff & \quad \{ \text{ definition of B-coalgebra homomorphism } \} \\
& h \cdot a_1 = a_2 \cdot \mathsf{S}\, h \ \ \wedge \ \ h : \langle X_1, c_1\rangle \to \langle X_2, c_2\rangle : \mathsf{B}\text{-}\mathbf{Coalg}(\mathscr{C}) \\
\iff & \quad \{ \text{ definition of } \mathsf{S}^{\lambda}\text{-algebra homomorphism } \} \\
& h : \langle \langle X_1, c_1\rangle, a_1\rangle \to \langle \langle X_2, c_2\rangle, a_2\rangle : \mathsf{S}^{\lambda}\text{-}\mathbf{Alg}(\mathsf{B}\text{-}\mathbf{Coalg}(\mathscr{C}))
\end{aligned}
$$

As a summary of our construction above, for the categorically enlightened, the final λ-bialgebra is determined by the final S^{λ}-algebra. Recall that the *final* S-algebra is $\langle 1, !_{\mathsf{S}\,1}\rangle$. Consequently, the final S^{λ}-algebra is $\langle 1, !_{\mathsf{S}^{\lambda}\,1}\rangle = \langle\langle \nu \mathsf{B}, \mathit{out}\rangle, [\![\mathsf{S}^{\lambda}\, \mathit{out}]\!]\rangle$ as $\langle \nu \mathsf{B}, \mathit{out}\rangle$ is the final object in $\mathsf{B}\text{-}\mathbf{Coalg}(\mathscr{C})$.

Dually, we can view a bialgebra as a coalgebra over algebras.

$$
\lambda\text{-}\mathbf{Bialg}(\mathscr{C}) \cong \begin{cases} \mathsf{S}^{\lambda}\text{-}\mathbf{Alg}(\mathsf{B}\text{-}\mathbf{Coalg}(\mathscr{C})) \\ \mathsf{B}_{\lambda}\text{-}\mathbf{Coalg}(\mathsf{S}\text{-}\mathbf{Alg}(\mathscr{C})) \end{cases} \tag{6.6}
$$

The double isomorphism says that there are actually two ways to determine initial and final objects in $\lambda\text{-}\mathbf{Bialg}(\mathscr{C})$. The reader is encouraged to work out the details.

7. A Step along the Categorical Brick Road...

Distributive laws of type $\mathsf{S}\circ\mathsf{B} \dot{\to} \mathsf{B}\circ\mathsf{S}$ are not sufficiently expressive to model the recursion equations for *bird* and *fib* as their right-hand sides consist of more than one layer of syntax. In general, we need terms, elements of the free monad for the syntax functor. However, rather than making a beeline for free monads, we will visit *pointed functors* as a stepping stone. This is an adventure with category theory after all, and the fun is in the journey.

Example 7.1. Suspend your disbelief and suppose that you need the identity operator on streams, defined by the equation,

$$
\mathit{id}\ (\mathit{Cons}\ m\ s) = m \prec s \ .
$$

A system containing this equation cannot be turned into a distributive law $\lambda : \mathsf{S}\circ\mathsf{B} \dot{\to} \mathsf{B}\circ\mathsf{S}$ as the stream s is not an element of the syntax functor S. To solve this, we can allow for variables or constructors of S.

$$
\begin{aligned}
&\mathbf{data}\ \mathsf{P}\ x = \mathit{Var}\ x \mid \mathit{Con}\ (\mathsf{S}\ x) \\
&\mathbf{data}\ \mathsf{S}\ x = \mathit{Id}\ x \mid \mathit{One} \mid \ldots
\end{aligned}
$$

A system of recursion equations is now captured by a natural transformation ρ of type $\mathsf{S}\circ\mathsf{B} \dot{\to} \mathsf{B}\circ\mathsf{P}$.

$$
\begin{aligned}
\rho\ (\mathit{Id}\ (\mathit{Cons}\ (m, s))) &= \mathit{Cons}\ (m, \mathit{Var}\ s) \\
\rho\ (\mathit{One}) &= \mathit{Cons}\ (1,\ \mathit{Con}\ \mathit{One}) \quad \ldots
\end{aligned}
$$

Note that we have only replaced S on the right-hand side, where there is a need. We shall later restore symmetry and show how to turn ρ into a distributive law (Section 7.3). Furthermore, this is a very limited introduction of variables: one can either have a variable, or a constructor, but no variables as arguments. □

The Haskell type P is the so-called free pointed functor of S [13]. We will discuss pointed functors in general and then return to the free construction in Section 7.1.

Definition 7.2. We say that an endofunctor $\mathsf{T} : \mathscr{C} \to \mathscr{C}$ is *pointed* if it is equipped with a natural transformation $\eta : \mathsf{Id} \dot{\to} \mathsf{T}$.

We are going to build on the picture we laid out in the previous section by replacing the plain endofunctor with a pointed functor. The extra structure that we have introduced with η has two implications: first with regards to the distributive law λ and second with regards to constructing algebras of pointed functors.

Condition 7.3. A distributive law $\lambda : \mathsf{T}\circ\mathsf{B} \dot{\to} \mathsf{B}\circ\mathsf{T}$ for a pointed functor T has an additional coherence condition to satisfy:

$$
\lambda \cdot \eta\circ\mathsf{B} = \mathsf{B}\circ\eta \ . \tag{7.1}
$$

Condition 7.4. If we construct an algebra $\langle X, a : \mathsf{T}\, X \to X\rangle$ of a pointed functor T, then it must respect η:

$$
a \cdot \eta X = \mathit{id}_X \ . \tag{7.2}
$$

For full specificity we will say that $(\mathsf{T}, \eta)\text{-}\mathbf{Alg}(\mathscr{C})$ is the category of T-algebras that respect η. This is a full subcategory of $\mathsf{T}\text{-}\mathbf{Alg}(\mathscr{C})$. Henceforth, we will be working with λ-bialgebras based on (T, η)-algebras and B-coalgebras.

The double isomorphism (6.6) succinctly tells the story of initial and final objects in $\lambda\text{-}\mathbf{Bialg}$. In a sense, Conditions 7.3 and 7.4 ensure that we can establish an analogous isomorphism for pointed functors. The following two properties prepare the ground.

Property 7.5. Let $c : X \to \mathsf{B}\, X$ be a B-coalgebra, then

$$
\eta X : c \to \mathsf{T}^{\lambda}\, c : \mathsf{B}\text{-}\mathbf{Coalg}(\mathscr{C}) \ , \tag{7.3}
$$

is the lifting of η to a B-coalgebra homomorphism.

Proof.

$$
\begin{aligned}
& \mathsf{T}^{\lambda}\, c \cdot \eta X \\
= & \quad \{ \text{ definition of } \mathsf{T}^{\lambda} \text{ (6.4) } \} \\
& \lambda X \cdot \mathsf{T}\, c \cdot \eta X \\
= & \quad \{ \eta : \mathsf{Id} \dot{\to} \mathsf{T} \text{ is natural } \} \\
& \lambda X \cdot \eta(\mathsf{B}\, X) \cdot c \\
= & \quad \{ \text{ coherence of } \lambda \text{ with } \eta \text{ (7.1) } \} \\
& \mathsf{B}\, (\eta X) \cdot c \quad \square
\end{aligned}
$$

In other words, the lifted functor T^{λ} is pointed as well and we can form $(\mathsf{T}^{\lambda}, \eta)\text{-}\mathbf{Alg}(\mathsf{B}\text{-}\mathbf{Coalg}(\mathscr{C}))$.

Property 7.6. The functor B_{λ} preserves respect for η.

$$
\mathsf{B}_{\lambda}\, a \cdot \eta(\mathsf{B}\, X) = \mathit{id}_{\mathsf{B}\, X} \iff a \cdot \eta X = \mathit{id}_X \tag{7.4}
$$

Proof.

$$\begin{aligned}
&\mathsf{B}_\lambda\, a \cdot \eta(\mathsf{B}\, X) \\
=\ & \{ \text{ definition of } \mathsf{B}_\lambda \text{ (6.1) } \} \\
&\mathsf{B}\, a \cdot \lambda\, X \cdot \eta(\mathsf{B}\, X) \\
=\ & \{ \text{ coherence of } \lambda \text{ with } \eta \text{ (7.1) } \} \\
&\mathsf{B}\, a \cdot \mathsf{B}\, (\eta X) \\
=\ & \{ \mathsf{B} \text{ functor and assumption } a \cdot \eta X = id_X \} \\
&id_{\mathsf{B}\, X} \quad \square
\end{aligned}$$

In other words, B_λ is an endofunctor on (T,η)-$\mathbf{Alg}(\mathscr{C})$ and we can form B_λ-$\mathbf{Coalg}((\mathsf{T},\eta)$-$\mathbf{Alg}(\mathscr{C}))$.

Summary

Properties 7.5 and 7.6 imply that the double isomorphism (6.6) carries over to the new setting.

$$\lambda\text{-}\mathbf{Bialg}(\mathscr{C}) \cong \begin{cases} (\mathsf{T}^\lambda,\eta)\text{-}\mathbf{Alg}(\mathsf{B}\text{-}\mathbf{Coalg}(\mathscr{C})) \\ \mathsf{B}_\lambda\text{-}\mathbf{Coalg}((\mathsf{T},\eta)\text{-}\mathbf{Alg}(\mathscr{C})) \end{cases} \tag{7.5}$$

7.1 Free Pointed Functor

Let $\mathsf{S} : \mathscr{C} \to \mathscr{C}$ be an endofunctor. There is a canonical pointed functor, with pleasant properties, that we can construct from S. This is the *free* pointed functor of S [13], the categorical version of the Haskell type P we saw in Example 7.1,

$$\mathsf{P}\, X = X + \mathsf{S}\, X \ . \tag{7.6}$$

The natural transformation $\eta : \mathsf{Id} \dot{\to} \mathsf{P}$ that equips the free pointed functor is simply $\eta = inl$. Our λ-bialgebras now have P-algebras, but what about all the S-algebras that we have used previously? All is not lost, in fact far from it.

Theorem 7.1. *The category of algebras for the free pointed functor is isomorphic to the category of* S*-algebras:*

$$(\mathsf{P},\eta)\text{-}\mathbf{Alg}(\mathscr{C}) \cong \mathsf{S}\text{-}\mathbf{Alg}(\mathscr{C}) \ .$$

The following definitions are the witnesses to this isomorphism.

$$\lceil \langle X, a : \mathsf{S}\, X \to X \rangle \rceil = \langle X, id_X \triangledown a : \mathsf{P}\, X \to X \rangle \quad \lceil h \rceil = h \tag{7.7}$$

$$\lfloor \langle X, b : \mathsf{P}\, X \to X \rangle \rfloor = \langle X, b \cdot inr : \mathsf{S}\, X \to X \rangle \quad \lfloor h \rfloor = h \tag{7.8}$$

In particular, $\lceil - \rceil$ *preserves and reflects homomorphisms.*

$$h : \lceil a \rceil \to \lceil b \rceil : (\mathsf{P},\eta)\text{-}\mathbf{Alg}(\mathscr{C}) \Longleftrightarrow h : a \to b : \mathsf{S}\text{-}\mathbf{Alg}(\mathscr{C}) \tag{7.9}$$

Proof. (i) Given an S-algebra a, we can cast it up to a P-algebra $\lceil a \rceil$. Likewise, we can cast a P-algebra b down to an S-algebra $\lfloor b \rfloor$. The following proves both directions of the isomorphism.

$$\begin{aligned}
&\lfloor \lceil a \rceil \rfloor && \lceil \lfloor b \rfloor \rceil \\
=\ & \{ \text{ definition of } \lceil - \rceil \ \} && =\ \{ \text{ defs. of } \lfloor - \rfloor \text{ and } \lceil - \rceil \ \} \\
&\lfloor id_X \triangledown a \rfloor && id_X \triangledown b \cdot inr \\
=\ & \{ \text{ definition of } \lfloor - \rfloor \ \} && =\ \{ \ b \text{ respects } \eta \text{ (7.2) } \} \\
&(id_X \triangledown a) \cdot inr && b \cdot inl \triangledown b \cdot inr \\
=\ & \{ \text{ join comp. (§A) } \} && =\ \{ \text{ join fusion and refl. (§A) } \} \\
&a && b
\end{aligned}$$

(ii) $\lfloor - \rfloor$ is functorial as $inr : \mathsf{S} \dot{\to} \mathsf{P}$ and $\lfloor - \rfloor = inr$-$\mathbf{Alg}$ (cf. §3).

(iii) $\lceil - \rceil$ maps S-homomorphisms to P-homomorphisms. For the proof we refer to the full paper [11].

(iv) Finally, $\lceil \langle X, a \rangle \rceil$ has to be an algebra for the pointed functor. That $\lceil a \rceil$ respects η (7.2), unfolds to $(id \triangledown a) \cdot inl = id$, and this is just an instance of the join computation law (§A). $\square$

7.2 Initial and Final Objects

The double isomorphism (7.5) immediately suggests how to define initial and final objects in the new setting. Nonetheless, we will slow down a bit and go through the construction step by step.

In Section 6 we explored λ-bialgebras over S and B, the functors representing syntax and behaviour, respectively. Despite the fact that we are now using the free pointed functor of S, the carrier of the initial λ-bialgebra will remain the same, as we are not changing our objects of syntax. Instead, we are generalizing the evaluation of our syntax. The initial λ-bialgebra will be $\langle \mu\mathsf{S}, a : \mathsf{P}\,(\mu\mathsf{S}) \to \mu\mathsf{S}, c : \mu\mathsf{S} \to \mathsf{B}\,(\mu\mathsf{S}) \rangle$, for some a and c that we will now determine.

Previously the algebra component of the initial λ-bialgebra was simply $in : \mathsf{S}\,(\mu\mathsf{S}) \to \mu\mathsf{S}$. This can no longer be the case; we need an algebra $a : \mathsf{P}\,(\mu\mathsf{S}) \to \mu\mathsf{S}$. However, now that we can freely cast between S and (P,η)-algebras, we can use $\lceil in \rceil : \mathsf{P}\,(\mu\mathsf{S}) \to \mu\mathsf{S}$.

The previous coalgebra component was $(\!|\mathsf{B}_\lambda\, in|\!)$, and this also no longer has the right type, as our λ has changed. Now B_λ lifts the functor B to a functor on (P,η)-algebras, not S-algebras; $(\!|-|\!)$ expects an S-algebra, and *in* is an S-algebra. We can satisfy these expectations with selective usage of casting: we can cast *in* up to a (P,η)-algebra so that we can apply B_λ, and furthermore, we can cast the image of $\mathsf{B}_\lambda \lceil in \rceil$ down so that it is an S-algebra that we can fold. The claim is that $\langle \mu\mathsf{S}, \lceil in \rceil, (\!|\lfloor \mathsf{B}_\lambda \lceil in \rceil \rfloor|\!) \rangle$ is the initial λ-bialgebra and $(\!|\lfloor a \rfloor|\!)$ is the unique homomorphism to any λ-bialgebra $\langle X, a, c \rangle$. There are the three usual proof obligations we must satisfy. For reasons that will become clear, we will start by showing that $(\!|\lfloor a \rfloor|\!)$ is (P,η)-algebra homomorphism.

$$(\!|\lfloor a \rfloor|\!) : \lceil in \rceil \to a : (\mathsf{P},\eta)\text{-}\mathbf{Alg} \tag{7.10}$$

This is a direct consequence of Theorem 7.1.

$$\begin{aligned}
&(\!|\lfloor a \rfloor|\!) : \lceil in \rceil \to a : (\mathsf{P},\eta)\text{-}\mathbf{Alg} \\
\Longleftrightarrow\ & \{ \text{ isomorphism } (\mathsf{P},\eta)\text{-}\mathbf{Alg} \cong \mathsf{S}\text{-}\mathbf{Alg} \text{ (7.9) } \} \\
&(\!|\lfloor a \rfloor|\!) : in \to \lfloor a \rfloor : \mathsf{S}\text{-}\mathbf{Alg}
\end{aligned}$$

Next we will show that $\langle \mu\mathsf{S}, \lceil in \rceil, (\!|\lfloor \mathsf{B}_\lambda \lceil in \rceil \rfloor|\!) \rangle$ is indeed a λ-bialgebra, in that it satisfies the pentagonal law (5.1).

$$\begin{aligned}
&(\!|\lfloor \mathsf{B}_\lambda \lceil in \rceil \rfloor|\!) \cdot \lceil in \rceil \\
=\ & \{ \ (\!|\lfloor \mathsf{B}_\lambda \lceil in \rceil \rfloor|\!) : \lceil in \rceil \to \mathsf{B}_\lambda \lceil in \rceil : (\mathsf{P},\eta)\text{-}\mathbf{Alg} \text{ (7.10) } \} \\
&\mathsf{B}_\lambda \lceil in \rceil \cdot \mathsf{P}\, (\!|\lfloor \mathsf{B}_\lambda \lceil in \rceil \rfloor|\!) \\
=\ & \{ \text{ definition of } \mathsf{B}_\lambda \text{ (6.1) } \} \\
&\mathsf{B} \lceil in \rceil \cdot \lambda\, (\mu\mathsf{S}) \cdot \mathsf{P}\, (\!|\lfloor \mathsf{B}_\lambda \lceil in \rceil \rfloor|\!)
\end{aligned}$$

Furthermore, (7.2) is satisfied since $\lceil - \rceil$ creates such an algebra.

Finally, we will show that $(\!|\lfloor a \rfloor|\!)$ is a B-coalgebra homomorphism. We know from the pentagonal law that c is a (P,η)-algebra homomorphism, $c : a \to \mathsf{B}_\lambda\, a : (\mathsf{P},\eta)$-$\mathbf{Alg}$. By (7.9) c is also an S-algebra homomorphism, $c : \lfloor a \rfloor \to \lfloor \mathsf{B}_\lambda\, a \rfloor : \mathsf{S}$-$\mathbf{Alg}$, and as a direct consequence of fold fusion (§A), $c \cdot (\!|\lfloor a \rfloor|\!) = (\!|\lfloor \mathsf{B}_\lambda\, a \rfloor|\!)$.

$$\begin{aligned}
&c \cdot (\!|\lfloor a \rfloor|\!) = \mathsf{B}\, (\!|\lfloor a \rfloor|\!) \cdot (\!|\lfloor \mathsf{B}_\lambda \lceil in \rceil \rfloor|\!) \\
\Longleftrightarrow\ & \{ \text{ fold fusion (§A) with } c : \lfloor a \rfloor \to \lfloor \mathsf{B}_\lambda\, a \rfloor : \mathsf{S}\text{-}\mathbf{Alg} \ \} \\
&(\!|\lfloor \mathsf{B}_\lambda\, a \rfloor|\!) = \mathsf{B}\, (\!|\lfloor a \rfloor|\!) \cdot (\!|\lfloor \mathsf{B}_\lambda \lceil in \rceil \rfloor|\!) \\
\Longleftarrow\ & \{ \text{ fold fusion (§A) } \} \\
&\mathsf{B}\, (\!|\lfloor a \rfloor|\!) : \lfloor \mathsf{B}_\lambda \lceil in \rceil \rfloor \to \lfloor \mathsf{B}_\lambda\, a \rfloor : \mathsf{S}\text{-}\mathbf{Alg} \\
\Longleftrightarrow\ & \{ \text{ isomorphism } (\mathsf{P},\eta)\text{-}\mathbf{Alg} \cong \mathsf{S}\text{-}\mathbf{Alg} \text{ (7.9) } \} \\
&\mathsf{B}\, (\!|\lfloor a \rfloor|\!) : \mathsf{B}_\lambda \lceil in \rceil \to \mathsf{B}_\lambda\, a : (\mathsf{P},\eta)\text{-}\mathbf{Alg} \\
\Longleftarrow\ & \{ \ \mathsf{B}_\lambda \text{ functor (6.3) } \} \\
&(\!|\lfloor a \rfloor|\!) : \lceil in \rceil \to a : (\mathsf{P},\eta)\text{-}\mathbf{Alg}
\end{aligned}$$

We have already shown that the last statement holds (7.10).

As before, the final λ-bialgebra is $\langle \nu\mathsf{B}, [\![\mathsf{P}^\lambda\ out]\!], out\rangle$. The unique λ-bialgebra homomorphism to the final λ-bialgebra from any λ-bialgebra $\langle X, a, c\rangle$ is $[\![c]\!]$. There is one final proof obligation: we have to show that $[\![\mathsf{P}^\lambda\ out]\!]$ respects η (7.2).

$$
\begin{array}{rl}
& [\![\mathsf{P}^\lambda\ out]\!] \cdot \eta(\nu\mathsf{B}) = id_{\nu\mathsf{B}} \\
\iff & \{\ \text{unfold reflection (§A)}\ \} \\
& [\![\mathsf{P}^\lambda\ out]\!] \cdot \eta(\nu\mathsf{B}) = [\![out]\!] \\
\Longleftarrow & \{\ \text{unfold fusion (§A)}\ \} \\
& \eta(\nu\mathsf{B}) : out \to \mathsf{P}^\lambda\ out
\end{array}
$$

The last statement holds as P^λ is pointed (7.3).

Putting things together, we can give a new statement of the semantic function $\mu\mathsf{S} \to \nu\mathsf{B}$.

$$
\begin{array}{ccc}
\mathsf{P}\,(\mu\mathsf{S}) & \dashrightarrow & \mathsf{P}\,(\nu\mathsf{B}) \\
{\scriptstyle \lceil in \rceil}\downarrow & & \downarrow{\scriptstyle [\![\mathsf{P}^\lambda\ out]\!]} \\
\mu\mathsf{S} & \dashrightarrow & \nu\mathsf{B} \\
{\scriptstyle (\!\lfloor \mathsf{B}_\lambda\ \lceil in \rceil \rfloor\!)}\downarrow & & \downarrow{\scriptstyle out} \\
\mathsf{B}\,(\mu\mathsf{S}) & \dashrightarrow & \mathsf{B}\,(\nu\mathsf{B})
\end{array}
\qquad
\begin{array}{c}
(\!\lfloor [\![\mathsf{P}^\lambda\ out]\!] \rfloor\!) \\
\| \\
[\![(\!\lfloor \mathsf{B}_\lambda\ \lceil in \rceil \rfloor\!)]\!]
\end{array}
$$

We are in a more expressive setting, yet thanks to Theorem 7.1, we can hold on to our resolution of Iniga and Finn's viewpoints.

7.3 Constructing a Distributive Law

In Section 6 we modelled a stream program by a distributive law of type $\mathsf{S}\circ\mathsf{B} \dot{\to} \mathsf{B}\circ\mathsf{S}$. With the introduction of the free pointed functor, stream equations have become slightly more expressive. A program, such as in Example 7.1, now gives rise to a natural transformation $\rho : \mathsf{S}\circ\mathsf{B} \dot{\to} \mathsf{B}\circ\mathsf{P}$. The pointed functor appears only on the right. On the left we keep S, as a stream equation defines a constructor of S, not a variable. From $\rho : \mathsf{S}\circ\mathsf{B} \dot{\to} \mathsf{B}\circ\mathsf{P}$ we seek to *construct* a distributive law $\lambda : \mathsf{P}\circ\mathsf{B} \dot{\to} \mathsf{B}\circ\mathsf{P}$ such that

$$c \cdot \lceil a \rceil = \mathsf{B}\,\lceil a \rceil \cdot \lambda X \cdot \mathsf{P}\,c \iff c \cdot a = \mathsf{B}\,\lceil a \rceil \cdot \rho X \cdot \mathsf{S}\,c \ . \quad (7.11)$$

Since P is a coproduct, λ has to be defined by a case analysis. Though obvious, we will calculate λ from the specification above as this will serve nicely as a blueprint for later sections.

$$
\begin{array}{rl}
& c \cdot a = \mathsf{B}\,\lceil a \rceil \cdot \rho X \cdot \mathsf{S}\,c \\
\iff & \{\ \text{equality of joins}\ \} \\
& c \triangledown c \cdot a = c \triangledown \mathsf{B}\,\lceil a \rceil \cdot \rho X \cdot \mathsf{S}\,c \\
\iff & \{\ \lceil a \rceil \text{ respects } \eta \text{ (7.2) and B functor}\ \} \\
& c \triangledown c \cdot a = \mathsf{B}\,\lceil a \rceil \cdot \mathsf{B}\,(\eta X) \cdot c \triangledown \mathsf{B}\,\lceil a \rceil \cdot \rho X \cdot \mathsf{S}\,c \\
\iff & \{\ \text{join fusion and functor fusion (§A)}\ \} \\
& c \cdot (id \triangledown a) = \mathsf{B}\,\lceil a \rceil \cdot (\mathsf{B}\,(\eta X) \triangledown \rho X) \cdot (c + \mathsf{S}\,c) \\
\iff & \{\ \text{definitions of } \lceil - \rceil \text{ (7.7) and P (7.6)}\ \} \\
& c \cdot \lceil a \rceil = \mathsf{B}\,\lceil a \rceil \cdot (\mathsf{B}\,(\eta X) \triangledown \rho X) \cdot \mathsf{P}\,c
\end{array}
$$

The specification (7.11) can be satisfied if we set $\lambda = \mathsf{B}\circ\eta \triangledown \rho$, which is easily seen to satisfy the coherence condition (7.1).

8. ...to Monad City

With pointed functors we made a limited introduction of variables. The next step is to allow constructors to be nested. In this section we are going to build on our picture of λ-bialgebras again, augmenting pointed functors to monads.

Example 8.1. Let us look at an example comparable to those of Section 2. Here is a stream equation for the natural numbers.

$$nat = 0 \prec nat + 1$$

We need more than a single syntax constructor to represent $nat + 1$. To solve this, we build terms with variables and constructors of S.

$$
\begin{array}{l}
\textbf{data}\ \mathsf{M}\ x = Var\ x \mid Com\ (\mathsf{S}\ (\mathsf{M}\ x)) \\
\textbf{data}\ \mathsf{S}\ x\ = One \mid Plus\ (x, x) \mid Nat
\end{array}
$$

A system of recursion equations is now captured by a natural transformation ρ of type $\mathsf{S}\circ\mathsf{B} \dot{\to} \mathsf{B}\circ\mathsf{M}$.

$$
\begin{array}{l}
\rho\ One \\
\quad = Cons\ (1, Com\ One) \\
\rho\ (Plus\ (Cons\ (m, s), Cons\ (n, t))) \\
\quad = Cons\ (m + n, Com\ (Plus\ (Var\ s, Var\ t))) \\
\rho\ Nat \\
\quad = Cons\ (0, Com\ (Plus\ (Com\ Nat, Com\ One)))
\end{array}
$$

Note that we only have terms on the right-hand side. Arguments of *Cons* on the left can be embedded into variables on the right, and as shown in the case of *Nat*, we can use more than one level of syntax. Again, we shall restore symmetry later, showing how to derive a distributive law from ρ (Section 8.3). □

The Haskell type M is the so-called free monad of S. We will discuss monads in general and then return to the free construction in Section 8.1.

Definition 8.2. We say that $\mathsf{T} : \mathscr{C} \to \mathscr{C}$ is a *monad* if there are natural transformations $\eta : \mathsf{Id} \dot{\to} \mathsf{T}$ and $\mu : \mathsf{T}\circ\mathsf{T} \dot{\to} \mathsf{T}$ such that

$$\mu \cdot \eta\circ\mathsf{T} = id_{\mathsf{T}} \ , \quad (8.1a)$$

$$\mu \cdot \mathsf{T}\circ\eta = id_{\mathsf{T}} \ , \quad (8.1b)$$

$$\mu \cdot \mu\circ\mathsf{T} = \mu \cdot \mathsf{T}\circ\mu \ . \quad (8.1c)$$

A monad extends a pointed functor with a second natural transformation $\mu : \mathsf{T}\circ\mathsf{T} \dot{\to} \mathsf{T}$. In the previous section we saw that η must be respected when constructing algebras and also by the distributive law of the λ-bialgebra; these same conditions extend to μ.

Condition 8.3. The following are the necessary coherence conditions for a distributive law $\lambda : \mathsf{T}\circ\mathsf{B} \dot{\to} \mathsf{B}\circ\mathsf{T}$ over a monad T:

$$\lambda \cdot \eta\circ\mathsf{B} = \mathsf{B}\circ\eta \ , \quad (8.2a)$$

$$\lambda \cdot \mu\circ\mathsf{B} = \mathsf{B}\circ\mu \cdot \lambda\circ\mathsf{T} \cdot \mathsf{T}\circ\lambda \ . \quad (8.2b)$$

Condition 8.4. If we construct an algebra $\langle X, a : \mathsf{T}\,X \to X\rangle$ of a monad T, then it must respect both η and μ.

$$a \cdot \eta X = id_X \ , \quad (8.3a)$$

$$a \cdot \mu X = a \cdot \mathsf{T}\,a \ . \quad (8.3b)$$

In the same manner as for pointed functors, we will say that (T, η, μ)-$\mathbf{Alg}(\mathscr{C})$ is the category of T-algebras that respect η and μ, a full subcategory of T-$\mathbf{Alg}(\mathscr{C})$. Henceforth, we will be working with λ-bialgebras based on (T, η, μ)-algebras and B-coalgebras.

As in Section 7, the additional conditions ensure that the double isomorphism (6.6) is maintained. We have shown previously that η can be lifted to a B-coalgebra homomorphism (7.3). There is an analogous property for μ:

Property 8.5. Let $c : X \to \mathsf{B}\,X$ be a B-coalgebra, then

$$\mu X : \mathsf{T}^\lambda\,(\mathsf{T}^\lambda\,c) \to \mathsf{S}^\lambda\,c : \mathsf{B}\text{-}\mathbf{Coalg}(\mathscr{C}) \ , \quad (8.4)$$

is the lifting of μ to a B-coalgebra homomorphism.

Proof.

$$\begin{aligned}
&\mathsf{T}^\lambda\, c \cdot \mu X \\
=\ & \{\ \text{definition of } \mathsf{T}^\lambda\ (6.4)\ \} \\
&\lambda X \cdot \mathsf{T}\, c \cdot \mu X \\
=\ & \{\ \mu : \mathsf{T}\circ\mathsf{T} \dot{\to} \mathsf{T} \text{ is natural}\ \} \\
&\lambda X \cdot \mu(\mathsf{B}\, X) \cdot \mathsf{T}\,(\mathsf{T}\, c) \\
=\ & \{\ \text{coherence of } \lambda \text{ with } \mu\ (8.2b)\ \} \\
&\mathsf{B}\,(\mu X) \cdot \lambda\,(\mathsf{T}\, X) \cdot \mathsf{T}\,(\lambda X) \cdot \mathsf{T}\,(\mathsf{T}\, c) \\
=\ & \{\ \mathsf{T} \text{ functor and definition of } \mathsf{T}^\lambda\ (6.4)\ \} \\
&\mathsf{B}\,(\mu X) \cdot \mathsf{T}^\lambda\,(\mathsf{T}^\lambda\, c) \quad \square
\end{aligned}$$

In other words, the lifted functor T^λ is a monad as well and we can form $(\mathsf{T}^\lambda, \eta, \mu)$-$\mathbf{Alg}(\mathsf{B}$-$\mathbf{Coalg}(\mathscr{C}))$.

We also have shown that B_λ preserves respect for η (7.4). Again, there is an analogous property for μ:

Property 8.6. The lifted functor B_λ preserves respect for μ.

$$\mathsf{B}_\lambda\, a \cdot \mu(\mathsf{B}\, X) = \mathsf{B}_\lambda\, a \cdot \mathsf{T}\,(\mathsf{B}_\lambda\, a) \iff a \cdot \mu X = a \cdot \mathsf{T}\, a \quad (8.5)$$

Proof.

$$\begin{aligned}
&\mathsf{B}_\lambda\, a \cdot \mu(\mathsf{B}\, X) \\
=\ & \{\ \text{definition of } \mathsf{B}_\lambda\ (6.1)\ \} \\
&\mathsf{B}\, a \cdot \lambda X \cdot \mu(\mathsf{B}\, X) \\
=\ & \{\ \text{coherence of } \lambda \text{ with } \mu\ (8.2b)\ \} \\
&\mathsf{B}\, a \cdot \mathsf{B}\,(\mu X) \cdot \lambda\,(\mathsf{T}\, X) \cdot \mathsf{T}\,(\lambda X) \\
=\ & \{\ \mathsf{B} \text{ functor and assumption } a \cdot \mu X = a \cdot \mathsf{T}\, a\ \} \\
&\mathsf{B}\, a \cdot \mathsf{B}\,(\mathsf{T}\, a) \cdot \lambda\,(\mathsf{T}\, X) \cdot \mathsf{T}\,(\lambda X) \\
=\ & \{\ \lambda : \mathsf{T}\circ\mathsf{B} \dot{\to} \mathsf{B}\circ\mathsf{T} \text{ is natural}\ \} \\
&\mathsf{B}\, a \cdot \lambda X \cdot \mathsf{T}\,(\mathsf{B}\, a) \cdot \mathsf{T}\,(\lambda X) \\
=\ & \{\ \mathsf{T} \text{ functor and definition of } \mathsf{B}_\lambda\ (6.1)\ \} \\
&\mathsf{B}_\lambda\, a \cdot \mathsf{T}\,(\mathsf{B}_\lambda\, a) \quad \square
\end{aligned}$$

Thus, B_λ is an endofunctor on (T, η, μ)-$\mathbf{Alg}(\mathscr{C})$ and we can form B_λ-$\mathbf{Coalg}((\mathsf{T}, \eta, \mu)$-$\mathbf{Alg}(\mathscr{C}))$.

Summary

As before the category of bialgebras can be seen as a category of algebras over coalgebras or as a category of coalgebras over algebras.

$$\lambda\text{-}\mathbf{Bialg}(\mathscr{C}) \cong \begin{cases} (\mathsf{T}^\lambda, \eta, \mu)\text{-}\mathbf{Alg}(\mathsf{B}\text{-}\mathbf{Coalg}(\mathscr{C})) \\ \mathsf{B}_\lambda\text{-}\mathbf{Coalg}((\mathsf{T}, \eta, \mu)\text{-}\mathbf{Alg}(\mathscr{C})) \end{cases} \quad (8.6)$$

8.1 Free Monad

Let $\mathsf{S} : \mathscr{C} \to \mathscr{C}$ be an endofunctor representing our syntax. There is a canonical monad, with pleasant properties, that we can construct from S. To do so we will first define the *free* S-algebra.

The free S-algebra over X is an algebra $\langle \mathsf{M}\, X, \mathit{com}\rangle$ equipped with an arrow $\mathit{var} : X \to \mathsf{M}\, X$. We think of elements of $\mathsf{M}\, X$ as terms built from our syntax functor S and variables drawn from X. There are two ways to construct a term: *var* embeds a variable into a term; and $\mathit{com} : \mathsf{S}\,(\mathsf{M}\, X) \to \mathsf{M}\, X$ constructs a composite term from a level of syntax over subterms.

If we have an algebra $a : \mathsf{S}\, X \to X$, we can evaluate a term with $(\!|a|\!) : \mathsf{M}\, X \to X$ (pronounce "eval"). Given an arrow $g : Y \to X$ to evaluate variables and an S-algebra a to evaluate composites, evaluation of terms is characterized by the uniqueness property,

$$f = (\!|a|\!) \cdot \mathsf{M}\, g \iff f \cdot \mathit{var} = g \ \wedge\ f \cdot \mathit{com} = a \cdot \mathsf{S} f\ , \quad (8.7)$$

for all $f : \mathsf{M}\, Y \to X$. The equivalence states that a compositional evaluation of a term, second conjunct, is uniquely defined by an evaluation of variables, first conjunct. (For the clued-in reader, all of this information comes from the adjunction of the free and forgetful functors between S-$\mathbf{Alg}(\mathscr{C})$ and $\mathscr{C}$.)

The initial algebra emerges as a special case: $\mu\mathsf{S} \cong \mathsf{M}\, 0$. It represents the closed terms. Modulo this isomorphism, we have $\mathit{in} = \mathit{com}_0$ and $(\!|a|\!) = (\!|a|\!) \cdot \mathsf{M}\, !_A$. (Again, this relation is induced by the aforementioned adjunction.)

There are two simple consequences of the uniqueness property. If we set the evaluation of variables to the identity ($g = \mathit{id}$), we get the computation laws:

$$(\!|a|\!) \cdot \mathit{var} = \mathit{id}\ , \quad (8.8a)$$

$$(\!|a|\!) \cdot \mathit{com} = a \cdot \mathsf{S}\,(\!|a|\!)\ . \quad (8.8b)$$

As *var* and *com* are the constructors of terms, we can read these as defining equations of $(\!|-|\!)$. The uniqueness property also implies that *var* and *com* are natural in X and that $(\!|-|\!)$ preserves naturality.

The free monad of the functor S is $\langle \mathsf{M}, \eta, \mu\rangle$, where $\eta = \mathit{var}$ and $\mu = (\!|\mathit{com}|\!)$. The $\mu : \mathsf{M}\circ\mathsf{M} \dot{\to} \mathsf{M}$ of the monad flattens a term whose variables are terms. It does so by evaluating the term with the composite constructor—the action of the free algebra.

Theorem 8.1. *The category of algebras for the free monad of* S *is isomorphic to the category of* S*-algebras:*

$$(\mathsf{M}, \eta, \mu)\text{-}\mathbf{Alg}(\mathscr{C}) \cong \mathsf{S}\text{-}\mathbf{Alg}(\mathscr{C})\ .$$

The following definitions are the witnesses to this isomorphism.

$$\lceil \langle X, a : \mathsf{S}\, X \to X\rangle \rceil = \langle X, (\!|a|\!) : \mathsf{M}\, X \to X\rangle \qquad \lceil h \rceil = h\ , \quad (8.9)$$

$$\lfloor \langle X, b : \mathsf{M}\, X \to X\rangle \rfloor = \langle X, b \cdot \theta X : \mathsf{S}\, X \to X\rangle \qquad \lfloor h \rfloor = h\ , \quad (8.10)$$

where $\theta = \mathit{com} \cdot \mathsf{S}\circ\eta : \mathsf{S} \dot{\to} \mathsf{M}$*, which turns a level of syntax into a term. The map* $\lceil - \rceil$ *preserves and reflects homomorphisms.*

$$h : \lceil a \rceil \to \lceil b \rceil : (\mathsf{M}, \eta, \mu)\text{-}\mathbf{Alg}(\mathscr{C}) \iff h : a \to b : \mathsf{S}\text{-}\mathbf{Alg}(\mathscr{C}) \quad (8.11)$$

Proof. (i) $\lfloor \lceil \langle X, a\rangle \rceil \rfloor = \langle X, a\rangle$:

$$\begin{aligned}
&\lfloor \lceil a \rceil \rfloor \\
=\ & \{\ \text{definitions of } \lceil - \rceil\ (8.9) \text{ and } \lfloor - \rfloor\ (8.10)\ \} \\
&(\!|a|\!) \cdot \mathit{com} \cdot \mathsf{S}\,(\eta X) \\
=\ & \{\ \text{eval computation (8.8b) and } \mathsf{S} \text{ functor}\ \} \\
&a \cdot \mathsf{S}\,((\!|a|\!) \cdot \eta X) \\
=\ & \{\ \text{eval computation (8.8a) and } \mathsf{S} \text{ functor}\ \} \\
&a
\end{aligned}$$

An instance of this property is $\mathit{com} = \lfloor \lceil \mathit{com} \rceil \rfloor = \mu X \cdot \theta(\mathsf{M}\, X)$. In the opposite direction, $\lceil \lfloor \langle X, b\rangle \rfloor \rceil = \langle X, b\rangle$:

$$\begin{aligned}
&\lceil \lfloor b \rfloor \rceil = b \\
\iff\ & \{\ \text{definitions of } \lceil - \rceil\ (8.9) \text{ and } \lfloor - \rfloor\ (8.10)\ \} \\
&(\!|b \cdot \theta X|\!) = b \\
\iff\ & \{\ \text{uniqueness of eval (8.7)}\ \} \\
&b \cdot \eta X = \mathit{id} \ \wedge\ b \cdot \mathit{com} = b \cdot \theta X \cdot \mathsf{S}\, b
\end{aligned}$$

The first conjunct follows from the fact that b respects η (8.3a). For the second conjunct we reason:

$$\begin{aligned}
&b \cdot \theta X \cdot \mathsf{S}\, b \\
=\ & \{\ \theta : \mathsf{S} \dot{\to} \mathsf{M} \text{ is natural}\ \} \\
&b \cdot \mathsf{M}\, b \cdot \theta(\mathsf{M}\, X) \\
=\ & \{\ b \text{ respects } \mu\ (8.3b)\ \}
\end{aligned}$$

$$
\begin{array}{ll}
& b \cdot \mu X \cdot \theta(\mathsf{M}\,X) \\
= & \quad \{\ \mu X \cdot \theta(\mathsf{M}\,X) = \mathit{com}\text{, see above}\ \} \\
& b \cdot \mathit{com}\ .
\end{array}
$$

(ii) $\lfloor - \rfloor$ is functorial as $\theta : \mathsf{S} \dot{\to} \mathsf{M}$ is natural and $\lfloor - \rfloor = \theta$-**Alg**.

(iii) $\lceil - \rceil$ maps S-homomorphisms to M-homomorphisms.

$$
\begin{array}{ll}
& h \cdot (\!|a|\!) = (\!|b|\!) \cdot \mathsf{M}\,h \\
\Longleftrightarrow & \quad \{\ \text{uniqueness of eval (8.7)}\ \} \\
& h \cdot (\!|a|\!) \cdot \eta X = h \ \ \wedge \ \ h \cdot (\!|a|\!) \cdot \mathit{com} = b \cdot \mathsf{S}\,(h \cdot (\!|a|\!))
\end{array}
$$

The first conjunct is a direct consequence of computation (8.8a). For the second conjunct we reason:

$$
\begin{array}{ll}
& h \cdot (\!|a|\!) \cdot \mathit{com} \\
= & \quad \{\ \text{eval computation (8.8b)}\ \} \\
& h \cdot a \cdot \mathsf{S}\,(\!|a|\!) \\
= & \quad \{\ \text{assumption } h : a \to b : \mathsf{S}\text{-}\mathbf{Alg} \text{ and } \mathsf{S} \text{ functor}\ \} \\
& b \cdot \mathsf{S}\,(h \cdot (\!|a|\!))
\end{array}
$$

(iv) Finally, $\lceil \langle A, a \rangle \rceil$ is an algebra for the monad. That $\lceil a \rceil$ respects η (8.3a), unfolds to, $(\!|a|\!) \cdot \eta X = \mathit{id}$, which is the first computation law (8.8a). That $\lceil a \rceil$ respects μ (8.3b), unfolds to, $(\!|a|\!) \cdot \mu X = (\!|a|\!) \cdot \mathsf{M}\,(\!|a|\!)$, and this follows from part (iii)

$$
\begin{array}{ll}
& (\!|a|\!) \cdot \mu X = (\!|a|\!) \cdot \mathsf{M}\,(\!|a|\!) \\
\Longleftarrow & \quad \{\ \lceil - \rceil \text{ maps S- to M-homomorphisms and } \mu = (\!|\mathit{com}|\!)\ \} \\
& (\!|a|\!) \cdot \mathit{com} = a \cdot \mathsf{S}\,(\!|a|\!)
\end{array}
$$

and the second computation law (8.8b). □

8.2 Initial and Final Objects

Now that we have completed another round of generalization, from free pointed functors to free monads, it is appropriate to examine what the new initial and final λ-bialgebras are. Again, they can be derived from the double isomorphism (8.6), and again, we will highlight the salient details.

Superficially, the initial λ-bialgebra has not changed: it remains $\langle \mu\mathsf{S}, \lceil \mathit{in} \rceil, (\!|\lfloor \mathsf{B}^{\lambda}\ \lceil \mathit{in} \rceil \rfloor|\!) \rangle$. What *has* changed are the definitions of $\lceil - \rceil$ and $\lfloor - \rfloor$. The usual three proof obligations are all discharged by the proofs provided in previous section. All of the proof steps have analogues in this section—in particular, Theorem 7.1 has been succeeded by Theorem 8.1.

The final λ-bialgebra is $\langle \nu\mathsf{B}, [\![\mathsf{M}^{\lambda}\ \mathit{out}]\!], \mathit{out} \rangle$; the single change is replacing P^{λ} with M^{λ}. The unique λ-bialgebra homomorphism to the final λ-bialgebra from any λ-bialgebra $\langle X, a, C \rangle$ is still $[\![c]\!]$. Just as in Section 7.2, there is one final proof obligation: we have to show that $[\![\mathsf{M}^{\lambda}\ \mathit{out}]\!]$ is an algebra for M. Previously we showed that $[\![\mathsf{P}^{\lambda}\ \mathit{out}]\!]$ respects η, and this proof suffices to show the same of $[\![\mathsf{M}^{\lambda}\ \mathit{out}]\!]$ (8.3a). It remains to show that μ is respected (8.3b):

$$
\begin{array}{ll}
& [\![\mathsf{M}^{\lambda}\ \mathit{out}]\!] \cdot \mu(\nu\mathsf{B}) = [\![\mathsf{M}^{\lambda}\ \mathit{out}]\!] \cdot \mathsf{M}\,[\![\mathsf{M}^{\lambda}\ \mathit{out}]\!] \\
\Longleftrightarrow & \quad \{\ \text{unfold fusion (§A)} \\
& \qquad \text{with } \mu(\nu\mathsf{B}) : \mathsf{M}^{\lambda}\,(\mathsf{M}^{\lambda}\ \mathit{out}) \to \mathsf{M}^{\lambda}\ \mathit{out}\ (8.4)\ \} \\
& [\![\mathsf{M}^{\lambda}\,(\mathsf{M}^{\lambda}\ \mathit{out})]\!] = [\![\mathsf{M}^{\lambda}\ \mathit{out}]\!] \cdot \mathsf{M}\,[\![\mathsf{M}^{\lambda}\ \mathit{out}]\!] \\
\Longleftarrow & \quad \{\ \text{unfold fusion (§A)}\ \} \\
& \mathsf{M}\,[\![\mathsf{M}^{\lambda}\ \mathit{out}]\!] : \mathsf{M}^{\lambda}\,(\mathsf{M}^{\lambda}\ \mathit{out}) \to \mathsf{M}^{\lambda}\ \mathit{out} \\
\Longleftarrow & \quad \{\ \mathsf{M}^{\lambda} \text{ functor}\ \} \\
& [\![\mathsf{M}^{\lambda}\ \mathit{out}]\!] : \mathsf{M}^{\lambda}\ \mathit{out} \to \mathit{out}\ .
\end{array}
$$

Finally, we can give another statement of the semantic function $\mu\mathsf{S} \to \nu\mathsf{B}$, in the setting of $\lambda : \mathsf{M} \circ \mathsf{B} \dot{\to} \mathsf{B} \circ \mathsf{M}$.

$$
\begin{array}{ccc}
\mathsf{M}\,(\mu\mathsf{S}) & \cdots\cdots\succ & \mathsf{M}\,(\nu\mathsf{B}) \\
{\scriptstyle \lceil \mathit{in} \rceil}\downarrow & & \downarrow{\scriptstyle [\![\mathsf{M}^{\lambda}\ \mathit{out}]\!]} \\
\mu\mathsf{S} & \cdots\cdots\succ & \nu\mathsf{B} \\
{\scriptstyle (\!|\lfloor \mathsf{B}_{\lambda}\ \lceil \mathit{in} \rceil \rfloor|\!)}\downarrow & & \downarrow{\scriptstyle \mathit{out}} \\
\mathsf{B}\,(\mu\mathsf{S}) & \cdots\cdots\succ & \mathsf{B}\,(\nu\mathsf{B})
\end{array}
\qquad (\!|\lfloor [\![\mathsf{M}^{\lambda}\ \mathit{out}]\!] \rfloor|\!) \qquad [\![(\!|\lfloor \mathsf{B}_{\lambda}\ \lceil \mathit{in} \rceil \rfloor|\!)]\!]
$$

We have upgraded pointed functors to monads and Theorem 8.1 ensures that Iniga and Finn still see eye to eye. However, we will need to repeat the exercise of Section 7.3.

8.3 Constructing a Distributive Law

Given a program that is modelled by a natural transformation of type $\rho : \mathsf{S} \circ \mathsf{B} \dot{\to} \mathsf{B} \circ \mathsf{M}$, we seek to derive a distributive law $\lambda : \mathsf{M} \circ \mathsf{B} \dot{\to} \mathsf{B} \circ \mathsf{M}$ such that

$$
c \cdot \lceil a \rceil = \mathsf{B}\,\lceil a \rceil \cdot \lambda X \cdot \mathsf{M}\,c \quad \Longleftrightarrow \quad c \cdot a = \mathsf{B}\,\lceil a \rceil \cdot \rho X \cdot \mathsf{S}\,c\ . \tag{8.12}
$$

Let us calculate.

$$
\begin{array}{ll}
& c \cdot a = \mathsf{B}\,\lceil a \rceil \cdot \rho X \cdot \mathsf{S}\,c \\
\Longleftrightarrow & \quad \{\ \text{isomorphism } (\mathsf{M}, \eta, \mu)\text{-}\mathbf{Alg} \cong \mathsf{S}\text{-}\mathbf{Alg}\ (8.11)\ \} \\
& c \cdot \lceil a \rceil = \lceil \mathsf{B}\,\lceil a \rceil \cdot \rho X \rceil \cdot \mathsf{M}\,c \\
\Longleftrightarrow & \quad \{\ \text{see below}\ \} \\
& c \cdot \lceil a \rceil = \mathsf{B}_{\lambda}\,\lceil a \rceil \cdot \mathsf{M}\,c \\
\Longleftrightarrow & \quad \{\ \text{definition of } \mathsf{B}_{\lambda}\ (6.1)\ \} \\
& c \cdot \lceil a \rceil = \mathsf{B}\,\lceil a \rceil \cdot \lambda X \cdot \mathsf{M}\,c
\end{array}
$$

The specification (8.12) holds if $\mathsf{B}_{\lambda}\,\lceil a \rceil = \lceil \mathsf{B}\,\lceil a \rceil \cdot \rho X \rceil$. To turn this property into a definition for λ, we have to delve a bit deeper into the theory. Applegate [3] discovered that distributive laws $\lambda : \mathsf{M} \circ \mathsf{B} \dot{\to} \mathsf{B} \circ \mathsf{M}$ are in one-to-one correspondence to lifted functors $\bar{\mathsf{B}} : (\mathsf{M}, \eta, \mu)\text{-}\mathbf{Alg} \to (\mathsf{M}, \eta, \mu)\text{-}\mathbf{Alg}$, where a functor $\bar{\mathsf{B}}$ is a lifting of B if its action on carriers and homomorphisms is given by B. It is useful to make explicit what it means for $\bar{\mathsf{B}}$ to preserve algebra homomorphisms (as before, $\bar{\mathsf{B}}\,a$ is synecdochic, see §6.1).

$$
\mathsf{B}\,h \cdot \bar{\mathsf{B}}\,a = \bar{\mathsf{B}}\,b \cdot \mathsf{M}\,(\mathsf{B}\,h) \quad \Longleftarrow \quad h \cdot a = b \cdot \mathsf{M}\,h \tag{8.13}
$$

This property immediately implies that $\bar{\mathsf{B}}$ takes natural algebras of type $\mathsf{M} \circ \mathsf{F} \dot{\to} \mathsf{F}$ to natural algebras of type $\mathsf{M} \circ \mathsf{B} \circ \mathsf{F} \dot{\to} \mathsf{B} \circ \mathsf{F}$.

Looking back, we note that we have already made extensive use of the correspondence in one direction, turning a distributive law into a lifting B_{λ}; now we need the opposite direction. Given a lifting $\bar{\mathsf{B}}$, we can construct a distributive law as follows. The uniqueness property (8.7) states that homomorphisms of type $\mathsf{M}\,X \to A$ are in one-to-one correspondence to arrows of type $X \to A$. We aim to construct $\lambda : \mathsf{M} \circ \mathsf{B} \dot{\to} \mathsf{B} \circ \mathsf{M}$, so we need a natural transformation of type $\mathsf{B} \dot{\to} \mathsf{B} \circ \mathsf{M}$. The composition $\mathsf{B} \circ \eta$ will do nicely. We obtain:

$$
\lambda_{\bar{\mathsf{B}}} = \bar{\mathsf{B}}\,\mu \cdot \mathsf{M} \circ \mathsf{B} \circ \eta\ , \tag{8.14}
$$

where $\bar{\mathsf{B}}\,\mu : \mathsf{M} \circ \mathsf{B} \circ \mathsf{M} \dot{\to} \mathsf{B} \circ \mathsf{M}$ is the M-algebra for the carrier $\mathsf{B} \circ \mathsf{M}$. We must show that $\lambda_{\bar{\mathsf{B}}}$ coheres with η and μ per equations (8.2a) and (8.2b). For the proof we refer to the full paper [11].

The mappings $\lambda \mapsto \mathsf{B}_{\lambda}$ and $\bar{\mathsf{B}} \mapsto \lambda_{\bar{\mathsf{B}}}$ then establish the one-to-one correspondence between distributive laws and lifted functors.

Returning to the task at hand, constructing a distributive law from ρ, we use the property $\mathsf{B}_{\lambda}\,\lceil a \rceil = \lceil \mathsf{B}\,\lceil a \rceil \cdot \rho X \rceil$ to define:

$$
\begin{array}{ll}
\bar{\mathsf{B}}\,\langle X, b : \mathsf{M}\,X \to X \rangle = \langle \mathsf{B}\,X, \lceil \mathsf{B}\,b \cdot \rho X \rceil : \mathsf{M}\,(\mathsf{B}\,X) \to \mathsf{B}\,X \rangle\ , \\
\bar{\mathsf{B}}\,h \qquad\qquad\qquad\qquad = \mathsf{B}\,h\ .
\end{array}
$$

This defines a lifting because $\lceil - \rceil = (-)$ is one that lifts $\mathsf{S}\,(\mathsf{B}\,X) \to \mathsf{B}\,X$ to $\mathsf{M}\,(\mathsf{B}\,X) \to \mathsf{B}\,X$. Putting things together, the distributive law $\lambda = \lambda_{\bar{\mathsf{B}}}$ expressed as a composition of natural transformations is:

$$\lambda = (\mathsf{B} \circ \mu \cdot \rho \circ \mathsf{M}) \cdot \mathsf{M} \circ \mathsf{B} \circ \eta \ . \tag{8.15}$$

8.4 Distributive Laws à la Carte

Distributive laws can be constructed modularly from a system of recursion equations. In this modular development we will have different syntax functors S and we need to construct the free monad for each, so we will replace the notation M by the more informative S^*. The mapping $(-)^*$ is actually a higher-order functor whose arrow part takes a natural transformation $\alpha : \mathsf{S} \dot{\to} \mathsf{T}$ to a natural transformation $\alpha^* : \mathsf{S}^* \dot{\to} \mathsf{T}^*$. Think of α^* as a term converter.

First let us consider an alternative definition of *fib* (cf. §2.2).

$$fib = 0 \prec (1 \prec fib) + fib$$

Note that there is a nested occurrence of $\prec$. We can support nested stream constructors if we embed the behaviour into the syntax. The new syntax-with-behaviour functor is $\mathsf{T}\,X = \mathsf{B}\,X + \mathsf{S}\,X$. Given a system of recursion equations $\rho : \mathsf{S} \circ \mathsf{B} \dot{\to} \mathsf{B} \circ \mathsf{T}^*$, we can construct a symmetric system σ as,

$$\sigma = \mathsf{B} \circ inl^* \cdot \mathsf{B} \circ \theta_{\mathsf{B}} \triangledown \rho = \mathsf{B} \circ (\theta_{\mathsf{T}} \cdot inl) \triangledown \rho : \mathsf{T} \circ \mathsf{B} \dot{\to} \mathsf{B} \circ \mathsf{T}^* \ ,$$

where $inl : \mathsf{B} \dot{\to} \mathsf{T}$ and $\theta_{\mathsf{F}} : \mathsf{F} \dot{\to} \mathsf{F}^*$. A distributive law $\lambda : \mathsf{T}^* \circ \mathsf{B} \dot{\to} \mathsf{B} \circ \mathsf{T}^*$ can then be constructed by Section 8.3.

Embedding behaviour into syntax is a special case of extending a system of recursion equations. Specifically, given a base system $\rho_1 : \mathsf{S}_1 \circ \mathsf{B} \dot{\to} \mathsf{B} \circ \mathsf{S}_1^*$ and an extension $\rho_2 : \mathsf{S}_2 \circ \mathsf{B} \dot{\to} \mathsf{B} \circ \mathsf{S}^*$, where $\mathsf{S}\,X = \mathsf{S}_1\,X + \mathsf{S}_2\,X$, we can form a combined system as follows:

$$\rho = \mathsf{B} \circ inl^* \cdot \rho_1 \triangledown \rho_2 : \mathsf{S} \circ \mathsf{B} \dot{\to} \mathsf{B} \circ \mathsf{S}^* \ .$$

The idea is that ρ_2 can use the operators of S_1 *and* S_2 to define the operators of S_2. Compare this with Example 8.1: we can model *One* and *Plus* with an S_1 and ρ_1, and *Nat* with an S_2 and ρ_2, where *Nat* is defined in terms of itself as well as *One* and *Plus*.

Returning to the embedding of behaviour into syntax, by setting $\mathsf{S}_1 = \mathsf{B}$ and $\rho_1 = \mathsf{B} \circ \theta_{\mathsf{B}}$, the embedding emerges as a special case. A minor variation is the merge of two independent systems of recursion equations,

$$\rho = \mathsf{B} \circ inl^* \cdot \rho_1 \triangledown \mathsf{B} \circ inr^* \cdot \rho_2 \ ,$$

where $\rho_1 : \mathsf{S}_1 \circ \mathsf{B} \dot{\to} \mathsf{B} \circ \mathsf{S}_1^*$ and $\rho_2 : \mathsf{S}_2 \circ \mathsf{B} \dot{\to} \mathsf{B} \circ \mathsf{S}_2^*$. We can further modularize our modelling of Example 8.1 as the recursion equations for *One* and *Plus* are independent. It is clear that we can develop distributive laws modularly: if we have a collection of recursion equations with acyclic dependencies, then we can combine them into a single system using the two techniques described above. In the same fashion as Swierstra's *Data types à la carte* [20], we can create distributive laws à la carte.

There is one final thing to be said on this topic. The embedding of behaviour makes the constructors of B available in the syntax. Often, one also wishes to embed an element of $\nu\mathsf{B}$: consider the equation $x = 0 \prec evenfib' + x$ from Section 2. The stream *evenfib'* is defined by a previous system, in fact, two systems; we wish to reuse it at this point. This can be accommodated by setting $\mathsf{S}_1\,X = \nu\mathsf{B}$ and $\rho_1 = \mathsf{B} \circ com \cdot out$. Here S_1 is a constant functor—elements of $\nu\mathsf{B}$ are embedded as constants. It is important to note that merging the systems for *fib'*, *even* and *x* is *not* an option as *even* uses a different definitional style and, as we have pointed out, we cannot mix styles. Of course, we have to show that *even* is uniquely defined and this is what we do in Section 9.

8.5 Proving the Unique Fixed-Point Principle Correct

Let us now return to our original problem of proving the unique fixed-point principle correct. Also, a brief summary is perhaps not amiss. A system of recursion equations is modelled by a natural transformation $\rho : \mathsf{S} \circ \mathsf{B} \dot{\to} \mathsf{B} \circ \mathsf{S}^*$, where S is the syntax functor and B the behaviour functor. The type of ρ captures the slogan *consume at most one, produce at least one*. Using the trick of embedding behaviour into syntax we can consume nothing (the argument is reassembled on the right) and we can produce more than one. Systems of this form are quite liberal; most, but not all of the examples in the literature satisfy the restrictions. We will get back to this point in Section 9.

A solution of a system modelled by ρ consists of an S-algebra and a B-coalgebra over a common carrier that satisfies:

$$c \cdot a = \mathsf{B}\,\lceil a \rceil \cdot \rho\,X \cdot \mathsf{S}\,c \ .$$

We can now replay the calculations of Section 4. If the coalgebra is final, then a is uniquely determined, which establishes the UFP:

$$\begin{array}{ll} & out \cdot a = \mathsf{B}\,\lceil a \rceil \cdot \rho\,(\nu\mathsf{B}) \cdot \mathsf{S}\,out \\ \iff & \{\ \lambda \text{ given by (8.15) which satisfies (8.12)}\ \} \\ & out \cdot \lceil a \rceil = \mathsf{B}\,\lceil a \rceil \cdot \lambda\,(\nu\mathsf{B}) \cdot \mathsf{M}\,out \\ \iff & \{\ \text{definition of } \mathsf{M}^\lambda \text{ and uniqueness of unfold (3.2)}\ \} \\ & \lceil a \rceil = [\![\mathsf{M}^\lambda\ out]\!] \\ \iff & \{\ \text{isomorphism } (\mathsf{M}, \eta, \mu)\text{-}\mathbf{Alg} \cong \mathsf{S}\text{-}\mathbf{Alg}\ (8.1)\ \} \\ & a = \lfloor [\![\mathsf{M}^\lambda\ out]\!] \rfloor \ . \end{array}$$

Conversely, if the algebra is initial, then c is fixed: $c = (\!|\lfloor \mathsf{B}_\lambda\ \lceil in \rceil \rfloor|\!)$. Since the data defines initial and final objects in $\lambda\text{-}\mathbf{Bialg}(\mathscr{C})$, we can furthermore conclude that the two ways of defining the semantic function of type $\mu\mathsf{S} \to \nu\mathsf{B}$ coincide: $(\!|a|\!) = [\![c]\!]$.

9. Echoes from the Second Dimension

Thus far, we have been living in a single dimension: we have incrementally augmented the syntax functor S, first to P, the free pointed functor of S, and then to M, the free monad of S. A second dimension arises as the dual of the first; just as we replaced S with P, we can do so dually with B and C, the *cofree copointed functor* of B. Of course, the progression continues predictably on to N, the *cofree comonad* of B. The developments of C and N are the duals of Sections 7 and 8, respectively; the details are spelled out in [11].

Let us take a moment to characterize the natural transformations with which we are modelling recursion equations: they take the general form of $\rho : \mathsf{S} \circ lhs \dot{\to} \mathsf{B} \circ rhs$. In Section 6 we took the simplest case, where $lhs = \mathsf{B}$, $rhs = \mathsf{S}$, and thus ρ was exactly our distributive law λ. The duality of the dimensions can be seen in how they affect the expressive power of these natural transformations.

The first dimension, the one we have focused on hitherto, corresponds to the sophistication with which we can build syntax on the right-hand side of equations; the progression first replaced a constructor by a constructor or a variable, and then by terms, nested constructors with variables. This culminated in a ρ where $lhs = \mathsf{B}$, $rhs = \mathsf{M}$, and λ is defined in terms of ρ using the structure of M.

The second dimension corresponds to the left-hand side of equations, and rather than constructing syntax, this is about destructing or patterning matching on behaviour. The *cofree copointed functor* C, defined as $\mathsf{C}\,X = X \times \mathsf{B}\,X$, gives a categorical modelling of Haskell's *as-patterns*, where *var@pat* gives the name *var* to the value being matched by *pat*—B represents a level of behaviour and C gives a label to that level. Therefore, a ρ, where $lhs = \mathsf{C}$, models an equation that consumes at most one, rather than strictly one. This can also be achieved, albeit in an indirect way, by embedding behaviour in syntax, as described in Section 8.4. For example, consider the stream operator that interleaves two streams:

$$interleave\ (Cons\ m\ s)\ (Cons\ n\ t) = m \prec interleave\ (n \prec t)\ s \ .$$

The result of *interleave* $(0\,2\,4\,..)\,(1\,3\,5\,..)$ is $0\,1\,2\,3\,..$, the natural numbers. In this definition we are unnecessarily deconstructing the second parameter into its head and tail, we simply need the whole stream. A more natural definition is:

$$interleave\ (Cons\ m\ s)\ t = m \prec interleave\ t\ s\ \ .$$

Sometimes we want the head, tail, *and* the whole stream. Consider the stream operator that performs an ordered merge of two streams:

$$\begin{aligned} &merge\ s@(Cons\ m\ s')\ t@(Cons\ n\ t') \\ &\quad = \textbf{if}\ m \leqslant n\ \textbf{then}\ m \prec merge\ s'\ t\ \textbf{else}\ n \prec merge\ s\ t'\ \ . \end{aligned}$$

From this we are able to construct a natural transformation $\rho : \mathsf{S}\circ\mathsf{C} \dot{\to} \mathsf{B}\circ\mathsf{S}$ to model *interleave* and *merge* without the need to reconstruct behaviour on the right-hand side.

$$\begin{aligned} &\textbf{data}\ \mathsf{C}\ x = As\ x\ (\mathsf{B}\ x) \\ &\rho\ (Interleave\ (As\ \_\ (Cons\ (m,s)), As\ t\ \_)) \\ &\quad = Cons\ (m, Interleave\ (t,s)) \\ &\rho\ (Merge\ (As\ s\ (Cons\ (m,s')), As\ t\ (Cons\ (n,t')))) \\ &\quad = \textbf{if}\ m \leqslant n\ \textbf{then} \\ &\qquad Cons\ (m, Merge\ (s',t)) \\ &\qquad \textbf{else} \\ &\qquad Cons\ (n,\ Merge\ (t,\ t')) \end{aligned}$$

Finally, the *cofree comonad* permits the inspection of behaviour to an arbitrary depth—unlimited consumption. This is exactly what we need to model the equations that consume more than they produce, such as the stream operator *even*, which we saw in Section 2.

$$even\ (Cons\ m\ (Cons\ n\ u)) = m \prec even\ u$$

We can render the *cofree comonad* in Haskell as,

$$\textbf{data}\ \mathsf{N}\ x = Root\ x\ (\mathsf{B}\ (\mathsf{N}\ x))\ \ ,$$

and *even* is captured by a natural transformation $\rho : \mathsf{S}\circ\mathsf{N} \dot{\to} \mathsf{B}\circ\mathsf{S}$,

$$\begin{aligned} &\rho\ (Even\ (Root\ s\ (Cons\ (m, Root\ t\ (Cons\ (n, Root\ u\ \_))))))) \\ &\quad = Cons\ (m, Even\ u)\ \ . \end{aligned}$$

We can form a distributive law $\lambda : \mathsf{S}\circ\mathsf{N} \dot{\to} \mathsf{N}\circ\mathsf{S}$ from ρ, by following the dual of the derivation outlined in Section 8.3.

There are three points in each dimension, leading to a total of nine different instantiations of $\rho : \mathsf{S}\circ lhs \dot{\to} \mathsf{B}\circ rhs$, which correspond to nine combinations of expressive power. A natural transformation $\rho : \mathsf{S}\circ\mathsf{N} \dot{\to} \mathsf{B}\circ\mathsf{M}$ is the most general: it captures recursion equations that have an arbitrary depth of pattern matching on the left-hand side, with an arbitrary term on the right-hand side. And in some sense it is too general, as it unclear how to derive the corresponding distributive law $\lambda : \mathsf{M}\circ\mathsf{N} \dot{\to} \mathsf{N}\circ\mathsf{M}$, or if there should be such a derivation. We leave this determination as future work.

The sweet spot of expressivity is $\rho : \mathsf{S}\circ\mathsf{C} \dot{\to} \mathsf{B}\circ\mathsf{M}$, where M is the free monad of syntax with embedded behaviour, which captures the slogan mentioned in Section 8.5: *consume at most one, produce at least one*. The use of C makes the nature of the consumption more explicit. Let us showcase this sweet spot.

There is a sequence of numbers called the Hamming numbers, which can be characterized as the numbers that only have 2, 3 or 5 as prime factors. They are named after the Turing award winner Richard Hamming, who posed the problem of generating these numbers in ascending order. Dijkstra [6] presented a solution in SASL, attributed to J. L. A. van de Snepscheut, and proved its correctness. Here we will replicate the same solution, which is in fact a slightly simplified version for numbers that only have 2 and 3 as prime factors. The stream is,

$$ham = 1 \prec merge\ (times\ 2\ ham, times\ 3\ ham)\ \ ,$$

where the definition of *merge* is given above and *times* is,

$$times\ n\ (Cons\ m\ s) = n \times m \prec times\ n\ s\ \ .$$

Again, we can capture the recursion equations *merge*, *times* and *ham* by a natural transformation $\rho : \mathsf{S}\circ\mathsf{C} \dot{\to} \mathsf{B}\circ\mathsf{M}$:

$$\begin{aligned} &\rho\ (Merge\ (As\ s\ (Cons\ (m,s')), As\ t\ (Cons\ (n,t')))) \\ &\quad = \textbf{if}\ m \leqslant n\ \textbf{then} \\ &\qquad Cons\ (m, Com\ (Merge\ (Var\ s', Var\ t))) \\ &\qquad \textbf{else} \\ &\qquad Cons\ (n,\ Com\ (Merge\ (Var\ t,\ \ Var\ t'))) \\ &\rho\ (Times\ n\ (As\ \_\ (Cons\ (m,s)))) \\ &\quad = Cons\ (n \times m, Com\ (Times\ n\ (Var\ s))) \\ &\rho\ Ham \\ &\quad = Cons\ (1, Com\ (Merge\ (Com\ (Times\ 2\ (Com\ Ham)), \\ &\qquad\qquad\qquad\qquad Com\ (Times\ 3\ (Com\ Ham)))))\ \ . \end{aligned}$$

The details of the construction of the distributive law $\lambda : \mathsf{M}\circ\mathsf{C} \dot{\to} \mathsf{C}\circ\mathsf{M}$ can be found in Hinze and James [11]. By the naturality of ρ and thus the constructed λ, we have a proof (as an alternative to Dijkstra's) that *ham* uniquely defines a stream.

10. Related Work

The theoretical foundations of our work exist in the literature, originally in Turi and Plotkin [21] and refined in Lenisa et al. [13]. We see our work as an application of, and an exercise in, this theory.

The work that is closest in spirit to ours is Bartels [4]. It is centered around the coinduction proof principle, in contrast to the UFP. Bartels looks at two out of the nine points that we have identified, the simplest $\lambda : \mathsf{S}\circ\mathsf{B} \dot{\to} \mathsf{B}\circ\mathsf{S}$, and our sweet spot $\lambda : \mathsf{M}\circ\mathsf{C} \dot{\to} \mathsf{C}\circ\mathsf{M}$, but for space reasons does not explore any others. Bartels introduces a construction *homomorphism up-to*, which is a homomorphism from a coalgebra to a bialgebra, and uses it as a definitional principle. We simply use bialgebra homomorphisms, following the original theory of Turi and Plotkin [21], which nicely exhibits the duality of Iniga and Finn's viewpoints.

Rutten and Silva have presented two coinductive calculi, one for streams [17] and one for binary trees [19], also using coinduction as a proof principle. They have a uniqueness proof for each: Theorem 3.1 and Appendix A in Rutten [17]; and Theorem 2 in Silva and Rutten [19]. Our approach treats streams and infinite trees, and *behaviour* in general, in a datatype generic way—the same proofs apply, only varying in the chosen functors for syntax and behaviour. Moreover, we emphasize a compositional, functional style.

Our task of determining that a recursion equation has a unique solution is related to the task of determining that corecursive definitions are productive [18]. This is crucial in dependently typed programming and proof languages, where the logical consistency of the system requires it. In Coq this is enforced by the guardedness condition [8], which is particularly conservative: it has no means to propagate information through function calls, so corecursive calls are forbidden to appear anywhere other than as a direct argument of a constructor. Compositionality is the first casualty. The situation is similar in Agda [2].

Hughes et al. [12] were the first to talk about the notion of *sized-types*, and used it as part of a type-based analysis that guarantees termination and liveness of embedded functional programs. Following this, there have been a whole host of proposed type systems incorporating size annotations. MiniAgda [1, 15] is a tangible implementation of a dependently typed core language with sized types, able to track the productivity of corecursive definitions. Type signatures are mandatory and contain sizes explicitly, which is in contrast to our ρ functions, the naturality of which is easy to infer.

Specific to streams, Endrullis et al. [7] introduce what they call *data-oblivious* productivity: productivity that can be decided without inspecting the stream elements. They present three classes of stream specifications. Their analysis is provably optimal for the *flat* class, where stream functions cannot contain nested function applications. Our slogan "consume at most one, produce at least one"

corresponds to their *friendly nesting* class. A competing approach appears in Zantema [22], who reduces the determination of uniqueness to the termination of a term rewriting system (TRS). A stream specification has a unique solution if its *observational variant* TRS is terminating, a TRS that is very like Rutten's stream definitions.

Acknowledgements

Ralf would like to thank Jan Rutten for pointing him to distributive laws and bialgebras. Daniel is funded by a DTA Studentship from the Engineering and Physical Sciences Research Council.

References

[1] A. Abel. MiniAgda: Integrating Sized and Dependent Types. *Electronic Proceedings in Theoretical Computer Science*, 43:14–28, 2010.

[2] A. Abel and T. Altenkirch. A predicative analysis of structural recursion. *JFP*, 12(1):1–41, 2002.

[3] H. Applegate. *Acyclic models and resolvent functors*. PhD thesis, Columbia University, 1965.

[4] F. Bartels. Generalised coinduction. *Mathematical Structures in Computer Science*, 13(2):321–348, 2003.

[5] R. S. Bird and O. De Moor. *Algebra of Programming*, volume 100 of *International Series in Computing Science*. Prentice Hall, 1997.

[6] E. W. Dijkstra. Hamming's exercise in SASL. Personal Note EWD792, 1981.

[7] J. Endrullis, C. Grabmayer, and D. Hendriks. Data-oblivious stream productivity. In *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 5330 of *LNCS*, pages 79–96. Springer, 2008.

[8] E. Giménez. Codifying guarded definitions with recursive schemes. In *Types for Proofs and Programs*, volume 996 of *LNCS*, pages 39–59. Springer, 1995.

[9] R. Hinze. The Bird tree. *JFP*, 19(5):491–508, 2009.

[10] R. Hinze. Concrete stream calculus—an extended study. *JFP*, 20(5–6):463–535, 2010.

[11] R. Hinze and D. W. H. James. Proving the Unique-Fixed Point Principle Correct. Technical Report RR-11-03, Department of Computer Science, University of Oxford, 2011.

[12] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *POPL*, pages 410–423. ACM, 1996.

[13] M. Lenisa, J. Power, and H. Watanabe. Distributivity for endofunctors, pointed and co-pointed endofunctors, monads and comonads. *Electronic Notes in Theoretical Computer Science*, 33:230–260, 2000.

[14] C. McBride and R. Paterson. Applicative programming with effects. *JFP*, 18(1):1–13, 2008.

[15] K. Mehltretter. Termination checking for a dependently typed language. Master's thesis, LMU Munich, 2007.

[16] M. Niqui and J. J. M. M. Rutten. Sampling, splitting and merging in coinductive stream calculus. In *MPC*, volume 6120 of *LNCS*, pages 310–330. Springer, 2010.

[17] J. J. M. M. Rutten. Fundamental study: Behavioural differential equations: A coinductive calculus of streams, automata, and power series. *Theoretical Computer Science*, 308:1–53, 2003.

[18] B. A. Sijtsma. On the productivity of recursive list definitions. *ACM Trans. Program. Lang. Syst.*, 11(4):633–649, 1989.

[19] A. Silva and J. J. M. M. Rutten. A coinductive calculus of binary trees. *Information and Computation*, 208:578–593, 2010.

[20] W. Swierstra. Data types à la carte. *JFP*, 18(04):423–436, 2008.

[21] D. Turi and G. Plotkin. Towards a mathematical operational semantics. In *Logic in Computer Science*, pages 280–291. IEEE, 1997.

[22] H. Zantema. Well-definedness of streams by transformation and termination. *Logical Methods in Computer Science*, 6(3:21), 2010.

A. Miscellaneous Laws

$$id = inl \triangledown inr \qquad \text{join reflection}$$

$$(g_1 \triangledown g_2) \cdot inl = g_1 \qquad \text{join computation}$$

$$(g_1 \triangledown g_2) \cdot inr = g_2 \qquad \text{join computation}$$

$$k \cdot (g_1 \triangledown g_2) = k \cdot g_1 \triangledown k \cdot g_2 \qquad \text{join fusion}$$

$$(g_1 \triangledown g_2) \cdot (h_1 + h_2) = g_1 \cdot h_1 \triangledown g_2 \cdot h_2 \qquad \text{join functor fusion}$$

$$(\!|a|\!) \cdot in = a \cdot \mathsf{F}\,(\!|a|\!) \qquad \text{fold computation}$$

$$h \cdot (\!|a|\!) = (\!|b|\!) \iff h \cdot a = b \cdot \mathsf{F}\,h \qquad \text{fold fusion}$$

$$[\!|out|\!] = id \qquad \text{unfold reflection}$$

$$[\!|d|\!] = [\!|c|\!] \cdot h \iff \mathsf{F}\,h \cdot d = c \cdot h \qquad \text{unfold fusion}$$

B. Lifting

The underlying or forgetful functor $\mathsf{U} : \mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C}) \to \mathscr{C}$ is defined

$$\mathsf{U}\,\langle A, a\rangle = A \ , \qquad \mathsf{U}\,h = h \ .$$

A functor $\bar{\mathsf{H}} : \mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C}) \to \mathsf{G}\text{-}\mathbf{Alg}(\mathscr{D})$ is a *lifting of* $\mathsf{H} : \mathscr{C} \to \mathscr{D}$ if $\mathsf{U} \circ \bar{\mathsf{H}} = \mathsf{H} \circ \mathsf{U}$.

$$\begin{array}{ccc} \mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C}) & \xrightarrow{\bar{\mathsf{H}}} & \mathsf{G}\text{-}\mathbf{Alg}(\mathscr{D}) \\ \mathsf{U}\downarrow & & \downarrow\mathsf{U} \\ \mathscr{C} & \xrightarrow[\mathsf{H}]{} & \mathscr{D} \end{array}$$

Given a natural transformation $\lambda : \mathsf{G} \circ \mathsf{H} \dot{\to} \mathsf{H} \circ \mathsf{F}$, we can define a lifting $\mathsf{H}_\lambda : \mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C}) \to \mathsf{G}\text{-}\mathbf{Alg}(\mathscr{D})$ of H as follows:

$$\mathsf{H}_\lambda\,\langle X, a : \mathsf{F}\,X \to X\rangle = \langle \mathsf{H}\,X, \mathsf{H}\,a \cdot \lambda\,X : \mathsf{G}\,(\mathsf{H}\,X) \to \mathsf{H}\,X\rangle \ , \quad \text{(B.1)}$$

$$\mathsf{H}_\lambda\,h = \mathsf{H}\,h \ . \quad \text{(B.2)}$$

Since H_λ's action on carriers and homomorphisms is given by H, it preserves identity and composition. It remains to show that it takes F-homomorphisms to G-homomorphisms.

$$\mathsf{H}\,h : \mathsf{H}_\lambda\,a \to \mathsf{H}_\lambda\,b : \mathsf{G}\text{-}\mathbf{Alg}(\mathscr{D}) \iff h : a \to b : \mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C}) \ ,$$

where $a : \mathsf{F}\,X \to X$ and $b : \mathsf{F}\,Y \to Y$. (As throughout this paper, and as explained in Section 6.1, we use lifted functors synecdochically.) We reason

$$\begin{aligned} & \mathsf{H}\,h \cdot \mathsf{H}_\lambda\,a \\ = \ & \{ \text{ definition of } \mathsf{H}_\lambda \text{ (B.1) } \} \\ & \mathsf{H}\,h \cdot \mathsf{H}\,a \cdot \lambda\,X \\ = \ & \{ \ \mathsf{H} \text{ functor and assumption } h : a \to b : \mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C}) \ \} \\ & \mathsf{H}\,b \cdot \mathsf{H}\,(\mathsf{F}\,h) \cdot \lambda\,X \\ = \ & \{ \ \lambda : \mathsf{G} \circ \mathsf{H} \dot{\to} \mathsf{H} \circ \mathsf{F} \text{ is natural } \} \\ & \mathsf{H}\,b \cdot \lambda\,Y \cdot \mathsf{G}\,(\mathsf{H}\,h) \\ = \ & \{ \text{ definition of } \mathsf{H}_\lambda \text{ (B.1) } \} \\ & \mathsf{H}_\lambda\,b \cdot \mathsf{G}\,(\mathsf{H}\,h) \ . \end{aligned}$$

The functor $\alpha\text{-}\mathbf{Alg}$ emerges as a special case with $\mathsf{H} = \mathsf{Id}$ and $\lambda = \alpha$. Also, S_λ is an instance of the scheme with $\mathsf{F} = \mathsf{G}$, which consequently restricts H to endofunctors.

The construction dualizes to categories of coalgebras.