Generic Programming with Adjunctions

Ralf Hinze

Department of Computer Science, University of Oxford
Wolfson Building, Parks Road, Oxford, OX1 3QD, England
`ralf.hinze@cs.ox.ac.uk`
`http://www.cs.ox.ac.uk/ralf.hinze/`

Abstract. Adjunctions are among the most important constructions in mathematics. These lecture notes show they are also highly relevant to datatype-generic programming. First, every fundamental datatype—sums, products, function types, recursive types—arises out of an adjunction. The defining properties of an adjunction give rise to well-known laws of the algebra of programming. Second, adjunctions are instrumental in unifying and generalising recursion schemes. We discuss a multitude of basic adjunctions and show that they are directly relevant to programming and to reasoning about programs.

1 Introduction

Haskell programmers have embraced functors [1], natural transformations [2], monads [3], monoidal functors [4] and, perhaps to a lesser extent, initial algebras [5] and final coalgebras [6]. It is time for them to turn their attention to adjunctions.

The notion of an adjunction was introduced by Daniel Kan in 1958 [7]. Very briefly, the functors L and R are adjoint if arrows of type $\mathsf{L}\,A \to B$ are in one-to-one correspondence to arrows of type $A \to \mathsf{R}\,B$ and if the bijection is furthermore natural in A and B. Adjunctions have proved to be one of the most important ideas in category theory, predominantly due to their ubiquity. Many mathematical constructions turn out to be adjoint functors that form adjunctions, with Mac Lane [8, p.vii] famously saying, "Adjoint functors arise everywhere."

The purpose of these lecture notes is to show that the notion of an adjunction is also highly relevant to programming, in particular, to datatype-generic programming. The concept is relevant in at least two different, but related ways.

First, every fundamental datatype—sums, products, function types, recursive types—arises out of an adjunction. The categorical ingredients of an adjunction correspond to introduction and elimination rules; the defining properties of an adjunction correspond to β-rules, η-rules and fusion laws, which codify basic optimisation principles.

Second, adjunctions are instrumental in unifying and generalising recursion schemes. Historically, the algebra of programming [9] is based on the theory of initial algebras: programs are expressed as folds, and program calculation is based on the universal property of folds. In a nutshell, the universal property formalises that a fold is the unique solution of its defining equation. It implies computation laws and optimisation laws such as fusion. The economy of reasoning is further enhanced by the principle of duality: initial algebras dualise to final coalgebras, and correspondingly folds dualise to unfolds. Two theories for the price of one.

However, all that glitters is not gold. Most, if not all, programs require some tweaking to be given the form of a fold or an unfold and thus make them amenable to formal manipulation. Somewhat ironically, this is in particular true of the "Hello, world!" programs of functional programming: factorial, the Fibonacci function and append. For instance, append does not have the form of a fold as it takes a second argument that is later used in the base case.

In response to this shortcoming a plethora of different recursion schemes has been introduced over the past two decades. Using the concept of an adjunction many of these schemes can be unified and generalised. The resulting scheme is called an adjoint fold. A standard fold insists on the idea that the control structure of a function ever follows the structure of its input data. Adjoint folds loosen this tight coupling—the control structure is given implicitly through the adjunction.

Technically, the central idea is to gain flexibility by allowing the argument of a fold or the result of an unfold to be wrapped up in a functor application. In the case of append, the functor is essentially pairing. Not every functor is admissible: to preserve the salient properties of folds and unfolds, we require the functor to have a right adjoint and, dually, a left adjoint for unfolds. Like folds, adjoint folds are then the unique solutions of their defining equations and, as is to be expected, this dualises to unfolds.

These lecture notes are organised into two major parts. The first part (Section 2) investigates the use of adjunctions for defining 'data structures'. It is partly based on the "Category Theory Primer" distributed at the Spring School [10]. This section includes some background material on category theory, with the aim of making the lecture notes accessible to readers without specialist knowledge.

The second part (Section 3) illustrates the use of adjunctions for giving a precise semantics to 'algorithms'. It is largely based on the forthcoming article "Adjoint Folds and Unfolds—An Extended Study" [11]. Some material has been omitted, some new material has been added (Sections 3.3.3 and 3.4.2); furthermore, all of the examples have been reworked.

The two parts can be read fairly independently. Indeed, on a first reading I recommend to skip to the second part even though it relies on the results of the first one. The development in Section 3 is accompanied by a series of examples in Haskell, which may help in motivating and comprehending the different constructions. The first part then hopefully helps in gaining a deeper understanding of the material.

The notes are complemented by a series of exercises, which can be used to check progress. Some of the exercises form independent threads that introduce more advanced material. As an example, adjunctions are closely related to the

Haskell programmer's favourite toy, monads: every adjunction induces a monad and a comonad; conversely, every (co)monad can be defined by an adjunction. These advanced exercises are marked with a '$*$'.

Enjoy reading!

2 Adjunctions for Data Structures

The first part of these lecture notes is structured as follows. Section 2.1 and 2.2 provide some background to category theory, preparing the ground for the remainder of these lecture notes. Sections 2.3 and 2.4 show how to model non-recursive datatypes, finite products and sums, categorically. We emphasise the calculational properties of the constructions, working carefully towards the central concept of an adjunction, which is then introduced in Section 2.5. This section discusses fundamental properties of adjunctions and illustrates the concept with further examples. In particular, it introduces exponentials, which model higher-order function types. Section 2.6 then shows how to capture recursive datatypes, introducing initial algebras and final coalgebras. Both constructions arise out of an adjunction, related to free algebras and cofree coalgebras, which are perhaps less well-known and which are studied in considerable depth. Finally, Section 2.7 introduces an important categorical tool, the Yoneda Lemma, used repeatedly in the second part of the lecture notes.

2.1 Category, Functor and Natural Transformation

This section introduces the categorical trinity: category, functor and natural transformation. If you are already familiar with the topic, then you can skip the section and the next, except perhaps for notation. If this is unexplored territory, try to absorb the definitions, study the examples and, most importantly, take your time to let the material sink in.

2.1.1 Category. A category consists of objects and arrows between objects. We let $\mathscr{C}$, $\mathscr{D}$ etc range over categories. We write $A : \mathscr{C}$ to express that A is an object of $\mathscr{C}$. We let A, B etc range over objects. For every pair of objects $A, B : \mathscr{C}$ there is a class of arrows from A to B, denoted $\mathscr{C}(A, B)$. If $\mathscr{C}$ is obvious from the context, we abbreviate $f : \mathscr{C}(A, B)$ by $f : A \to B$ or by $f : B \leftarrow A$. We will also loosely speak of $A \to B$ as the type of f. We let f, g etc range over arrows.

For every object $A : \mathscr{C}$ there is an arrow $id_A : A \to A$, called the identity. Two arrows can be composed if their types match: If $f : A \to B$ and $g : B \to C$, then $g \cdot f : A \to C$. We require composition to be associative with identity as its neutral element.

A category is often identified with its class of objects. For instance, we say that **Set** is the category of sets. However, equally, if not more, important are the arrows of a category. So, **Set** is really the category of sets and total functions. (There is also **Rel**, the category of sets and relations.) For **Set**, the identity arrow is the identity function and composition is functional composition. If the objects have additional structure (monoids, groups etc), then the arrows are typically structure-preserving maps.

Exercise 1. Define the category **Mon**, whose objects are monoids and whose arrows are monoid homomorphisms. □

However, the objects of a category are not necessarily sets and the arrows are not necessarily functions:

Exercise 2. A preorder $\precsim$ is an extreme example of a category: $\mathscr{C}(A, B)$ is inhabited if and only if $A \precsim B$. So each $\mathscr{C}(A, B)$ has at most one element. Spell out the details. □

Exercise 3. A monoid is another extreme example of a category: there is exactly one object. Spell out the details. □

A *subcategory* $\mathscr{S}$ of a category $\mathscr{C}$ is a collection of some of the objects and some of the arrows of $\mathscr{C}$, such that identity and composition are preserved to ensure $\mathscr{S}$ constitutes a category. In a full subcategory, $\mathscr{S}(A, B) = \mathscr{C}(A, B)$, for all objects $A, B : \mathscr{S}$.

An arrow $f : A \to B$ is invertible if there is an arrow $g : A \leftarrow B$ with $g \cdot f = id_A$ and $f \cdot g = id_B$. If the inverse arrow exists, it is unique and is written as f°. Two objects A and B are isomorphic, $A \cong B$, if there is an invertible arrow $f : A \to B$. We also write $f : A \cong B : f^\circ$ to express that the arrows $f : A \to B$ and $f^\circ : A \leftarrow B$ witness the isomorphism $A \cong B$.

Exercise 4. Show that the inverse of an arrow is unique. □

2.1.2 Functor.

Every mathematical structure comes equipped with structure-preserving maps; so do categories, where these maps are called *functors*. (Indeed, category theory can be seen as the study of structure-preserving maps. Mac Lane [8, p.30] writes: “Category theory asks of every type of Mathematical object: ‘What are the morphisms?’”) Since a category consists of two parts, objects and arrows, a functor $\mathsf{F} : \mathscr{C} \to \mathscr{D}$ consists of a mapping on objects and a mapping on arrows. It is common practice to denote both mappings by the same symbol. We will also loosely speak of F’s arrow part as a ‘map’. The action on arrows has to respect the types: if $f : \mathscr{C}(A, B)$, then $\mathsf{F} f : \mathscr{D}(\mathsf{F} A, \mathsf{F} B)$. Furthermore, F has to preserve identity and composition:

$$\mathsf{F}\, id_A = id_{\mathsf{F} A} \quad , \tag{1}$$

$$\mathsf{F}\,(g \cdot f) = \mathsf{F}\, g \cdot \mathsf{F} f \quad . \tag{2}$$

The force of functoriality lies in the action on arrows and in the preservation of composition. We let F, G etc range over functors. A functor $\mathsf{F} : \mathscr{C} \to \mathscr{C}$ over a category $\mathscr{C}$ is called an *endofunctor*.

Exercise 5. Show that functors preserve isomorphisms.

$$\mathsf{F}\,f : \mathsf{F}\,A \cong \mathsf{F}\,B : \mathsf{F}\,f^{\circ} \quad \Longleftarrow \quad f : A \cong B : f^{\circ} \square$$

⋆ *Exercise 6.* The category **Mon** has more structure than **Set**. Define a functor $\mathsf{U} : \mathbf{Mon} \to \mathbf{Set}$ that forgets about the additional structure. (The functor U is called the forgetful or underlying functor.) □

There is an identity functor, $\mathsf{Id}_{\mathscr{C}} : \mathscr{C} \to \mathscr{C}$, and functors can be composed: $(\mathsf{G}\circ\mathsf{F})\,A = \mathsf{G}\,(\mathsf{F}\,A)$ and $(\mathsf{G}\circ\mathsf{F})\,f = \mathsf{G}\,(\mathsf{F}\,f)$. This data turns small categories[1] and functors into a category, called **Cat**.

Exercise 7. Show that $\mathsf{Id}_{\mathscr{C}}$ and $\mathsf{G}\circ\mathsf{F}$ are indeed functors. □

2.1.3 Natural Transformation.

Let $\mathsf{F}, \mathsf{G} : \mathscr{C} \to \mathscr{D}$ be two parallel functors. A *transformation* $\alpha : \mathsf{F} \to \mathsf{G}$ is a collection of arrows, so that for each object $A : \mathscr{C}$ there is an arrow $\alpha\,A : \mathscr{D}(\mathsf{F}\,A, \mathsf{G}\,A)$. In other words, a transformation is a mapping from objects to arrows. A transformation is *natural*, $\alpha : \mathsf{F} \dot{\to} \mathsf{G}$, if

$$\mathsf{G}\,h \cdot \alpha\,\hat{A} = \alpha\,\check{A} \cdot \mathsf{F}\,h \ , \tag{3}$$

for all objects $\hat{A}$ and $\check{A}$ and for all arrows $h : \mathscr{C}(\hat{A}, \check{A})$. Note that α is used at two different instances: $\mathscr{D}(\mathsf{F}\,\hat{A}, \mathsf{G}\,\hat{A})$ and $\mathscr{D}(\mathsf{F}\,\check{A}, \mathsf{G}\,\check{A})$—we will adopt the habit of decorating instances with a circumflex (ˆ) and with an inverted circumflex (ˇ). Now, given α and h, there are essentially two ways of turning $\mathsf{F}\,\hat{A}$ things into $\mathsf{G}\,\check{A}$ things. The coherence condition (3) demands that they are equal. The condition is visualised below using a commuting diagram: all paths from the same source to the same target lead to the same result by composition.

$$\begin{tikzcd}
\mathsf{F}\,\hat{A} \arrow[r, "\mathsf{F}\,h"] \arrow[d, "\alpha\,\hat{A}"'] & \mathsf{F}\,\check{A} \arrow[d, "\alpha\,\check{A}"] \\
\mathsf{G}\,\hat{A} \arrow[r, "\mathsf{G}\,h"'] & \mathsf{G}\,\check{A}
\end{tikzcd}$$

We write $\alpha : \mathsf{F} \cong \mathsf{G}$, if α is a natural isomorphism. As an example, the identity is a natural isomorphism of type $\mathsf{F} \cong \mathsf{F}$. We let α, β etc range over natural transformations.

2.2 Opposite, Product and Functor Category

In the previous section we have encountered a few examples of categories. Next, we show how to create new categories from old.

[1] To avoid paradoxes, we have to require that the objects of **Cat** are small, where a category is called small if the class of objects and the class of all arrows are sets. By that token, **Set** and **Cat** are not themselves small.

2.2.1 Opposite Category. Let $\mathscr{C}$ be a category. The *opposite category* $\mathscr{C}^{\mathsf{op}}$ has the same objects as $\mathscr{C}$; the arrows of $\mathscr{C}^{\mathsf{op}}$ are in one-to-one correspondence to the arrows in $\mathscr{C}$, that is, $f^{\mathsf{op}} : \mathscr{C}^{\mathsf{op}}(A, B)$ if and only if $f : \mathscr{C}(B, A)$. Identity and composition are defined flip-wise:

$$id = id^{\mathsf{op}} \qquad \text{and} \qquad f^{\mathsf{op}} \cdot g^{\mathsf{op}} = (g \cdot f)^{\mathsf{op}} \ .$$

A functor of type $\mathscr{C}^{\mathsf{op}} \to \mathscr{D}$ or $\mathscr{C} \to \mathscr{D}^{\mathsf{op}}$ is sometimes called a *contravariant* functor from $\mathscr{C}$ to $\mathscr{D}$, the usual kind being styled *covariant*. The operation $(-)^{\mathsf{op}}$ itself can be extended to a covariant functor $(-)^{\mathsf{op}} : \mathbf{Cat} \to \mathbf{Cat}$, whose arrow part is defined $\mathsf{F}^{\mathsf{op}}\, A = \mathsf{F}\, A$ and $\mathsf{F}^{\mathsf{op}} f^{\mathsf{op}} = (\mathsf{F} f)^{\mathsf{op}}$. We agree that $(f^{\mathsf{op}})^{\mathsf{op}} = f$ so that the operation is an involution. (In later sections, we will often be sloppy and omit the bijection $(-)^{\mathsf{op}}$ on arrows.)

A somewhat incestuous example of a contravariant functor is pre-composition $\mathscr{C}(-, B) : \mathscr{C}^{\mathsf{op}} \to \mathbf{Set}$, whose action on arrows is given by $\mathscr{C}(h^{\mathsf{op}}, B)\, f = f \cdot h$. (Partial applications of mappings and operators are written using 'categorical dummies', where $-$ marks the first and $=$ the second argument if any.) The functor $\mathscr{C}(-, B)$ maps an object A to the *set* of arrows $\mathscr{C}(A, B)$ from A to a fixed B, and it takes an arrow $h^{\mathsf{op}} : \mathscr{C}^{\mathsf{op}}(\hat{A}, \check{A})$ to a *function* $\mathscr{C}(h^{\mathsf{op}}, B) : \mathscr{C}(\hat{A}, B) \to \mathscr{C}(\check{A}, B)$. Dually, post-composition $\mathscr{C}(A, -) : \mathscr{C} \to \mathbf{Set}$ is a covariant functor defined $\mathscr{C}(A, k)\, f = k \cdot f$.

Exercise 8. Show that $\mathscr{C}(A, -)$ and $\mathscr{C}(-, B)$ are functors. □

2.2.2 Product Category. Let $\mathscr{C}_1$ and $\mathscr{C}_2$ be categories. An object of the *product category* $\mathscr{C}_1 \times \mathscr{C}_2$ is a pair $\langle A_1, A_2 \rangle$ of objects $A_1 : \mathscr{C}_1$ and $A_2 : \mathscr{C}_2$; an arrow of $(\mathscr{C}_1 \times \mathscr{C}_2)(\langle A_1, A_2 \rangle, \langle B_1, B_2 \rangle)$ is a pair $\langle f_1, f_2 \rangle$ of arrows $f_1 : \mathscr{C}_1(A_1, B_1)$ and $f_2 : \mathscr{C}_2(A_2, B_2)$. Identity and composition are defined component-wise:

$$id = \langle id, id \rangle \qquad \text{and} \qquad \langle g_1, g_2 \rangle \cdot \langle f_1, f_2 \rangle = \langle g_1 \cdot f_1, g_2 \cdot f_2 \rangle \ .$$

The projection functors $\mathsf{Outl} : \mathscr{C}_1 \times \mathscr{C}_2 \to \mathscr{C}_1$ and $\mathsf{Outr} : \mathscr{C}_1 \times \mathscr{C}_2 \to \mathscr{C}_2$ are given by $\mathsf{Outl}\,\langle A_1, A_2 \rangle = A_1$, $\mathsf{Outl}\,\langle f_1, f_2 \rangle = f_1$ and $\mathsf{Outr}\,\langle A_1, A_2 \rangle = A_2$, $\mathsf{Outr}\,\langle f_1, f_2 \rangle = f_2$. Product categories avoid the need for functors of several arguments. Functors such as Outl and Outr from a product category are sometimes called *bifunctors*. The diagonal functor $\Delta : \mathscr{C} \to \mathscr{C} \times \mathscr{C}$ is an example of a functor into a product category: it duplicates its argument $\Delta A = \langle A, A \rangle$ and $\Delta f = \langle f, f \rangle$.

If we fix one argument of a bifunctor, we obtain a functor. The converse is not true: functoriality in each argument separately does not imply functoriality in both. Rather, we have the following: $(- \otimes =) : \mathscr{C} \times \mathscr{D} \to \mathscr{E}$ is a bifunctor if and only if the partial application $(A \otimes -) : \mathscr{D} \to \mathscr{E}$ is a functor for all $A : \mathscr{C}$, the partial application $(- \otimes B) : \mathscr{C} \to \mathscr{E}$ is a functor for all $B : \mathscr{D}$, *and* if furthermore the two collections of unary functors satisfy the *exchange law*

$$(\check{A} \otimes g) \cdot (f \otimes \hat{B}) = (f \otimes \check{B}) \cdot (\hat{A} \otimes g) \ , \tag{4}$$

for all objects $\hat{A}$, $\check{A}$, $\hat{B}$ and $\check{B}$ and for all arrows $f : \mathscr{C}(\hat{A}, \check{A})$ and $g : \mathscr{D}(\hat{B}, \check{B})$. Given f and g there are two ways of turning $\hat{A} \otimes \hat{B}$ things into $\check{A} \otimes \check{B}$ things:

$$\begin{tikzcd}
\hat{A} \otimes \hat{B} \arrow[r, "\hat{A} \otimes g"] \arrow[d, "f \otimes \hat{B}"'] \arrow[dr, "f \otimes g" sloped] & \hat{A} \otimes \check{B} \arrow[d, "f \otimes \check{B}"] \\
\check{A} \otimes \hat{B} \arrow[r, "\check{A} \otimes g"'] & \check{A} \otimes \check{B} \ .
\end{tikzcd}$$

The coherence condition (4) demands that they are equal. The arrow part of the bifunctor, the diagonal, is then given by either side of (4). The exchange law can also be read as two naturality conditions, stating that $f \otimes -$ and $- \otimes g$ are natural transformations!

Exercise 9. Prove the characterisation of bifunctors. □

The corresponding notion of a 'binatural' transformation is more straightforward. Let $\mathsf{F}, \mathsf{G} : \mathscr{C} \times \mathscr{D} \to \mathscr{E}$ be two parallel functors. The transformation $\alpha : \mathsf{F} \dot{\to} \mathsf{G}$ is natural in both arguments if and only if it is natural in each argument separately.

Exercise 10. Spell out the details and prove the claim. □

We have noted that pre-composition $\mathscr{C}(A, -)$ and post-composition $\mathscr{C}(-, B)$ are functors. Pre-composition commutes with post-composition:

$$\mathscr{C}(\check{A}, g) \cdot \mathscr{C}(f^{\mathsf{op}}, \hat{B}) = \mathscr{C}(f^{\mathsf{op}}, \check{B}) \cdot \mathscr{C}(\hat{A}, g) \ , \qquad (5)$$

for all $f^{\mathsf{op}} : \mathscr{C}^{\mathsf{op}}(\hat{A}, \check{A})$ and $g : \mathscr{C}(\hat{B}, \check{B})$. This is an instance of the exchange law (4), so it follows that the so-called *hom-functor* $\mathscr{C}(-, =) : \mathscr{C}^{\mathsf{op}} \times \mathscr{C} \to \mathbf{Set}$ is a bifunctor. It maps a pair of objects to the set of arrows between them, the so-called hom-set; its action on arrows is given by

$$\mathscr{C}(f^{\mathsf{op}}, g)\, h = g \cdot h \cdot f \ . \qquad (6)$$

2.2.3 Functor Category. There is an identity natural transformation $id_{\mathsf{F}} : \mathsf{F} \dot{\to} \mathsf{F}$ defined $id_{\mathsf{F}}\, A = id_{\mathsf{F}\, A}$. Natural transformations can be composed: if $\alpha : \mathsf{F} \dot{\to} \mathsf{G}$ and $\beta : \mathsf{G} \dot{\to} \mathsf{H}$, then $\beta \cdot \alpha : \mathsf{F} \dot{\to} \mathsf{H}$ is defined $(\beta \cdot \alpha)\, A = \beta\, A \cdot \alpha\, A$. Thus, functors of type $\mathscr{C} \to \mathscr{D}$ and natural transformations between them form a category, the functor category $\mathscr{D}^{\mathscr{C}}$. (Functor categories are exponentials in **Cat**, hence the notation. The next paragraph makes a first step towards proving this fact.)

The application of a functor to an object is itself functorial. Specifically, it is a bifunctor of type $(- =) : \mathscr{D}^{\mathscr{C}} \times \mathscr{C} \to \mathscr{D}$. Using the characterisation of bifunctors, we have to show that $(\mathsf{F}\, -) : \mathscr{C} \to \mathscr{D}$ is a functor for each $\mathsf{F} : \mathscr{D}^{\mathscr{C}}$, that $(-\, A) : \mathscr{D}^{\mathscr{C}} \to \mathscr{D}$ is a functor for each $A : \mathscr{C}$, and that the two collections satisfy the exchange law (4). The former is immediate since $(\mathsf{F}\, -)$ is just F. The arrow part of the latter is $(-\, A)\, \alpha = \alpha\, A$. That this action preserves identity and composition is a consequence of the definition of $\mathscr{D}^{\mathscr{C}}$. Finally, the coherence condition for bifunctors (4) is just the naturality condition (3). (Indeed, one could argue the other way round: the desire to turn functor application into a higher-order functor determines the concept of a natural transformation and in turn the definition of $\mathscr{D}^{\mathscr{C}}$.) For reference, we record that functor application is a bifunctor, whose action on arrows is defined

$$\alpha\, f = \check{\mathsf{F}}\, f \cdot \alpha\, \hat{A} = \alpha\, \check{A} \cdot \hat{\mathsf{F}}\, f \quad . \tag{7}$$

Let $\mathsf{F} : \mathscr{C} \to \mathscr{D}$ be a functor. Pre-composition $-\circ\mathsf{F}$ is itself a functor, one between functor categories $-\circ\mathsf{F} : \mathscr{E}^{\mathscr{D}} \to \mathscr{E}^{\mathscr{C}}$. The action on arrows, that is, natural transformations, is defined $(\alpha\circ\mathsf{F})\, A = \alpha\, (\mathsf{F}\, A)$. Dually, post-composition $\mathsf{F}\circ-$ is a functor of type $\mathsf{F}\circ- : \mathscr{C}^{\mathscr{E}} \to \mathscr{D}^{\mathscr{E}}$ defined $(\mathsf{F}\circ\alpha)\, A = \mathsf{F}\, (\alpha\, A)$.

Exercise 11. Show that $\alpha\circ\mathsf{F}$ and $\mathsf{F}\circ\alpha$ are natural transformations. Prove that $-\circ\mathsf{F}$ and $\mathsf{F}\circ-$ preserve identity and composition. □

Pre-composition commutes with post-composition:

$$(\check{\mathsf{F}}\circ\beta)\cdot(\alpha\circ\hat{\mathsf{G}}) = (\alpha\circ\check{\mathsf{G}})\cdot(\hat{\mathsf{F}}\circ\beta) \quad ,$$

for all $\alpha : \hat{\mathsf{F}} \dot{\to} \check{\mathsf{F}}$ and $\beta : \hat{\mathsf{G}} \dot{\to} \check{\mathsf{G}}$. Again, it follows that functor composition $(-\circ=) : \mathscr{E}^{\mathscr{D}} \times \mathscr{D}^{\mathscr{C}} \to \mathscr{E}^{\mathscr{C}}$ is a bifunctor.

2.3 Product and Coproduct

Definitions in category theory often take the form of universal constructions, a concept we explore in this section. The paradigmatic example of this approach is the definition of products—in fact, this is also historically the first example.

2.3.1 Product.

A *product* of two objects B_1 and B_2 consists of an object written $B_1 \times B_2$ and a pair of arrows $\mathit{outl} : B_1 \times B_2 \to B_1$ and $\mathit{outr} : B_1 \times B_2 \to B_2$. These three things have to satisfy the following *universal property*: for each object A and for each pair of arrows $f_1 : A \to B_1$ and $f_2 : A \to B_2$, there exists a unique arrow $g : A \to B_1 \times B_2$ such that $f_1 = \mathit{outl} \cdot g$ and $f_2 = \mathit{outr} \cdot g$. (The unique arrow is also called the *mediating arrow*).

The universal property can be stated more attractively if we replace the existentially quantified variable g by a Skolem function[2]: for each object A and for each pair of arrows $f_1 : A \to B_1$ and $f_2 : A \to B_2$, there exists an arrow $f_1 \vartriangle f_2 : A \to B_1 \times B_2$ (pronounce "f_1 split f_2") such that

$$f_1 = \mathit{outl} \cdot g \ \wedge\ f_2 = \mathit{outr} \cdot g \quad\Longleftrightarrow\quad f_1 \vartriangle f_2 = g \quad , \tag{8}$$

[2] The existentially quantified variable g is in scope of a universal quantifier, hence the need for a Skolem *function*.

for all $g : A \to B_1 \times B_2$. The equivalence captures the existence of an arrow satisfying the property on the left and furthermore states that $f_1 \vartriangle f_2$ is the unique such arrow. The following diagram summarises the type information.

$$\begin{tikzcd}
 & A \arrow[dl, "f_1"'] \arrow[dr, "f_2"] \arrow[dd, dotted, "f_1 \vartriangle f_2"] & \\
B_1 & & B_2 \\
 & B_1 \times B_2 \arrow[ul, "outl"] \arrow[ur, "outr"'] &
\end{tikzcd}$$

The dotted arrow indicates that $f_1 \vartriangle f_2$ is the unique arrow from A to $B_1 \times B_2$ that makes the diagram commute. Any two products of B_1 and B_2 are isomorphic, which is why we usually speak of *the* product (see Exercise 12). The fact that the definition above determines products only up to isomorphism is a feature, not a bug. A good categorical definition serves as a specification. Think of it as an interface, which may enjoy many different implementations.

A universal property such as (8) has two immediate consequences that are worth singling out. If we substitute the right-hand side into the left-hand side, we obtain the *computation laws* (also known as β-rules):

$$f_1 = outl \cdot (f_1 \vartriangle f_2) \;, \tag{9}$$

$$f_2 = outr \cdot (f_1 \vartriangle f_2) \;. \tag{10}$$

They can be seen as defining equations for the arrow $f \vartriangle g$.

Instantiating g in (8) to the identity $id_{B_1 \times B_2}$ and substituting into the right-hand side, we obtain the *reflection law* (also known as the simple η-rule or η-rule 'light'):

$$outl \vartriangle outr = id_{B_1 \times B_2} \;. \tag{11}$$

The law expresses an extensionality property: taking a product apart and then re-assembling it yields the original.

The universal property enjoys two further consequences, which we shall later identify as naturality properties. The first consequence is the *fusion law* that allows us to fuse a split with an arrow to form another split:

$$(f_1 \vartriangle f_2) \cdot h = f_1 \cdot h \vartriangle f_2 \cdot h \;, \tag{12}$$

for all $h : \hat{A} \to \check{A}$. The law states that $\vartriangle$ is natural in A. For the proof we reason

$$\begin{aligned}
& f_1 \cdot h \vartriangle f_2 \cdot h = (f_1 \vartriangle f_2) \cdot h \\
\iff\; & \{ \text{ universal property (8) } \} \\
& f_1 \cdot h = outl \cdot (f_1 \vartriangle f_2) \cdot h \;\wedge\; f_2 \cdot h = outr \cdot (f_1 \vartriangle f_2) \cdot h \\
\iff\; & \{ \text{ computation (9)–(10) } \} \\
& f_1 \cdot h = f_1 \cdot h \;\wedge\; f_2 \cdot h = f_2 \cdot h \;.
\end{aligned}$$

Exercise 12. Use computation, reflection and fusion to show that any two products of B_1 and B_2 are isomorphic. More precisely, the product of B_1 and B_2 is unique up to a *unique* isomorphism that makes the diagram

$$\begin{tikzcd}
& B_1 \times B_2 \arrow[dl, "\mathit{outl}"'] \arrow[dr, "\mathit{outr}"] \arrow[dd, dotted, shift right, "\cong"'] & \\
B_1 & & B_2 \\
& B_1 \times' B_2 \arrow[ul, "\mathit{outl}'"] \arrow[ur, "\mathit{outr}'"'] \arrow[uu, dotted, shift right] &
\end{tikzcd}$$

commute. (It is *not* the case that there is a unique isomorphism *per se.* For example, there are two isomorphisms between $B \times B$ and $B \times B$: the identity $\mathit{id}_{B\times B} = \mathit{outl} \vartriangle \mathit{outr}$ and $\mathit{outr} \vartriangle \mathit{outl}$.) □

Exercise 13. Show

$$f_1 \vartriangle f_2 = g_1 \vartriangle g_2 \iff f_1 = g_1 \;\wedge\; f_2 = g_2 \ ,$$
$$f = g \iff \mathit{outl} \cdot f = \mathit{outl} \cdot g \;\wedge\; \mathit{outr} \cdot f = \mathit{outr} \cdot g \ .$$

Again, try to use all of the laws above. □

Let us now assume that the product $B_1 \times B_2$ exists for every combination of B_1 and B_2. In this case, the definition of products is also functorial in B_1 and B_2—both objects are totally passive in the description above. We capture this property by turning $\times$ into a functor of type $\mathscr{C} \times \mathscr{C} \to \mathscr{C}$. Indeed, there is a unique way to turn $\times$ into a functor so that the projection arrows, *outl* and *outr*, are natural in B_1 and B_2:

$$k_1 \cdot \mathit{outl} = \mathit{outl} \cdot (k_1 \times k_2) \ , \qquad (13)$$
$$k_2 \cdot \mathit{outr} = \mathit{outr} \cdot (k_1 \times k_2) \ . \qquad (14)$$

We appeal to the universal property

$$\begin{aligned}
& k_1 \cdot \mathit{outl} = \mathit{outl} \cdot (k_1 \times k_2) \;\wedge\; k_2 \cdot \mathit{outr} = \mathit{outr} \cdot (k_1 \times k_2) \\
\iff & \quad \{ \text{ universal property (8) } \} \\
& k_1 \cdot \mathit{outl} \vartriangle k_2 \cdot \mathit{outr} = k_1 \times k_2 \ ,
\end{aligned}$$

which suggests that the arrow part of $\times$ is defined

$$f_1 \times f_2 = f_1 \cdot \mathit{outl} \vartriangle f_2 \cdot \mathit{outr} \ . \qquad (15)$$

We postpone the proof that $\times$ preserves identity and composition.

The *functor fusion law* states that we can fuse a map after a split to form another split:

$$(k_1 \times k_2) \cdot (f_1 \vartriangle f_2) = k_1 \cdot f_1 \vartriangle k_2 \cdot f_2 \ , \qquad (16)$$

for all $k_1 : \hat{B}_1 \to \check{B}_1$ and $k_2 : \hat{B}_2 \to \check{B}_2$. The law formalises that $\vartriangle$ is natural in B_1 and B_2. The proof of (16) builds on fusion and computation:

$$\begin{aligned}
& (k_1 \times k_2) \cdot (f_1 \vartriangle f_2) \\
= \;& \{ \text{ definition of } \times \text{ (15) } \} \\
& (k_1 \cdot \mathit{outl} \vartriangle k_2 \cdot \mathit{outr}) \cdot (f_1 \vartriangle f_2) \\
= \;& \{ \text{ fusion (12) } \} \\
& k_1 \cdot \mathit{outl} \cdot (f_1 \vartriangle f_2) \vartriangle k_2 \cdot \mathit{outr} \cdot (f_1 \vartriangle f_2) \\
= \;& \{ \text{ computation (9)–(10) } \} \\
& k_1 \cdot f_1 \vartriangle k_2 \cdot f_2 \quad .
\end{aligned}$$

Given these prerequisites, it is straightforward to show that $\times$ preserves identity

$$\begin{aligned}
& \mathit{id}_A \times \mathit{id}_B \\
= \;& \{ \text{ definition of } \times \text{ (15) } \} \\
& \mathit{id}_A \cdot \mathit{outl} \vartriangle \mathit{id}_B \cdot \mathit{outr} \\
= \;& \{ \text{ identity and reflection (11) } \} \\
& \mathit{id}_{A \times B}
\end{aligned}$$

and composition

$$\begin{aligned}
& (g_1 \times g_2) \cdot (f_1 \times f_2) \\
= \;& \{ \text{ definition of } \times \text{ (15) } \} \\
& (g_1 \times g_2) \cdot (f_1 \cdot \mathit{outl} \vartriangle f_2 \cdot \mathit{outr}) \\
= \;& \{ \text{ functor fusion (16) } \} \\
& g_1 \cdot f_1 \cdot \mathit{outl} \vartriangle g_2 \cdot f_2 \cdot \mathit{outr} \\
= \;& \{ \text{ definition of } \times \text{ (15) } \} \\
& g_1 \cdot f_1 \times g_2 \cdot f_2 \quad .
\end{aligned}$$

The naturality of $\vartriangle$ can be captured precisely using product categories and hom-functors (we use $\forall X \,.\, \mathsf{F}\,X \to \mathsf{G}\,X$ as a shorthand for $\mathsf{F} \dot{\to} \mathsf{G}$).

$$(\vartriangle) : \forall A, B \,.\, (\mathscr{C} \times \mathscr{C})(\varDelta A, B) \to \mathscr{C}(A, \times B)$$

Split takes a pair of arrows as an argument and delivers an arrow to a product. The object B lives in a product category, so $\times B$ is the product functor applied to B; the object A on the other hand lives in $\mathscr{C}$, the diagonal functor sends it to an object in $\mathscr{C} \times \mathscr{C}$. Do not confuse the diagonal functor $\varDelta$ (the Greek letter Delta) with the mediating arrow $\vartriangle$ (an upwards pointing triangle). The fusion law (12) captures naturality in A,

$$\mathscr{C}(h, \times B) \cdot (\vartriangle) = (\vartriangle) \cdot (\mathscr{C} \times \mathscr{C})(\varDelta h, B) \quad ,$$

and the functor fusion law (16) naturality in B,

$$\mathscr{C}(A, \times k) \cdot (\vartriangle) = (\vartriangle) \cdot (\mathscr{C} \times \mathscr{C})(\varDelta A, k) \quad .$$

The naturality of *outl* and *outr* can be captured using the diagonal functor:

$$\langle outl, outr \rangle : \forall B \,.\, (\mathscr{C} \times \mathscr{C})(\Delta(\times B), B) \enspace .$$

The naturality conditions (13) and (14) amount to

$$k \cdot \langle outl, outr \rangle = \langle outl, outr \rangle \cdot \Delta(\times k) \enspace .$$

The import of all this is that $\times$ is right adjoint to the diagonal functor Δ. We will have to say a lot more about adjoint situations later on (Section 2.5).

Exercise 14. Show $A \times B \cong B \times A$ and $A \times (B \times C) \cong (A \times B) \times C$. □

Exercise 15. What is the difference between $\langle A, B \rangle$ and $A \times B$? □

2.3.2 Coproduct. The construction of products nicely dualises to coproducts, which are products in the opposite category. The *coproduct* of two objects A_1 and A_2 consists of an object written $A_1 + A_2$ and a pair of arrows $inl : A_1 \to A_1 + A_2$ and $inr : A_2 \to A_1 + A_2$. These three things have to satisfy the following *universal property*: for each object B and for each pair of arrows $g_1 : A_1 \to B$ and $g_2 : A_2 \to B$, there exists an arrow $g_1 \triangledown g_2 : A_1 + A_2 \to B$ (pronounce "g_1 join g_2") such that

$$f = g_1 \triangledown g_2 \iff f \cdot inl = g_1 \;\wedge\; f \cdot inr = g_2 \enspace , \tag{17}$$

for all $f : A_1 + A_2 \to B$.

$$\begin{tikzcd} & A_1 + A_2 \arrow[dd, dotted, "g_1 \triangledown g_2"] & \\ A_1 \arrow[ur, "inl"] \arrow[dr, "g_1"'] & & A_2 \arrow[ul, "inr"'] \arrow[dl, "g_2"] \\ & B & \end{tikzcd}$$

As with products, the universal property implies computation, reflection, fusion and functor fusion laws. *Computation laws*:

$$(g_1 \triangledown g_2) \cdot inl = g_1 \enspace , \tag{18}$$
$$(g_1 \triangledown g_2) \cdot inr = g_2 \enspace . \tag{19}$$

Reflection law:

$$id_{A+B} = inl \triangledown inr \enspace . \tag{20}$$

Fusion law:

$$k \cdot (g_1 \triangledown g_2) = k \cdot g_1 \triangledown k \cdot g_2 \enspace . \tag{21}$$

There is a unique way to turn $+$ into a functor so that the injection arrows are natural in A_1 and A_2:

$$(h_1 + h_2) \cdot inl = inl \cdot h_1 \enspace , \tag{22}$$
$$(h_1 + h_2) \cdot inr = inr \cdot h_2 \enspace . \tag{23}$$

The arrow part of the coproduct functor is then given by

$$g_1 + g_2 = \mathit{inl} \cdot g_1 \triangledown \mathit{inr} \cdot g_2 \ . \tag{24}$$

Functor fusion law:

$$(g_1 \triangledown g_2) \cdot (h_1 + h_2) = g_1 \cdot h_1 \triangledown g_2 \cdot h_2 \ . \tag{25}$$

The two fusion laws identify $\triangledown$ as a natural transformation:

$$(\triangledown) : \forall A\, B \,.\, (\mathscr{C} \times \mathscr{C})(A, \Delta B) \to \mathscr{C}(+A, B) \ .$$

The naturality of *inl* and *inr* can be captured as follows.

$$\langle \mathit{inl}, \mathit{inr} \rangle : \forall A \,.\, (\mathscr{C} \times \mathscr{C})(A, \Delta(+A)) \ .$$

The import of all this is that + is left adjoint to the diagonal functor Δ.

2.4 Initial and Final Object

An object A is called initial if for each object $B : \mathscr{C}$ there is exactly one arrow from A to B. Any two initial objects are isomorphic, which is why we usually speak of *the* initial object. It is denoted 0, and the unique arrow from 0 to B is written $0 \dashrightarrow B$ or $B \dashleftarrow 0$.

$$\begin{tikzcd} 0 \arrow[r, dashed, "0 \dashrightarrow B"] & B \end{tikzcd}$$

The uniqueness can also be expressed as a *universal property*:

$$f = 0 \dashrightarrow B \iff \mathit{true} \ , \tag{26}$$

for all $f : 0 \to B$. Instantiating f to the identity id_0, we obtain the *reflection law*: $\mathit{id}_0 = 0 \dashrightarrow 0$. An arrow after a unique arrow can be fused into a single unique arrow.

$$k \cdot (\hat{B} \dashleftarrow 0) = (\check{B} \dashleftarrow 0) \ ,$$

for all $k : \check{B} \leftarrow \hat{B}$. The *fusion law* expresses that $0 \dashrightarrow B$ is natural in B.

Exercise 16. Show that any two initial objects are isomorphic. More precisely, the initial object is unique up to *unique* isomorphism. □

Dually, 1 is a final object if for each object $A : \mathscr{C}$ there is a unique arrow from A to 1, written $A \dashrightarrow 1$ or $1 \dashleftarrow A$.

$$\begin{tikzcd} A \arrow[r, dashed, "A \dashrightarrow 1"] & 1 \end{tikzcd}$$

We have adopted arithmetic notation to denote coproducts and products, initial and final objects. This choice is justified since the constructions satisfy many of the laws of high-school algebra (see Exercise 14). The following exercises ask you to explore the analogy a bit further.

Exercise 17. Show $A + 0 \cong A$ and dually $A \times 1 \cong A$. □

⋆ *Exercise 18.* What about $A \times 0 \cong 0$ and $A \times (B + C) \cong A \times B + A \times C$? □

Exercise 17 suggests that 0 can be seen as a nullary coproduct and, dually, 1 as a nullary product. In general, a category is said to have finite products if it has a final object and binary products.

2.5 Adjunction

We have noted in Section 2.3 that products and coproducts are part of an adjunction. In this section, we explore the notion of an adjunction in depth.

Let $\mathscr{C}$ and $\mathscr{D}$ be categories. The functors $\mathsf{L} : \mathscr{C} \leftarrow \mathscr{D}$ and $\mathsf{R} : \mathscr{C} \rightarrow \mathscr{D}$ are *adjoint*, written $\mathsf{L} \dashv \mathsf{R}$,

$$\begin{tikzcd}\mathscr{C} \arrow[r, phantom, "\perp"] \arrow[r, bend right, "\mathsf{R}"'] & \mathscr{D} \arrow[l, bend right, "\mathsf{L}"']\end{tikzcd}$$

if and only if there is a bijection between the hom-sets

$$\lfloor - \rfloor : \mathscr{C}(\mathsf{L}\,A, B) \cong \mathscr{D}(A, \mathsf{R}\,B) : \lceil - \rceil \ ,$$

that is natural both in A and B. The functor L is said to be a *left adjoint* for R, while R is L's *right adjoint*. The isomorphism $\lfloor - \rfloor$ is called the *left adjunct* with $\lceil - \rceil$ being the *right adjunct*. The notation $\lfloor - \rfloor$ for the left adjunct is chosen as the opening bracket resembles an 'L'. Likewise—but this is admittedly a bit laboured—the opening bracket of $\lceil - \rceil$ can be seen as an angular 'r'. An alternative name for the left adjunct is *adjoint transposition*, which is why $\lfloor f \rfloor$ is commonly called the *transpose* of f (often named f').

That $\lfloor - \rfloor : \mathscr{C}(\mathsf{L}\,A, B) \rightarrow \mathscr{D}(A, \mathsf{R}\,B)$ and $\lceil - \rceil : \mathscr{C}(\mathsf{L}\,A, B) \leftarrow \mathscr{D}(A, \mathsf{R}\,B)$ are mutually inverse can be captured using an equivalence.

$$f = \lceil g \rceil \iff \lfloor f \rfloor = g \tag{27}$$

The equation on the left lives in $\mathscr{C}$, and the equation on the right in $\mathscr{D}$.

As a simple example, the identity functor is self-adjoint: $\mathsf{Id} \dashv \mathsf{Id}$. More generally, if the functor F is invertible, then F is simultaneously a left and a right adjoint: $\mathsf{F} \dashv \mathsf{F}^{\circ} \dashv \mathsf{F}$. (Note that in general $\mathsf{F} \dashv \mathsf{G} \dashv \mathsf{H}$ does not imply $\mathsf{F} \dashv \mathsf{H}$.)

2.5.1 Product and Coproduct Revisited. The equivalence (27) is reminiscent of the universal property of products. That the latter indeed defines an adjunction can be seen more clearly if we re-formulate (8) in terms of the categories involved (again, do not confuse $\mathit{\Delta}$ and $\vartriangle$).

$$f = \langle \mathit{outl}, \ \mathit{outr} \rangle \cdot \mathit{\Delta} g \iff \vartriangle f = g \tag{28}$$

The right part of the diagram below explicates the categories.

$$\begin{tikzcd}\mathscr{C} \arrow[r, phantom, "\perp"] \arrow[r, bend right, "\mathit{\Delta}"'] & \mathscr{C} \times \mathscr{C} \arrow[l, bend right, "+"'] \arrow[r, phantom, "\perp"] \arrow[r, bend right, "\times"'] & \mathscr{C} \arrow[l, bend right, "\mathit{\Delta}"']\end{tikzcd}$$

We actually have a double adjunction with $+$ being left adjoint to $\mathit{\Delta}$. Rewritten in terms of product categories, the universal property of coproducts (17) becomes

$$f = \triangledown g \iff \mathit{\Delta} f \cdot \langle \mathit{inl}, \ \mathit{inr} \rangle = g \ . \tag{29}$$

2.5.2 Initial and Final Object Revisited. Initial object and final object also define an adjunction, though a rather trivial one.

$$\begin{tikzcd} \mathscr{C} \arrow[r, bend right, "\Delta"'] \arrow[r, phantom, "\perp"] & \mathbf{1} \arrow[l, bend right, "0"'] \arrow[r, bend right, "1"'] \arrow[r, phantom, "\perp"] & \mathscr{C} \arrow[l, bend right, "\Delta"'] \end{tikzcd}$$

The category $\mathbf{1}$ consists of a single object $*$ and a single arrow id_*. The diagonal functor is now defined $\Delta A = *$ and $\Delta f = id_*$. The objects 0 and 1 are seen as constant functors from $\mathbf{1}$. (An object $A : \mathscr{C}$ seen as a functor $A : \mathbf{1} \to \mathscr{C}$ maps $*$ to A and id_* to id_A.)

$$f = (0 \rightarrowtail B) \cdot 0\,g \iff \Delta f \cdot id_* = g \tag{30}$$

$$f = id_* \cdot \Delta g \iff 1 f \cdot (1 \rightarrowtail A) = g \tag{31}$$

The universal properties are somewhat degenerated as the right-hand side of (30) and the left-hand side of (31) are vacuously true. Furthermore, $0\,g$ and $1 f$ are both the identity, so (30) simplifies to (26) and (31) simplifies to the equivalence $\mathit{true} \iff A \rightarrowtail 1 = g$.

2.5.3 Counit and Unit. An adjunction can be defined in a variety of ways. Recall that the adjuncts $\lfloor - \rfloor$ and $\lceil - \rceil$ have to be natural both in A and B.

$$\lceil g \rceil \cdot \mathsf{L}\,h = \lceil g \cdot h \rceil$$
$$\mathsf{R}\,k \cdot \lfloor f \rfloor = \lfloor k \cdot f \rfloor$$

This implies $\lceil id \rceil \cdot \mathsf{L}\,h = \lceil h \rceil$ and $\mathsf{R}\,k \cdot \lfloor id \rfloor = \lfloor k \rfloor$. Consequently, the adjuncts are uniquely defined by their images of the identity: $\epsilon = \lceil id \rceil$ and $\eta = \lfloor id \rfloor$. An alternative definition of adjunctions is based on these two natural transformations, which are called the *counit* $\epsilon : \mathsf{L} \circ \mathsf{R} \dot{\to} \mathsf{Id}$ and the *unit* $\eta : \mathsf{Id} \dot{\to} \mathsf{R} \circ \mathsf{L}$ of the adjunction. The units must satisfy the so-called *triangle identities*

$$(\epsilon \circ \mathsf{L}) \cdot (\mathsf{L} \circ \eta) = id_{\mathsf{L}} \quad \text{and} \quad (\mathsf{R} \circ \epsilon) \cdot (\eta \circ \mathsf{R}) = id_{\mathsf{R}} \ . \tag{32}$$

The diagrammatic rendering explains the name triangle identities.

$$\begin{tikzcd} & \mathsf{L}\circ\mathsf{R}\circ\mathsf{L} \arrow[dr, "\epsilon\circ\mathsf{L}"] & \\ \mathsf{L} \arrow[ur, "\mathsf{L}\circ\eta"] \arrow[rr, "id_{\mathsf{L}}"] & & \mathsf{L} \end{tikzcd} \qquad \begin{tikzcd} & \mathsf{R}\circ\mathsf{L}\circ\mathsf{R} \arrow[dr, "\mathsf{R}\circ\epsilon"] & \\ \mathsf{R} \arrow[ur, "\eta\circ\mathsf{R}"] \arrow[rr, "id_{\mathsf{R}}"] & & \mathsf{R} \end{tikzcd}$$

All in all, an adjunction consists of six entities: two functors, two adjuncts, and two units. Every single one of those can be defined in terms of the others:

$$\begin{aligned} \lceil g \rceil &= \epsilon\,B \cdot \mathsf{L}\,g & \epsilon &= \lceil id \rceil & \mathsf{L}\,h &= \lceil \eta\,B \cdot h \rceil \\ \lfloor f \rfloor &= \mathsf{R} f \cdot \eta\,A & \eta &= \lfloor id \rfloor & \mathsf{R}\,k &= \lfloor k \cdot \epsilon\,A \rfloor \ , \end{aligned} \tag{33}$$

for all $f : \mathscr{C}(\mathsf{L}\,A, B)$, $g : \mathscr{D}(A, \mathsf{R}\,B)$, $h : \mathscr{D}(A, B)$ and $k : \mathscr{C}(A, B)$.

Inspecting (28) we note that the counit of the adjunction $\Delta \dashv \times$ is the pair $\langle outl, outr\rangle$ of projection arrows. The unit is the so-called diagonal arrow $\delta = id \vartriangle id$. Dually, equation (29) suggests that the unit of $+ \dashv \Delta$ is the pair $\langle inl, inr\rangle$ of injection arrows. The counit is the so-called codiagonal $id \triangledown id$.

Exercise 19. Show the equivalence of the two ways of defining an adjunction:

1. Assume that an adjunction is given in terms of adjuncts satisfying (27). Show that the units defined $\epsilon = \lceil id \rceil$ and $\eta = \lfloor id \rfloor$ are natural and satisfy the triangle identities (32).
2. Conversely, assume that an adjunction is given in terms of units satisfying the triangle identities (32). Show that the adjuncts defined $\lceil g \rceil = \epsilon\, B \cdot \mathsf{L}\, g$ and $\lfloor f \rfloor = \mathsf{R} f \cdot \eta\, A$ are natural and satisfy the equivalence (27). □

2.5.4 Adjunctions and Programming Languages. In terms of programming language concepts, adjuncts correspond to introduction and elimination rules: split $\vartriangle$ introduces a pair, join $\triangledown$ eliminates a tagged value. The units can be seen as simple variants of these rules: the counit $\langle outl, outr\rangle$ eliminates pairs and the unit $\langle inl, inr\rangle$ introduces tagged values. When we discussed products, we derived a variety of laws from the universal property. Table 1 re-formulates these laws using the new vocabulary. The name of the law is found by identifying the cell in which the law occurs and reading off the label to the left or to the right of the slash. For instance, from the perspective of the right adjoint the identity $f = \lceil \lfloor f \rfloor \rceil$ corresponds to a computation law or β-rule, viewed from the left it is an η-rule.[3] An adjunction typically involves a simple or primitive functor. In our running example, this is the diagonal functor Δ whose adjuncts are the interesting new concepts. It is the new concept that determines the view. Hence we view the equivalence (28) and its consequences from the right and its dual (29) from the left. The table merits careful study.

2.5.5 Universal Arrow Revisited. We looked at universal constructions in Section 2.3. Let us now investigate how this generalises all. Since the components of an adjunction are inter-definable, an adjunction can be specified by providing only part of the data. Surprisingly little is needed: for products only the functor $\mathsf{L} = \Delta$ and the universal arrow $\epsilon = \langle outl, outr\rangle$ were given, the other ingredients were derived from those. In the rest of this section, we replay the derivation in the more abstract setting of adjunctions.

Let $\mathsf{L} : \mathscr{C} \leftarrow \mathscr{D}$ be a functor, let $\mathsf{R} : \mathscr{C} \rightarrow \mathscr{D}$ be an object mapping, and let $\epsilon : \mathscr{C}(\mathsf{L}\,(\mathsf{R}\,B), B)$ be a *universal arrow.* Universality means that for each $f : \mathscr{C}(\mathsf{L}\,A, B)$ there exists a unique arrow $g : \mathscr{D}(A, \mathsf{R}\,B)$ such that $f = \epsilon \cdot \mathsf{L}\,g$. As in Section 2.3.1, we replace the existentially quantified variable g by a skolem function, cunningly written $\lfloor - \rfloor$. Then the statement reads: for each $f : \mathscr{C}(\mathsf{L}\,A, B)$

[3] It is a coincidence that the same Greek letter is used both for extensionality (η-rule) and for the unit of an adjunction.

Table 1. Adjunctions and laws (view from the left / right).

$\lceil - \rceil$ *introduction / elimination* $\lceil - \rceil : \mathscr{D}(A, \mathsf{R}\,B) \to \mathscr{C}(\mathsf{L}\,A, B)$	$\lfloor - \rfloor$ *elimination / introduction* $\lfloor - \rfloor : \mathscr{C}(\mathsf{L}\,A, B) \to \mathscr{D}(A, \mathsf{R}\,B)$
$f : \mathscr{C}(\mathsf{L}\,A, B)$ *universal property* $g : \mathscr{D}(A, \mathsf{R}\,B)$ $f = \lceil g \rceil \iff \lfloor f \rfloor = g$	
$\epsilon : \mathscr{C}(\mathsf{L}\,(\mathsf{R}\,B), B)$ $\epsilon = \lceil id \rceil$	$\eta : \mathscr{D}(A, \mathsf{R}\,(\mathsf{L}\,A))$ $\lfloor id \rfloor = \eta$
— / *computation law* η-rule / β-rule $f = \lceil \lfloor f \rfloor \rceil$	*computation law* / — β-rule / η-rule $\lfloor \lceil g \rceil \rfloor = g$
reflection law / — simple η-rule / simple β-rule $id = \lceil \eta \rceil$	— / *reflection law* simple β-rule / simple η-rule $\lfloor \epsilon \rfloor = id$
functor fusion law / — $\lceil - \rceil$ is natural in A $\lceil g \rceil \cdot \mathsf{L}\,h = \lceil g \cdot h \rceil$	— / *fusion law* $\lfloor - \rfloor$ is natural in A $\lfloor f \rfloor \cdot h = \lfloor f \cdot \mathsf{L}\,h \rfloor$
fusion law / — $\lceil - \rceil$ is natural in B $k \cdot \lceil g \rceil = \lceil \mathsf{R}\,k \cdot g \rceil$	— / *functor fusion law* $\lfloor - \rfloor$ is natural in B $\mathsf{R}\,k \cdot \lfloor f \rfloor = \lfloor k \cdot f \rfloor$
ϵ is natural in B $k \cdot \epsilon = \epsilon \cdot \mathsf{L}\,(\mathsf{R}\,k)$	η is natural in A $\mathsf{R}\,(\mathsf{L}\,h) \cdot \eta = \eta \cdot h$

there exists an arrow $\lfloor f \rfloor : \mathscr{D}(A, \mathsf{R}\,B)$ such that

$$f = \epsilon \cdot \mathsf{L}\,g \iff \lfloor f \rfloor = g \ , \tag{34}$$

for all $g : \mathscr{D}(A, \mathsf{R}\,B)$. The formula suggests that $\epsilon \cdot \mathsf{L}\,g = \lceil g \rceil$. *Computation law:* substituting the right-hand side into the left-hand side, we obtain

$$f = \epsilon \cdot \mathsf{L}\,\lfloor f \rfloor \ . \tag{35}$$

Reflection law: setting $f = \epsilon$ and $g = id$, yields

$$\lfloor \epsilon \rfloor = id \ . \tag{36}$$

Fusion law: to establish

$$\lfloor f \cdot \mathsf{L}\,h \rfloor = \lfloor f \rfloor \cdot h \ , \tag{37}$$

we appeal to the universal property:

$$f \cdot \mathsf{L}\,h = \epsilon \cdot \mathsf{L}\,(\lfloor f \rfloor \cdot h) \iff \lfloor f \cdot \mathsf{L}\,h \rfloor = \lfloor f \rfloor \cdot h \ .$$

To show the left-hand side, we calculate

$$\begin{aligned} & \epsilon \cdot \mathsf{L}\,(\lfloor f \rfloor \cdot h) \\ = \;& \{\ \mathsf{L} \text{ functor } (2)\ \} \\ & \epsilon \cdot \mathsf{L}\,\lfloor f \rfloor \cdot \mathsf{L}\,h \\ = \;& \{\ \text{computation } (35)\ \} \\ & f \cdot \mathsf{L}\,h \enspace . \end{aligned}$$

There is a unique way to turn the object mapping R into a functor so that the counit ϵ is natural in B:

$$k \cdot \epsilon = \epsilon \cdot \mathsf{L}\,(\mathsf{R}\,k) \enspace .$$

We simply appeal to the universal property (34)

$$k \cdot \epsilon = \epsilon \cdot \mathsf{L}\,(\mathsf{R}\,k) \iff \lfloor k \cdot \epsilon \rfloor = \mathsf{R}\,k \enspace ,$$

which suggests to define

$$\mathsf{R}f = \lfloor f \cdot \epsilon \rfloor \enspace . \tag{38}$$

Functor fusion law:

$$\mathsf{R}\,k \cdot \lfloor f \rfloor = \lfloor k \cdot f \rfloor \enspace . \tag{39}$$

For the proof, we reason

$$\begin{aligned} & \mathsf{R}\,k \cdot \lfloor f \rfloor \\ = \;& \{\ \text{definition of } \mathsf{R} \text{ } (38)\ \} \\ & \lfloor k \cdot \epsilon \rfloor \cdot \lfloor f \rfloor \\ = \;& \{\ \text{fusion } (37)\ \} \\ & \lfloor k \cdot \epsilon \cdot \mathsf{L}\,\lfloor f \rfloor \rfloor \\ = \;& \{\ \text{computation } (35)\ \} \\ & \lfloor k \cdot f \rfloor \enspace . \end{aligned}$$

Functoriality: R preserves identity

$$\begin{aligned} & \mathsf{R}\,id \\ = \;& \{\ \text{definition of } \mathsf{R} \text{ } (38)\ \} \\ & \lfloor id \cdot \epsilon \rfloor \\ = \;& \{\ \text{identity and reflection } (36)\ \} \\ & id \end{aligned}$$

and composition

$$\begin{aligned}
& \mathsf{R}\,g \cdot \mathsf{R} f \\
= \; & \{ \text{ definition of } \mathsf{R} \text{ (38) } \} \\
& \mathsf{R}\,g \cdot \lfloor f \cdot \epsilon \rfloor \\
= \; & \{ \text{ functor fusion (39) } \} \\
& \lfloor g \cdot f \cdot \epsilon \rfloor \\
= \; & \{ \text{ definition of } \mathsf{R} \text{ (38) } \} \\
& \mathsf{R}\,(g \cdot f) \;\; .
\end{aligned}$$

Fusion and functor fusion show that $\lfloor - \rfloor$ is natural both in A and in B.

Dually, a functor R and a universal arrow $\eta : \mathscr{C}(A, \mathsf{R}\,(\mathsf{L}\,A))$ are sufficient to form an adjunction.

$$f = \lceil g \rceil \iff \mathsf{R} f \cdot \eta = g \;\; .$$

Define $\lfloor f \rfloor = \mathsf{R} f \cdot \eta$ and $\mathsf{L}\,g = \lceil \eta \cdot g \rceil$.

Exercise 20. Make the relation between naturality and fusion precise. □

2.5.6 Exponential. Let us instantiate the abstract concept of an adjunction to another concrete example. In **Set**, a function of two arguments $A \times X \to B$ can be treated as a function of the first argument $A \to B^X$ whose values are functions of the second argument. In general, the object B^X is called the *exponential* of X and B. An element of B^X is eliminated using application $\mathit{apply} : \mathscr{C}(B^X \times X, B)$. Application is an example of a universal arrow: for each $f : \mathscr{C}(A \times X, B)$ there exists an arrow $\Lambda f : \mathscr{C}(A, B^X)$ (pronounce "curry f") such that

$$f = \mathit{apply} \cdot (g \times \mathit{id}_X) \iff \Lambda f = g \;\; , \tag{40}$$

for all $g : \mathscr{C}(A, B^X)$. The function Λ turns a two argument function into a curried function, hence its name. We recognise an adjoint situation, $- \times X \dashv (-)^X$.

$$\begin{tikzcd} \mathscr{C} \arrow[r, phantom, "\perp"] & \mathscr{C} \arrow[l, bend right, "- \times X"'] \arrow[from=1-1, to=1-2, bend right, "(-)^X"'] \end{tikzcd} \qquad \Lambda : \mathscr{C}(A \times X, B) \cong \mathscr{C}(A, B^X) : \Lambda^\circ$$

The left adjoint is pairing with X, the right adjoint is the exponential from X.

Turning to the laws, since the exponential is right adjoint, we have to view Table 1 from the right. *Computation law*:

$$f = \mathit{apply} \cdot (\Lambda f \times \mathit{id}) \;\; . \tag{41}$$

Reflection law:

$$\Lambda\, \mathit{apply} = \mathit{id} \;\; . \tag{42}$$

Fusion law:

$$\Lambda f \cdot h = \Lambda\,(f \cdot (h \times \mathit{id})) \;\; . \tag{43}$$

There is a unique way to turn $(-)^X$ into a functor so that application is natural in B:

$$k \cdot apply = apply \cdot (k^X \times id) \ . \tag{44}$$

The arrow part of the exponential functor is then given by

$$f^X = \Lambda\,(f \cdot apply) \ . \tag{45}$$

Functor fusion law:

$$k^X \cdot \Lambda f = \Lambda\,(k \cdot f) \ . \tag{46}$$

Exponentials have some extra structure: if all the necessary exponentials exist, then we do not have a single adjunction, but rather a family of adjunctions, $- \times X \dashv (-)^X$, one for each choice of X. The exponential is functorial in the parameter X, and the adjuncts are natural in that parameter. Here are the details:

We already know that $\times$ is a bifunctor. There is a *unique* way to turn the exponential into a bifunctor, necessarily contravariant in its first argument, so that the bijection $\Lambda : \mathscr{C}(A \times X, B) \cong \mathscr{C}(A, B^X) : \Lambda^\circ$ is also natural in X:

$$\begin{tikzcd}
\mathscr{C}(A \times \hat{X}, B) \arrow[r, "\Lambda"] \arrow[d, "\mathscr{C}(A \times p, B)"'] & \mathscr{C}(A, B^{\hat{X}}) \arrow[d, "\mathscr{C}(A, B^p)"] \\
\mathscr{C}(A \times \check{X}, B) \arrow[r, "\Lambda"'] & \mathscr{C}(A, B^{\check{X}})
\end{tikzcd}$$

for all $p : \mathscr{C}(\check{X}, \hat{X})$. We postpone a high-level proof until Section 2.7. For now we construct the bifunctor manually via its partial applications. Fix an object B. The arrow part of the contravariant functor $B^{(-)} : \mathscr{C}^{\mathsf{op}} \to \mathscr{C}$ is given by

$$B^p = \Lambda\,(apply \cdot (id \times p)) \ . \tag{47}$$

Exercise 21. Show that $B^{(-)}$ preserves identity and composition. □

Pre-composition $B^{(-)}$ commutes with post-composition $(-)^A$:

$$g^{\check{A}} \cdot \hat{B}^f = \check{B}^f \cdot g^{\hat{A}} \ , \tag{48}$$

for all $f : \mathscr{C}(\check{A}, \hat{A})$ and $g : \mathscr{C}(\hat{B}, \check{B})$. Consequently, the so-called *internal hom-functor* $(=)^{(-)} : \mathscr{C}^{\mathsf{op}} \times \mathscr{C} \to \mathscr{C}$ is a bifunctor. It maps a pair of objects to their exponential; its action on arrows is given by $g^f = \Lambda\,(g \cdot apply \cdot (id \times f))$.

Since Λ is also natural in X, we have yet another fusion law, the *parameter fusion law*:

$$B^p \cdot \Lambda f = \Lambda\,(f \cdot (A \times p))) \ . \tag{49}$$

We can also merge the three fusion laws into a single law:

$$k^p \cdot \Lambda f \cdot h = \Lambda\,(k \cdot f \cdot (h \times p))) \ .$$

However, it is *not* the case that *apply* is natural in X—its source type $B^X \times X$ is not even functorial in X. (Rather, *apply* is an example of a dinatural transformation [12], see also [8, Exercise IX.4.1].)

As an aside, a category with finite products ($\Delta \dashv 1$ and $\Delta \dashv \times$ exist) and exponentials ($- \times X \dashv (-)^X$ exists for each choice of X) is called *cartesian closed.*

Exercise 22. Show that the contravariant functor $B^{(-)} : \mathscr{C}^{\mathsf{op}} \to \mathscr{C}$ is self-adjoint: $(B^{(-)})^{\mathsf{op}} \dashv B^{(-)}$. □

2.5.7 Power and Copower. A finite product can be formed by nesting binary products: $A_1 \times (A_2 \times (\cdots \times A_n))$. (By definition, the n-ary product is the final object 1 for $n = 0$.) Alternatively, we can generalise products and coproducts to n components (or, indeed, to an infinite number of components).

Central to the double adjunction $+ \dashv \Delta \dashv \times$ is the notion of a product category. The product category $\mathscr{C} \times \mathscr{C}$ can be regarded as a simple functor category: $\mathscr{C}^2$, where 2 is some two-element set. To be able to deal with an arbitrary number of components we generalise from 2 to an arbitrary index set.

A set forms a so-called *discrete category*: the objects are the elements of the set and the only arrows are the identities. Consequently, a functor from a discrete category is uniquely defined by its action on objects. The *category of indexed objects and arrows* $\mathscr{C}^I$, where I is some arbitrary index set, is a functor category from a discrete category: $A : \mathscr{C}^I$ if and only if $\forall i \in I \,.\, A_i : \mathscr{C}$ and $f : \mathscr{C}^I(A, B)$ if and only if $\forall i \in I \,.\, f_i : \mathscr{C}(A_i, B_i)$. The diagonal functor $\Delta : \mathscr{C} \to \mathscr{C}^I$ now sends each index to the same object: $(\Delta A)_i = A$. Left and right adjoints of the diagonal functor generalise the binary constructions. The left adjoint of the diagonal functor is a simple form of a *dependent sum* (also called a dependent product).

$$\mathscr{C}(\textstyle\sum i \in I \,.\, A_i, B) \cong \mathscr{C}^I(A, \Delta B)$$

Its right adjoint is a *dependent product* (also called a dependent function space).

$$\mathscr{C}^I(\Delta A, B) \cong \mathscr{C}(A, \textstyle\prod i \in I \,.\, B_i)$$

The following diagram summarises the type information.

$$\begin{tikzcd}
\mathscr{C} \arrow[r, bend right, "\Delta"'] \arrow[r, phantom, "\perp"] & \mathscr{C}^I \arrow[l, bend right, "\Sigma i \in I \,.\, (-)_i"'] \arrow[r, bend right, "\Pi i \in I \,.\, (-)_i"'] \arrow[r, phantom, "\perp"] & \mathscr{C} \arrow[l, bend right, "\Delta"']
\end{tikzcd}$$

Let us spell out the underlying universal properties. The family of arrows $\iota_k : A_k \to (\Sigma\, i \in I \,.\, A_i)$ generalises the binary injections *inl* and *inr*. For each family of arrows $\forall i \in I \,.\, g_i : A_i \to B$, there exists an arrow $(\bigtriangledown\, i \in I \,.\, g_i) : (\Sigma\, i \in I \,.\, A_i) \to B$ such that

$$f = (\bigtriangledown\, i \in I \,.\, g_i) \iff (\forall i \in I \,.\, f \cdot \iota_i = g_i) \qquad (50)$$

for all $f : (\Sigma\, i \in I \,.\, A_i) \to B$.

Dually, the family $\pi_k : (\mathit{\Pi}\, i \in I \,.\, B_i) \to B_k$ generalises the binary projections *outl* and *outr*. For each family of arrows $\forall\, i \in I \,.\, f_i : A \to B_i$, there exists an arrow $(\bigtriangleup\, i \in I \,.\, f_i) : A \to (\mathit{\Pi}\, i \in I \,.\, B_i)$ such that

$$(\forall\, i \in I \,.\, f_i = \pi_i \cdot g) \iff (\bigtriangleup\, i \in I \,.\, f_i) = g \tag{51}$$

for all $g : A \to (\mathit{\Pi}\, i \in I \,.\, B_i)$.

It is worth singling out a special case of the construction that we shall need later on. First of all, note that $\mathscr{C}^I(\Delta X, \Delta Y) \cong (\mathscr{C}(X, Y))^I \cong I \to \mathscr{C}(X, Y)$. Consequently, if the summands of the sum and the factors of the product are the same, $A_i = X$ and $B_i = Y$, we obtain another adjoint situation:

$$\mathscr{C}(\textstyle\sum I \,.\, X, Y) \cong I \to \mathscr{C}(X, Y) \cong \mathscr{C}(X, \textstyle\prod I \,.\, Y) \ . \tag{52}$$

The degenerated sum $\sum I \,.\, A$ is also called a *copower*, sometimes written $I \bullet A$. The degenerated product $\prod I \,.\, A$ is also called a *power*, sometimes written A^I. In **Set**, we have $\sum I \,.\, A = I \times A$ and $\prod I \,.\, A = I \to A$. (Hence, $\mathit{\Sigma}\, I \dashv \mathit{\Pi}\, I$ is essentially a variant of currying).

2.5.8 Properties of Adjunctions. Adjunctions satisfy a myriad of properties. A property well worth memorising is that both the left and right adjoint of a functor is unique up to natural isomorphism. For the proof assume that the functor $\mathsf{L} : \mathscr{C} \leftarrow \mathscr{D}$ has two right adjoints:

$$\begin{aligned} \lfloor - \rfloor &: \mathscr{C}(\mathsf{L}\, A, B) \cong \mathscr{D}(A, \mathsf{R}\, B) \ , \\ \lfloor - \rfloor' &: \mathscr{C}(\mathsf{L}\, A, B) \cong \mathscr{D}(A, \mathsf{R}'\, B) \ . \end{aligned}$$

The natural isomorphism is given by

$$\lfloor \epsilon \rfloor' : \mathsf{R} \cong \mathsf{R}' : \lfloor \epsilon' \rfloor \ .$$

We show one half of the isomorphism, the proof of the other half proceeds completely analogously (this solves Exercise 12, albeit in the abstract).

$$\begin{aligned} & \lfloor \epsilon \rfloor' \cdot \lfloor \epsilon' \rfloor \\ = \ & \{ \text{ fusion: } \lfloor - \rfloor' \text{ is natural in } A \text{ (Table 1) } \} \\ & \lfloor \epsilon \cdot \mathsf{L}\, \lfloor \epsilon' \rfloor \rfloor' \\ = \ & \{ \text{ computation (35) } \} \\ & \lfloor \epsilon' \rfloor' \\ = \ & \{ \text{ reflection (Table 1) } \} \\ & id \end{aligned}$$

We shall give an application in Section 2.6.5 (where we show that $\mathsf{F}^*\, A \cong \mu \mathsf{F}_A$).

Here is another property worth memorising: left adjoints preserve initial objects and coproducts and, dually, right adjoints preserve final objects and products. (In general, left adjoints preserve so-called colimits and right adjoints preserve so-called limits.) In what follows let $\mathsf{L} : \mathscr{C} \leftarrow \mathscr{D}$ and $\mathsf{R} : \mathscr{C} \to \mathscr{D}$ be an adjoint pair of functors.

A functor $\mathsf{F} : \mathscr{C} \to \mathscr{D}$ preserves initial objects if it takes an initial object in $\mathscr{C}$ to an initial object in $\mathscr{D}$. To prove that the left adjoint L preserves initial objects we show that for each object $B : \mathscr{C}$ there is a unique arrow from $\mathsf{L}\,0$ to B. The required arrow is simply the transpose of the unique arrow to $\mathsf{R}\,B$.

$$\begin{aligned}
& f = \lceil 0 \dashrightarrow \mathsf{R}\,B \rceil \\
\iff & \quad \{ \text{ adjunction: } f = \lceil g \rceil \iff \lfloor f \rfloor = g \ (27) \ \} \\
& \lfloor f \rfloor = 0 \dashrightarrow \mathsf{R}\,B \\
\iff & \quad \{ \ 0 \text{ is initial: universal property } (26) \ \} \\
& \mathit{true}
\end{aligned}$$

Since the initial object is unique up to unique isomorphism (see Exercise 16), we conclude that

$$\mathsf{L}\,0 \cong 0 \ .$$

A functor $\mathsf{F} : \mathscr{C} \to \mathscr{D}$ preserves the product $B_1 \times B_2$ if $\mathsf{F}\,(B_1 \times B_2)$ with $\mathsf{F}\,\mathit{outl} : \mathsf{F}\,(B_1 \times B_2) \to \mathsf{F}\,B_1$ and $\mathsf{F}\,\mathit{outr} : \mathsf{F}\,(B_1 \times B_2) \to \mathsf{F}\,B_2$ is a product of $\mathsf{F}\,B_1$ and $\mathsf{F}\,B_2$. To show that the right adjoint R preserves $B_1 \times B_2$, we establish the universal property of products:

$$\begin{aligned}
& f_1 = \mathsf{R}\,\mathit{outl} \cdot g \ \wedge \ f_2 = \mathsf{R}\,\mathit{outr} \cdot g \\
\iff & \quad \{ \text{ adjunction: } f = \lceil g \rceil \iff \lfloor f \rfloor = g \ (27) \ \} \\
& \lceil f_1 \rceil = \lceil \mathsf{R}\,\mathit{outl} \cdot g \rceil \ \wedge \ \lceil f_2 \rceil = \lceil \mathsf{R}\,\mathit{outr} \cdot g \rceil \\
\iff & \quad \{ \text{ fusion: } \lceil - \rceil \text{ is natural in } B \text{ (Table 1) } \} \\
& \lceil f_1 \rceil = \mathit{outl} \cdot \lceil g \rceil \ \wedge \ \lceil f_2 \rceil = \mathit{outr} \cdot \lceil g \rceil \\
\iff & \quad \{ \ B_1 \times B_2 \text{ is a product: universal property } (8) \ \} \\
& \lceil f_1 \rceil \vartriangle \lceil f_2 \rceil = \lceil g \rceil \\
\iff & \quad \{ \text{ adjunction: } f = \lceil g \rceil \iff \lfloor f \rfloor = g \ (27) \ \} \\
& \lfloor \lceil f_1 \rceil \vartriangle \lceil f_2 \rceil \rfloor = g \ .
\end{aligned}$$

The calculation shows that $\lfloor \lceil f_1 \rceil \vartriangle \lceil f_2 \rceil \rfloor$ is the required mediating arrow, the split of f_1 and f_2. Since the product is unique up to a unique isomorphism relating the projections (see Exercise 12), we have

$$\tau = \mathsf{R}\,\mathit{outl} \vartriangle \mathsf{R}\,\mathit{outr} : \mathsf{R}\,(B_1 \times B_2) \cong \mathsf{R}\,B_1 \times \mathsf{R}\,B_2 \ ,$$

and consequently

$$\mathsf{R}\,\mathit{outl} = \mathit{outl} \cdot \tau \ \wedge \ \mathsf{R}\,\mathit{outr} = \mathit{outr} \cdot \tau \ . \tag{53}$$

To illustrate the 'preservation properties', let us instantiate $\mathsf{L} \dashv \mathsf{R}$ to the 'curry' adjunction $- \times X \dashv (-)^X$. For the left adjoint we obtain familiar looking laws:

$$\begin{aligned}
0 \times X &\cong 0 \ , \\
(A_1 + A_2) \times X &\cong A_1 \times X + A_2 \times X \ .
\end{aligned}$$

These laws are the requirements for a *distributive category* (see also Exercise 17), which demonstrates that a cartesian closed category with finite coproducts is automatically distributive. For the right adjoint we obtain two of the laws of exponentials:

$$1^X \cong 1 \ ,$$
$$(B_1 \times B_2)^X \cong B_1^X \times B_2^X \ .$$

Another interesting example is provided by the adjunction $(B^{(-)})^{\mathsf{op}} \dashv B^{(-)}$ of Exercise 22. Since the self-adjoint functor $B^{(-)}$ is contravariant, it takes the initial object to the final object and coproducts to products:

$$X^0 \cong 1 \ ,$$
$$X^{A_1+A_2} \cong X^{A_1} \times X^{A_2} \ .$$

We obtain two more of the laws of exponentials.

2.6 Initial Algebra and Final Coalgebra

Products model pair types, coproducts model sum types, and exponentials model higher-order function types. In this section we study initial algebras and final coalgebras, which give a meaning to recursively defined types. We shall say a lot more about recursive types and functions over recursive types in the second part of these notes (Section 3).

2.6.1 Initial Algebra.

2.6.1 Initial Algebra. 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 $A : \mathscr{C}$ (the carrier of the algebra) and an arrow $a : \mathscr{C}(\mathsf{F}\,A, A)$ (the action of the algebra). An F-*algebra homomorphism* between algebras $\langle A,\ a\rangle$ and $\langle B,\ b\rangle$ is an arrow $h : \mathscr{C}(A, B)$ such that $h \cdot a = b \cdot \mathsf{F}\,h$. The diagram below illustrates F-algebras and their homomorphisms.

$$\begin{tikzcd}
\mathsf{F}\,A \arrow[d, "a"'] & \mathsf{F}\,A \arrow[r, "\mathsf{F}\,h"] \arrow[d, "a"'] & \mathsf{F}\,B \arrow[d, "b"] & \mathsf{F}\,B \arrow[d, "b"] \\
A & A \arrow[r, "h"'] & B & B
\end{tikzcd}$$

There are two ways to turn $\mathsf{F}\,A$ things into B things; the coherence property for F-algebra homomorphisms demands that they are equal.

Identity is an F-algebra homomorphism and homomorphisms compose. Thus, the data defines a category, called F-**Alg**$(\mathscr{C})$ or just F-**Alg** if the underlying category is obvious from the context. The initial object in this category—if it exists—is the so-called *initial* F-*algebra* $\langle \mu\mathsf{F},\ \mathit{in}\rangle$. The import of initiality is that there is a unique arrow from $\langle \mu\mathsf{F},\, in\rangle$ to any F-algebra $\langle B,\, b\rangle$. This unique arrow is written $(\!|b|\!)$ and is called *fold* or *catamorphism.*[4] Expressed in terms of the base category, it satisfies the following *uniqueness property.*

$$f = (\!|b|\!) \quad\Longleftrightarrow\quad f \cdot in = b \cdot \mathsf{F}f \quad (\Longleftrightarrow\quad f : \langle \mu\mathsf{F},\, in\rangle \to \langle B,\, b\rangle) \tag{54}$$

Similar to products, the uniqueness property has two immediate consequences. Substituting the left-hand side into the right-hand side gives the *computation law*:

$$(\!|b|\!) \cdot in = b \cdot \mathsf{F}\,(\!|b|\!) \quad (\Longleftrightarrow\quad (\!|b|\!) : \langle \mu\mathsf{F},\, in\rangle \to \langle B,\, b\rangle)\ . \tag{55}$$

Setting $f = id$ and $b = in$, we obtain the *reflection law*:

$$id = (\!|in|\!)\ . \tag{56}$$

Since the initial algebra is an initial object, we also have a *fusion law* for fusing an arrow with a fold to form another fold.

$$k \cdot (\!|\hat{b}|\!) = (\!|\check{b}|\!) \quad\Longleftarrow\quad k \cdot \hat{b} = \check{b} \cdot \mathsf{F}\,k \quad (\Longleftrightarrow\quad k : \langle \hat{B},\, \hat{b}\rangle \to \langle \check{B},\, \check{b}\rangle) \tag{57}$$

The proof is trivial if phrased in terms of the category F-$\mathbf{Alg}(\mathscr{C})$. However, we can also execute the proof in the underlying category $\mathscr{C}$.

$$\begin{aligned}
& k \cdot (\!|\hat{b}|\!) = (\!|\check{b}|\!) \\
\Longleftrightarrow\quad & \{\ \text{uniqueness property (54)}\ \} \\
& k \cdot (\!|\hat{b}|\!) \cdot in = \check{b} \cdot \mathsf{F}\,(k \cdot (\!|\hat{b}|\!)\,) \\
\Longleftrightarrow\quad & \{\ \text{computation (55)}\ \} \\
& k \cdot \hat{b} \cdot \mathsf{F}\,(\!|\hat{b}|\!) = \check{b} \cdot \mathsf{F}\,(k \cdot (\!|\hat{b}|\!)\,) \\
\Longleftrightarrow\quad & \{\ \mathsf{F}\ \text{functor (2)}\ \} \\
& k \cdot \hat{b} \cdot \mathsf{F}\,(\!|\hat{b}|\!) = \check{b} \cdot \mathsf{F}\,k \cdot \mathsf{F}\,(\!|\hat{b}|\!) \\
\Longleftarrow\quad & \{\ \text{cancel } - \cdot \mathsf{F}\,(\!|\hat{b}|\!) \text{ on both sides}\ \} \\
& k \cdot \hat{b} = \check{b} \cdot \mathsf{F}\,k\ .
\end{aligned}$$

The fusion law states that $(\!|-|\!)$ is natural in $\langle B,\, b\rangle$, that is, as an arrow in F-$\mathbf{Alg}(\mathscr{C})$. This does *not* imply naturality in the underlying category $\mathscr{C}$. (As an arrow in $\mathscr{C}$ the fold $(\!|-|\!)$ is a strong dinatural transformation.)

Using these laws we can show *Lambek's Lemma* [13], which states that $\mu\mathsf{F}$ is a *fixed point* of the functor: $\mathsf{F}\,(\mu\mathsf{F}) \cong \mu\mathsf{F}$. The isomorphism is witnessed by $in : \mathscr{C}(\mathsf{F}\,(\mu\mathsf{F}), \mu\mathsf{F}) : (\!|\mathsf{F}\,in|\!)$. We calculate

$$\begin{aligned}
& in \cdot (\!|\mathsf{F}\,in|\!) = id \\
\Longleftrightarrow\quad & \{\ \text{reflection (56)}\ \} \\
& in \cdot (\!|\mathsf{F}\,in|\!) = (\!|in|\!) \\
\Longleftarrow\quad & \{\ \text{fusion (57)}\ \} \\
& in \cdot \mathsf{F}\,in = in \cdot \mathsf{F}\,in\ .
\end{aligned}$$

[4] The term catamorphism was coined by Meertens, the notation $(\!|-|\!)$ is due to Malcolm, and the name banana bracket is attributed to Van der Woude.

For the reverse direction, we reason

$$\begin{aligned}
& (\!|\mathsf{F}\,in|\!) \cdot in \\
= \;& \{\ \text{computation (55)}\ \} \\
& \mathsf{F}\,in \cdot \mathsf{F}\,(\!|\mathsf{F}\,in|\!) \\
= \;& \{\ \mathsf{F}\ \text{functor (2)}\ \} \\
& \mathsf{F}\,(in \cdot (\!|\mathsf{F}\,in|\!)) \\
= \;& \{\ \text{see proof above}\ \} \\
& \mathsf{F}\,id \\
= \;& \{\ \mathsf{F}\ \text{functor (1)}\ \} \\
& id \ .
\end{aligned}$$

As an example, $Bush = \mu\mathsf{B}$ where $\mathsf{B}\,A = \mathbb{N} + (A \times A)$ defines the type of binary leaf trees: a tree is either a leaf, labelled with a natural number, or a node consisting of two subtrees. Binary leaf trees can be used to represent non-empty sequences of natural numbers. To define a function that computes the sum of such a sequence, we need to provide an algebra of type $\mathsf{B}\,\mathbb{N} \to \mathbb{N}$. The arrow $id \triangledown plus$ where $plus$ is addition will do nicely. Consequently, the function that computes the total is given by $(\!|id \triangledown plus|\!)$.

Exercise 23. Explore the category $\mathsf{Id}\text{-}\mathbf{Alg}(\mathscr{C})$ where Id is the identity functor. Determine the initial Id-algebra. □

⋆ *Exercise 24.* The inclusion functor $\mathsf{Incl} : \mathscr{C} \to \mathsf{Id}\text{-}\mathbf{Alg}(\mathscr{C})$, defined $\mathsf{Incl}\,A = \langle A, id \rangle$ and $\mathsf{Incl}\,f = f$, embeds the underlying category in the category of Id-algebras. Does Incl have a left or a right adjoint? □

Exercise 25. Explore the category $\mathsf{K}\text{-}\mathbf{Alg}(\mathscr{C})$ where $\mathsf{K}\,A = C$ is the constant functor. Determine the initial K-algebra. □

Exercise 26. Is there such a thing as a *final* F-algebra? □

If all the necessary initial algebras exist, we can turn μ into a higher-order functor of type $\mathscr{C}^{\mathscr{C}} \to \mathscr{C}$. The object part of this functor maps a functor to its initial algebra; the arrow part maps a natural transformation $\alpha : \mathsf{F} \dot{\to} \mathsf{G}$ to an arrow $\mu\alpha : \mathscr{C}(\mu\mathsf{F}, \mu\mathsf{G})$. There is a unique way to define this arrow so that the arrow $in : \mathsf{F}\,(\mu\mathsf{F}) \to \mu\mathsf{F}$ is natural in F:

$$\mu\alpha \cdot in = in \cdot \alpha\,(\mu\alpha) \ . \tag{58}$$

Note that the higher-order functor $\lambda\,\mathsf{F}\ .\ \mathsf{F}\,(\mu\mathsf{F})$, whose action on arrows is $\lambda\,\alpha\ .\ \alpha\,(\mu\alpha) = \lambda\,\alpha\ .\ \alpha\,(\mu\mathsf{G}) \cdot \mathsf{F}\,(\mu\alpha)$, involves the 'application functor' (7). To derive $\mu\alpha$ we simply appeal to the universal property (54):

$$\mu\alpha \cdot in = in \cdot \alpha\,(\mu\mathsf{G}) \cdot \mathsf{F}\,(\mu\alpha) \iff \mu\alpha = (\!|in \cdot \alpha\,(\mu\mathsf{G})|\!) \ .$$

To reduce clutter we will usually omit the type argument of α on the right-hand side and define

$$\mu\alpha = (\!|in \cdot \alpha|\!) \ . \tag{59}$$

As with products, we postpone the proof that μ preserves identity and composition.

Folds enjoy a second fusion law that we christen *base functor fusion* or just *base fusion law*. It states that we can fuse a fold after a map to form another fold:

$$(\!|b \cdot \alpha|\!) = (\!|b|\!) \cdot \mu\alpha \ , \tag{60}$$

for all $\alpha : \hat{\mathsf{F}} \mathrel{\dot{\to}} \check{\mathsf{F}}$. To establish base fusion we reason

$$\begin{aligned}
& (\!|b|\!) \cdot \mu\alpha = (\!|b \cdot \alpha|\!) \\
\iff & \quad \{ \text{ definition of } \mu \text{ (59) } \} \\
& (\!|b|\!) \cdot (\!|in \cdot \alpha|\!) = (\!|b \cdot \alpha|\!) \\
\Longleftarrow & \quad \{ \text{ fusion (57) } \} \\
& (\!|b|\!) \cdot in \cdot \alpha = b \cdot \alpha \cdot \hat{\mathsf{F}}\,(\!|b|\!) \\
\iff & \quad \{ \text{ computation (55) } \} \\
& b \cdot \check{\mathsf{F}}\,(\!|b|\!) \cdot \alpha = b \cdot \alpha \cdot \hat{\mathsf{F}}\,(\!|b|\!) \\
\iff & \quad \{ \ \alpha \text{ is natural: } \check{\mathsf{F}}\,h \cdot \alpha = \alpha \cdot \hat{\mathsf{F}}\,h \ \} \\
& b \cdot \alpha \cdot \hat{\mathsf{F}}\,(\!|b|\!) = b \cdot \alpha \cdot \hat{\mathsf{F}}\,(\!|b|\!) \ .
\end{aligned}$$

Given these prerequisites, it is straightforward to show that μ preserves identity

$$\begin{aligned}
& \mu id \\
= & \quad \{ \text{ definition of } \mu \text{ (59) } \} \\
& (\!|in \cdot id|\!) \\
= & \quad \{ \text{ identity and reflection (56) } \} \\
& id
\end{aligned}$$

and composition

$$\begin{aligned}
& \mu\beta \cdot \mu\alpha \\
= & \quad \{ \text{ definition of } \mu \text{ (59) } \} \\
& (\!|in \cdot \beta|\!) \cdot \mu\alpha \\
= & \quad \{ \text{ base fusion (60) } \} \\
& (\!|in \cdot \beta \cdot \alpha|\!) \\
= & \quad \{ \text{ definition of } \mu \text{ (59) } \} \\
& \mu(\beta \cdot \alpha) \ .
\end{aligned}$$

To summarise, base fusion expresses that $(\!|-|\!)$ is natural in F:

$$(\!|-|\!) : \forall \mathsf{F} \,.\, \mathscr{C}(\mathsf{F}\,B, B) \to \mathscr{C}(\mu\mathsf{F}, B) \ .$$

Note that $\mathscr{C}(-B, B)$ and $\mathscr{C}(\mu -, B)$ are contravariant, higher-order functors of type $\mathscr{C}^{\mathscr{C}} \to \mathbf{Set}^{\mathsf{op}}$.

As an example, $\alpha = \mathit{succ} \triangledown \mathit{id}$ is a natural transformation of type $\mathsf{B} \mathrel{\dot{\to}} \mathsf{B}$. The arrow $\mu\alpha$ increments the labels contained in a binary leaf tree.

2.6.2 Final Coalgebra. The development nicely dualises to F-*coalgebras* and *unfolds*. An F-*coalgebra* is a pair $\langle C,\ c\rangle$ consisting of an object $C : \mathscr{C}$ and an arrow $c : \mathscr{C}(C, \mathsf{F}\,C)$. An F-*coalgebra homomorphism* between coalgebras $\langle C,\ c\rangle$ and $\langle D,\ d\rangle$ is an arrow $h : \mathscr{C}(C, D)$ such that $\mathsf{F}\,h \cdot c = d \cdot h$. Identity is an F-coalgebra homomorphism and homomorphisms compose. Consequently, the data defines a category, called F-**Coalg**($\mathscr{C}$) or just F-**Coalg**. The final object in this category—if it exists—is the so-called *final* F-*coalgebra* $\langle \nu\mathsf{F},\ \mathit{out}\rangle$. The import of finality is that there is a unique arrow to $\langle \nu\mathsf{F},\ \mathit{out}\rangle$ from any F-coalgebra $\langle C,\ c\rangle$. This unique arrow is written $[\![c]\!]$ and is called *unfold* or *anamorphism*. Expressed in terms of the base category, it satisfies the following *uniqueness property*.

$$(g : \langle C,\ c\rangle \to \langle \nu\mathsf{F},\ \mathit{out}\rangle \iff) \quad \mathsf{F}\,g \cdot c = \mathit{out} \cdot g \iff [\![c]\!] = g \tag{61}$$

As with initial algebras, the uniqueness property implies computation, reflection, fusion and base fusion laws. *Computation law*:

$$([\![c]\!] : \langle C,\ c\rangle \to \langle \nu\mathsf{F},\ \mathit{out}\rangle \iff) \quad \mathsf{F}\,[\![c]\!] \cdot c = \mathit{out} \cdot [\![c]\!] \ . \tag{62}$$

Reflection law:

$$[\![\mathit{out}]\!] = \mathit{id} \ . \tag{63}$$

Fusion law:

$$[\![\hat{c}]\!] = [\![\check{c}]\!] \cdot h \impliedby \mathsf{F}\,h \cdot \hat{c} = \check{c} \cdot h \quad (\iff h : \langle \hat{C},\ \hat{c}\rangle \to \langle \check{C},\ \check{c}\rangle) \ . \tag{64}$$

There is a unique way to turn ν into a functor so that out is a natural in F:

$$\alpha\,(\nu\alpha) \cdot \mathit{out} = \mathit{out} \cdot \nu\alpha \ .$$

The arrow part of the functor ν is then given by

$$\nu\alpha = [\![\alpha \cdot \mathit{out}]\!] \ . \tag{65}$$

Base fusion law:

$$\nu\alpha \cdot [\![c]\!] = [\![\alpha \cdot c]\!] \ . \tag{66}$$

As an example, $\mathit{Tree} = \nu\mathsf{T}$ where $\mathsf{T}\,A = A \times \mathbb{N} \times A$ defines the type of bifurcations, infinite binary trees of naturals. (A bifurcation is a division of a state or an action into two branches.) The unfold $\mathit{generate} = [\![\mathit{shift0} \vartriangle \mathit{id} \vartriangle \mathit{shift1}]\!]$, where $\mathit{shift0}\ n = 2 * n + 0$ and $\mathit{shift1}\ n = 2 * n + 1$, generates an infinite tree: $\mathit{generate}\,1$ contains all the positive naturals.

Exercise 27. Explore the category Id-**Coalg**($\mathscr{C}$) where Id is the identity functor. Determine the final Id-coalgebra. □

Exercise 28. Explore the category K-**Coalg**($\mathscr{C}$) where $\mathsf{K}\,A = C$ is the constant functor. Determine the final K-coalgebra. □

Exercise 29. Is there such a thing as an *initial* F-coalgebra? □

2.6.3 Free Algebra and Cofree Coalgebra. We have explained coproducts, products and exponentials in terms of adjunctions. Can we do the same for initial algebras and final coalgebras? Well, the initial algebra is an initial object, hence it is part of a trivial adjunction between $\mathsf{F}\text{-}\mathbf{Alg}$ and $\mathbf{1}$. Likewise, the final coalgebra is a final object giving rise to an adjunction between $\mathbf{1}$ and $\mathsf{F}\text{-}\mathbf{Coalg}$. A more satisfactory answer is provided by the following:

The category $\mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C})$ has more structure than $\mathscr{C}$. The forgetful or underlying functor $\mathsf{U} : \mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C}) \to \mathscr{C}$ forgets about the additional structure: $\mathsf{U}\,\langle A,\ a\rangle = A$ and $\mathsf{U}\,h = h$. An analogous functor can be defined for $\mathsf{F}\text{-}\mathbf{Coalg}(\mathscr{C})$. While the definitions of the forgetful functors are deceptively simple, they give rise to two interesting concepts via two adjunctions.

$$\begin{tikzcd} \mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C}) \arrow[r, bend right, "\mathsf{U}"'] \arrow[r, phantom, "\perp"] & \mathscr{C} \arrow[l, bend right, "\mathsf{Free}"'] \arrow[r, bend right, "\mathsf{Cofree}"'] \arrow[r, phantom, "\perp"] & \mathsf{F}\text{-}\mathbf{Coalg}(\mathscr{C}) \arrow[l, bend right, "\mathsf{U}"'] \end{tikzcd}$$

The functor Free maps an object A to the so-called free F-algebra over A. Dually, Cofree maps an object A to the cofree F-coalgebra over A.

2.6.4 Free Algebra. Let us explore the notion of the free algebra in more depth. First of all, $\mathsf{Free}\,A$ is an F-algebra. We christen its action *com* for reasons to become clear in a moment. In $\mathbf{Set}$, the elements of $\mathsf{U}\,(\mathsf{Free}\,A)$ are terms built from constructors determined by F and variables drawn from A. Think of the functor F as a grammar describing the syntax of a language. The action $\mathit{com} : \mathscr{C}(\mathsf{F}\,(\mathsf{U}\,(\mathsf{Free}\,A)), \mathsf{U}\,(\mathsf{Free}\,A))$ constructs a *com*posite term from an F-structure of subterms. There is also an operation $\mathit{var} : \mathscr{C}(A, \mathsf{U}\,(\mathsf{Free}\,A))$ for embedding a *var*iable into a term. This operation is a further example of a universal arrow: for each $g : \mathscr{C}(A, \mathsf{U}\,B)$ there exists an F-algebra homomorphism $\mathit{eval}\,g : \mathsf{F}\text{-}\mathbf{Alg}(\mathsf{Free}\,A, B)$ (pronounce "*eval*uate with g") such that

$$f = \mathit{eval}\,g \iff \mathsf{U}\,f \cdot \mathit{var} = g \ , \tag{67}$$

for all $f : \mathsf{F}\text{-}\mathbf{Alg}(\mathsf{Free}\,A, B)$. In words, the meaning of a term is uniquely determined by the meaning of the variables. The fact that $\mathit{eval}\,g$ is a homomorphism entails that the meaning function is compositional: the meaning of a composite term is defined in terms of the meanings of its constituent parts.

The universal property implies the usual smörgåsbord of laws. Even though U's action on arrows is a no-op, $\mathsf{U}\,h = h$, we shall not omit applications of U, because it provides valuable 'type information': it makes precise that h is an F-algebra homomorphism, not just an arrow in $\mathscr{C}$.

Computation law:

$$\mathsf{U}\,(\mathit{eval}\,g) \cdot \mathit{var} = g \ . \tag{68}$$

Reflection law:

$$\mathit{id} = \mathit{eval}\,\mathit{var} \ . \tag{69}$$

Fusion law:

$$k \cdot \mathit{eval}\,g = \mathit{eval}\,(\mathsf{U}\,k \cdot g) \ . \tag{70}$$

Note that the computation law lives in the underlying category $\mathscr{C}$, whereas the reflection law and the fusion law live in $\mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C})$.

As usual, there is a unique way to turn Free into a functor so that the unit *var* is natural in A:

$$\mathsf{U}\,(\mathsf{Free}\,h) \cdot \mathit{var} = \mathit{var} \cdot h \quad . \tag{71}$$

The arrow part of the functor Free is then given by

$$\mathsf{Free}\,g = \mathit{eval}\,(\mathit{var} \cdot g) \quad . \tag{72}$$

Functor fusion law:

$$\mathit{eval}\,g \cdot \mathsf{Free}\,h = \mathit{eval}\,(g \cdot h) \quad . \tag{73}$$

The algebra $\mathit{com} : \mathscr{C}(\mathsf{F}\,(\mathsf{U}\,(\mathsf{Free}\,A)), \mathsf{U}\,(\mathsf{Free}\,A))$ is also natural in A.

$$\mathsf{U}\,(\mathsf{Free}\,h) \cdot \mathit{com} = \mathit{com} \cdot \mathsf{F}\,(\mathsf{U}\,(\mathsf{Free}\,h)) \quad . \tag{74}$$

This is just a reformulation of the fact that $\mathsf{Free}\,h : \mathsf{Free}\,A \to \mathsf{Free}\,B$ is an F-algebra homomorphism.

As an example, the free algebra of the squaring functor $\mathsf{Sq}\,A = A \times A$ generalises the type of binary leaf trees abstracting away from the type of natural numbers: *var* creates a leaf and *com* an inner node (see also Example 21 and Exercise 42).

$\ast$ *Exercise 30.* Every adjunction $\mathsf{L} \dashv \mathsf{R}$ gives rise to a monad $\mathsf{R}\circ\mathsf{L}$. This exercise asks you to explore $\mathsf{M} = \mathsf{U}\circ\mathsf{Free}$, the so-called *free monad* of the functor F. The unit of the monad is $\mathit{var} : \mathsf{Id} \dot{\to} \mathsf{M}$, which embeds a variable into a term. The multiplication of the monad, $\mathit{join} : \mathsf{M}\circ\mathsf{M} \dot{\to} \mathsf{M}$, implements substitution. Define *join* using *eval* and prove the monad laws ($\circ$ binds more tightly than $\cdot$):

$$\begin{aligned} \mathit{join} \cdot \mathit{var}\circ\mathsf{M} &= \mathit{id}_{\mathsf{M}} \quad , \\ \mathit{join} \cdot \mathsf{M}\circ\mathit{var} &= \mathit{id}_{\mathsf{M}} \quad , \\ \mathit{join} \cdot \mathit{join}\circ\mathsf{M} &= \mathit{join} \cdot \mathsf{M}\circ\mathit{join} \quad . \end{aligned}$$

The laws capture fundamental properties of substitution. Explain. □

Exercise 31. What is the free monad of the functor Id? Is the free monad of the constant functor $\mathsf{K}\,A = C$ useful? □

$\ast$ *Exercise 32.* Exercise 6 asked you to define the forgetful functor $\mathsf{U} : \mathbf{Mon} \to \mathbf{Set}$ that forgets about the additional structure of $\mathbf{Mon}$. Show that the functor $\mathsf{Free} : \mathbf{Set} \to \mathbf{Mon}$, which maps a set A to the free monoid on A, is left adjoint to U. The units of this adjunction are familiar list-processing functions. Which ones? □

Free algebras have some extra structure. As with products, we do not have a single adjunction, but rather a family of adjunctions, one for each choice of F. The construction of the free algebra is functorial in the underlying base functor F and the operations are natural in that functor. Compared to products, the situation is more complicated as each choice of F gives rise to a different category of algebras. We need some infrastructure to switch swiftly between those different categories:

Rather amazingly, the construction $(-)$-**Alg** can be turned into a contravariant functor of type $\mathscr{C}^{\mathscr{C}} \to \mathbf{Cat}^{\mathsf{op}}$: it sends the functor F to the category of F-algebras and the natural transformation $\alpha : \mathsf{F} \dot{\to} \mathsf{G}$ to the functor $\alpha\text{-}\mathbf{Alg} : \mathsf{G}\text{-}\mathbf{Alg} \to \mathsf{F}\text{-}\mathbf{Alg}$, defined $\alpha\text{-}\mathbf{Alg}\,\langle A,\, a\rangle = \langle A,\, a \cdot \alpha\, A\rangle$ and $\alpha\text{-}\mathbf{Alg}\, h = h$. (As an example, $\mu\alpha$ is an F-algebra homomorphism of type $\langle \mu\mathsf{F},\, \mathit{in}\rangle \to \alpha\text{-}\mathbf{Alg}\,\langle \mu\mathsf{G},\, \mathit{in}\rangle$.) A number of proof obligations arise. We have to show that the G-algebra homomorphism $h : A \to B$ is also a F-algebra homomorphism of type $\alpha\text{-}\mathbf{Alg}\, A \to \alpha\text{-}\mathbf{Alg}\, B$.

$$\begin{aligned}
& h \cdot a \cdot \alpha\, A \\
= \; & \{\ \text{assumption: } h \text{ is an } \mathsf{G}\text{-algebra homomorphism: } h \cdot a = b \cdot \mathsf{G}\, h\ \} \\
& b \cdot \mathsf{G}\, h \cdot \alpha\, A \\
= \; & \{\ \alpha \text{ is natural: } \mathsf{G}\, h \cdot \alpha\, \hat{A} = \alpha\, \check{A} \cdot \mathsf{F}\, h\ \} \\
& b \cdot \alpha\, B \cdot \mathsf{F}\, h
\end{aligned}$$

Furthermore, $(-)$-**Alg** has to preserve identity and composition.

$$(\mathit{id}_{\mathsf{F}})\text{-}\mathbf{Alg} = \mathsf{Id}_{\mathsf{F}\text{-}\mathbf{Alg}} \tag{75}$$

$$(\alpha \cdot \beta)\text{-}\mathbf{Alg} = \beta\text{-}\mathbf{Alg} \circ \alpha\text{-}\mathbf{Alg} \tag{76}$$

The proofs are straightforward as the functor $(-)$-**Alg** does not change the carrier of an algebra. This is an important property worth singling out: $\mathsf{U}_{\mathsf{F}} \circ \alpha\text{-}\mathbf{Alg} = \mathsf{U}_{\mathsf{G}}$.

$$\begin{tikzcd}
\mathsf{G}\text{-}\mathbf{Alg}(\mathscr{C}) \arrow[rr, "\alpha\text{-}\mathbf{Alg}"] \arrow[dr, "\mathsf{U}_{\mathsf{G}}"'] & & \mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C}) \arrow[dl, "\mathsf{U}_{\mathsf{F}}"] \\
& \mathscr{C} &
\end{tikzcd} \tag{77}$$

Since several base functors are involved, we have indexed the constructions with the respective functor.

Equipped with the new machinery we can now generalise the fusion law (70) to homomorphisms of different types. Assuming $\alpha : \mathsf{F} \dot{\to} \mathsf{G}$, we have

$$\mathsf{U}_{\mathsf{G}}\, k \cdot \mathsf{U}_{\mathsf{F}}\,(\mathit{eval}_{\mathsf{F}}\, g) = \mathsf{U}_{\mathsf{F}}\,(\mathit{eval}_{\mathsf{F}}\,(\mathsf{U}_{\mathsf{G}}\, k \cdot g)) \ . \tag{78}$$

The original fusion law lives in F-**Alg**$(\mathscr{C})$, whereas this one lives in the underlying category $\mathscr{C}$. The proof makes essential use of property (77).

$$\begin{aligned}
& \mathsf{U}_{\mathsf{G}}\, k \cdot \mathsf{U}_{\mathsf{F}}\,(\mathit{eval}_{\mathsf{F}}\, g) \\
= \; & \{\ \mathsf{U}_{\mathsf{G}} = \mathsf{U}_{\mathsf{F}} \circ \alpha\text{-}\mathbf{Alg}\ (77)\ \} \\
& \mathsf{U}_{\mathsf{F}}\,(\alpha\text{-}\mathbf{Alg}\, k) \cdot \mathsf{U}_{\mathsf{F}}\,(\mathit{eval}_{\mathsf{F}}\, g) \\
= \; & \{\ \mathsf{U}_{\mathsf{F}} \text{ functor } (2)\ \} \\
& \mathsf{U}_{\mathsf{F}}\,(\alpha\text{-}\mathbf{Alg}\, k \cdot \mathit{eval}_{\mathsf{F}}\, g)
\end{aligned}$$

$$
\begin{aligned}
&= \quad \{\ \text{fusion (70)}\ \} \\
&\quad\ \mathsf{U_F}\,(\mathit{eval}_{\mathsf{F}}\,(\mathsf{U_F}\,(\alpha\text{-}\mathbf{Alg}\,k)\cdot g)) \\
&= \quad \{\ \mathsf{U_G} = \mathsf{U_F}\circ\alpha\text{-}\mathbf{Alg}\ (77)\ \} \\
&\quad\ \mathsf{U_F}\,(\mathit{eval}_{\mathsf{F}}\,(\mathsf{U_G}\,k\cdot g))
\end{aligned}
$$

The application of α-**Alg** can be seen as an adaptor. The algebras are invisible in the calculation—they can be made explicit using the type information provided by the adaptor.

Let us now turn to the heart of the matter. It is convenient to introduce a shortcut for the carrier of the free algebra:

$$\mathsf{F}^* = \mathsf{U_F}\circ\mathsf{Free_F}\ . \tag{79}$$

This defines a functor whose arrow part is $\mathsf{F}^*\,g = \mathsf{U_F}\,(\mathit{eval}_{\mathsf{F}}\,(\mathit{var}_{\mathsf{F}}\cdot g))$. Using F^* we can assign more succinct types to the constructors: $\mathit{var}_{\mathsf{F}} : \mathscr{C}(A, \mathsf{F}^*\,A)$ and $\mathit{com}_{\mathsf{F}} : \mathscr{C}(\mathsf{F}\,(\mathsf{F}^*\,A), \mathsf{F}^*\,A)$.

We claim that $(-)^*$ is a higher-order functor of type $\mathscr{C}^{\mathscr{C}} \to \mathscr{C}^{\mathscr{C}}$ that maps a base functor F to the so-called free monad of the functor F. As usual, we would like to *derive* the definition of the arrow part, which takes a natural transformation $\alpha : \mathsf{F} \dot{\to} \mathsf{G}$ to a natural transformation $\alpha^* : \mathsf{F}^* \dot{\to} \mathsf{G}^*$. One would hope that the constructors $\mathit{var}_{\mathsf{F}} : \mathscr{C}(A, \mathsf{F}^*\,A)$ and $\mathit{com}_{\mathsf{F}} : \mathscr{C}(\mathsf{F}\,(\mathsf{F}^*\,A), \mathsf{F}^*\,A)$ are natural in F:

$$
\begin{aligned}
\alpha^*\,A\cdot \mathit{var}_{\mathsf{F}} &= \mathit{var}_{\mathsf{G}}\ , \\
\alpha^*\,A\cdot \mathit{com}_{\mathsf{F}} &= \mathit{com}_{\mathsf{G}}\cdot\alpha\,(\alpha^*\,A)\ .
\end{aligned}
$$

Note that the functor $\lambda\,\mathsf{F}\,.\,\mathsf{F}\,(\mathsf{F}^*\,A)$, whose action on arrows is $\lambda\,\alpha\,.\,\alpha\,(\alpha^*\,A) = \lambda\,\alpha\,.\,\alpha\,(\mu\mathsf{G})\cdot\mathsf{F}\,(\alpha^*\,A)$, involves the 'application functor' (7). Consequently, the second condition expresses that the arrow $\alpha^*\,A$ is an F-homomorphism of type $\mathsf{Free_F}\,A \to \alpha\text{-}\mathbf{Alg}\,(\mathsf{Free_G}\,A)$:

$$
\begin{aligned}
&\alpha^*\,A\cdot\mathit{com}_{\mathsf{F}} = \mathit{com}_{\mathsf{G}}\cdot\alpha\,(\mu\mathsf{G})\cdot\mathsf{F}\,\alpha^*\,A \\
\iff\quad &\alpha^*\,A : \langle \mathsf{F}^*\,A,\ \mathit{com}_{\mathsf{F}}\rangle \to \langle \mathsf{G}^*\,A,\ \mathit{com}_{\mathsf{G}}\cdot\alpha\,(\mu\mathsf{G})\rangle\ .
\end{aligned}
$$

To derive the arrow part of $\alpha^*\,A$ we reason

$$
\begin{aligned}
&\alpha^*\,A\cdot\mathit{var}_{\mathsf{F}} = \mathit{var}_{\mathsf{G}} \\
\iff\quad &\{\ \alpha^*\,A \text{ is an } \mathsf{F}\text{-homomorphism, see above}\ \} \\
&\mathsf{U_F}\,(\alpha^*\,A)\cdot\mathit{var}_{\mathsf{F}} = \mathit{var}_{\mathsf{G}} \\
\iff\quad &\{\ \text{universal property (67)}\ \} \\
&\alpha^*\,A = \mathit{eval}_{\mathsf{F}}\ \mathit{var}_{\mathsf{G}} \\
\iff\quad &\{\ \mathit{eval}_{\mathsf{F}}\ \mathit{var}_{\mathsf{G}} \text{ is an } \mathsf{F}\text{-homomorphism}\ \} \\
&\alpha^*\,A = \mathsf{U_F}\,(\mathit{eval}_{\mathsf{F}}\ \mathit{var}_{\mathsf{G}})\ .
\end{aligned}
$$

Two remarks are in order. First, the universal property is applicable in the second step as $\mathit{var}_{\mathsf{G}} : A \to \mathsf{U}_{\mathsf{G}}\,(\mathsf{Free}_{\mathsf{G}}\,A) = A \to \mathsf{U}_{\mathsf{F}}\,(\alpha\text{-}\mathbf{Alg}\,(\mathsf{Free}_{\mathsf{G}}\,A))$. Second, the equations live in different categories: the last equation lives in $\mathscr{C}$, whereas the second, but last equation lives in $\mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C})$. To summarise, the arrow part of the higher-order functor α^* is defined

$$\alpha^*\,A = \mathsf{U}_{\mathsf{F}}\,(\mathit{eval}_{\mathsf{F}}\,\mathit{var}_{\mathsf{G}}) \ . \qquad (80)$$

Turning to the proofs we first have to show that α^* is indeed natural. Let $h : \mathscr{C}(\hat{A}, \check{A})$, then

$$\begin{aligned}
&\mathsf{G}^*\,h \cdot \alpha^*\,\hat{A}\\
=\;&\{\text{ definition of } \mathsf{G}^* \text{ (79) and } \alpha^* \text{ (80) }\}\\
&\mathsf{U}_{\mathsf{G}}\,(\mathit{eval}_{\mathsf{G}}\,(\mathit{var}_{\mathsf{G}} \cdot h)) \cdot \mathsf{U}_{\mathsf{F}}\,(\mathit{eval}_{\mathsf{F}}\,\mathit{var}_{\mathsf{G}})\\
=\;&\{\text{ generalised fusion (78) }\}\\
&\mathsf{U}_{\mathsf{F}}\,(\mathit{eval}_{\mathsf{F}}\,(\mathsf{U}_{\mathsf{G}}\,(\mathit{eval}_{\mathsf{G}}\,(\mathit{var}_{\mathsf{G}} \cdot h)) \cdot \mathit{var}_{\mathsf{G}}))\\
=\;&\{\text{ computation (68) }\}\\
&\mathsf{U}_{\mathsf{F}}\,(\mathit{eval}_{\mathsf{F}}\,(\mathit{var}_{\mathsf{G}} \cdot h))\\
=\;&\{\text{ functor fusion (73) }\}\\
&\mathsf{U}_{\mathsf{F}}\,(\mathit{eval}_{\mathsf{F}}\,\mathit{var}_{\mathsf{G}} \cdot \mathsf{Free}_{\mathsf{F}}\,h)\\
=\;&\{\ \mathsf{U}_{\mathsf{F}} \text{ functor (2) }\}\\
&\mathsf{U}_{\mathsf{F}}\,(\mathit{eval}_{\mathsf{F}}\,\mathit{var}_{\mathsf{G}}) \cdot \mathsf{U}_{\mathsf{F}}\,(\mathsf{Free}_{\mathsf{F}}\,h)\\
=\;&\{\text{ definition of } \alpha^* \text{ (80) and } \mathsf{F}^* \text{ (79) }\}\\
&\alpha^*\,\check{A} \cdot \mathsf{F}^*\,h \ .
\end{aligned}$$

As usual, we postpone the proof that $(-)^*$ preserves identity and composition.

The *base functor fusion law* states that

$$\mathsf{U}_{\mathsf{G}}\,(\mathit{eval}_{\mathsf{G}}\,g) \cdot \alpha^*\,A = \mathsf{U}_{\mathsf{F}}\,(\mathit{eval}_{\mathsf{F}}\,g) \ . \qquad (81)$$

We reason

$$\begin{aligned}
&\mathsf{U}_{\mathsf{G}}\,(\mathit{eval}_{\mathsf{G}}\,g) \cdot \alpha^*\,A\\
=\;&\{\text{ definition of } (-)^* \text{ (80) }\}\\
&\mathsf{U}_{\mathsf{G}}\,(\mathit{eval}_{\mathsf{G}}\,g) \cdot \mathsf{U}_{\mathsf{F}}\,(\mathit{eval}_{\mathsf{F}}\,\mathit{var}_{\mathsf{G}})\\
=\;&\{\text{ generalised fusion (78) }\}\\
&\mathsf{U}_{\mathsf{F}}\,(\mathit{eval}_{\mathsf{F}}\,(\mathsf{U}_{\mathsf{G}}\,(\mathit{eval}_{\mathsf{G}}\,g) \cdot \mathit{var}_{\mathsf{G}}))\\
=\;&\{\text{ computation (68) }\}\\
&\mathsf{U}_{\mathsf{F}}\,(\mathit{eval}_{\mathsf{F}}\,g) \ .
\end{aligned}$$

Given these prerequisites, it is straightforward to show that $(-)^*$ preserves identity ($id : \mathsf{F} \dot{\to} \mathsf{F}$)

$$\begin{aligned}
& id^*\, A \\
= \;& \{ \text{ definition of } (-)^* \text{ (80) and (75) } \} \\
& \mathsf{U}_{\mathsf{F}}\, (eval_{\mathsf{F}}\; var_{\mathsf{F}}) \\
= \;& \{ \text{ reflection (69) } \} \\
& \mathsf{U}_{\mathsf{F}}\; id \\
= \;& \{ \; \mathsf{U}_{\mathsf{F}} \text{ functor (1) } \} \\
& id \;\; ,
\end{aligned}$$

and composition ($\beta : \mathsf{G} \dot{\to} \mathsf{H}$ and $\alpha : \mathsf{F} \dot{\to} \mathsf{G}$)

$$\begin{aligned}
& \beta^* \cdot \alpha^* \\
= \;& \{ \text{ definition of } (-)^* \text{ (80) } \} \\
& \mathsf{U}_{\mathsf{G}}\, (eval_{\mathsf{G}}\; var_{\mathsf{H}}) \cdot \alpha^* \\
= \;& \{ \text{ base functor fusion (81) } \} \\
& \mathsf{U}_{\mathsf{F}}\, (eval_{\mathsf{F}}\; var_{\mathsf{H}}) \\
= \;& \{ \text{ definition of } (-)^* \text{ (80) and (76) } \} \\
& (\beta \cdot \alpha)^* \;\; .
\end{aligned}$$

Base functor fusion expresses that $\mathsf{U}_{\mathsf{F}}\, (eval_{\mathsf{F}}\, -) : \mathscr{C}(A, \mathsf{U}\, B) \to \mathscr{C}(\mathsf{F}^*\, A, \mathsf{U}\, B)$ is natural in F—note that F occurs in a contravariant position.

2.6.5 Relating F^* and $\mu\mathsf{F}$. Since left adjoints preserve initial objects, we have $\mathsf{Free}\, 0 \cong \langle \mu\mathsf{F},\, in \rangle$ and consequently $\mathsf{F}^*\, 0 = \mathsf{U}\, (\mathsf{Free}\, 0) \cong \mathsf{U}\, \langle \mu\mathsf{F},\, in \rangle = \mu\mathsf{F}$—this step uses the fact that functors, here U, preserve isomorphisms (see Exercise 5). In words, the elements of $\mu\mathsf{F}$ are closed terms, terms without variables. Conversely, free algebras can be expressed in terms of initial algebras: $\langle \mathsf{F}^*\, A,\, com \rangle \cong \langle \mu\mathsf{F}_A,\, in \cdot inr \rangle$ where $\mathsf{F}_A\, X = A + \mathsf{F}\, X$. The functor F_A formalises that a term is either a variable or a composite term. For this representation, $in \cdot inl$ plays the rôle of var and $in \cdot inr$ plays the rôle of com. To prove the isomorphism we show that the data above determines an adjunction with U as the right adjoint. Since left adjoints are unique up to natural isomorphism, the result follows. (The isomorphism $\langle \mathsf{F}^*\, A,\, com \rangle \cong \langle \mu\mathsf{F}_A,\, in \cdot inr \rangle$ between algebras is even natural in A.)

Turning to the proof, we show that for each $g : \mathscr{C}(A, \mathsf{U}\, B)$ there exists an F-algebra homomorphism $\mathsf{F}\text{-}\mathbf{Alg}(\langle \mu\mathsf{F}_A,\, in \cdot inr \rangle, \langle B,\, b \rangle)$ satisfying the universal property (67). We claim that $(\!| g \triangledown b |\!)$ is the required homomorphism. The following calculation shows that $(\!|g \triangledown b|\!)$ is indeed an F-algebra homomorphism.

$$
\begin{aligned}
& (\!|g \triangledown b|\!) \cdot in \cdot inr \\
= \;& \{ \text{ computation (55) } \} \\
& (g \triangledown b) \cdot \mathsf{F}_A \, (\!|g \triangledown b|\!) \cdot inr \\
= \;& \{ \; \mathsf{F}_A \, X = A + \mathsf{F} \, X \text{ and } inr \text{ is natural (23) } \} \\
& (g \triangledown b) \cdot inr \cdot \mathsf{F} \, (\!|g \triangledown b|\!) \\
= \;& \{ \text{ computation (19) } \} \\
& b \cdot \mathsf{F} \, (\!|g \triangledown b|\!)
\end{aligned}
$$

To establish the universal property (67) we reason

$$
\begin{aligned}
& f = (\!|g \triangledown b|\!) \\
\iff \;& \{ \text{ uniqueness property (54) } \} \\
& f \cdot in = (g \triangledown b) \cdot \mathsf{F}_A \, f \\
\iff \;& \{ \; \mathsf{F}_A \, X = A + \mathsf{F} \, X \text{ and functor fusion (25) } \} \\
& f \cdot in = g \triangledown b \cdot \mathsf{F} \, f \\
\iff \;& \{ \text{ universal property (17) } \} \\
& f \cdot in \cdot inl = g \;\; \wedge \;\; f \cdot in \cdot inr = b \cdot \mathsf{F} \, f \\
\iff \;& \{ \; f : \langle \mu \mathsf{F}_A, \, in \cdot inr \rangle \to \langle B, \, b \rangle \; \} \\
& \mathsf{U} \, f \cdot in \cdot inl = g \;\; .
\end{aligned}
$$

The last step makes use of the fact that f ranges over F-algebra homomorphisms.

2.6.6 Banana-split. The adjunction $\mathsf{Free} \dashv \mathsf{U}$ tells us a lot about the structure of the category of algebras. We have already made use of the fact that left adjoints preserve initial objects: $\mathsf{Free}\, 0 \cong 0$. Since right adjoints preserve final objects and products, we furthermore know that

$$\mathsf{U} \, 1 \cong 1 \;\; , \tag{82}$$

$$\mathsf{U} \, (B_1 \times B_2) \cong \mathsf{U} \, B_1 \times \mathsf{U} \, B_2 \;\; . \tag{83}$$

Since U is a forgetful functor, we can use these properties to *derive* the definition of final objects and products in the category of algebras. Property (82) suggests that the final algebra is given by (this solves Exercise 26)

$$1 = \langle 1, \, \mathsf{F} \, 1 \dashrightarrow 1 \rangle \;\; .$$

The action of the final algebra is determined since there is exactly one arrow from $\mathsf{F}\,1$ to 1. The unique homomorphism from any algebra $\langle A, \, a \rangle$ to the final algebra is simply $A \dashrightarrow 1$. The homomorphism condition, $(1 \dashleftarrow A) \cdot a = (1 \dashleftarrow \mathsf{F}\,1) \cdot \mathsf{F}\,(1 \dashleftarrow A)$, follows from fusion.

Likewise, Property (83) determines the carrier of the product algebra. To determine its action, we reason as follows. Preservation of products also implies

$\mathsf{U}\,\mathit{outl} = \mathit{outl}$ and $\mathsf{U}\,\mathit{outr} = \mathit{outr}$—these are instances of (53) assuming equality rather than isomorphism in (83). In other words, *outl* and *outr* have to be F-algebra homomorphisms: $\mathit{outl} : \langle B_1 \times B_2,\, x\rangle \to \langle B_1,\, b_1\rangle$ and $\mathit{outr} : \langle B_1 \times B_2,\, x\rangle \to \langle B_2,\, b_2\rangle$ where x is the to-be-determined action. Let's calculate.

$$\begin{aligned}
& \mathit{outl} : \langle B_1 \times B_2,\, x\rangle \to \langle B_1,\, b_1\rangle \ \wedge\ \mathit{outr} : \langle B_1 \times B_2,\, x\rangle \to \langle B_2,\, b_2\rangle \\
\iff\ & \{\ \text{homomorphism condition}\ \} \\
& \mathit{outl} \cdot x = b_1 \cdot \mathsf{F}\,\mathit{outl} \ \wedge\ \mathit{outr} \cdot x = b_2 \cdot \mathsf{F}\,\mathit{outr} \\
\iff\ & \{\ \text{universal property (8)}\ \} \\
& x = b_1 \cdot \mathsf{F}\,\mathit{outl} \vartriangle b_2 \cdot \mathsf{F}\,\mathit{outr}
\end{aligned}$$

Consequently, the product of algebras is defined

$$\langle B_1,\, b_1\rangle \times \langle B_2,\, b_2\rangle = \langle B_1 \times B_2,\, b_1 \cdot \mathsf{F}\,\mathit{outl} \vartriangle b_2 \cdot \mathsf{F}\,\mathit{outr}\rangle \ .$$

There is one final proof obligation: we have to show that the mediating arrow $\vartriangle$ takes homomorphisms to homomorphisms.

$$\begin{aligned}
& f_1 \vartriangle f_2 : \langle A,\, a\rangle \to \langle B_1,\, b_1\rangle \times \langle B_2,\, b_2\rangle \\
& \quad \Longleftarrow \quad f_1 : \langle A,\, a\rangle \to \langle B_1,\, b_1\rangle \ \wedge\ f_2 : \langle A,\, a\rangle \to \langle B_2,\, b_2\rangle
\end{aligned}$$

We reason

$$\begin{aligned}
& (f_1 \vartriangle f_2) \cdot a \\
=\ & \{\ \text{fusion (12)}\ \} \\
& f_1 \cdot a \vartriangle f_2 \cdot a \\
=\ & \{\ \text{assumption: } f_1 : \langle A,\, a\rangle \to \langle B_1,\, b_1\rangle \wedge f_2 : \langle A,\, a\rangle \to \langle B_2,\, b_2\rangle\ \} \\
& b_1 \cdot \mathsf{F}\,f_1 \vartriangle b_2 \cdot \mathsf{F}\,f_2 \\
=\ & \{\ \text{computation (9)–(10)}\ \} \\
& b_1 \cdot \mathsf{F}\,(\mathit{outl} \cdot (f_1 \vartriangle f_2)) \vartriangle b_2 \cdot \mathsf{F}\,(\mathit{outr} \cdot (f_1 \vartriangle f_2)) \\
=\ & \{\ \mathsf{F}\ \text{functor (2)}\ \} \\
& b_1 \cdot \mathsf{F}\,\mathit{outl} \cdot \mathsf{F}\,(f_1 \vartriangle f_2) \vartriangle b_2 \cdot \mathsf{F}\,\mathit{outr} \cdot \mathsf{F}\,(f_1 \vartriangle f_2) \\
=\ & \{\ \text{fusion (12)}\ \} \\
& (b_1 \cdot \mathsf{F}\,\mathit{outl} \vartriangle b_2 \cdot \mathsf{F}\,\mathit{outr}) \cdot \mathsf{F}\,(f_1 \vartriangle f_2) \ .
\end{aligned}$$

Using product algebras we can justify the *banana-split law* [9], an important program optimisation which replaces a double tree traversal by a single one.

$$(\!|b_1|\!) \vartriangle (\!|b_2|\!) = (\!|b_1 \cdot \mathsf{F}\,\mathit{outl} \vartriangle b_2 \cdot \mathsf{F}\,\mathit{outr}|\!) : \langle \mu\mathsf{F},\, \mathit{in}\rangle \to \langle B_1,\, b_1\rangle \times \langle B_2,\, b_2\rangle$$

The double traversal on the left is transformed into the single traversal on the right. (The law is called 'banana-split', because the fold brackets are like bananas and $\vartriangle$ is pronounced 'split'.) The law can now be justified in two different ways: because $(\!|b_1|\!) \vartriangle (\!|b_2|\!)$ is the *unique* homomorphism *to* the product algebra, and because $(\!|b_1 \cdot \mathsf{F}\,\mathit{outl} \vartriangle b_2 \cdot \mathsf{F}\,\mathit{outr}|\!)$ is the *unique* F-algebra homomorphism *from* the initial algebra.

Exercise 33. Formalise the dual of the banana-split law (which involves unfolds and coproducts). □

2.6.7 Cofree Coalgebra. The dual of the free algebra is the cofree coalgebra. In **Set**, the elements of $\mathsf{U}\,(\mathsf{Cofree}\,A)$ are infinite trees whose branching structure is determined by F with labels drawn from A. The action of the coalgebra, $\mathit{subtrees} : \mathscr{C}(\mathsf{U}\,(\mathsf{Cofree}\,A), \mathsf{F}\,(\mathsf{U}\,(\mathsf{Cofree}\,A)))$, maps a tree to an F-structure of subtrees. Think of the functor F as a static description of all possible behaviours of a system. Additionally, there is an operation $\mathit{label} : \mathscr{C}(\mathsf{U}\,(\mathsf{Cofree}\,A), A)$ for extracting the label of (the root of) a tree. This operation is universal: for each $f : \mathscr{C}(\mathsf{U}\,A, B)$ there is an F-coalgebra homomorphism $\mathit{trace}\,f : \mathsf{F}\text{-}\mathbf{Coalg}(A, \mathsf{Cofree}\,B)$ such that

$$f = \mathit{label} \cdot \mathsf{U}\,g \iff \mathit{trace}\,f = g \ , \tag{84}$$

for all $g : \mathsf{F}\text{-}\mathbf{Coalg}(A, \mathsf{Cofree}\,B)$. Think of the coalgebra A as a type of states, whose action is a mapping from states to successor states. The universal property expresses that the infinite tree of behaviours for a given start state is uniquely determined by a labelling function for the states. As an aside, the carrier of $\mathsf{Cofree}\,A$, written $\mathsf{F}^{\infty}\,A$, is also known as a *generalised rose tree*.

As an example, the cofree coalgebra of the squaring functor $\mathsf{Sq}\,A = A \times A$ generalises the type of bifurcations abstracting away from the type of natural numbers (see also Exercise 43).

⋆ *Exercise 34.* The composition $\mathsf{U} \circ \mathsf{Cofree}$ is the cofree comonad of the functor F. Explore the structure for $\mathsf{F} = \mathsf{Id}$ and $\mathsf{F} = \mathsf{K}$, where K is the constant functor. □

Since right adjoints preserve final objects, we have $\mathsf{Cofree}\,1 \cong \langle \nu\mathsf{F}, \mathit{out} \rangle$ and consequently $\mathsf{F}^{\infty}\,1 = \mathsf{U}\,(\mathsf{Cofree}\,1) \cong \mathsf{U}\,\langle \nu\mathsf{F}, \mathit{out} \rangle = \nu\mathsf{F}$. In words, the elements of $\nu\mathsf{F}$ are infinite trees with trivial labels. Conversely, cofree coalgebras can be expressed in terms of final coalgebras: $\langle \mathsf{F}^{\infty}\,A, \mathit{subtrees} \rangle \cong \langle \nu\mathsf{F}_A, \mathit{outr} \cdot \mathit{out} \rangle$ where $\mathsf{F}_A\,X = A \times \mathsf{F}\,X$.

Table 2 summarises the adjunctions discussed in this section.

2.7 The Yoneda Lemma

This section introduces an important categorical tool: the Yoneda Lemma. It is related to continuation-passing style and induces an important proof technique, the principle of indirect proof [14].

Recall that the contravariant hom-functor $\mathscr{C}(-, X) : \mathscr{C}^{\mathsf{op}} \to \mathbf{Set}$ maps an arrow $f : \mathscr{C}(B, A)$ to a *function* $\mathscr{C}(f, X) : \mathscr{C}(A, X) \to \mathscr{C}(B, X)$. This function is natural in X—this is the import of identity (5). Furthermore, every natural transformation of type $\mathscr{C}(A, -) \dot{\to} \mathscr{C}(B, -)$ is obtained as the image of $\mathscr{C}(f, -)$ for some f. So we have the following isomorphism between arrows and natural transformations.

$$\mathscr{C}(B, A) \cong \mathscr{C}(A, -) \dot{\to} \mathscr{C}(B, -)$$

Table 2. Examples of adjunctions.

adjunction	initial object	final object	coproduct	product	general coproduct copower	general product power	exponential to	free algebra	cofree coalgebra
L	0	$\varDelta$	$+$	$\varDelta$	$\Sigma\, i \in I \,.\, (-)_i$	$\varDelta$	$- \times X$	Free	U
R	$\varDelta$	1	$\varDelta$	$\times$	$\varDelta$	$\Pi\, i \in I \,.\, (-)_i$	$(-)^X$	U	Cofree
$\lceil - \rceil$			$\triangledown$		$\bigtriangledown\, i \in I \,.\, (-)_i$		Λ°	eval	
$\lfloor - \rfloor$				$\vartriangle$		$\bigtriangleup\, i \in I \,.\, (-)_i$	Λ		trace
ϵ	¡		$\mathit{id} \triangledown \mathit{id}$	$\langle \mathit{outl},\ \mathit{outr} \rangle$		$\pi_{(-)}$	apply		label
η		$!$	$\langle \mathit{inl},\ \mathit{inr} \rangle$	$\delta = \mathit{id} \vartriangle \mathit{id}$	$\iota_{(-)}$			var	

This isomorphism is an instance of a more general result, known as the *Yoneda Lemma* [8]. Let $\mathsf{H} : \mathscr{C} \to \mathbf{Set}$ be a set-valued functor, then

$$\mathsf{H}\, A \cong \mathscr{C}(A, -) \mathrel{\dot{\to}} \mathsf{H} \ . \tag{85}$$

(The isomorphism is natural in H and in A.) The following arrows are the witnesses of the Yoneda isomorphism:

$$\mathsf{y}\, s\, X = \lambda f : \mathscr{C}(A, X) \,.\, \mathsf{H} f\, s \qquad \text{and} \qquad \mathsf{y}^{\circ}\, \alpha = \alpha\, A\, \mathit{id}_A \ . \tag{86}$$

Observe that y is just H with the two arguments swapped. It is easy to see that y° is the left-inverse of y.

$$\begin{aligned}
& \mathsf{y}^{\circ}\, (\mathsf{y}\, s) \\
= \ & \{ \text{ definition of } \mathsf{y}^{\circ} \text{ (86) } \} \\
& \mathsf{y}\, s\, A\, \mathit{id}_A \\
= \ & \{ \text{ definition of } \mathsf{y} \text{ (86) } \} \\
& \mathsf{H}\, \mathit{id}_A\, s \\
= \ & \{ \ \mathsf{H} \text{ functor (2) } \} \\
& s
\end{aligned}$$

For the opposite direction, we make use of the naturality of α, that is, $\mathsf{H}\, h \cdot \alpha\, \hat{X} = \alpha\, \check{X} \cdot \mathscr{C}(A, h)$, or written in a pointwise style: $\mathsf{H}\, h\, (\alpha\, \hat{X}\, g) = \alpha\, \check{X}\, (h \cdot g)$, with

$h : \mathscr{C}(\hat{X}, \check{X})$ and $g : \mathscr{C}(A, \hat{X})$.

$$\begin{aligned}
& \mathsf{y}\,(\mathsf{y}^{\circ}\,\alpha)\,X \\
= \;& \{ \text{ definition of } \mathsf{y} \text{ (86) } \} \\
& \lambda f \,.\, \mathsf{H} f\,(\mathsf{y}^{\circ}\,\alpha) \\
= \;& \{ \text{ definition of } \mathsf{y}^{\circ} \text{ (86) } \} \\
& \lambda f \,.\, \mathsf{H} f\,(\alpha\,A\,id_A) \\
= \;& \{ \; \alpha \text{ is natural: } \mathsf{H}\,h\,(\alpha\,\hat{X}\,g) = \alpha\,\check{X}\,(h \cdot g) \; \} \\
& \lambda f \,.\, \alpha\,X\,(f \cdot id_A) \\
= \;& \{ \text{ identity } \} \\
& \lambda f \,.\, \alpha\,X\,f \\
= \;& \{ \text{ extensionality—}\alpha\,X \text{ is a function } \} \\
& \alpha\,X
\end{aligned}$$

For $\mathsf{H} : \mathscr{C} \to \mathbf{Set}$ with $\mathsf{H}\,X = \mathscr{C}(B, X)$ and B fixed, we have $\mathscr{C}(B, A) \cong \mathscr{C}(A, -) \dot{\to} \mathscr{C}(B, -)$. Furthermore, the isomorphism simplifies to $\mathsf{y}\,g = \mathscr{C}(g, -)$ as a quick calculation shows.

$$\mathsf{y}\,g\,X = \lambda f \,.\, \mathscr{C}(B, f)\,g = \lambda f \,.\, \mathscr{C}(g, X)\,f = \mathscr{C}(g, X)$$

Conversely, for $\mathsf{H} : \mathscr{C}^{\mathsf{op}} \to \mathbf{Set}$ with $\mathsf{H}\,X = \mathscr{C}(X, B)$ and B fixed, we have $\mathscr{C}(A, B) \cong \mathscr{C}^{\mathsf{op}}(A, -) \dot{\to} \mathscr{C}(-, B) \cong \mathscr{C}(-, A) \dot{\to} \mathscr{C}(-, B)$. Furthermore, the isomorphism simplifies to $\mathsf{y}\,g = \mathscr{C}(-, g)$.

These special cases give rise to the principle of *indirect proof*.

$$f = g \iff \mathscr{C}(f, -) = \mathscr{C}(g, -) \tag{87}$$
$$f = g \iff \mathscr{C}(-, f) = \mathscr{C}(-, g) \tag{88}$$

Instead of proving the equality of f and g directly, we show the equality of their Yoneda images $\mathsf{y}\,f$ and $\mathsf{y}\,g$.

When we discussed exponentials (see Section 2.5), we noted that there is a unique way to turn the exponential B^X into a bifunctor, so that the bijection

$$\Lambda : \mathscr{C}(A \times X, B) \cong \mathscr{C}(A, B^X) : \Lambda^{\circ} \tag{89}$$

is also natural in X. The proof of this fact makes essential use of the Yoneda Lemma. Recall that a transformation between n-ary functors is natural if and only if it is natural in each argument separately. Let $p : \mathscr{C}(\check{X}, \hat{X})$, then the naturality condition (89) implies

$$\begin{aligned}
& \mathscr{C}(-, B^{p}) \cdot \Lambda = \Lambda \cdot \mathscr{C}(- \times p, B) \\
\iff & \quad \{ \text{ adjunction: } \Lambda \cdot \Lambda^{\circ} = id \text{ and } \Lambda^{\circ} \cdot \Lambda = id \text{ (27) } \} \\
& \mathscr{C}(-, B^{p}) = \Lambda \cdot \mathscr{C}(- \times p, B) \cdot \Lambda^{\circ} \\
\iff & \quad \{ \text{ Yoneda Lemma: } \mathscr{C}(B^{\hat{X}}, B^{\check{X}}) \cong \mathscr{C}(-, B^{\hat{X}}) \dot{\to} \mathscr{C}(-, B^{\check{X}}) \text{ (85) } \} \\
& \mathsf{y}^{\circ}\,(\mathscr{C}(-, B^{p})) = \mathsf{y}^{\circ}\,(\Lambda \cdot \mathscr{C}(- \times p, B) \cdot \Lambda^{\circ}) \\
\iff & \quad \{ \text{ definition of } \mathsf{y}^{\circ} \} \\
& \mathscr{C}(-, B^{p})\, B^{\hat{X}}\, id = (\Lambda \cdot \mathscr{C}(- \times p, B) \cdot \Lambda^{\circ})\, B^{\hat{X}}\, id \\
\iff & \quad \{ \text{ composition of natural transformations } \} \\
& \mathscr{C}(-, B^{p})\, B^{\hat{X}}\, id = (\Lambda \cdot \mathscr{C}(B^{\hat{X}} \times p, B) \cdot \Lambda^{\circ})\, id \\
\iff & \quad \{ \text{ definition of hom-functors (6) } \} \\
& B^{p} = \Lambda\,(\Lambda^{\circ}\, id \cdot (B^{\hat{X}} \times p)) \\
\iff & \quad \{ \; apply = \Lambda^{\circ}\, id \text{ (Table 1) } \} \\
& B^{p} = \Lambda\,(apply \cdot (B^{\hat{X}} \times p)) \enspace .
\end{aligned}$$

The reflection law (42) implies that $B^{(-)}$ preserves the identity. Since the naturality condition (89) uniquely determines $B^{(-)}$'s action on arrows, it furthermore preserves composition:

$$\begin{aligned}
& \mathscr{C}(-, B^{p \cdot q}) \\
= & \quad \{ \text{ naturality condition (89) } \} \\
& \Lambda \cdot \mathscr{C}(- \times (p \cdot q), B) \cdot \Lambda^{\circ} \\
= & \quad \{ \; A \times - \text{ covariant functor and } \mathscr{C}(-, B) \textit{ contravariant} \text{ functor } \} \\
& \Lambda \cdot \mathscr{C}(- \times q, B) \cdot \mathscr{C}(- \times p, B) \cdot \Lambda^{\circ} \\
= & \quad \{ \text{ naturality condition (89) } \} \\
& \Lambda \cdot \mathscr{C}(- \times q, B) \cdot \Lambda^{\circ} \cdot \mathscr{C}(-, B^{p}) \\
= & \quad \{ \text{ naturality condition (89) and adjunction } \Lambda \cdot \Lambda^{\circ} = id \text{ (27) } \} \\
& \mathscr{C}(-, B^{q}) \cdot \mathscr{C}(-, B^{p}) \\
= & \quad \{ \; \mathscr{C}(A, -) \text{ covariant functor } \} \\
& \mathscr{C}(-, B^{q} \cdot B^{p}) \enspace .
\end{aligned}$$

Applying the principle of indirect proof (88), we conclude that $B^{p \cdot q} = B^{q} \cdot B^{p}$.

Exercise 35. Generalise the argument above to an arbitrary adjunction with a parameter. Let $\mathsf{L} : \mathscr{C} \leftarrow \mathscr{D} \times \mathscr{X}$ be a bifunctor, written $\mathsf{L}_X\, A$ for clarity, so that the partial application $\mathsf{L}_X : \mathscr{C} \leftarrow \mathscr{D}$ has a right adjoint $\mathsf{R}_X : \mathscr{C} \to \mathscr{D}$ for each choice of $X : \mathscr{X}$:

$$\lfloor - \rfloor : \mathscr{C}(\mathsf{L}_X\, A, B) \cong \mathscr{D}(A, \mathsf{R}_X\, B) \enspace . \qquad (90)$$

1. Show that there is a unique way to turn R into a bifunctor of type $\mathscr{X}^{\mathsf{op}} \times \mathscr{C} \to \mathscr{D}$ so that the bijection (90) is natural in all three variables A, B and X.

$$\begin{tikzcd}
\mathscr{C}(\mathsf{L}_{\hat{X}}\, A, B) \arrow[r, "\lfloor - \rfloor"] \arrow[d, "\mathscr{C}(\mathsf{L}_p\, A, B)"'] & \mathscr{D}(A, \mathsf{R}_{\hat{X}}\, B) \arrow[d, "\mathscr{D}(A, \mathsf{R}_p\, B)"] \\
\mathscr{C}(\mathsf{L}_{\check{X}}\, A, B) \arrow[r, "\lfloor - \rfloor"'] & \mathscr{D}(A, \mathsf{R}_{\check{X}}\, B)
\end{tikzcd}$$

   Explain why R_X is necessarily contravariant in the parameter X.
2. Let $\eta_X : \mathsf{Id} \dot{\to} \mathsf{R}_X \circ \mathsf{L}_X$ be the unit of the adjunction with a parameter. What property of η corresponds to the naturality of the bijection (90) in X? (The unit is *not* natural in X since $\mathsf{R}_X \circ \mathsf{L}_X$ is *not* functorial in X. Why?) □

3 Adjunctions for Algorithms

In the first part of these lecture notes we have seen that every fundamental 'data structure' arises out of an adjunction. In the second part we turn our attention to 'algorithms'. Our goal is to give a precise semantics to a large class of recursion equations—equations as they might arise in a Haskell program.

This second part is organised as follows. Section 3.1 reviews conventional folds and unfolds as introduced in Section 2.6. We take a somewhat non-standard approach and re-introduce them as solutions of so-called Mendler-style equations. Section 3.2 generalises these equations to adjoint equations and demonstrates that many Haskell functions fall under this umbrella. Section 3.3 specialises adjoint equations to a variety of basic adjunctions and explores the resulting recursion schemes. Section 3.4 develops the calculational properties of adjoint folds. Like their vanilla counterparts, they enjoy reflection, computation and fusion laws.

Some knowledge of the functional programming language Haskell [15] is useful, as the formal development is paralleled by a series of programming examples.

3.1 Fixed-Point Equations

In this section we review the semantics of datatypes, albeit with a slight twist. The following two Haskell programs serve as running examples.

Example 1. The datatype *Bush* models binary leaf trees, a representation of non-empty sequences of natural numbers.

$$\mathbf{data}\, \mathit{Bush} = \mathit{Leaf}\ \mathit{Nat} \mid \mathit{Fork}\ (\mathit{Bush}, \mathit{Bush})$$

The type (A, B) is Haskell syntax for the cartesian product $A \times B$.

The function *total* computes the sum of a bush of natural numbers.

$$\begin{aligned} &\mathit{total} : \mathit{Bush} && \to \mathit{Nat} \\ &\mathit{total}\ \ (\mathit{Leaf}\ n) && = n \\ &\mathit{total}\ \ (\mathit{Fork}\ (l, r)) && = \mathit{total}\ l + \mathit{total}\ r \end{aligned}$$

This is a typical example of a *fold*, a function that *consumes* data. □

Example 2. The type *Tree* captures bifurcations, infinite binary trees of naturals. (A bifurcation is a division of a state or an action into two branches.)

$$\mathbf{data}\ \mathit{Tree} = \mathit{Branch}\ (\mathit{Tree}, \mathit{Nat}, \mathit{Tree})$$

The call *generate* 1 constructs an infinite tree, labelled with the naturals from 1 onwards.

$$\begin{aligned} &\mathit{generate} : \mathit{Nat} \to \mathit{Tree} \\ &\mathit{generate}\ \ \ n \quad = \mathit{Branch}\ (\mathit{generate}\ (2 * n + 0), n, \mathit{generate}\ (2 * n + 1)) \end{aligned}$$

This is a typical example of an *unfold*, a function that *produces* data. □

Both the types, *Bush* and *Tree*, and the functions, *total* and *generate*, are given by recursion equations. At the outset, it is not at all clear that these equations have solutions and if so whether the solutions are unique. It is customary to rephrase the problem of solving recursion equations as a fixed-point problem: a recursion equation of the form $x = \Psi\, x$ implicitly defines a function Ψ in the unknown x, the so-called *base function* of the recursion equation. A fixed-point of the base function is then a solution of the recursion equation and vice versa.

Consider the type equation defining *Bush*. Its base function or, rather, its *base functor* is given by

$$\begin{aligned} &\mathbf{data}\ \mathfrak{Bush}\ \mathit{bush} = \mathfrak{Leaf}\ \mathit{Nat} \mid \mathfrak{Fork}\ (\mathit{bush}, \mathit{bush}) \\ &\mathbf{instance}\ \mathit{Functor}\ \mathfrak{Bush}\ \mathbf{where} \\ &\quad \mathit{fmap}\ f\ (\mathfrak{Leaf}\ n) \qquad = \mathfrak{Leaf}\ n \\ &\quad \mathit{fmap}\ f\ (\mathfrak{Fork}\ (l, r)) = \mathfrak{Fork}\ (f\ l, f\ r) \quad . \end{aligned}$$

We adopt the convention that the base functor is named after the underlying type, using $\mathfrak{this}$ font for the former and *this* font for the latter. The type argument of $\mathfrak{Bush}$ marks the recursive component. In Haskell, the object part of a functor is defined by a **data** declaration; the arrow part is given by a *Functor* instance. Using arithmetic notation $\mathfrak{Bush}$ is written $\mathfrak{Bush}\ B = \mathit{Nat} + B \times B$.

All functors underlying first-order datatype declarations (sums of products, no function types) have two extremal fixed points: the *initial* F-*algebra* $\langle \mu\mathsf{F}, \mathit{in} \rangle$ and the *final* F-*coalgebra* $\langle \nu\mathsf{F}, \mathit{out} \rangle$, where $\mathsf{F} : \mathscr{C} \to \mathscr{C}$ is the functor in question (Section 2.6). Some programming languages such as Charity [16] or Coq [17] allow the user to choose between initial and final solutions—the datatype declarations are flagged as *inductive* or *coinductive*. Haskell is not one of them. Since Haskell's underlying category is **SCpo**, the category of complete partial orders and strict continuous functions, initial algebras and final coalgebras actually coincide [18, 19]—further background is provided at the end of this section. In contrast, in **Set** elements of an inductive type are finite, whereas elements of a coinductive type are potentially infinite. Operationally, an element of an inductive type can be constructed in a finite sequence of steps, whereas an element of a coinductive type allows any finite sequence of observations.

Turning to our running examples, we view *Bush* as an initial algebra—though inductive and coinductive trees are both equally useful. For bifurcations, only the coinductive reading is useful since in **Set** the initial algebra of *Tree*'s base functor is the empty set.

Definition 3. In Haskell, initial algebras and final coalgebras can be defined as follows.

$$\begin{aligned}&\mathbf{newtype}\,\mu f = \mathit{In} \quad\ \{\mathit{in}^{\circ} : f\,(\mu f)\}\\&\mathbf{newtype}\,\nu f = \mathit{Out}^{\circ}\,\{\mathit{out} : f\,(\nu f)\}\end{aligned}$$

The definitions use Haskell's record syntax to introduce the destructors in° and *out* in addition to the constructors *In* and Out°. The **newtype** declaration guarantees that μf and $f\,(\mu f)$ share the same representation at run-time, and likewise for νf and $f\,(\nu f)$. In other words, the constructors and destructors are no-ops. Of course, since initial algebras and final coalgebras coincide in Haskell, they could be defined by a single **newtype** definition. However, since we wish to use Haskell as a meta-language for **Set**, we keep them separate. □

Working towards a semantics for *total*, let us first adapt its definition to the new 'two-level type' $\mu\mathfrak{Bush}$. The term is due to [20]; one level describes the structure of the data, the other level ties the recursive knot.

$$\begin{aligned}&\mathit{total} : \mu\mathfrak{Bush} && \to \mathit{Nat}\\&\mathit{total}\ \ (\mathit{In}\,(\mathfrak{Leaf}\,n)) && = n\\&\mathit{total}\ \ (\mathit{In}\,(\mathfrak{Fork}\,(l,r))) && = \mathit{total}\,l + \mathit{total}\,r\end{aligned}$$

Now, if we abstract away from the recursive call, we obtain a non-recursive base function of type $(\mu\mathfrak{Bush} \to \mathit{Nat}) \to (\mu\mathfrak{Bush} \to \mathit{Nat})$. As with functors, we adopt the convention that the base function is named after the underlying function, using $\mathfrak{this}$ font for the former and *this* font for the latter.

$$\begin{aligned}&\mathfrak{total} : (\mu\mathfrak{Bush} \to \mathit{Nat}) \to (\mu\mathfrak{Bush} && \to \mathit{Nat})\\&\mathfrak{total}\ \ \mathit{total}\qquad\qquad\ (\mathit{In}\,(\mathfrak{Leaf}\,n)) && = n\\&\mathfrak{total}\ \ \mathit{total}\qquad\qquad\ (\mathit{In}\,(\mathfrak{Fork}\,(l,r))) && = \mathit{total}\,l + \mathit{total}\,r\end{aligned}$$

Functions of this type possibly have many fixed points—consider as an extreme example the identity base function, which has an infinite number of fixed points. Interestingly, the problem of ambiguity disappears into thin air, if we additionally remove the constructor *In*.

$$\begin{aligned}&\mathfrak{total} : \forall x\,.\,(x \to \mathit{Nat}) \to (\mathfrak{Bush}\,x && \to \mathit{Nat})\\&\mathfrak{total}\qquad \mathit{total}\qquad\ (\mathfrak{Leaf}\,n) && = n\\&\mathfrak{total}\qquad \mathit{total}\qquad\ (\mathfrak{Fork}\,(l,r)) && = \mathit{total}\,l + \mathit{total}\,r\end{aligned}$$

The type of the base function has become polymorphic in the argument of the recursive call. We shall show in the next section that this type guarantees that the recursive definition of *total*

$$\begin{aligned}&\mathit{total} : \mu\mathfrak{Bush} \to \mathit{Nat}\\ &\mathit{total}\ \ (\mathit{In}\, s) \ = \mathfrak{total}\, \mathit{total}\, s\end{aligned}$$

is well-defined in the sense that the equation has exactly one solution.

Applying an analogous transformation to the type *Tree* and the function *generate* we obtain

$$\begin{aligned}&\mathbf{data}\, \mathfrak{Tree}\, \mathit{tree} = \mathfrak{Branch}\, (\mathit{tree}, \mathit{Nat}, \mathit{tree})\\ &\mathfrak{generate} : \forall x\ .\ (\mathit{Nat} \to x) \to (\mathit{Nat} \to \mathfrak{Tree}\, x)\\ &\mathfrak{generate}\qquad \mathit{generate}\qquad n\\ &\quad = \mathfrak{Branch}\, (\mathit{generate}\, (2 * n + 0), n, \mathit{generate}\, (2 * n + 1))\\ &\mathit{generate} : \mathit{Nat} \to \nu\mathfrak{Tree}\\ &\mathit{generate}\ \ n \ \ = \ \mathit{Out}^{\circ}\, (\mathfrak{generate}\, \mathit{generate}\, n)\ \ .\end{aligned}$$

Again, the base function enjoys a polymorphic type that guarantees that the recursive function is well-defined.

Abstracting away from the particulars of the syntax, the examples suggest the consideration of fixed-point equations of the form

$$x \cdot \mathit{in} = \Psi\, x, \qquad \text{and dually} \qquad \mathit{out} \cdot x = \Psi\, x\ \ , \tag{91}$$

where the unknown x has type $\mathscr{C}(\mu\mathsf{F}, A)$ on the left and $\mathscr{C}(A, \nu\mathsf{G})$ on the right. The Haskell definitions above are pointwise versions of these equations: $x\, (\mathit{In}\, a) = \Psi\, x\, a$ and $x\, a = \mathit{Out}^{\circ}\, (\Psi\, x\, a)$. Arrows defined by equations of this form are known as *Mendler-style folds and unfolds*, because they were originally introduced by Mendler [21] in the setting of type theory. We shall usually drop the qualifier and call the solutions simply folds and unfolds. In fact, the abuse of language is justified as each Mendler-style equation is equivalent to the defining equation of a standard (un)fold. This is what we show next, considering folds first.

3.1.1 Initial Fixed-Point Equations. Let $\mathscr{C}$ be some base category and let $\mathsf{F} : \mathscr{C} \to \mathscr{C}$ be some endofunctor. An *initial fixed-point equation* in the unknown $x : \mathscr{C}(\mu\mathsf{F}, A)$ has the syntactic form

$$x \cdot \mathit{in} = \Psi\, x\ \ , \tag{92}$$

where the base function Ψ has type

$$\Psi : \forall X\ .\ \mathscr{C}(X, A) \to \mathscr{C}(\mathsf{F}\, X, A)\ \ .$$

In the fixed-point equation (92) the natural transformation Ψ is instantiated to the initial algebra: $x \cdot \mathit{in} = \Psi\, (\mu\mathsf{F})\, x$. For reasons of readability we will usually omit the 'type arguments' of natural transformations. The diagram below displays the types involved.

$$\begin{tikzcd}
\mathsf{F}\,(\mu\mathsf{F}) \arrow[d, "in"'] \arrow[dr, "\Psi\,(\mu\mathsf{F})\,x"] & \\
\mu\mathsf{F} \arrow[r, "x"'] & A
\end{tikzcd}$$

The naturality condition can be seen as the semantic counterpart of the *guarded-by-destructors* condition [22]. This becomes visible if we move the isomorphism $in : \mathsf{F}\,(\mu\mathsf{F}) \cong \mu\mathsf{F}$ to the right-hand side: $x = \Psi\, x \cdot in^{\circ}$. Here in° is the destructor that guards the recursive calls. The equation has a straightforward operational reading. The argument of x is destructed yielding an element of type $\mathsf{F}\,(\mu\mathsf{F})$. The base function Ψ then works on the F-structure, possibly applying its first argument, the recursive call of x, to elements of type $\mu\mathsf{F}$. These elements are proper sub-terms of the original argument—recall that the type argument of F marks the recursive components. The naturality of Ψ ensures that *only* these sub-terms can be passed to the recursive calls.

Does this imply that x is terminating? Termination is an operational notion; how the notion translates to a denotational setting depends on the underlying category. Our primary goal is to show that Equation (92) has a *unique solution*. When working in **Set** this result implies that the equation admits a solution that is indeed a total function. Furthermore, the operational reading of $x = \Psi\, x \cdot in^{\circ}$ suggests that x is terminating, as elements of an inductive type can only be destructed a finite number of times. (Depending on the evaluation strategy this claim is also subject to the proviso that the F-structures themselves are finite.) On the other hand, if the underlying category is **SCpo**, then the solution is a continuous function that does not necessarily terminate for all its inputs, since initial algebras in **SCpo** possibly contain infinite elements.

While the definition of *total* fits nicely into the framework above, the following program does not.

Example 4. The naturality condition is sufficient but not necessary as the example of the binary increment demonstrates.

$$\begin{aligned}
&\mathbf{data}\, Nat = N \mid O\, Nat \mid I\, Nat\\
&succ : Nat \to Nat\\
&succ\ (N) \quad = I\, N\\
&succ\ (O\, b) = I\, b\\
&succ\ (I\ b) \ = O\,(succ\, b)
\end{aligned}$$

As with *total*, we split the datatype into two levels.

$$\begin{aligned}
&\mathbf{type}\, Nat = \mu\mathfrak{Nat}\\
&\mathbf{data}\, \mathfrak{Nat}\, nat = \mathfrak{N} \mid \mathfrak{O}\, nat \mid \mathfrak{I}\, nat
\end{aligned}$$

$$\begin{array}{l} \textbf{instance}\ \mathit{Functor}\ \mathfrak{Nat}\ \textbf{where} \\ \quad \mathit{fmap}\ f\ (\mathfrak{N}) \quad = \mathfrak{N} \\ \quad \mathit{fmap}\ f\ (\mathfrak{O}\ b) = \mathfrak{O}\ (f\ b) \\ \quad \mathit{fmap}\ f\ (\mathfrak{I}\ \ b) = \mathfrak{I}\ \ (f\ b) \end{array}$$

In **Set**, the implementation of the successor function is clearly terminating. However, the associated base function

$$\begin{array}{llll} \mathfrak{succ} : (\mathit{Nat} \to \mathit{Nat}) \to (\mathfrak{Nat}\,\mathit{Nat} \to \mathit{Nat}) & & & \\ \mathfrak{succ}\ \ \mathit{succ} & (\mathfrak{N}) & = & \mathit{In}\,(\mathfrak{I}\,(\mathit{In}\,\mathfrak{N})) \\ \mathfrak{succ}\ \ \mathit{succ} & (\mathfrak{O}\,b) & = & \mathit{In}\,(\mathfrak{I}\,b) \\ \mathfrak{succ}\ \ \mathit{succ} & (\mathfrak{I}\ \ b) & = & \mathit{In}\,(\mathfrak{O}\,(\mathit{succ}\,b)) \end{array}$$

lacks naturality. In a sense, its type is too concrete, as it reveals that the recursive call is passed a binary number. An adversary can make use of this information turning the terminating program into a non-terminating one:

$$\begin{array}{llll} \mathfrak{bogus} : (\mathit{Nat} \to \mathit{Nat}) \to (\mathfrak{Nat}\,\mathit{Nat} \to \mathit{Nat}) & & & \\ \mathfrak{bogus}\ \ \mathit{succ} & (\mathfrak{N}) & = & \mathit{In}\,(\mathfrak{I}\,(\mathit{In}\,\mathfrak{N})) \\ \mathfrak{bogus}\ \ \mathit{succ} & (\mathfrak{O}\,b) & = & \mathit{In}\,(\mathfrak{I}\,b) \\ \mathfrak{bogus}\ \ \mathit{succ} & (\mathfrak{I}\ \ b) & = & \mathit{succ}\,(\mathit{In}\,(\mathfrak{I}\,b))\ \ . \end{array}$$

We will get back to this example in Section 3.3.2 (Example 19). □

Turning to the proof of uniqueness, let us spell out the naturality property of the base function Ψ. If $h : \mathscr{C}(X_1, X_2)$, then $\mathscr{C}(\mathsf{F}\,h, id) \cdot \Psi = \Psi \cdot \mathscr{C}(h, id)$. Using the definition of hom-functors (6), this unfolds to

$$\Psi\,f \cdot \mathsf{F}\,h = \Psi\,(f \cdot h)\ \ , \tag{93}$$

for all arrows $f : \mathscr{C}(X_2, A)$. This property implies, in particular, that Ψ is completely determined by its image of id as $\Psi\,h = \Psi\,id \cdot \mathsf{F}\,h$. Now, to prove that equation $x \cdot in = \Psi\,x$ (92) has a *unique* solution, we show that x is a solution if and only if x is a standard fold.

$$\begin{array}{ll} & x \cdot in = \Psi\,x \\ \Longleftrightarrow & \{\ \Psi \text{ is natural (93)}\ \} \\ & x \cdot in = \Psi\,id \cdot \mathsf{F}\,x \\ \Longleftrightarrow & \{\ \text{uniqueness property of standard folds (54)}\ \} \\ & x = (\!|\Psi\,id|\!) \end{array}$$

Overloading the banana brackets, the unique solution of the fixed-point equation $x \cdot in = \Psi\,x$ (92) is written $(\!|\Psi|\!)$.

Let us explore the relation between standard folds and Mendler-style folds in more depth. The proof above rests on the fact that the type of Ψ is isomorphic to $\mathscr{C}(\mathsf{F}\,A, A)$, the type of F-algebras.

$$\mathscr{C}(\mathsf{F}\,A, A) \cong (\forall X : \mathscr{C}\,.\,\mathscr{C}(X, A) \to \mathscr{C}(\mathsf{F}\,X, A))\ \ .$$

This bijection between arrows and natural transformations is an instance of the *Yoneda Lemma* (Section 2.7), where the contravariant functor $\mathsf{H} : \mathscr{C}^{\mathsf{op}} \to \mathbf{Set}$ is given by $\mathsf{H} = \mathscr{C}(\mathsf{F}\,-, A)$. Consequently, Mendler-style folds and standard folds are related by $(\!|\Psi|\!) = (\!|\mathsf{y}^{\circ}\,\Psi|\!) = (\!|\Psi\ id|\!)$ and $(\!|\lambda\, x\ .\ a \cdot \mathsf{F}\, x|\!) = (\!|\mathsf{y}\ a|\!) = (\!|a|\!)$.

Example 1. The standard fold for computing the total of a bush of natural numbers is $(\!|id \triangledown plus|\!)$, see Section 2.6.1. Written in a pointwise style, the algebra $id \triangledown plus$ reads

$$\begin{aligned}
&\mathsf{total} : \mathfrak{Bush}\ Nat && \to Nat\\
&\mathsf{total}\ \ (\mathfrak{Leaf}\ n) && = n\\
&\mathsf{total}\ \ (\mathfrak{Fork}\ (l, r)) && = l + r\ \ .
\end{aligned}$$

The algebra and the base function are related by $\mathsf{total} = \mathfrak{total}\ id$ and $\mathfrak{total}\ total = \mathsf{total} \cdot fmap\ total$, which implies $(\!|\mathsf{total}|\!) = (\!|\mathfrak{total}|\!)$. □

3.1.2 Final Fixed-Point Equations. The development of the previous section dualises to final coalgebras. For completeness, let us spell out the details.

A *final fixed-point equation* in the unknown $x : \mathscr{C}(A, \nu\mathsf{G})$ has the form

$$out \cdot x = \Psi\, x\ \ , \tag{94}$$

where the base function Ψ has type

$$\Psi : \forall X\ .\ \mathscr{C}(A, X) \to \mathscr{C}(A, \mathsf{G}\, X)\ \ .$$

Overloading the lens brackets, the unique solution of (94) is denoted $[\![\Psi]\!]$.

In **Set**, the naturality condition captures the *guarded-by-constructors* condition [22] ensuring *productivity*. Again, this can be seen more clearly if we move the isomorphism $out : \nu\mathsf{G} \cong \mathsf{G}\,(\nu\mathsf{G})$ to the right-hand side: $x = out^{\circ} \cdot \Psi\, x$. Here out° is the constructor that guards the recursive calls. The base function Ψ has to produce a $\mathsf{G}\,(\nu\mathsf{G})$ structure. To create the recursive components of type $\nu\mathsf{G}$, the base function Ψ can use its first argument, the recursive call of x. However, the naturality of Ψ ensures that these calls can *only* be made in guarded positions.

The type of Ψ is isomorphic to $\mathscr{C}(A, \mathsf{G}\, A)$, the type of G-coalgebras.

$$\mathscr{C}(A, \mathsf{G}\, A) \cong (\forall X : \mathscr{C}\ .\ \mathscr{C}(A, X) \to \mathscr{C}(A, \mathsf{G}\, X))\ \ .$$

Again, this is an instance of the Yoneda Lemma: now $\mathsf{H} = \mathscr{C}(A, \mathsf{G}\,-)$ is a covariant functor $\mathsf{H} : \mathscr{D} \to \mathbf{Set}$.

Example 2. The standard unfold for constructing an infinite tree of natural numbers is $[\![shift0 \vartriangle id \vartriangle shift1]\!]$, see Section 2.6.2. Written in a pointwise style, the coalgebra $shift0 \vartriangle id \vartriangle shift1$ reads

$$\begin{aligned}
&\mathsf{generate} : Nat \to \mathfrak{Tree}\ Nat\\
&\mathsf{generate}\ \ n \quad = \mathfrak{Branch}\,(2 * n + 0, n, 2 * n + 1)\ \ .
\end{aligned}$$

The coalgebra and the base function are related by $\mathsf{generate} = \mathfrak{generate}\ id$ and $\mathfrak{generate}\ gen = fmap\ gen \cdot \mathsf{generate}$, which implies $[\![\mathsf{generate}]\!] = [\![\mathfrak{generate}]\!]$. □

In the following sections we show that fixed-point equations are quite general. More functions fit under this umbrella than one might initially think.

3.1.3 Mutual Type Recursion. In Haskell, datatypes can be defined by mutual recursion.

Example 5. Imagine a simple imperative programming language. The abstract syntax of expressions and statements is typically defined by mutual type recursion (this is a very stripped-down example).

$$\begin{aligned}&\mathbf{data}\, \mathit{Expr} = \mathit{Var}\ \mathit{Var} \mid \mathit{Block}\,(\mathit{Stat}, \mathit{Expr})\\ &\mathbf{data}\, \mathit{Stat}\ \ = \mathit{Assign}\,(\mathit{Var}, \mathit{Expr}) \mid \mathit{Seq}\,(\mathit{Stat}, \mathit{Stat})\end{aligned}$$

As function follows form, functions that consume an abstract syntax tree are typically defined by mutual value recursion.

$$\begin{aligned}
&\mathbf{type}\ \mathit{Vars} = \mathsf{Set}\ \mathit{Var}\\
&\mathit{varsExpr} : \mathit{Expr} \to \mathit{Vars}\\
&\mathit{varsExpr}\ \ (\mathit{Var}\,x) = \{x\}\\
&\mathit{varsExpr}\ \ (\mathit{Block}\,(s, e)) = \mathit{varsStat}\,s \cup \mathit{varsExpr}\,e\\
&\mathit{varsStat} : \mathit{Stat} \to \mathit{Vars}\\
&\mathit{varsStat}\ \ (\mathit{Assign}\,(x, e)) = \{x\} \cup \mathit{varsExpr}\,e\\
&\mathit{varsStat}\ \ (\mathit{Seq}\,(s_1, s_2)) = \mathit{varsStat}\,s_1 \cup \mathit{varsStat}\,s_2
\end{aligned}$$

The functions determine the variables of either an expression or a statement, assuming a suitable collection type Set with operations $\{-\}$ and $\cup$. □

Can we fit the above definitions into the framework of the previous section? Yes, we only have to choose a suitable base category: in this case, a product category (Section 2.2.2). The base functor underlying *Expr* and *Stat* is an endofunctor over a product category:

$$\mathsf{Grammar}\,\langle A,\ B\rangle = \langle\, \mathit{Var} + B \times A,\ \mathit{Var} \times A + B \times B\,\rangle\ .$$

The Haskell types *Expr* and *Stat* are then the components of the fixed point: $\mu\mathsf{Grammar} = \langle \mathit{Expr}, \mathit{Stat}\rangle$. The functions *varsExpr* and *varsStat* are handled accordingly: we bundle them to a single arrow

$$\mathit{vars} = \langle \mathit{varsExpr}, \mathit{varsStat}\rangle : (\mathscr{C} \times \mathscr{C})(\mu\mathsf{Grammar}, \langle\, \mathit{Vars},\ \mathit{Vars}\rangle)\ .$$

The following calculation makes explicit that an initial fixed-point equation in $\mathscr{C} \times \mathscr{D}$ corresponds to two equations, one in $\mathscr{C}$ and one in $\mathscr{D}$.

$$\begin{aligned}
&x \cdot \mathit{in} = \Psi\, x : (\mathscr{C} \times \mathscr{D})(\mathsf{F}\,(\mu\mathsf{F}), \langle A_1,\ A_2\rangle)\\
\iff&\ \{\ \text{surjective pairing: } f = \langle \mathsf{Outl}\,f,\ \mathsf{Outr}\,f\rangle\ \}\\
&\langle \mathsf{Outl}\,x,\ \mathsf{Outr}\,x\rangle \cdot \langle \mathsf{Outl}\,\mathit{in},\ \mathsf{Outr}\,\mathit{in}\rangle = \Psi\,\langle \mathsf{Outl}\,x,\ \mathsf{Outr}\,x\rangle\\
\iff&\ \{\ \text{set } x_1 = \mathsf{Outl}\,x,\ x_2 = \mathsf{Outr}\,x \text{ and } \mathit{in}_1 = \mathsf{Outl}\,\mathit{in},\ \mathit{in}_2 = \mathsf{Outr}\,\mathit{in}\ \}\\
&\langle x_1,\ x_2\rangle \cdot \langle \mathit{in}_1,\ \mathit{in}_2\rangle = \Psi\,\langle x_1,\ x_2\rangle\\
\iff&\ \{\ \text{definition of composition in } \mathscr{C} \times \mathscr{D}\ \}\\
&\langle x_1 \cdot \mathit{in}_1,\ x_2 \cdot \mathit{in}_2\rangle = \Psi\,\langle x_1,\ x_2\rangle\\
\iff&\ \{\ \text{surjective pairing: } f = \langle \mathsf{Outl}\,f,\ \mathsf{Outr}\,f\rangle\ \}\\
&\langle x_1 \cdot \mathit{in}_1,\ x_2 \cdot \mathit{in}_2\rangle = \langle \mathsf{Outl}\,(\Psi\,\langle x_1,\ x_2\rangle),\ \mathsf{Outr}\,(\Psi\,\langle x_1,\ x_2\rangle)\rangle
\end{aligned}$$

$$\begin{aligned}
&\iff \quad \{\ \text{set } \Psi_1 = \mathsf{Outl} \circ \Psi \text{ and } \Psi_2 = \mathsf{Outr} \circ \Psi\ \} \\
&\qquad \langle x_1 \cdot in_1,\ x_2 \cdot in_2 \rangle = \langle \Psi_1\, \langle x_1,\ x_2 \rangle,\ \Psi_2\, \langle x_1,\ x_2 \rangle \rangle \\
&\iff \quad \{\ \text{equality of arrows in } \mathscr{C} \times \mathscr{D}\ \} \\
&\qquad x_1 \cdot in_1 = \Psi_1\, \langle x_1,\ x_2 \rangle : \mathscr{C}(\mathsf{Outl}\,(\mathsf{F}\,(\mu\mathsf{F})), A_1) \quad \text{and} \\
&\qquad\quad x_2 \cdot in_2 = \Psi_2\, \langle x_1,\ x_2 \rangle : \mathscr{D}(\mathsf{Outr}\,(\mathsf{F}\,(\mu\mathsf{F})), A_2)
\end{aligned}$$

The base functions Ψ_1 and Ψ_2 are parametrised both with x_1 and x_2. Other than that, the syntactic form is identical to a standard fixed-point equation.

It is a simple exercise to bring the equations of Example 5 into this form:

Definition 6. In Haskell, mutually recursive types can be modelled as follows.

$$\begin{aligned}
&\textbf{newtype}\, \mu_1\, f_1\, f_2 = In_1\, \{\, in_1^{\circ} : f_1\, (\mu_1\, f_1\, f_2)\, (\mu_2\, f_1\, f_2)\, \} \\
&\textbf{newtype}\, \mu_2\, f_1\, f_2 = In_2\, \{\, in_2^{\circ} : f_2\, (\mu_1\, f_1\, f_2)\, (\mu_2\, f_1\, f_2)\, \}
\end{aligned}$$

Since Haskell has no concept of pairs on the type level, that is, no product kinds, we have to curry the type constructors: $\mu_1\, f_1\, f_2 = \mathsf{Outl}\,(\mu\langle f_1,\ f_2\rangle)$ and $\mu_2\, f_1\, f_2 = \mathsf{Outr}\,(\mu\langle f_1,\ f_2\rangle)$. □

Example 7. The base functors of *Expr* and *Stat* are

$$\begin{aligned}
&\textbf{data}\, \mathfrak{Expr}\ expr\ stat = \mathfrak{Var}\ Var \mid \mathfrak{Block}\,(stat, expr) \\
&\textbf{data}\, \mathfrak{Stat}\ expr\ stat = \mathfrak{Assign}\,(Var, expr) \mid \mathfrak{Seq}\,(stat, stat)\ .
\end{aligned}$$

Since all Haskell functions live in the same category, we have to represent arrows in $\mathscr{C} \times \mathscr{C}$ by pairs of arrows in $\mathscr{C}$.

$$\begin{aligned}
&\mathfrak{varsExpr} : \forall x_1\, x_2\ . \\
&\qquad (x_1 \to Vars, x_2 \to Vars) \to (\mathfrak{Expr}\, x_1\, x_2 \to Vars) \\
&\mathfrak{varsExpr}\ (varsExpr,\ varsStat)\ (\mathfrak{Var}\, x) = \{x\} \\
&\mathfrak{varsExpr}\ (varsExpr,\ varsStat)\ (\mathfrak{Block}\,(s, e)) = varsStat\ s \cup varsExpr\ e \\
&\mathfrak{varsStat} : \forall x_1\, x_2\ . \\
&\qquad (x_1 \to Vars, x_2 \to Vars) \to (\mathfrak{Stat}\, x_1\, x_2 \to Vars) \\
&\mathfrak{varsStat}\ (varsExpr,\ varsStat)\ (\mathfrak{Assign}\,(x, e)) = \{x\} \cup varsExpr\ e \\
&\mathfrak{varsStat}\ (varsExpr,\ varsStat)\ (\mathfrak{Seq}\,(s_1, s_2)) = varsStat\ s_1 \cup varsStat\ s_2
\end{aligned}$$

The definitions of *varsExpr* and *varsStat* match exactly the scheme above.

$$\begin{aligned}
&varsExpr : \mu_1\ \mathfrak{Expr}\ \mathfrak{Stat} \to Vars \\
&varsExpr\ \ (In_1\, e) = \mathfrak{varsExpr}\,(varsExpr, varsStat)\ e \\
&varsStat : \ \mu_2\ \mathfrak{Expr}\ \mathfrak{Stat} \to Vars \\
&varsStat\ \ (In_2\, s) = \mathfrak{varsStat}\,(varsExpr, varsStat)\ s
\end{aligned}$$

Since the two equations are equivalent to an initial fixed-point equation in $\mathscr{C} \times \mathscr{C}$, they indeed have unique solutions. □

No new theory is needed to deal with mutually recursive datatypes and mutually recursive functions over them. By duality, the same is true for final coalgebras. For final fixed-point equations we have the following correspondence.

$$out \cdot x = \Psi\, x \quad\Longleftrightarrow\quad out_1 \cdot x_1 = \Psi_1\,\langle x_1,\, x_2\rangle \text{ and } out_2 \cdot x_2 = \Psi_2\,\langle x_1,\, x_2\rangle$$

3.1.4 Type Functors. In Haskell, datatypes can be parametrised by types.

Example 8. The type of random-access lists [23] is given by

```
data Array a = Null | Zero (Array (a, a)) | One (a, Array (a, a))
instance Functor Array where
  fmap f (Null)       = Null
  fmap f (Zero s)     = Zero (fmap (f × f) s)
  fmap f (One (a, s)) = One (f a, fmap (f × f) s)
(×) : (â → ǎ) → (b̂ → b̌) → ((â, b̂) → (ǎ, b̌))
(f × g) (a, b) = (f a, g b)  .
```

The type Array is a so-called *nested datatype* [24] as the type argument changes from a to (a, a) in the recursive calls. Random-access lists are a numerical representation, a container type that is modelled after a number system, here the binary numbers.

```
size : ∀ a . Array a    → Nat
size      (Null)        = 0
size      (Zero s)      = 2 ∗ size s + 0
size      (One (a, s))  = 2 ∗ size s + 1
```

The function *size* calculates the size of a random-access list, illustrating the correspondence between random-access lists and binary numbers. The definition requires *polymorphic recursion* [25], as the recursive calls have type $\mathsf{Array}\,(a, a) \to Nat$, which is a substitution instance of the declared type. □

Can we fit the definitions above into the framework of Section 3.1.1? Again, the answer is yes. We only have to choose a suitable base category: this time, a functor category (Section 2.2.3). The base functor of Array is an endofunctor over a functor category:

$$\mathfrak{Array}\,\mathsf{F}\,A = 1 + \mathsf{F}\,(A \times A) + A \times \mathsf{F}\,(A \times A) \enspace .$$

The second-order functor $\mathfrak{Array}$ sends a functor to a functor. Since its fixed point $\mathsf{Array} = \mu\mathfrak{Array}$ lives in a functor category, folds over random-access lists are necessarily natural transformations. The function *size* is a natural transformation, as we can assign it the type

$$size : \mu\mathfrak{Array} \mathrel{\dot{\to}} \mathsf{K}\,Nat \enspace ,$$

where $\mathsf{K} : \mathscr{D} \to \mathscr{D}^{\mathscr{C}}$ is the constant functor defined $\mathsf{K}\,A\,B = A$. Again, we can replay the development in Haskell.

Definition 9. The definition of second-order initial algebras and final coalgebras is identical to that of Definition 3, except for an additional type argument.

$$\begin{aligned}&\textbf{newtype}\,\mu f\ a = \mathit{In}\quad\ \{\mathit{in}^{\circ} : f\ (\mu f)\ a\}\\&\textbf{newtype}\,\nu f\ a = \mathit{Out}^{\circ}\,\{\mathit{out} : f\ (\nu f)\ a\}\end{aligned}$$

To capture the fact that μf and νf are functors whenever f is a second-order functor, we need an extension of the Haskell 2010 class system.

$$\begin{aligned}&\textbf{instance}\,(\forall x\ .\ (\mathit{Functor}\ x) \Rightarrow \mathit{Functor}\ (f\ x)) \Rightarrow \mathit{Functor}\ (\mu f)\ \textbf{where}\\&\quad \mathit{fmap}\ f\ (\mathit{In}\quad s) = \mathit{In}\quad (\mathit{fmap}\ f\ s)\\&\textbf{instance}\,(\forall x\ .\ (\mathit{Functor}\ x) \Rightarrow \mathit{Functor}\ (f\ x)) \Rightarrow \mathit{Functor}\ (\nu f)\ \textbf{where}\\&\quad \mathit{fmap}\ f\ (\mathit{Out}^{\circ}\ s) = \mathit{Out}^{\circ}\ (\mathit{fmap}\ f\ s)\end{aligned}$$

The declarations use a so-called *polymorphic predicate* [26], which precisely captures the requirement that f sends functors to functors. Unfortunately, the extension has not been implemented yet. It can be simulated within Haskell 2010 [27], but the resulting code is somewhat clumsy. Alternatively, one can use 'recursive dictionaries'

$$\begin{aligned}&\textbf{instance}\,\mathit{Functor}\ (f\ (\mu f)) \Rightarrow \mathit{Functor}\ (\mu f)\ \textbf{where}\\&\quad \mathit{fmap}\ f\ (\mathit{In}\quad s) = \mathit{In}\quad (\mathit{fmap}\ f\ s)\\&\textbf{instance}\,\mathit{Functor}\ (f\ (\nu f)) \Rightarrow \mathit{Functor}\ (\nu f)\ \textbf{where}\\&\quad \mathit{fmap}\ f\ (\mathit{Out}^{\circ}\ s) = \mathit{Out}^{\circ}\ (\mathit{fmap}\ f\ s)\end{aligned}$$

and rely on the compiler to tie the recursive knot [28]. □

Let us specialise fixed-point equations to functor categories.

$$\begin{aligned}&\quad x \cdot \mathit{in} = \Psi\ x\\\iff&\quad \{\ \text{equality of arrows in } \mathscr{D}^{\mathscr{C}}\ \}\\&\quad \forall A : \mathscr{C}\ .\ (x \cdot \mathit{in})\ A = \Psi\ x\ A\\\iff&\quad \{\ \text{definition of composition in } \mathscr{D}^{\mathscr{C}}\ \}\\&\quad \forall A : \mathscr{C}\ .\ x\ A \cdot \mathit{in}\ A = \Psi\ x\ A\end{aligned}$$

In Haskell, type application is invisible, so fixed-point equations in functor categories cannot be distinguished from equations in the base category.

Example 10. Continuing Example 8, the base functor of Array maps functors to functors: it has kind $(\star \to \star) \to (\star \to \star)$.

$$\begin{aligned}&\textbf{data}\,\mathfrak{Array}\ \mathit{array}\ a = \mathfrak{Null} \mid \mathfrak{Zero}\ (\mathit{array}\ (a, a)) \mid \mathfrak{One}\ (a, \mathit{array}\ (a, a))\\&\textbf{instance}\ (\mathit{Functor}\ \mathit{array}) \Rightarrow \mathit{Functor}\ (\mathfrak{Array}\ \mathit{array})\ \textbf{where}\\&\quad \mathit{fmap}\ f\ (\mathfrak{Null}) \qquad\quad = \mathfrak{Null}\\&\quad \mathit{fmap}\ f\ (\mathfrak{Zero}\ s) \qquad = \mathfrak{Zero}\ (\mathit{fmap}\ (f \times f)\ s)\\&\quad \mathit{fmap}\ f\ (\mathfrak{One}\ (a, s)) = \mathfrak{One}\ (f\ a, \mathit{fmap}\ (f \times f)\ s)\end{aligned}$$

Table 3. Initial algebras and final coalgebras in different categories.

category	initial fixed-point equation $x \cdot in = \Psi\, x$	final fixed-point equation $out \cdot x = \Psi\, x$
Set	inductive type standard fold	coinductive type standard unfold
Cpo	—	continuous coalgebra (domain) continuous unfold (F locally continuous in **SCpo**)
SCpo	continuous algebra (domain) strict continuous fold (F locally continuous in **SCpo**, $\mu\mathsf{F} \cong \nu\mathsf{F}$)	continuous coalgebra (domain) strict continuous unfold
$\mathscr{C} \times \mathscr{D}$	mutually recursive inductive types mutually recursive folds $x_1 \cdot in_1 = \Psi_1\, \langle x_1,\, x_2\rangle$ $x_2 \cdot in_2 = \Psi_2\, \langle x_1,\, x_2\rangle$	mutually recursive coinductive types mutually recursive unfolds $out_1 \cdot x_1 = \Psi_1\, \langle x_1,\, x_2\rangle$ $out_2 \cdot x_2 = \Psi_2\, \langle x_1,\, x_2\rangle$
$\mathscr{D}^{\mathscr{C}}$	inductive type functor higher-order fold $x\, A \cdot in\, A = \Psi\, x\, A$	coinductive type functor higher-order unfold $out\, A \cdot x\, A = \Psi\, x\, A$

Its action on arrows, not shown above, maps natural transformations to natural transformations. Accordingly, the base function of *size* is a second-order natural transformation that takes natural transformations to natural transformations.

$$\begin{aligned}
&\mathfrak{size} : \forall x\, .\, (\forall a\, .\, x\, a \to Nat) \to (\forall a\, .\, \mathfrak{Array}\, x\, a \to Nat)\\
&\mathfrak{size}\ \ size\ \ (\mathfrak{Null}) = 0\\
&\mathfrak{size}\ \ size\ \ (\mathfrak{Zero}\, s) = 2 * size\, s + 0\\
&\mathfrak{size}\ \ size\ \ (\mathfrak{One}\, (a, s)) = 2 * size\, s + 1
\end{aligned}$$

$$\begin{aligned}
&size : \forall a\, .\, \mu\mathfrak{Array}\, a \to Nat\\
&size\ \ (In\, s) = \mathfrak{size}\, size\, s
\end{aligned}$$

The resulting equation fits the pattern of an initial fixed-point equation (type application is invisible in Haskell). Consequently, it has a unique solution. □

Table 3 summarises our findings so far. To provide some background, **Cpo** is the category of complete partial orders and continuous functions; **SCpo** is the full subcategory of strict functions. A functor $\mathsf{F} : \mathbf{SCpo} \to \mathbf{SCpo}$ is *locally continuous* if its action on arrows $\mathbf{SCpo}(A, B) \to \mathbf{SCpo}(\mathsf{F}\, A, \mathsf{F}\, B)$ is continuous for any pair of objects A and B. A continuous algebra is just an algebra whose carrier is a complete partial order and whose action is a continuous function. In **SCpo**, every locally continuous functor has an initial algebra and, furthermore, the initial algebra coincides with the final coalgebra. This is the reason why **SCpo** is commonly considered to be Haskell's ambient category. It may seem odd at first that lazy programs are modelled by strict functions. Non-strict functions, however, are in one-to-one correspondence to strict functions from a lifted domain: $\mathbf{SCpo}(A_\perp, B) \cong \mathbf{Cpo}(A, B)$. (In other words, we have an adjunction

$(-)_\perp \dashv \mathsf{Incl}$ between lifting and the inclusion functor $\mathsf{Incl} : \mathbf{SCpo} \to \mathbf{Cpo}$.) The denotational notion of lifting, adding a new least element, models the operational notion of a thunk (also known as a closure, laze or recipe).

3.2 Adjoint Fixed-Point Equations

We have seen in the previous section that initial and final fixed-point equations are quite general. However, there are obviously a lot of definitions that do not fit the pattern. We have mentioned list concatenation and others in the introduction.

Example 11. The datatype *Stack* models stacks of natural numbers.

$$\mathbf{data}\, \mathit{Stack} = \mathit{Empty} \mid \mathit{Push}\,(\mathit{Nat}, \mathit{Stack})$$

The function *cat* concatenates two stacks.

$$\begin{aligned}
&\mathit{cat} : (\mathit{Stack}, && \mathit{Stack}) \to \mathit{Stack} \\
&\mathit{cat}\ \ (\mathit{Empty}, && \mathit{ns}) \quad = \mathit{ns} \\
&\mathit{cat}\ \ (\mathit{Push}\,(m, \mathit{ms}), \mathit{ns}) && \quad\ \ = \mathit{Push}\,(m, \mathit{cat}\,(\mathit{ms}, \mathit{ns}))
\end{aligned}$$

The definition does not fit the pattern of an initial fixed-point equation as it takes two arguments and recurses only over the first one. □

Example 12. The functions *left* and *right* generate infinite trees labelled with zeros and ones.

$$\begin{aligned}
&\mathit{left} : () \to \mathit{Tree} \\
&\mathit{left}\ \ () = \mathit{Branch}\,(\mathit{left}\,(), 0, \mathit{right}\,()) \\
&\mathit{right} : () \to \mathit{Tree} \\
&\mathit{right}\ \ () = \mathit{Branch}\,(\mathit{left}\,(), 1, \mathit{right}\,())
\end{aligned}$$

The two definitions are not instances of final fixed-point equations, because even though the functions are mutually recursive the datatype is not. □

In Example 11 the element of the initial algebra is embedded in a context. Written in a point-free style the definition of *cat* is of the form $x \cdot (\mathit{in} \times \mathit{id}) = \Psi\, x$. The central idea of these lecture notes is to model this context by a functor, generalising fixed-point equations to

$$x \cdot \mathsf{L}\,\mathit{in} = \Psi\, x, \qquad \text{and dually} \qquad \mathsf{R}\,\mathit{out} \cdot x = \Psi\, x \ \ , \tag{95}$$

where the unknown x has type $\mathscr{C}(\mathsf{L}\,(\mu\mathsf{F}), A)$ on the left and $\mathscr{C}(A, \mathsf{R}\,(\nu\mathsf{G}))$ on the right. The functor L models the context of $\mu\mathsf{F}$. In the case of *cat* the functor is $\mathsf{L} = - \times \mathit{Stack}$. Dually, R allows x to return an element of $\nu\mathsf{G}$ embedded in a context. Section 3.3.2 discusses a suitable choice for R in Example 12.

Of course, the functors L and R cannot be arbitrary. For instance, for $\mathsf{L} = \mathsf{K}\,A$ where $\mathsf{K} : \mathscr{C} \to \mathscr{C}^{\mathscr{D}}$ is the constant functor and $\Psi = \mathit{id}$, the equation $x \cdot \mathsf{L}\,\mathit{in} = \Psi\, x$ simplifies to $x = x$, which every arrow of the appropriate type satisfies. One approach for ensuring uniqueness is to require L and R to be adjoint: $\mathsf{L} \dashv \mathsf{R}$ (Section 2.5). The adjoint transposition allows us to trade L in the source for R in the target of an arrow, which is the key for showing that generalised fixed-point equations (95) have unique solutions. This is what we do next.

3.2.1 Adjoint Initial Fixed-Point Equations. Let $\mathscr{C}$ and $\mathscr{D}$ be categories, let $\mathsf{L} \dashv \mathsf{R}$ be an adjoint pair of functors $\mathsf{L} : \mathscr{C} \leftarrow \mathscr{D}$ and $\mathsf{R} : \mathscr{C} \rightarrow \mathscr{D}$, and let $\mathsf{F} : \mathscr{D} \rightarrow \mathscr{D}$ be some endofunctor. An *adjoint initial fixed-point equation* in the unknown $x : \mathscr{C}(\mathsf{L}\,(\mu\mathsf{F}), A)$ has the syntactic form

$$x \cdot \mathsf{L}\,\mathit{in} = \Psi\,x \ , \tag{96}$$

where the base function Ψ has type

$$\Psi : \forall X : \mathscr{D} \,.\, \mathscr{C}(\mathsf{L}\,X, A) \rightarrow \mathscr{C}(\mathsf{L}\,(\mathsf{F}\,X), A) \ .$$

The unique solution of (96) is called an *adjoint fold*, denoted $(\!|\Psi|\!)_{\mathsf{L}}$. The diagrams below summarise the type information.

$$\begin{tikzcd}
\mathsf{L}\,(\mathsf{F}\,(\mu\mathsf{F})) \arrow[d, "\mathsf{L}\,\mathit{in}"'] \arrow[dr, "\Psi\,(\mu\mathsf{F})\,x"] & \\
\mathsf{L}\,(\mu\mathsf{F}) \arrow[r, "x"'] & A
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathscr{C} \arrow[r, bend right, "\mathsf{R}"'] \arrow[r, phantom, "\perp"] & \mathscr{D} \arrow[l, bend right, "\mathsf{L}"'] \arrow[loop right, "\mathsf{F}"]
\end{tikzcd}$$

The proof of uniqueness makes essential use of the fact that the left adjunct is natural in A.

$$\begin{aligned}
& x \cdot \mathsf{L}\,\mathit{in} = \Psi\,x \\
\iff & \quad \{ \text{ adjunction: } \lceil \lfloor f \rfloor \rceil = f \ (27) \ \} \\
& \lfloor x \cdot \mathsf{L}\,\mathit{in} \rfloor = \lfloor \Psi\,x \rfloor \\
\iff & \quad \{ \text{ fusion: } \lfloor - \rfloor \text{ is natural in } A \ (37) \ \} \\
& \lfloor x \rfloor \cdot \mathit{in} = \lfloor \Psi\,x \rfloor \\
\iff & \quad \{ \text{ adjunction: } \lceil \lfloor f \rfloor \rceil = f \ \} \\
& \lfloor x \rfloor \cdot \mathit{in} = \lfloor \Psi\,\lceil \lfloor x \rfloor \rceil \rfloor \\
\iff & \quad \{ \text{ Section 3.1.1 } \} \\
& \lfloor x \rfloor = (\!|\lambda\,x \,.\, \lfloor \Psi\,\lceil x \rceil \rfloor|\!) \\
\iff & \quad \{ \text{ adjunction: } f = \lceil g \rceil \iff \lfloor f \rfloor = g \ \ (27) \ \} \\
& x = \lceil (\!|\lambda\,x \,.\, \lfloor \Psi\,\lceil x \rceil \rfloor|\!) \rceil
\end{aligned}$$

In three simple steps we have transformed the adjoint fold $x : \mathscr{C}(\mathsf{L}\,(\mu\mathsf{F}), A)$ into the standard fold $\lfloor x \rfloor : \mathscr{D}(\mu\mathsf{F}, \mathsf{R}\,A)$ and, alongside, the adjoint base function $\Psi : \forall X \,.\, \mathscr{C}(\mathsf{L}\,X, A) \rightarrow \mathscr{C}(\mathsf{L}\,(\mathsf{F}\,X), A)$ into the standard base function $\lambda\,x \,.\, \lfloor \Psi\,\lceil x \rceil \rfloor : \forall X \,.\, \mathscr{D}(X, \mathsf{R}\,A) \rightarrow \mathscr{D}(\mathsf{F}\,X, \mathsf{R}\,A)$. We have shown in Section 3.1.1 that the resulting equation has a unique solution. To summarise,

$$(\!|\Psi|\!)_{\mathsf{L}} = \lceil (\!|\lambda\,x \,.\, \lfloor \Psi\,\lceil x \rceil \rfloor|\!) \rceil \quad \text{or, equivalently,} \quad \lfloor (\!|\Psi|\!)_{\mathsf{L}} \rfloor = (\!|\lambda\,x \,.\, \lfloor \Psi\,\lceil x \rceil \rfloor|\!) \ .$$

3.2.2 Adjoint Final Fixed-Point Equations. Dually, an *adjoint final fixed-point equation* in the unknown $x : \mathscr{D}(A, \mathsf{R}\,(\nu\mathsf{G}))$ has the syntactic form

$$\mathsf{R}\,out \cdot x = \Psi\,x \enspace , \tag{97}$$

where the base function Ψ has type

$$\Psi : \forall X : \mathscr{C} \,.\, \mathscr{D}(A, \mathsf{R}\,X) \to \mathscr{D}(A, \mathsf{R}\,(\mathsf{G}\,X)) \enspace .$$

The unique solution of (97) is called an *adjoint unfold*, denoted $[\![\Psi]\!]_{\mathsf{R}}$.

3.3 Exploring Adjunctions

The simplest example of an adjunction is $\mathsf{Id} \dashv \mathsf{Id}$, which demonstrates that adjoint fixed-point equations (95) subsume fixed-point equations (91).

$$\begin{tikzcd} \mathscr{C} \arrow[r, phantom, "\perp"] \arrow[r, bend right, "\mathsf{Id}"'] & \mathscr{C} \arrow[l, bend right, "\mathsf{Id}"'] \end{tikzcd}$$

In the following sections we explore more interesting examples. Each section is structured as follows: we introduce an adjunction, specialise Equations (95) to the adjoint functors, and then provide some Haskell examples that fit the pattern.

3.3.1 Currying. The Haskell programmer's favourite adjunction is perhaps currying: $- \times X \dashv (-)^X$ (Section 2.5.6). Let us specialise the adjoint equations to $\mathsf{L} = - \times X$ and $\mathsf{R} = (-)^X$ in **Set**.

$$\begin{aligned} x \cdot \mathsf{L}\,in = \Psi\,x \quad &\iff \quad \forall\, a, c \,.\, x\,(in\,a, c) = \Psi\,x\,(a, c) \\ \mathsf{R}\,out \cdot x = \Psi\,x \quad &\iff \quad \forall\, a, c \,.\, out\,(x\,a\,c) = \Psi\,x\,a\,c \end{aligned}$$

The adjoint fold takes two arguments, an element of an initial algebra and a second argument (often an accumulator, see Example 14), both of which are available on the right-hand side. The transposed fold (not shown) is a higher-order function that yields a function. Dually, a curried unfold is transformed into an uncurried unfold.

Example 13. To turn the definition of *cat*, see Example 11, into the form of an adjoint equation, we follow the same steps as in Section 3.1. First, we turn *Stack* into a two-level type.

```
type Stack = μ𝔖𝔱𝔞𝔠𝔨
data 𝔖𝔱𝔞𝔠𝔨 stack = 𝔈𝔪𝔭𝔱𝔶 | 𝔓𝔲𝔰𝔥 (Nat, stack)
instance Functor 𝔖𝔱𝔞𝔠𝔨 where
  fmap f (𝔈𝔪𝔭𝔱𝔶)        = 𝔈𝔪𝔭𝔱𝔶
  fmap f (𝔓𝔲𝔰𝔥 (n, s)) = 𝔓𝔲𝔰𝔥 (n, f s)
```

Second, we determine the base function abstracting away from the recursive call, additionally removing in, and then we tie the recursive knot ($\mathsf{L} = - \times \mathit{Stack}$).

$$\begin{array}{llll}
\mathfrak{cat} : \forall x \,.\, (\mathsf{L}\, x \to \mathit{Stack}) \to (\mathsf{L}\,(\mathfrak{Stack}\, x) & & \to \mathit{Stack}) \\
\mathfrak{cat} \quad \mathit{cat} & (\mathfrak{Empty}, & ns) & = ns \\
\mathfrak{cat} \quad \mathit{cat} & (\mathfrak{Push}\,(m, ms), & ns) & = \mathit{Push}\,(m, \mathit{cat}\,(ms, ns))
\end{array}$$

$$\begin{array}{ll}
\mathit{cat} : \mathsf{L}\, \mathit{Stack} & \to \mathit{Stack} \\
\mathit{cat} \quad (\mathit{In}\, ms, ns) & = \mathfrak{cat}\, \mathit{cat}\,(ms, ns)
\end{array}$$

The defining equation fits the pattern of an adjoint initial fixed-point equation, $x \cdot (in \times id) = \Psi\, x$. Since $\mathsf{L} = - \times \mathit{Stack}$ has a right adjoint, cat is uniquely defined. The transposed fold, $cat' = \lfloor cat \rfloor$,

$$\begin{array}{ll}
\mathit{cat}' : \mathit{Stack} & \to \mathsf{R}\, \mathit{Stack} \\
\mathit{cat}' \quad (\mathit{In}\, \mathfrak{Empty}) & = \lambda ns \to ns \\
\mathit{cat}' \quad (\mathit{In}\,(\mathfrak{Push}\,(m, ms))) & = \lambda ns \to \mathit{Push}\,(m, (\mathit{cat}'\, ms)\, ns)
\end{array}$$

is simply the curried variant of cat. □

Example 14. The function $shunt$ pushes the elements of the first onto the second stack.

$$\begin{array}{lll}
\mathit{shunt} : (\mu\mathfrak{Stack}, & \mathit{Stack}) & \to \mathit{Stack} \\
\mathit{shunt} \quad (\mathit{In}\, \mathfrak{Empty}, & ns) & = ns \\
\mathit{shunt} \quad (\mathit{In}\,(\mathfrak{Push}\,(m, ms)), & ns) & = \mathit{shunt}\,(ms, \mathit{Push}\,(m, ns))
\end{array}$$

Unlike cat, the parameter of $shunt$ is changed in the recursive call—it serves as an accumulator. Nonetheless, $shunt$ fits into the framework, as its base function

$$\begin{array}{llll}
\mathfrak{shunt} : \forall x \,.\, (\mathsf{L}\, x \to \mathit{Stack}) \to (\mathsf{L}\,(\mathfrak{Stack}\, x) & & \to \mathit{Stack}) \\
\mathfrak{shunt} \quad \mathit{shunt} & (\mathfrak{Empty}, & ns) & = ns \\
\mathfrak{shunt} \quad \mathit{shunt} & (\mathfrak{Push}\,(m, ms), & ns) & = \mathit{shunt}\,(ms, \mathit{Push}\,(m, ns))
\end{array}$$

has the required naturality property. The revised definition of $shunt$

$$\begin{array}{ll}
\mathit{shunt} : \mathsf{L}\,(\mu\mathfrak{Stack}) & \to \mathit{Stack} \\
\mathit{shunt} \quad (\mathit{In}\, ms, ns) & = \mathfrak{shunt}\, \mathit{shunt}\,(ms, ns)
\end{array}$$

matches exactly the scheme for adjoint initial fixed-point equations. □

Exercise 36. Is the following tail-recursive variant of $total$ (see Example 1)

$$\begin{array}{ll}
\mathit{totalPlus} : (\mathit{Bush}, \mathit{Nat}) & \to \mathit{Nat} \\
\mathit{totalPlus} \quad (\mathit{Leaf}\, n, s) & = n + s \\
\mathit{totalPlus} \quad (\mathit{Fork}\,(l, r), s) & = \mathit{totalPlus}\,(l, \mathit{totalPlus}\,(r, s))
\end{array}$$

an adjoint fold? □

Lists are parametric in Haskell. Can we adopt the above reasoning to parametric types and polymorphic functions?

Example 15. The type of lists is given as the initial algebra of a higher-order base functor of kind $(\star \to \star) \to (\star \to \star)$.

$$\begin{array}{l}
\mathbf{type}\,\mathsf{List} = \mu\mathfrak{List} \\
\mathbf{data}\,\mathfrak{List}\,\mathit{list}\;a = \mathfrak{Nil} \mid \mathfrak{Cons}\,(a, \mathit{list}\;a) \\
\mathbf{instance}\,(\mathit{Functor}\;\mathit{list}) \Rightarrow \mathit{Functor}\,(\mathfrak{List}\,\mathit{list})\,\mathbf{where} \\
\quad \mathit{fmap}\,f\,\mathfrak{Nil} \qquad\qquad\quad = \mathfrak{Nil} \\
\quad \mathit{fmap}\,f\,(\mathfrak{Cons}\,(a, \mathit{as})) = \mathfrak{Cons}\,(f\;a, \mathit{fmap}\,f\;\mathit{as})
\end{array}$$

(Again, we do not need the functor's action on arrows, which maps natural transformations to natural transformations.) Lists generalise stacks, sequences of natural numbers, to an arbitrary element type. Likewise, the function

$$\begin{array}{llll}
\mathit{append} : \forall a\,.\,(\mu\mathfrak{List}\,a, & \mathsf{List}\,a) & \to & \mathsf{List}\,a \\
\mathit{append} \quad (\mathit{In}\,\mathfrak{Nil}, & \mathit{bs}) & = & \mathit{bs} \\
\mathit{append} \quad (\mathit{In}\,(\mathfrak{Cons}\,(a, \mathit{as})), & \mathit{bs}) & = & \mathit{In}\,(\mathfrak{Cons}\,(a, \mathit{append}\,(\mathit{as}, \mathit{bs})))
\end{array}$$

generalises *cat* (Example 11) to sequences of an arbitrary element type. □

If we lift products pointwise to functors, $(\mathsf{F} \mathbin{\dot{\times}} \mathsf{G})\,A = \mathsf{F}\,A \times \mathsf{G}\,A$, we can view *append* as a natural transformation of type

$$\mathit{append} : \mathsf{List} \mathbin{\dot{\times}} \mathsf{List} \mathbin{\dot{\to}} \mathsf{List} \;\; .$$

All that is left to do is to find the right adjoint of the lifted product $- \mathbin{\dot{\times}} \mathsf{H}$. One could be led to think that $\mathsf{F} \mathbin{\dot{\times}} \mathsf{H} \mathbin{\dot{\to}} \mathsf{G} \cong \mathsf{F} \mathbin{\dot{\to}} (\mathsf{H} \mathbin{\dot{\to}} \mathsf{G})$, but this does not work as $\mathsf{H} \mathbin{\dot{\to}} \mathsf{G}$ is not a functor in any sensible way (recall that $\mathsf{H} \mathbin{\dot{\to}} \mathsf{G}$ is the set of natural transformations from H to G). Also, lifting exponentials pointwise $\mathsf{G}^{\mathsf{H}}\,A = (\mathsf{G}\,A)^{\mathsf{H}\,A}$ does not work, because again the data does not define a functor as the exponential is contravariant in its first argument. To make progress, let us assume that the functor category is $\mathbf{Set}^{\mathscr{C}}$ so that $\mathsf{G}^{\mathsf{H}} : \mathscr{C} \to \mathbf{Set}$. (The category $\mathbf{Set}^{\mathscr{C}^{\mathsf{op}}}$ of contravariant, set-valued functors and natural transformations is known as the category of *pre-sheaves*.) We reason as follows:

$$\begin{array}{ll}
 & \mathsf{G}^{\mathsf{H}}\,A \\
\cong & \{\text{ Yoneda Lemma (85) }\} \\
 & \mathscr{C}(A, -) \mathbin{\dot{\to}} \mathsf{G}^{\mathsf{H}} \\
\cong & \{\text{ requirement: } - \mathbin{\dot{\times}} \mathsf{H} \dashv (-)^{\mathsf{H}} \text{ }\} \\
 & \mathscr{C}(A, -) \mathbin{\dot{\times}} \mathsf{H} \mathbin{\dot{\to}} \mathsf{G} \;\; .
\end{array}$$

The derivation suggests that the exponential of the functors H and G is given by $\mathscr{C}(A, -) \mathbin{\dot{\times}} \mathsf{H} \mathbin{\dot{\to}} \mathsf{G}$. However, the calculation does not prove that the functor thus defined is actually right adjoint to $- \mathbin{\dot{\times}} \mathsf{H}$, as its existence is assumed in the second step. We leave the proof as a (laborious) exercise to the reader—a more general result abstracting away from **Set** can be found in [11].

Exercise 37. Show that $- \mathbin{\dot{\times}} \mathsf{H}$ is left adjoint to $(-)^{\mathsf{H}}$.

1. Show that $\mathsf{G}^{\mathsf{H}}\,A = \mathscr{C}(A,-) \mathbin{\dot\times} \mathsf{H} \mathbin{\dot\to} \mathsf{G}$ is functorial in A. (The functor G^{H} takes an object to a set of natural transformations and an arrow to a function that in turn takes a natural transformation to a natural transformation.)
2. The adjuncts of $- \mathbin{\dot\times} \mathsf{H} \dashv (-)^{\mathsf{H}}$ are defined

$$\begin{aligned}\lfloor\sigma\rfloor &= \lambda\,A\,.\,\lambda\,s\,.\,\lambda\,X\,.\,\lambda\,(k,t)\,.\,\sigma\,X\,(\mathsf{F}\,k\,s,t)\ ,\\ \lceil\tau\rceil &= \lambda\,A\,.\,\lambda\,(s,t)\,.\,\tau\,A\,s\,A\,(id,t)\ .\end{aligned}$$

Prove that they are natural in F and G and mutually inverse. □

⋆ *Exercise 38.* Can you make sense of the functors $\mathsf{Id}^{\mathsf{Id}}$ and $\mathsf{Id}^{\mathsf{Sq}}$? □

Definition 16. The definition of exponentials goes beyond Haskell 2010, as it requires rank-2 types (the data constructor *Exp* has a rank-2 type).

$$\begin{aligned}&\mathbf{newtype}\,\mathsf{Exp}\,h\,g\,a = \mathit{Exp}\,\{\,\mathit{exp}^{\circ} : \forall x\,.\,(a \to x, h\,x) \to g\,x\,\}\\ &\mathbf{instance}\,\mathit{Functor}\,(\mathsf{Exp}\,h\,g)\,\mathbf{where}\\ &\quad \mathit{fmap}\,f\,(\mathit{Exp}\,h) = \mathit{Exp}\,(\lambda(k,t) \to h\,(k\cdot f,t))\end{aligned}$$

Morally, h and g are functors, as well. However, their mapping functions are not needed to define the $\mathsf{Exp}\,h\,g$ instance of *Functor*. The adjuncts are defined

$$\begin{aligned}\lfloor-\rfloor_{\mathsf{Exp}} &: (\mathit{Functor}\,f) \Rightarrow (\forall x\,.\,(f\,x, h\,x) \to g\,x) \to (\forall x\,.\,f\,x \to \mathsf{Exp}\,h\,g\,x)\\ \lfloor\sigma\rfloor_{\mathsf{Exp}} &= \lambda s \to \mathit{Exp}\,(\lambda(k,t) \to \sigma\,(\mathit{fmap}\,k\,s,t))\\ \lceil-\rceil_{\mathsf{Exp}} &: (\forall x\,.\,f\,x \to \mathsf{Exp}\,h\,g\,x) \to (\forall x\,.\,(f\,x,h\,x) \to g\,x)\\ \lceil\tau\rceil_{\mathsf{Exp}} &= \lambda(s,t) \to \mathit{exp}^{\circ}\,(\tau\,s)\,(id,t)\ .\end{aligned}$$

The type variables f, g and h are implicitly universally quantified. Again, most of the functor instances are not needed. □

Example 17. Continuing Example 15, we may conclude that the defining equation of *append* has a unique solution. Its transpose of type $\mathsf{List} \mathbin{\dot\to} \mathsf{List}^{\mathsf{List}}$ is interesting as it combines *append* with *fmap*:

$$\begin{aligned}&\mathit{append}' : \forall a\,.\,\mathsf{List}\,a \to \forall x\,.\,(a \to x) \to (\mathsf{List}\,x \to \mathsf{List}\,x)\\ &\mathit{append}'\ \ \mathit{as}\ = \ \lambda f \ \to \lambda \mathit{bs} \ \to \mathit{append}\,(\mathit{fmap}\,f\,\mathit{as},\mathit{bs})\ .\end{aligned}$$

For clarity, we have inlined the definition of $\mathsf{Exp}\,\mathsf{List}\,\mathsf{List}$. □

3.3.2 Mutual Value Recursion.

The functions *left* and *right* introduced in Example 12 are defined by mutual recursion. The program is similar to Example 5, which defines *varsExpr* and *varsStat*, with the notable difference that only one datatype is involved, rather than a pair of mutually recursive datatypes. Nonetheless, the correspondence suggests to view *left* and *right* as a single arrow in a product category.

$$\mathit{trees} : \langle 1, 1\rangle \to \Delta(\nu\mathfrak{Tree})$$

The arrow *trees* is an adjoint unfold since the diagonal functor $\Delta : \mathscr{C} \to \mathscr{C}\times\mathscr{C}$ has a left adjoint, the coproduct (Sections 2.3.2 and 2.5.1). Using a similar reasoning as in Section 3.1.3, we can unfold the adjoint final fixed-point equation specialised to the diagonal functor:

$$\Delta out \cdot x = \Psi\, x \quad\iff\quad out \cdot x_1 = \Psi_1\,\langle x_1,\, x_2\rangle \text{ and } out \cdot x_2 = \Psi_2\,\langle x_1,\, x_2\rangle \ ,$$

where $x_1 = \mathsf{Outl}\, x$, $x_2 = \mathsf{Outr}\, x$, $\Psi_1 = \mathsf{Outl}\circ\Psi$ and, $\Psi_2 = \mathsf{Outr}\circ\Psi$. The resulting equations are similar to those of Section 3.1.3, except that now the destructor *out* is the same in both equations.

Example 18. Continuing Example 12, the base functions of *left* and *right* are given by

$$\begin{aligned}
&\mathfrak{left} : \forall x\ .\ ((() \to x, () \to x) \to (() \to \mathfrak{Tree}\, x)\\
&\mathfrak{left}\quad (\mathit{left},\ \ \mathit{right})\quad () \ = \mathfrak{Branch}\,(\mathit{left}\,(), 0, \mathit{right}\,())\\
&\mathfrak{right} : \forall x\ .\ ((() \to x, () \to x) \to (() \to \mathfrak{Tree}\, x)\\
&\mathfrak{right}\quad (\mathit{left},\ \ \mathit{right})\quad () \ = \mathfrak{Branch}\,(\mathit{left}\,(), 1, \mathit{right}\,())\ .
\end{aligned}$$

The recursion equations

$$\begin{aligned}
&\mathit{left} : () \to \nu\mathfrak{Tree}\\
&\mathit{left}\ \ () \ = \ \mathit{Out}^\circ\,(\mathfrak{left}\,(\mathit{left}, \mathit{right})\,())\\
&\mathit{right} : () \to \nu\mathfrak{Tree}\\
&\mathit{right}\ \ () \ = \ \mathit{Out}^\circ\,(\mathfrak{right}\,(\mathit{left}, \mathit{right})\,())
\end{aligned}$$

exactly fit the pattern above (if we move Out° to the left-hand side). Hence, both functions are uniquely defined. Their transpose, $\lceil\langle \mathit{left},\ \mathit{right}\rangle\rceil$, combines the two functions into a single one using a coproduct.

$$\begin{aligned}
&\mathit{trees} : \mathsf{Either}\,()\,() \to \nu\mathfrak{Tree}\\
&\mathit{trees}\ \ (\mathit{Left}\ \ ())\ = \ \mathit{Out}^\circ\,(\mathfrak{Branch}\,(\mathit{trees}\,(\mathit{Left}\,()), 0, \mathit{trees}\,(\mathit{Right}\,())))\\
&\mathit{trees}\ \ (\mathit{Right}\,())\ = \ \mathit{Out}^\circ\,(\mathfrak{Branch}\,(\mathit{trees}\,(\mathit{Left}\,()), 1, \mathit{trees}\,(\mathit{Right}\,())))
\end{aligned}$$

The predefined datatype Either given by $\mathbf{data}\,\mathsf{Either}\, a\, b \ = \ \mathit{Left}\ a \ \mid\ \mathit{Right}\ b$ is Haskell's coproduct. □

Let us turn to the dual case. To handle folds defined by mutual recursion, we need the right adjoint of the diagonal functor, which is the *product* (Sections 2.3.1 and 2.5.1). Specialising the adjoint initial fixed-point equation yields

$$\langle x_1,\, x_2\rangle \cdot \Delta in = \Psi\,\langle x_1,\, x_2\rangle \quad\iff\quad x_1 \cdot in = \Psi_1\,\langle x_1,\, x_2\rangle \text{ and } x_2 \cdot in = \Psi_2\,\langle x_1,\, x_2\rangle \ .$$

Example 19. We can use mutual value recursion to fit the definition of the binary increment (Example 4) into the framework. The definition of *succ* has the form of a *paramorphism* [29], as the argument that drives the recursion is not exclusively used in the recursive call. The idea is to 'guard' the other occurrence by the identity function and to pretend that both functions are defined by mutual recursion.

$$\begin{array}{llll} succ : \mu\mathfrak{Nat} & \to Nat & id : \mu\mathfrak{Nat} & \to Nat \\ succ\ \ (In\,(\mathfrak{N})) & = In\,(\mathfrak{I}\,(In\,\mathfrak{N})) & id\ \ (In\,(\mathfrak{N})) & = In\,(\mathfrak{N}) \\ succ\ \ (In\,(\mathfrak{O}\,b)) & = In\,(\mathfrak{I}\,(id\,b)) & id\ \ (In\,(\mathfrak{O}\,b)) & = In\,(\mathfrak{O}\,(id\,b)) \\ succ\ \ (In\,(\mathfrak{I}\ b)) & = In\,(\mathfrak{O}\,(succ\,b)) & id\ \ (In\,(\mathfrak{I}\ b)) & = In\,(\mathfrak{I}\,(id\,b)) \end{array}$$

If we abstract away from the recursive calls, we find that the two base functions have indeed the required polymorphic types.

$$\begin{array}{llll} \mathfrak{succ} : \forall x\,.\,(x \to Nat, x \to Nat) \to (\mathfrak{Nat}\,x \to Nat) \\ \mathfrak{succ}\ \ (succ,\ \ id) & (\mathfrak{N}) & = In\,(\mathfrak{I}\,(In\,\mathfrak{N})) \\ \mathfrak{succ}\ \ (succ,\ \ id) & (\mathfrak{O}\,b) & = In\,(\mathfrak{I}\,(id\,b)) \\ \mathfrak{succ}\ \ (succ,\ \ id) & (\mathfrak{I}\ b) & = In\,(\mathfrak{O}\,(succ\,b)) \\ \mathfrak{id} : \forall x\,.\,(x \to Nat, x \to Nat) \to (\mathfrak{Nat}\,x \to Nat) \\ \mathfrak{id}\ \ (succ,\ \ id) & (\mathfrak{N}) & = In\,(\mathfrak{N}) \\ \mathfrak{id}\ \ (succ,\ \ id) & (\mathfrak{O}\,b) & = In\,(\mathfrak{O}\,(id\,b)) \\ \mathfrak{id}\ \ (succ,\ \ id) & (\mathfrak{I}\ b) & = In\,(\mathfrak{I}\,(id\,b)) \end{array}$$

The transposed fold has type $\mu\mathfrak{Nat} \to Nat \times Nat$ and corresponds to the usual encoding of paramorphisms as folds (using tupling). The trick does not work for the 'base function' $\mathfrak{bogus}$ as the resulting function still lacks naturality. □

Exercise 39. Show that the factorial function

$$\begin{array}{lll} \textbf{data}\,Peano = Z \mid S\,Peano \\ fac : Peano \to Peano \\ fac\ \ (Z) & = 1 \\ fac\ \ (S\,n) & = S\,n * fac\,n \end{array}$$

is an adjoint fold. □

Exercise 40. Can you also fit the Fibonacci function

$$\begin{array}{ll} fib : Peano & \to Peano \\ fib\ \ (Z) & = Z \\ fib\ \ (S\,Z) & = S\,Z \\ fib\ \ (S\,(S\,n)) & = fib\,n + fib\,(S\,n) \end{array}$$

into the framework of adjoint folds? *Hint:* introduce a second function $fib'\,n = fib\,(S\,n)$ and transform the nested recursion above into mutual recursion. □

3.3.3 Single Value Recursion. We have discussed mutually recursive functions over mutually recursive datatypes and mutually recursive functions over datatypes defined by single recursion. But what about a single function that recurses over a datatype defined by mutual recursion?

Example 20. The following datatypes (see also Example 5) model the abstract syntax of a simple functional programming language (as usual, this is a very stripped-down example).

$$\begin{array}{l}
\mathbf{data}\, \mathit{Expr} = \mathit{Var}\ \mathit{Var} \mid \mathit{Let}\,(\mathit{Decl}, \mathit{Expr}) \\
\mathbf{data}\, \mathit{Decl}\ = \mathit{Def}\,(\mathit{Var}, \mathit{Expr}) \mid \mathit{And}\,(\mathit{Decl}, \mathit{Decl})
\end{array}$$

$$\begin{array}{lll}
\mathit{bound} : \mathit{Decl} & \to & \mathit{Vars} \\
\mathit{bound}\ \ (\mathit{Def}\,(x, e)) & = & \{x\} \\
\mathit{bound}\ \ (\mathit{And}\,(d_1, d_2)) & = & \mathit{bound}\ d_1 \cup \mathit{bound}\ d_2
\end{array}$$

The function *bound* determines the variables that are defined by a declaration. □

The function *bound* proceeds by structural recursion, but it is not a fold simply because *Decl* is not an initial algebra: $\mathit{Decl} = \mathsf{Outr}\,(\mu\mathsf{Grammar})$. We can view *bound* as an adjoint fold provided Outr is part of an adjoint situation. It turns out that the projection functors Outl and Outr have both left and right adjoints if the base categories have initial and final objects. We show that Outl has a right adjoint—the other proofs proceed completely analogously.

$$\begin{array}{ll}
 & \mathscr{C}(\mathsf{Outl}\,A, B) \\
\cong & \{\ S \times 1 \cong S\ \} \\
 & \mathscr{C}(\mathsf{Outl}\,A, B) \times 1 \\
\cong & \{\ \text{assumption: } \mathscr{D} \text{ has a final object}\ \} \\
 & \mathscr{C}(\mathsf{Outl}\,A, B) \times \mathscr{D}(\mathsf{Outr}\,A, 1) \\
\cong & \{\ \text{definition of } \mathscr{C} \times \mathscr{D}\ \} \\
 & (\mathscr{C} \times \mathscr{D})(A, \langle B, 1\rangle)
\end{array}$$

The isomorphism is natural in A and B since each step is. The following diagram summarises the adjoint situations.

$$\begin{tikzcd}
\mathscr{C} \arrow[r, bend right, "\langle -, 1\rangle"'] \arrow[r, phantom, "\perp"] & \mathscr{C} \times \mathscr{D} \arrow[l, bend right, "\mathsf{Outl}"'] \arrow[r, bend right, "\mathsf{Outl}"'] \arrow[r, phantom, "\perp"] & \mathscr{C} \arrow[l, bend right, "\langle -, 0\rangle"']
\end{tikzcd}$$

Specialising the adjoint fixed-point equations to Outl yields

$$x \cdot \mathit{in}_1 = \Psi\, x, \qquad \text{and} \qquad \mathit{out}_1 \cdot x = \Psi\, x\ \ ,$$

where $\mathit{in}_1 = \mathsf{Outl}\,\mathit{in}$ and $\mathit{out}_1 = \mathsf{Outl}\,\mathit{out}$.

Exercise 41. Define the Haskell functions

$$\begin{array}{l}
\mathit{freeExpr} : \mathit{Expr} \to \mathit{Vars} \\
\mathit{freeDecl}\ : \mathit{Decl}\ \to \mathit{Vars}
\end{array}$$

that determine the free variables of expressions and declarations, respectively. Try to capture them as adjoint folds. (This is more involved than you might initially think, since *freeExpr* very likely also depends on the function *bound* from Example 20). □

An alternative approach to giving a semantics to *bound* is to make use of the fact that a fixed point of a functor over product categories can be expressed in terms of fixed points of unary functors [30]:

$$\mu\,\mathsf{F} \cong \langle \mu\,X\,.\,\mathsf{F}_1\,\langle X,\,\mu\,Y\,.\,\mathsf{F}_2\,\langle X,\ Y\rangle\rangle,\ \mu\,Y\,.\,\mathsf{F}_2\,\langle \mu\,X\,.\,\mathsf{F}_1\,\langle X,\ Y\rangle,\ Y\rangle\rangle\ ,$$

where $\mathsf{F}_1 = \mathsf{Outl}\circ\mathsf{F}$ and $\mathsf{F}_2 = \mathsf{Outr}\circ\mathsf{F}$.

3.3.4 Type Application. Folds of higher-order initial algebras are necessarily natural transformations as they live in a functor category. However, many Haskell functions that recurse over a parametric datatype are actually monomorphic.

Example 21. The type Sequ generalises the type of binary leaf trees, abstracting away from the type *Nat*.

$$\begin{array}{l}
\mathbf{type}\,\mathsf{Sequ} = \mu\mathfrak{Sequ}\\
\mathbf{data}\,\mathfrak{Sequ}\,sequ\ a = \mathfrak{Single}\ a \mid \mathfrak{Cat}\,(sequ\ a, sequ\ a)\\
\mathbf{instance}\,(Functor\ sequ) \Rightarrow Functor\,(\mathfrak{Sequ}\,sequ)\,\mathbf{where}\\
\quad fmap\,f\,(\mathfrak{Single}\ a) \ \ = \mathfrak{Single}\,(f\ a)\\
\quad fmap\,f\,(\mathfrak{Cat}\,(l, r)) = \mathfrak{Cat}\,(fmap\,f\,l, fmap\,f\,r)
\end{array}$$

The function *sums* defined

$$\begin{array}{lll}
sums : \mu\mathfrak{Sequ}\,Nat & \to & Nat\\
sums\ \ (In\,(\mathfrak{Single}\,n)) & = & n\\
sums\ \ (In\,(\mathfrak{Cat}\,(l, r))) & = & sums\,l + sums\,r
\end{array}$$

sums a non-empty sequence of natural numbers. It is the adaptation of *total* (Example 1) to the type of parametric leaf trees. □

The definition of *sums* looks suspiciously like a fold, but it is not as it does not have the right type. The corresponding function on random-access lists does not even resemble a fold.

Example 22. The function *suma* sums a random-access list.

$$\begin{array}{lll}
suma : \mu\mathfrak{Array}\,Nat & \to & Nat\\
suma\ \ (In\,(\mathfrak{Null})) & = & 0\\
suma\ \ (In\,(\mathfrak{Zero}\,s)) & = & suma\,(fmap\,plus\,s)\\
suma\ \ (In\,(\mathfrak{One}\,(a, s))) & = & a + suma\,(fmap\,plus\,s)\\
plus :\ \ (Nat, Nat) \to Nat & & \\
plus\ \ \ (a,\ \ \ b) & = & a + b
\end{array}$$

Note that the recursive calls of *suma* are not applied to a subterm of the input. In fact, they cannot as the parameter s has type $\mathsf{Array}\,(Nat, Nat)$, not $\mathsf{Array}\,Nat$. As an aside, this definition requires the functor instance for μ (Definition 9). □

Perhaps surprisingly, the definitions above fit into the framework of adjoint fixed-point equations. We already know that type application is a functor (Section 2.2.3). Using this higher-order functor we can assign *suma* the type $(-\,Nat)\,(\mu\mathfrak{Array}) \to Nat$. All that is left to do is to check whether $-\,X$ is part of an adjunction. It turns out that under some mild conditions (existence of copowers and powers) $-\,X$ has both a left and a right adjoint. We choose to derive the left adjoint.

$$\begin{aligned}
& \mathscr{C}(A, \mathsf{B}\,X) \\
\cong\;& \{\text{ Yoneda Lemma (85) }\} \\
& \forall Y : \mathscr{D}\,.\,\mathscr{D}(X, Y) \to \mathscr{C}(A, \mathsf{B}\,Y) \\
\cong\;& \{\text{ copower: } \mathscr{C}(\textstyle\sum I\,.\,X, Y) \cong I \to \mathscr{C}(X, Y)\ (52)\ \} \\
& \forall Y : \mathscr{D}\,.\,\mathscr{C}(\textstyle\sum \mathscr{D}(X, Y)\,.\,A, \mathsf{B}\,Y) \\
\cong\;& \{\text{ define } \mathsf{Lsh}_X\,A = \lambda\,Y : \mathscr{D}\,.\,\textstyle\sum \mathscr{D}(X, Y)\,.\,A\ \} \\
& \forall Y : \mathscr{D}\,.\,\mathscr{C}(\mathsf{Lsh}_X\,A\,Y, \mathsf{B}\,Y) \\
\cong\;& \{\text{ natural transformations }\} \\
& \mathsf{Lsh}_X\,A \mathbin{\dot{\to}} \mathsf{B}
\end{aligned}$$

Since each step is natural in A and B, the composite isomorphism is natural in A and B, as well. We call Lsh_X the *left shift* of X, for want of a better name. Dually, the right adjoint is $\mathsf{Rsh}_X\,B = \lambda\,Y : \mathscr{D}\,.\,\prod \mathscr{D}(Y, X)\,.\,B$, the *right shift* of X. The following diagram summarises the type information.

$$\begin{tikzcd}
\mathscr{C}^{\mathscr{D}} \arrow[r, bend right, "-\,X"'] \arrow[r, phantom, "\perp"] & \mathscr{C} \arrow[l, bend right, "\mathsf{Lsh}_X"'] \arrow[r, bend right, "\mathsf{Rsh}_X"'] \arrow[r, phantom, "\perp"] & \mathscr{C}^{\mathscr{D}} \arrow[l, bend right, "-\,X"']
\end{tikzcd}$$

Recall that in **Set**, the copower $\sum I\,.\,A$ is the cartesian product $I \times A$ and the power $\prod I\,.\,A$ is the set of functions $I \to A$. This correspondence suggests the Haskell implementation below. However, it is important to keep in mind that I is a set, not an object in the ambient category (like A).

Definition 23. The functors Lsh and Rsh can be defined as follows.

```
newtype Lsh_x a y = Lsh (x → y, a)
instance Functor (Lsh_x a) where
  fmap f (Lsh (k, a)) = Lsh (f · k, a)
newtype Rsh_x b y = Rsh {rsh° : (y → x) → b}
instance Functor (Rsh_x b) where
  fmap f (Rsh g)   = Rsh (λk → g (k · f))
```

The type $\mathsf{Lsh}_x\,a\,y$ can be seen as an abstract datatype: a is the internal state and $x \to y$ is the observer function—often, but not always, the types a and x are identical ($\mathsf{Lsh}_x\,x$ is a comonad, similar to the costate comonad). Dually, $\mathsf{Rsh}_x\,b\,y$ implements a continuation type—again, the types x and b are likely to be identical ($\mathsf{Rsh}_x\, x$ is the continuation monad, see Exercise 44). The adjuncts are defined

$$\begin{aligned}
\lfloor - \rfloor_{\mathsf{Lsh}} &: \forall x\, a\, b\, .\, (\forall y\, .\, \mathsf{Lsh}_x\, a\, y \to b\, y) \to (a \to b\, x) \\
\lfloor \alpha \rfloor_{\mathsf{Lsh}} &= \lambda s \to \alpha\, (\mathit{Lsh}\, (\mathit{id}, s)) \\
\lceil - \rceil_{\mathsf{Lsh}} &: \forall x\, a\, b\, .\, (\mathit{Functor}\, b) \Rightarrow (a \to b\, x) \to (\forall y\, .\, \mathsf{Lsh}_x\, a\, y \to b\, y) \\
\lceil g \rceil_{\mathsf{Lsh}} &= \lambda(\mathit{Lsh}\, (k, s)) \to \mathit{fmap}\, k\, (g\, s) \\
\lfloor - \rfloor_{\mathsf{Rsh}} &: \forall x\, a\, b\, .\, (\mathit{Functor}\, a) \Rightarrow (a\, x \to b) \to (\forall y\, .\, a\, y \to \mathsf{Rsh}_x\, b\, y) \\
\lfloor f \rfloor_{\mathsf{Rsh}} &= \lambda s \to \mathit{Rsh}\, (\lambda k \to f\, (\mathit{fmap}\, k\, s)) \\
\lceil - \rceil_{\mathsf{Rsh}} &: \forall x\, a\, b\, .\, (\forall y\, .\, a\, y \to \mathsf{Rsh}_x\, b\, y) \to (a\, x \to b) \\
\lceil \beta \rceil_{\mathsf{Rsh}} &= \lambda s \to \mathit{rsh}^{\circ}\, (\beta\, s)\, \mathit{id} \quad .
\end{aligned}$$

Note that the adjuncts are also natural in x, the parameter of the adjunctions. (Exercise 45 asks you to explore this fact.) □

As usual, let us specialise the adjoint equations (in **Set**).

$$\begin{aligned}
x \cdot (-\, X)\, \mathit{in} = \Psi\, x &\iff \forall s\, .\, x\, (\mathit{in}\, X\, s) = \Psi\, x\, s \\
(-\, X)\, \mathit{out} \cdot x = \Psi\, x &\iff \forall a\, .\, \mathit{out}\, X\, (x\, a) = \Psi\, x\, a
\end{aligned}$$

Since both type abstraction and type application are invisible in Haskell, the adjoint equations are, in fact, indistinguishable from standard fixed-point equations.

Example 24. Continuing Example 22, the base function of *suma* is

$$\begin{aligned}
&\mathfrak{suma} : \forall x\, .\, (\mathit{Functor}\, x) \Rightarrow \\
&\qquad (x\, \mathit{Nat} \to \mathit{Nat}) \to (\mathfrak{Array}\, x\, \mathit{Nat} \to \mathit{Nat}) \\
&\mathfrak{suma}\ \mathit{suma}\ (\mathfrak{Null}) = 0 \\
&\mathfrak{suma}\ \mathit{suma}\ (\mathfrak{Zero}\, s) = \mathit{suma}\, (\mathit{fmap}\, \mathit{plus}\, s) \\
&\mathfrak{suma}\ \mathit{suma}\ (\mathfrak{One}\, (a, s)) = a + \mathit{suma}\, (\mathit{fmap}\, \mathit{plus}\, s) \quad .
\end{aligned}$$

The definition requires the $\mathfrak{Array}\, x$ functor instance, which in turn induces the *Functor x* context. The transpose of *suma* is a fold that returns a higher-order function: $\mathit{suma}' : \mathsf{Array} \dot{\to} \mathsf{Rsh}_{\mathit{Nat}}\, \mathit{Nat}$.

$$\begin{aligned}
&\mathit{suma}' : \forall x\, .\, \mathsf{Array}\, x \to (x \to \mathit{Nat}) \to \mathit{Nat} \\
&\mathit{suma}'\ (\mathit{Null}) = \lambda k \to 0 \\
&\mathit{suma}'\ (\mathit{Zero}\, s) = \lambda k \to \mathit{suma}'\, s\, (\mathit{plus} \cdot (k \times k)) \\
&\mathit{suma}'\ (\mathit{One}\, (a, s)) = \lambda k \to k\, a + \mathit{suma}'\, s\, (\mathit{plus} \cdot (k \times k)) \quad .
\end{aligned}$$

Quite interestingly, the transformation turns a *generalised fold* in the sense of [31] into an *efficient generalised fold* in the sense of [32]. Both versions have a linear running time, but *suma′* avoids the repeated invocations of the mapping function (*fmap plus*). □

⋆ *Exercise 42.* The type of non-empty sequences (see Example 21)

$$\mathbf{data}\, \mathsf{Sequ}\, a = \mathit{Single}\, a \mid \mathit{Cat}\, (\mathsf{Sequ}\, a, \mathsf{Sequ}\, a)$$

can alternatively be seen as the free monad of the squaring functor $\mathsf{Sq}\, A = A \times A$, see Section 2.6.4. Express *sums* in terms of *eval*. □

⋆ *Exercise 43.* The type

$$\mathbf{data}\, \mathsf{Tree}\, a = \mathit{Branch}\, (\mathsf{Tree}\, a, a, \mathsf{Tree}\, a)$$

generalises the type of bifurcations (Example 2), abstracting away from the type of labels. Capture $\mathit{generate} : \mathit{Nat} \to \mathsf{Tree}\, \mathit{Nat}$ as an adjoint unfold. The type Tree can also be seen as the cofree comonad of the squaring functor $\mathsf{Sq}\, A = A \times A$ (see Exercise 34). Express *generate* in terms of *trace*. □

⋆ *Exercise 44.* The purpose of this exercise is to show that the functor $\mathsf{M} = \mathsf{Rsh}_X\, X$ is a monad for each choice of $X : \mathscr{C}$. There are at least two approaches: via the specification or via the implementation.

1. If we specialise the specification of Rsh_X, the adjunction $-X \dashv \mathsf{Rsh}_X$, to (setting $B := X$)

$$\lfloor - \rfloor : \mathscr{C}(\mathsf{A}\, X, X) \cong \mathscr{C}^{\mathscr{C}}(\mathsf{A}, \mathsf{M}) : \lceil - \rceil \enspace ,$$

   we obtain an isomorphism natural in the functor A. This suggests to define the unit and the multiplication of the monad by

$$\begin{aligned} \mathit{return} &= \lfloor \mathit{id} \rfloor \enspace , \\ \mathit{join} &= \lfloor e \cdot \mathsf{M}\, e \rfloor\, \mathbf{where}\, e = \lceil \mathit{id} \rceil \enspace . \end{aligned}$$

   Note that the arrow $e : \mathscr{C}(\mathsf{M}\, X, X)$ runs a computation. Show that *return* and *join* satisfy the monad laws (see Exercise 30).
2. If the monad is implemented by $\mathsf{M}\, A = \prod \mathscr{C}(A, X)\, .\, X$, then we can use the combinators of Section 2.5.7 to define

$$\begin{aligned} \mathit{return} &= \bigtriangleup\, k\, .\, k \enspace , \\ \mathit{join} &= \bigtriangleup\, k\, .\, \pi_{\pi_k} \enspace . \end{aligned}$$

   Show that *return* and *join* thus defined satisfy the monad laws. Try to relate the definitions to the implementation of the continuation monad from Haskell's standard libraries. □

⋆ *Exercise 45.* Like the curry adjunction, $-X \dashv \mathsf{Rsh}_X$ is an adjunction with a parameter. Apply Exercise 35 to show that there is a unique way to turn Rsh into a bifunctor, so that the bijection $\mathscr{C}(\mathsf{A}\, X, B) \cong \mathscr{C}^{\mathscr{D}}(\mathsf{A}, \mathsf{Rsh}_X\, B)$ is also natural in X:

$$\begin{tikzcd}
\mathscr{C}(\mathsf{A}\, \hat{X}, B) \arrow[r, "\lfloor - \rfloor"] \arrow[d, "\mathscr{C}(\mathsf{A}\, p, B)"'] & \mathscr{C}^{\mathscr{D}}(\mathsf{A}, \mathsf{Rsh}_{\hat{X}}\, B) \arrow[d, "\mathscr{C}^{\mathscr{D}}(\mathsf{A}, \mathsf{Rsh}_p\, B)"] \\
\mathscr{C}(\mathsf{A}\, \check{X}, B) \arrow[r, "\lfloor - \rfloor"'] & \mathscr{C}^{\mathscr{D}}(\mathsf{A}, \mathsf{Rsh}_{\check{X}}\, B) \enspace ,
\end{tikzcd}$$

where $p : \mathscr{D}(\hat{X}, \check{X})$. Explore. □

3.3.5 Type Composition. Continuing the theme of the last section, functions over parametric types, consider the following example.

Example 25. The function *join* defined

$$\begin{aligned}
&\mathit{join} : \forall a \,.\, \mu\mathfrak{Sequ}\,(\mathsf{Sequ}\,a) \to \mathsf{Sequ}\,a\\
&\mathit{join} \quad (\mathit{In}\,(\mathfrak{Single}\,s)) = s\\
&\mathit{join} \quad (\mathit{In}\,(\mathfrak{Cat}\,(l, r))) = \mathit{In}\,(\mathfrak{Cat}\,(\mathit{join}\,l, \mathit{join}\,r))
\end{aligned}$$

flattens a non-empty sequence of non-empty sequences. □

The definition has the structure of an ordinary fold, but again the type is not quite right: we need a natural transformation of type $\mu\mathfrak{Sequ} \dot{\to} \mathsf{G}$, but *join* has type $\mu\mathfrak{Sequ}\circ\mathsf{Sequ} \dot{\to} \mathsf{Sequ}$. Can we fit the definition into the framework of adjoint equations? The answer is an emphatic "Yes, we Kan!" Similar to the development of the previous section, the first step is to identify a left adjoint. We already know that pre-composition is a functor (Section 2.2.3). Using this higher-order functor we can assign *join* the type $(-\circ\mathsf{Sequ})\,(\mu\mathfrak{Sequ}) \dot{\to} \mathsf{Sequ}$. (We interpret $\mathsf{Sequ}\circ\mathsf{Sequ}$ as $(-\circ\mathsf{Sequ})\,\mathsf{Sequ}$ rather than $(\mathsf{Sequ}\circ-)\,\mathsf{Sequ}$ because the outer list, written $\mu\mathfrak{Sequ}$ for emphasis, drives the recursion.)

As a second step, we have to construct the right adjoint of the higher-order functor. It turns out that this is a well-studied problem in category theory. Similar to the situation of the previous section, under some conditions $-\circ\mathsf{J}$ has both a left and a right adjoint. For variety, we derive the latter.

$$\begin{aligned}
&\mathsf{F}\circ\mathsf{J} \dot{\to} \mathsf{G}\\
\cong\;&\{\text{ natural transformation as an end [8, p.223] }\}\\
&\forall Y : \mathscr{C} \,.\, \mathscr{E}(\mathsf{F}\,(\mathsf{J}\,Y), \mathsf{G}\,Y)\\
\cong\;&\{\text{ Yoneda Lemma (85) }\}\\
&\forall Y : \mathscr{C} \,.\, \forall X : \mathscr{D} \,.\, \mathscr{D}(X, \mathsf{J}\,Y) \to \mathscr{E}(\mathsf{F}\,X, \mathsf{G}\,Y)\\
\cong\;&\{\text{ power: } I \to \mathscr{C}(Y, B) \cong \mathscr{C}(Y, \textstyle\prod I \,.\, B) \text{ (52) }\}\\
&\forall Y : \mathscr{C} \,.\, \forall X : \mathscr{D} \,.\, \mathscr{E}(\mathsf{F}\,X, \textstyle\prod \mathscr{D}(X, \mathsf{J}\,Y) \,.\, \mathsf{G}\,Y)\\
\cong\;&\{\text{ interchange of quantifiers [8, p.231f] }\}\\
&\forall X : \mathscr{D} \,.\, \forall Y : \mathscr{C} \,.\, \mathscr{E}(\mathsf{F}\,X, \textstyle\prod \mathscr{D}(X, \mathsf{J}\,Y) \,.\, \mathsf{G}\,Y)\\
\cong\;&\{\text{ the hom-functor } \mathscr{E}(A, -) \text{ preserves ends [8, p.225] }\}\\
&\forall X : \mathscr{D} \,.\, \mathscr{E}(\mathsf{F}\,X, \forall Y : \mathscr{C} \,.\, \textstyle\prod \mathscr{D}(X, \mathsf{J}\,Y) \,.\, \mathsf{G}\,Y)\\
\cong\;&\{\text{ define } \mathsf{Ran}_{\mathsf{J}}\,\mathsf{G} = \lambda X : \mathscr{D} \,.\, \forall Y : \mathscr{C} \,.\, \textstyle\prod \mathscr{D}(X, \mathsf{J}\,Y) \,.\, \mathsf{G}\,Y \;\}\\
&\forall X : \mathscr{D} \,.\, \mathscr{E}(\mathsf{F}\,X, \mathsf{Ran}_{\mathsf{J}}\,\mathsf{G}\,X)\\
\cong\;&\{\text{ natural transformation as an end [8, p.223] }\}\\
&\mathsf{F} \dot{\to} \mathsf{Ran}_{\mathsf{J}}\,\mathsf{G}
\end{aligned}$$

Since each step is natural in F and G, the composite isomorphism is also natural in F and G. The functor $\mathsf{Ran}_{\mathsf{J}}\,\mathsf{G}$ is called the *right Kan extension* of G along J. (If we view $\mathsf{J} : \mathscr{C} \to \mathscr{D}$ as an inclusion functor, then $\mathsf{Ran}_{\mathsf{J}}\,\mathsf{G} : \mathscr{D} \to \mathscr{E}$ extends

$\mathsf{G} : \mathscr{C} \to \mathscr{E}$ to the whole of $\mathscr{D}$.) The universally quantified object in the definition of Ran_J is a so-called *end*, which corresponds to a polymorphic type in Haskell. An end is usually written with an integral sign; I prefer the universal quantifier, in particular, as it blends with the notation for natural transformations. And indeed, natural transformations are an example of an end: $\mathscr{D}^\mathscr{C}(\mathsf{F}, \mathsf{G}) = \forall X : \mathscr{C} \,.\, \mathscr{D}(\mathsf{F}\,X, \mathsf{G}\,X)$. We refer the interested reader to [8] for further details.

Dually, the left adjoint of $-\circ\mathsf{J}$ is called the *left Kan extension* and is defined $\mathsf{Lan}_\mathsf{J}\,\mathsf{F} = \lambda X : \mathscr{D} \,.\, \exists Y : \mathscr{C} \,.\, \sum \mathscr{D}(\mathsf{J}\,Y, X) \,.\, \mathsf{F}\,Y$. The existentially quantified object is a *coend*, which corresponds to an existential type in Haskell (hence the notation). The following diagrams summarise the type information.

$$\begin{tikzcd}
 & \mathscr{C} \arrow[dl, "\mathsf{F}"'] \arrow[d, "\mathsf{J}"] \\
\mathscr{E} & \mathscr{D} \arrow[l, shift right, "\mathsf{G}"'] \arrow[l, shift left, "\mathsf{Lan}_\mathsf{J}\,\mathsf{F}"]
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathscr{E}^\mathscr{D} \arrow[r, bend right, "-\circ\mathsf{J}"'] \arrow[r, phantom, "\perp"] & \mathscr{E}^\mathscr{C} \arrow[l, bend right, "\mathsf{Lan}_\mathsf{J}"'] \arrow[r, bend right, "\mathsf{Ran}_\mathsf{J}"'] \arrow[r, phantom, "\perp"] & \mathscr{E}^\mathscr{D} \arrow[l, bend right, "-\circ\mathsf{J}"']
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathscr{C} \arrow[d, "\mathsf{J}"'] \arrow[dr, "\mathsf{G}"] & \\
\mathscr{D} \arrow[r, shift left, "\mathsf{F}"] \arrow[r, shift right, "\mathsf{Ran}_\mathsf{J}\,\mathsf{G}"'] & \mathscr{E}
\end{tikzcd}$$

Definition 26. Like Exp, the definition of the right Kan extension requires rank-2 types (the data constructor *Ran* has a rank-2 type).

```
newtype Ranⱼ g x = Ran {ran° : ∀a . (x → j a) → g a}
instance Functor (Ranⱼ g) where
  fmap f (Ran h) = Ran (λk → h (k · f))
```

The type $\mathsf{Ran}_j\, g$ can be seen as a *generalised continuation type*—often, but not always, the type constructors j and g are identical ($\mathsf{Ran}_\mathsf{J}\,\mathsf{J}$ is known as the *codensity monad*, see Exercise 46). Morally, j and g are functors. However, their mapping functions are not needed to define the $\mathsf{Ran}_j\, g$ instance of *Functor*. Hence, we omit the $(\mathit{Functor}\, j, \mathit{Functor}\, g)$ context. The adjuncts are defined

```
⌊−⌋Ran : ∀j f g . (Functor f) ⇒ (∀x . f (j x) → g x) → (∀x . f x → Ranⱼ g x)
⌊α⌋Ran = λs → Ran (λk → α (fmap k s))
⌈−⌉Ran : ∀j f g . (∀x . f x → Ranⱼ g x) → (∀x . f (j x) → g x)
⌈β⌉Ran = λs → ran° (β s) id  .
```

Note that the adjuncts are also natural in j, the parameter of the adjunction. (Exercise 47 asks you to explore this fact.)

Turning to the definition of the left Kan extension we require another extension of the Haskell 2010 type system: existential types.

```
data Lanⱼ f x = ∀a . Lan (j a → x, f a)
instance Functor (Lanⱼ f) where
  fmap f (Lan (k, s)) = Lan (f · k, s)  .
```

The existential quantifier is written as a universal quantifier *in front of* the data constructor *Lan*. Ideally, Lan_j should be given by a **newtype** declaration, but **newtype** constructors must not have an existential context (in GHC). For similar reasons, we cannot use a destructor, that is, a selector function lan°. The type $\mathsf{Lan}_j\, f$ can be seen as a *generalised abstract data type*: $f\ a$ is the internal state and $j\ a \to x$ the observer function—again, the type constructors j and f are likely to be identical ($\mathsf{Lan}_{\mathsf{J}}\, \mathsf{J}$ is known as the *density comonad*). The adjuncts are given by

$$\begin{aligned}
&\lfloor - \rfloor_{\mathsf{Lan}} : \forall j\, f\, g\, .\, (\forall x\, .\, \mathsf{Lan}_j\, f\, x \to g\, x) \to (\forall x\, .\, f\, x \to g\, (j\, x)) \\
&\lfloor \alpha \rfloor_{\mathsf{Lan}} = \lambda s \to \alpha\, (Lan\, (id, s)) \\
&\lceil - \rceil_{\mathsf{Lan}} : \forall j\, f\, g\, .\, (Functor\ g) \Rightarrow (\forall x\, .\, f\, x \to g\, (j\, x)) \to (\forall x\, .\, \mathsf{Lan}_j\, f\, x \to g\, x) \\
&\lceil \beta \rceil_{\mathsf{Lan}} = \lambda (Lan\, (k, s)) \to fmap\ k\, (\beta\, s)\ .
\end{aligned}$$

The duality of the construction is somewhat obscured in Haskell. □

As usual, let us specialise the adjoint equations (in **Set**).

$$\begin{aligned}
x \cdot (-\circ \mathsf{J})\, in = \Psi\, x &\iff \forall A\, .\, \forall s\, .\, x\, A\, (in\, (\mathsf{J}\, A)\, s) = \Psi\, x\, A\, s \\
(-\circ \mathsf{J})\, out \cdot x = \Psi\, x &\iff \forall A\, .\, \forall a\, .\, out\, (\mathsf{J}\, A)\, (x\, A\, a) = \Psi\, x\, A\, a
\end{aligned}$$

The usual caveat applies when reading the equations as Haskell definitions: as type application is invisible, the derived equation is indistinguishable from the original one.

Example 27. Continuing Example 25, the base function of *join* is straightforward, except perhaps for the types.

$$\begin{aligned}
&\mathfrak{join} : \forall x\, .\, (\forall a\, .\, x\, (\mathsf{Sequ}\, a) \to \mathsf{Sequ}\, a) \to \\
&\qquad\qquad (\forall a\, .\, \mathfrak{Sequ}\, x\, (\mathsf{Sequ}\, a) \to \mathsf{Sequ}\, a) \\
&\mathfrak{join}\quad join\quad (\mathfrak{Single}\, s) \quad = s \\
&\mathfrak{join}\quad join\quad (\mathfrak{Cat}\, (l, r)) \quad = In\, (\mathfrak{Cat}\, (join\, l, join\, r))
\end{aligned}$$

The base function $\mathfrak{join}$ is a second-order natural transformation. The transpose of *join* is quite revealing. First of all, its type is

$$join' : \mathsf{Sequ} \dot{\to} \mathsf{Ran}_{\mathsf{Sequ}}\, \mathsf{Sequ} \cong \forall a\, .\, \mathsf{Sequ}\, a \to \forall b\, .\, (a \to \mathsf{Sequ}\, b) \to \mathsf{Sequ}\, b\ .$$

The type suggests that $join'$ is the bind of the monad Sequ (Exercise 42) and this is indeed the case!

$$\begin{aligned}
&join' : \forall a\, b\, .\, \mu \mathfrak{Sequ}\, a \to (a \to \mathsf{Sequ}\, b) \to \mathsf{Sequ}\, b \\
&join'\quad as \quad = \lambda k \quad \to join\, (fmap\ k\ as)
\end{aligned}$$

For clarity, we have inlined $\mathsf{Ran}_{\mathsf{Sequ}}\, \mathsf{Sequ}$. □

Kan extensions generalise the constructions of the previous section: If the category $\mathscr{C}$ is non-empty ($\mathscr{C} \neq \mathbf{0}$), then we have $\mathsf{Lsh}_A\, B \cong \mathsf{Lan}_{(\mathsf{K}\, A)}\, (\mathsf{K}\, B)$ and $\mathsf{Rsh}_A\, B \cong \mathsf{Ran}_{(\mathsf{K}\, A)}\, (\mathsf{K}\, B)$, where K is the constant functor. Here is the proof for the right adjoint:

$$\begin{aligned}
& \mathsf{F}\,A \to B \\
\cong\; & \{\ \text{arrows as natural transformations: } A \to B \cong \mathsf{K}\,A \dot{\to} \mathsf{K}\,B \text{ if } \mathscr{C} \neq \mathbf{0}\ \} \\
& \mathsf{K}\,(\mathsf{F}\,A) \dot{\to} \mathsf{K}\,B \\
=\; & \{\ \mathsf{K}\,(\mathsf{F}\,A) = \mathsf{F}\circ\mathsf{K}\,A\ \} \\
& \mathsf{F}\circ\mathsf{K}\,A \dot{\to} \mathsf{K}\,B \\
\cong\; & \{\ (-\circ\mathsf{J}) \dashv \mathsf{Ran}_{\mathsf{J}}\ \} \\
& \mathsf{F} \dot{\to} \mathsf{Ran}_{\mathsf{K}\,A}\,(\mathsf{K}\,B)\ .
\end{aligned}$$

Since adjoints are unique up to isomorphism (Section 2.5.8), we conclude that $\mathsf{Ran}_{\mathsf{K}\,A}\circ\mathsf{K} \cong \mathsf{Rsh}_A$.

⋆ *Exercise 46.* Kan extensions generalise shifts. Likewise, the codensity monad generalises the continuation monad. This exercise is due to Mac Lane [8, Exercise X.7.3]; it is solved in the forthcoming paper [33].

1. Generalise the argument of Exercise 44 to show that $\mathsf{M} = \mathsf{Ran}_{\mathsf{J}}\,\mathsf{J}$ is a monad for each choice of $\mathsf{J} : \mathscr{C} \to \mathscr{D}$. The functor M is called the *codensity monad of* J. (Specifically, if we specialise the adjunction $(-\circ\mathsf{J}) \dashv \mathsf{Ran}_{\mathsf{J}}$ to $(\mathsf{G} := \mathsf{J})$

$$\lfloor - \rfloor : \mathscr{D}^{\mathscr{C}}(\mathsf{F}\circ\mathsf{J}, \mathsf{J}) \cong \mathscr{D}^{\mathscr{D}}(\mathsf{F}, \mathsf{M}) : \lceil - \rceil\ ,$$

we obtain a bijection natural in the functor $\mathsf{F} : \mathscr{D} \to \mathscr{D}$. Unit and multiplication of the codensity monad are given by

$$\begin{aligned}
\mathit{return} &= \lfloor \mathit{id} \rfloor\ , \\
\mathit{join} &= \lfloor e \cdot \mathsf{M}\circ e \rfloor\ \mathbf{where}\ e = \lceil \mathit{id} \rceil\ ,
\end{aligned}$$

where the natural transformation $e : \mathscr{D}^{\mathscr{C}}(\mathsf{M}\circ\mathsf{J}, \mathsf{J})$ runs a computation.)
2. Show that if $\mathsf{R} : \mathscr{C} \to \mathscr{D}$ has a left adjoint, $\mathsf{L} \dashv \mathsf{R}$, then the codensity monad of R is the monad induced by the adjunction, $(\mathsf{R}\circ\mathsf{L}, \eta, \mathsf{R}\circ\epsilon\circ\mathsf{L})$. □

⋆ *Exercise 47.* The adjunction $(-\circ\mathsf{J}) \dashv \mathsf{Ran}_{\mathsf{J}}$ is yet another example of an adjunction with a parameter. Apply Exercise 35 to show that there is a unique way to turn Ran into a higher-order bifunctor, so that the bijection $\mathscr{E}^{\mathscr{C}}(\mathsf{F}\circ\mathsf{J}, \mathsf{G}) \cong \mathscr{E}^{\mathscr{D}}(\mathsf{F}, \mathsf{Ran}_{\mathsf{J}}\,\mathsf{G})$ is also natural in J:

$$\begin{tikzcd}
\mathscr{E}^{\mathscr{C}}(\mathsf{F}\circ\hat{\mathsf{J}}, \mathsf{G}) \arrow[r, "\lfloor - \rfloor"] \arrow[d, "\mathscr{E}^{\mathscr{C}}(\mathsf{F}\circ\alpha, \mathsf{G})"'] & \mathscr{E}^{\mathscr{D}}(\mathsf{F}, \mathsf{Ran}_{\hat{\mathsf{J}}}\,\mathsf{G}) \arrow[d, "\mathscr{E}^{\mathscr{D}}(\mathsf{F}, \mathsf{Ran}_{\alpha}\,\mathsf{G})"] \\
\mathscr{E}^{\mathscr{C}}(\mathsf{F}\circ\check{\mathsf{J}}, \mathsf{G}) \arrow[r, "\lfloor - \rfloor"'] & \mathscr{E}^{\mathscr{D}}(\mathsf{F}, \mathsf{Ran}_{\check{\mathsf{J}}}\,\mathsf{G})\ ,
\end{tikzcd}$$

where $\alpha : \mathscr{D}^{\mathscr{C}}(\hat{\mathsf{J}}, \check{\mathsf{J}})$. Remember that $-\circ=$ is a bifunctor (Section 2.2.3). Consequently, $-\circ\alpha$ is a higher-order natural transformation. Explore. □

3.3.6 Swapping Arguments. So far we have considered inductive and coinductive types only in isolation. The following example introduces two functions that combine an inductive with a coinductive type.

Example 28. Bifurcations and functions over the binary numbers are in one-to-one correspondence. The functions *tabulate* and *lookup* witness the isomorphism.

$$\begin{aligned}
&\mathit{tabulate} : (\mathit{Nat} \to \mathit{Nat}) \to \mathit{Tree}\\
&\mathit{tabulate}\ f = \mathit{Branch}\,(\mathit{tabulate}\,(f \cdot O), f\,N, \mathit{tabulate}\,(f \cdot I))\\
&\mathit{lookup} : \mathit{Tree} \to (\mathit{Nat} \to \mathit{Nat})\\
&\mathit{lookup}\ (\mathit{Branch}\,(l, v, r))\,(N) = v\\
&\mathit{lookup}\ (\mathit{Branch}\,(l, v, r))\,(O\ b) = \mathit{lookup}\ l\ b\\
&\mathit{lookup}\ (\mathit{Branch}\,(l, v, r))\,(I\ b) = \mathit{lookup}\ r\ b
\end{aligned}$$

The first isomorphism tabulates a given function, producing an infinite tree of its values. Its inverse looks up a binary number at a given position. □

Tabulation is a standard unfold, but what about *lookup*? Its type involves exponentials: $\mathit{lookup} : \mathscr{C}(\mathit{Tree}, \mathit{Nat}^{\mu\mathfrak{Nat}})$. However, the curry adjunction $- \times X \dashv (-)^X$ is not applicable here, as the right adjoint fixes the source object. We need its counterpart, the functor $X^{(-)} : \mathscr{C}^{\mathsf{op}} \to \mathscr{C}$, which fixes the target object (Exercise 22). Since this functor is contravariant, the type of *lookup* is actually $\mathscr{C}^{\mathsf{op}}((\mathit{Nat}^{(-)})^{\mathsf{op}}\,(\mu\mathfrak{Nat}), \mathit{Tree})$, which suggests that the arrow is an adjoint fold!

If we specialise the adjoint equation to $\mathscr{C} = \mathbf{Set}$ and $\mathsf{L} = X^{(-)}$, we obtain

$$x \cdot \mathsf{L}\,\mathit{in} = \Psi\,x \iff \forall s\,.\,\forall a\,.\,x\,a\,(\mathit{in}\,s) = \Psi\,x\,a\,s\ .$$

So x is simply a curried function that recurses over the *second* argument.

We have not mentioned unfolds so far. The reason is perhaps surprising. In this particular case, an adjoint unfold is the same as an adjoint fold! Consider the type of an adjoint unfold: $\mathscr{C}(A, \mathsf{R}\,(\nu\mathsf{F}))$. Since $\mathsf{R} = X^{(-)}$ is contravariant, the final coalgebra in $\mathscr{C}^{\mathsf{op}}$ is the initial algebra in $\mathscr{C}$. Since furthermore $X^{(-)}$ is self-adjoint, we obtain the type of an adjoint fold: $\mathscr{C}(A, \mathsf{L}\,(\mu\mathsf{F})) = \mathscr{C}^{\mathsf{op}}(\mathsf{L}\,(\mu\mathsf{F}), A)$.

Table 4 summarises the adjunctions considered in this section.

Exercise 48. Exercise 32 asked you to explore the adjunction $\mathsf{Free} \dashv \mathsf{U}$, where $\mathsf{U} : \mathbf{Mon} \to \mathbf{Set}$ is the functor that forgets about the additional structure of **Mon**. Explore adjoint folds of type $\mathsf{Free}\,(\mu\mathsf{F}) \to A$. □

3.4 Program Verification

In this section we develop the calculational properties of adjoint folds—the reader is invited to dualise the results to adjoint unfolds. Sections 3.4.1 is concerned with laws that support structured point-free reasoning. Section 3.4.2 then shifts the focus from point-free to pointwise style. It presents a unifying proof method that can be readily adapted to support effective pointwise calculations.

Table 4. Adjunctions and types of recursion.

adjunction	initial fixed-point equation	final fixed-point equation
$\mathsf{L} \dashv \mathsf{R}$	$x \cdot \mathsf{L}\, in = \Psi\, x$ $x' \cdot in = \lfloor \Psi \lceil x' \rceil \rfloor$	$\mathsf{R}\, out \cdot x = \Psi\, x$ $out \cdot x' = \lceil \Psi \lfloor x' \rfloor \rceil$
$\mathsf{Id} \dashv \mathsf{Id}$	standard fold standard fold	standard unfold standard unfold
$- \times X \dashv (-)^X$	parametrised fold $x \cdot (in \times X) = \Psi\, x$ fold to an exponential	curried unfold $out^X \cdot x = \Psi\, x$ unfold from a product
$(X^{(-)})^{\mathsf{op}} \dashv X^{(-)}$	swapped curried fold $X^{in} \cdot x = \Psi\, x$ fold to an exponential	
$(+) \dashv \Delta$	recursion from a coproduct of mutually recursive types mutual value recursion on mutually recursive types	mutual value recursion $out \cdot x_1 = \Psi_1\, \langle x_1, x_2 \rangle$ $out \cdot x_2 = \Psi_2\, \langle x_1, x_2 \rangle$ single recursion from a coproduct domain
$\Delta \dashv (\times)$	mutual value recursion $x_1 \cdot in = \Psi_1\, \langle x_1, x_2 \rangle$ $x_2 \cdot in = \Psi_2\, \langle x_1, x_2 \rangle$ single recursion to a product domain	recursion to a product of mutually recursive types mutual value recursion on mutually recursive types
$\langle -, 0 \rangle \dashv \mathsf{Outl}$	—	single value recursion on mutually recursive types $out_1 \cdot x = \Psi\, x$ 'mutual' value recursion on mutually recursive types
$\mathsf{Outl} \dashv \langle -, 1 \rangle$	single value recursion on mutually recursive types $x \cdot in_1 = \Psi\, x$ 'mutual' value recursion on mutually recursive types	—
$\mathsf{Lsh}_X \dashv (-\, X)$	—	monomorphic unfold $out\, X \cdot x = \Psi\, x$ unfold from a left shift
$(-\, X) \dashv \mathsf{Rsh}_X$	monomorphic fold $x \cdot in\, X = \Psi\, x$ fold to a right shift	—
$\mathsf{Lan}_{\mathsf{J}} \dashv (- \circ \mathsf{J})$	—	polymorphic unfold $out \circ \mathsf{J} \cdot x = \Psi\, x$ unfold from a left Kan extension
$(- \circ \mathsf{J}) \dashv \mathsf{Ran}_{\mathsf{J}}$	polymorphic fold $x \cdot in \circ \mathsf{J} = \Psi\, x$ fold to a right Kan extension	—

3.4.1 Uniqueness Property. Adjoint folds enjoy the usual plethora of properties. The fact that an adjoint initial fixed-point equation has a unique solution can be captured by the following equivalence, the *uniqueness property*.

$$x = (\!|\Psi|\!)_{\mathsf{L}} \iff x \cdot \mathsf{L}\, in = \Psi\, x \tag{98}$$

The uniqueness property has two simple consequences. First, substituting the left-hand side into the right-hand side gives the *computation law*.

$$(\!|\Psi|\!)_{\mathsf{L}} \cdot \mathsf{L}\, in = \Psi\, (\!|\Psi|\!)_{\mathsf{L}} \tag{99}$$

The law has a straightforward operational reading: an application of an adjoint fold is replaced by the body of the fold.

Second, instantiating x to id, we obtain the *reflection law*.

$$(\!|\Psi|\!)_{\mathsf{L}} = id \iff \Psi\, id = \mathsf{L}\, in \tag{100}$$

As an application of these identities, let us generalise the banana-split law [9]. In Section 2.6.6 we have stated the law in terms of standard folds. However, it can be readily ported to adjoint folds. First we introduce the counterpart of the product of two algebras (see also Exercise 49):

$$(\Phi \otimes \Psi)\, x = \Phi\, (outl \cdot x) \vartriangle \Psi\, (outr \cdot x) \enspace . \tag{101}$$

It is worth pointing out that the definition of $\otimes$ mentions neither the base functor F nor the adjoint functor L—in a sense, the base functions are hiding unnecessary detail.

Exercise 49. We have seen in Section 3.1.1 that base functions of type $\mathscr{C}(-, A) \dot{\to} \mathscr{C}(\mathsf{F}\,-, A)$ and F-algebras are in one-to-one correspondence. Show that $\otimes$ corresponds to the product of algebras (for $\mathsf{L} = \mathsf{Id}$). □

The generalised banana-split law then states

$$(\!|\Phi|\!)_{\mathsf{L}} \vartriangle (\!|\Psi|\!)_{\mathsf{L}} = (\!|\Phi \otimes \Psi|\!)_{\mathsf{L}} \enspace . \tag{102}$$

For the proof we appeal to the uniqueness property (98); the obligation is discharged as follows.

$$\begin{aligned}
& ((\!|\Phi|\!)_{\mathsf{L}} \vartriangle (\!|\Psi|\!)_{\mathsf{L}}) \cdot \mathsf{L}\, in \\
= \;& \{ \text{ fusion (12) } \} \\
& (\!|\Phi|\!)_{\mathsf{L}} \cdot \mathsf{L}\, in \vartriangle (\!|\Psi|\!)_{\mathsf{L}} \cdot \mathsf{L}\, in \\
= \;& \{ \text{ computation (99) } \} \\
& \Phi\, (\!|\Phi|\!)_{\mathsf{L}} \vartriangle \Psi\, (\!|\Psi|\!)_{\mathsf{L}} \\
= \;& \{ \text{ computation (9)–(10) } \} \\
& \Phi\, (outl \cdot ((\!|\Phi|\!)_{\mathsf{L}} \vartriangle (\!|\Psi|\!)_{\mathsf{L}})) \vartriangle \Psi\, (outr \cdot ((\!|\Phi|\!)_{\mathsf{L}} \vartriangle (\!|\Psi|\!)_{\mathsf{L}})) \\
= \;& \{ \text{ definition of } \otimes \text{ (101) } \} \\
& (\Phi \otimes \Psi)\, ((\!|\Phi|\!)_{\mathsf{L}} \vartriangle (\!|\Psi|\!)_{\mathsf{L}})
\end{aligned}$$

Exercise 50. Consider the following string of isomorphisms.

$$\begin{aligned}
&\mathsf{L}\,A \to B_1 \times B_2\\
&\cong (\mathsf{L}\,A \to B_1) \times (\mathsf{L}\,A \to B_2)\\
&\cong (A \to \mathsf{R}\,B_1) \times (A \to \mathsf{R}\,B_2)\\
&\cong A \to \mathsf{R}\,B_1 \times \mathsf{R}\,B_2\\
&\cong A \to \mathsf{R}\,(B_1 \times B_2)\\
&\cong \mathsf{L}\,A \to B_1 \times B_2
\end{aligned}$$

Justify each step. Why is the generalised banana-split law in a sense unsurprising? □

The *fusion law* states a condition for fusing an arrow $h : \mathscr{C}(A, B)$ with an adjoint fold $(\!|\Phi|\!)_{\mathsf{L}} : \mathscr{C}(\mathsf{L}\,(\mu\mathsf{F}), A)$ to form another adjoint fold $(\!|\Psi|\!)_{\mathsf{L}} : \mathscr{C}(\mathsf{L}\,(\mu\mathsf{F}), B)$. The condition can be easily calculated.

$$\begin{aligned}
& h \cdot (\!|\Phi|\!)_{\mathsf{L}} = (\!|\Psi|\!)_{\mathsf{L}}\\
\iff\ & \{\ \text{uniqueness property (98)}\ \}\\
& h \cdot (\!|\Phi|\!)_{\mathsf{L}} \cdot \mathsf{L}\,\mathit{in} = \Psi\,(h \cdot (\!|\Phi|\!)_{\mathsf{L}})\\
\iff\ & \{\ \text{computation (99)}\ \}\\
& h \cdot \Phi\,(\!|\Phi|\!)_{\mathsf{L}} = \Psi\,(h \cdot (\!|\Phi|\!)_{\mathsf{L}})\\
\Longleftarrow\ & \{\ \text{abstracting away from } (\!|\Phi|\!)_{\mathsf{L}}\ \}\\
& \forall f\ .\ h \cdot \Phi\,f = \Psi\,(h \cdot f)
\end{aligned}$$

Consequently,

$$h \cdot (\!|\Phi|\!)_{\mathsf{L}} = (\!|\Psi|\!)_{\mathsf{L}} \quad\Longleftarrow\quad \forall f\ .\ h \cdot \Phi\,f = \Psi\,(h \cdot f)\ . \tag{103}$$

As for generalised banana-split, the fusion condition $h \cdot \Phi\,f = \Psi\,(h \cdot f)$ mentions neither the base functor F nor the adjoint functor L, which makes the law easy to use.

Exercise 51. Let a and b be the algebras corresponding to the base functions Φ and Ψ (for $\mathsf{L} = \mathsf{Id}$). Show that

$$h \cdot a = b \cdot \mathsf{F}\,h \quad\iff\quad \forall f\ .\ h \cdot \Phi\,f = \Psi\,(h \cdot f)\ .$$

In other words, the fusion condition requires h to be an F-algebra homomorphism. □

Example 29. The function *height* determines the height of a stack.

$$\begin{aligned}
&\mathit{height} : \mathit{Stack} && \to \mathit{Nat}\\
&\mathit{height}\ \ \mathit{Empty} && = 0\\
&\mathit{height}\ \ (\mathit{Push}\,(n, s)) && = 1 + \mathit{height}\,s
\end{aligned}$$

Let us show that *height* is a monoid homomorphism from the stack monoid to the monoid of natural numbers with addition, $\mathit{height} : (\mathit{Stack}, \mathit{Empty}, \diamond) \to (\mathit{Nat}, 0, +)$:

$$\mathit{height}\,\mathit{Empty} = 0 \ , \tag{104}$$

$$\mathit{height}\,(x \diamond y) = \mathit{height}\,x + \mathit{height}\,y \ , \tag{105}$$

or, written in a point-free style,

$$\mathit{height} \cdot \mathit{empty} = \mathit{zero} \ , \tag{106}$$

$$\mathit{height} \cdot \mathit{cat} = \mathit{plus} \cdot (\mathit{height} \times \mathit{height}) \ . \tag{107}$$

Here *zero* is the constant arrow that yields 0, *empty* is the constant arrow that yields *Empty*, and, finally, *cat* and *plus* are $\diamond$ and $+$ written prefix. The first condition (106) is an immediate consequence of *height*'s definition. Regarding the second condition (107), there is no obvious zone of attack, as neither the left- nor the right-hand side is an adjoint fold. Consequently, we proceed in two steps: we first demonstrate that the left-hand side can be fused to an adjoint fold, and then we show that the right-hand side satisfies the adjoint fixed-point equation of this fold.

For the first step, we are seeking a base function $\mathfrak{height2}$ so that

$$\mathit{height} \cdot (\!|\mathfrak{cat}|\!)_{\mathsf{L}} = (\!|\mathfrak{height2}|\!)_{\mathsf{L}} \ ,$$

where $\mathsf{L} = - \times \mathit{Stack}$. The base function $\mathfrak{cat}$ is defined in Example 13. Fusion (103) immediately gives us

$$\forall \mathit{cat} \, . \, \mathit{height} \cdot \mathfrak{cat}\,\mathit{cat} = \mathfrak{height2}\,(\mathit{height} \cdot \mathit{cat}) \ , \tag{108}$$

from which we can easily synthesise a definition of $\mathfrak{height2}$:

Case $\mathfrak{Empty}$:

$$\begin{aligned} & \mathfrak{height2}\,(\mathit{height} \cdot \mathit{cat})\,(\mathfrak{Empty}, y) \\ = \ & \{ \text{ specification of } \mathfrak{height2} \text{ (108) } \} \\ & \mathit{height}\,(\mathfrak{cat}\,\mathit{cat}\,(\mathfrak{Empty}, y)) \\ = \ & \{ \text{ definition of } \mathfrak{cat} \text{ (Example 13) } \} \\ & \mathit{height}\,y \ . \end{aligned}$$

Case $\mathfrak{Push}\,(a, x)$:

$$\begin{aligned} & \mathfrak{height2}\,(\mathit{height} \cdot \mathit{cat})\,(\mathfrak{Push}\,(a, x), y) \\ = \ & \{ \text{ specification of } \mathfrak{height2} \text{ (108) } \} \\ & \mathit{height}\,(\mathfrak{cat}\,\mathit{cat}\,(\mathfrak{Push}\,(a, x), y)) \\ = \ & \{ \text{ definition of } \mathfrak{cat} \text{ (Example 13) and } \mathit{In} \cdot \mathfrak{Push} = \mathit{Push} \ \} \\ & \mathit{height}\,(\mathit{Push}\,(a, \mathit{cat}\,(x, y))) \\ = \ & \{ \text{ definition of } \mathit{height} \ \} \\ & 1 + (\mathit{height} \cdot \mathit{cat})\,(x, y) \ . \end{aligned}$$

Abstracting away from $height \cdot cat$, we obtain

$$\begin{aligned}
&\mathfrak{height2} : \forall x \,.\, (\mathsf{L}\, x \to Nat) \to (\mathsf{L}\, (\mathfrak{Stack}\, x) && \to Nat) \\
&\mathfrak{height2} \quad height2 \quad (\mathfrak{Empty}, \quad y) && = height\, y \\
&\mathfrak{height2} \quad height2 \quad (\mathfrak{Push}\, (a, x), y) && = 1 + height2\, (x, y) \ .
\end{aligned}$$

For the second step, we have to show

$$plus \cdot (height \times height) = (\!|\mathfrak{height2}|\!)_{\mathsf{L}} \ .$$

Appealing to uniqueness (98), we are left with the proof obligation

$$plus \cdot (height \times height) \cdot \mathsf{L}\, in = \mathfrak{height2}\, (plus \cdot (height \times height)) \ ,$$

which is straightforward to discharge. □

3.4.2 Unique Fixed-point Principle. Assume that you want to prove the equality of two arrows. In the fortunate case that one of the arrows takes the form of an adjoint fold, we can either appeal to the uniqueness property, or preferably, invoke the fusion law. Unfortunately, more often than not neither arrow is given explicitly as an adjoint fold, in which case none of the laws is directly applicable. Property (107) illustrates this observation: both sides of the equation involve adjoint folds, but they are not themselves adjoint folds.

The following proof method, the *unique fixed-point principle*, provides a way out of this dilemma. The idea is to demonstrate $f \cdot \mathsf{L}\, in = \Theta\, f$ and $\Theta\, g = g \cdot \mathsf{L}\, in$. If the equation $x \cdot \mathsf{L}\, in = \Theta\, x$ has a unique solution, then we may conclude that $f = g$. The important point is that we discover the base function Θ on the fly during the calculation. A proof in this style is laid out as follows.

$$\begin{aligned}
& f \cdot \mathsf{L}\, in \\
=\ & \{ \text{ why? } \} \\
& \Theta\, f \\
\propto\ & \{ \ x \cdot \mathsf{L}\, in = \Theta\, x \text{ has a unique solution } \} \\
& \Theta\, g \\
=\ & \{ \text{ why? } \} \\
& g \cdot \mathsf{L}\, in
\end{aligned}$$

The symbol $\propto$ is meant to suggest a link connecting the upper and the lower part. Overall, the proof establishes that $f = g$. An analogous approach can be used to prove the equality of two adjoint unfolds.

Example 30. Let us show $height\, (x \diamond y) = height\, x + height\, y$ (105) a second time, this time using the unique fixed-point principle. For reasons of brevity it is useful to condense the definitions of cat and $height$ into single equations (we abbreviate

$\mathfrak{Empty}$ and $\mathfrak{Push}$ by $\mathfrak{E}$ and $\mathfrak{P}$).

$$\begin{aligned}
&\mathit{cat} : (\mu\mathfrak{Stack}, \mu\mathfrak{Stack}) \to \mu\mathfrak{Stack} \\
&\mathit{cat}\ \ (\mathit{In}\, s,\ \ u) \quad = \mathbf{case}\, s\, \mathbf{of}\, \{\mathfrak{E} \to u; \mathfrak{P}\,(a, t) \to \mathit{In}\,(\mathfrak{P}\,(a, \mathit{cat}\,(t, u)))\} \\
&\mathit{height} : \mu\mathfrak{Stack} \to \mathit{Nat} \\
&\mathit{height}\ \ (\mathit{In}\, s) \ = \mathbf{case}\, s\, \mathbf{of}\, \{\mathfrak{E} \to 0; \mathfrak{P}\,(a, t) \to 1 + \mathit{height}\, t\}
\end{aligned}$$

Expressing the calculation in a pointwise style leads to a more attractive proof, which proceeds as follows:

$$\begin{aligned}
&\mathit{height}\,(\mathit{cat}\,(\mathit{In}\, s, u)) \\
={}& \quad \{\ \text{definition of } \mathit{cat}\ \} \\
&\mathit{height}\,(\mathbf{case}\, s\, \mathbf{of}\, \{\mathfrak{E} \to u; \mathfrak{P}\,(a, t) \to \mathit{In}\,(\mathfrak{P}\,(a, \mathit{cat}\,(t, u)))\}) \\
={}& \quad \{\ \mathbf{case}\text{-fusion}\ \} \\
&\mathbf{case}\, s\, \mathbf{of}\, \{\mathfrak{E} \to \mathit{height}\, u; \mathfrak{P}\,(a, t) \to \mathit{height}\,(\mathit{In}\,(\mathfrak{P}\,(a, \mathit{cat}\,(t, u))))\} \\
={}& \quad \{\ \text{definition of } \mathit{height}\ \} \\
&\mathbf{case}\, s\, \mathbf{of}\, \{\mathfrak{E} \to \mathit{height}\, u; \mathfrak{P}\,(a, t) \to 1 + \mathit{height}\,(\mathit{cat}\,(t, u))\} \\
\propto{}& \quad \{\ x\,(\mathit{In}\, s, u) = \mathbf{case}\, s\, \mathbf{of}\, \{\mathfrak{E} \to \mathit{height}\, u; \mathfrak{P}\,(a, t) \to 1 + x\,(t, u)\}\ \} \\
&\mathbf{case}\, s\, \mathbf{of}\, \{\mathfrak{E} \to \mathit{height}\, u; \mathfrak{P}\,(a, t) \to 1 + (\mathit{height}\, t + \mathit{height}\, u)\} \\
={}& \quad \{\ (\mathit{Nat}, 0, +) \text{ is a monoid}\ \} \\
&\mathbf{case}\, s\, \mathbf{of}\, \{\mathfrak{E} \to 0 + \mathit{height}\, u; \mathfrak{P}\,(a, t) \to (1 + \mathit{height}\, t) + \mathit{height}\, u\} \\
={}& \quad \{\ \mathbf{case}\text{-fusion}\ \} \\
&(\mathbf{case}\, s\, \mathbf{of}\, \{\mathfrak{E} \to 0; \mathfrak{P}\,(a, t) \to 1 + \mathit{height}\, t\}) + \mathit{height}\, u \\
={}& \quad \{\ \text{definition of } \mathit{height}\ \} \\
&\mathit{height}\,(\mathit{In}\, s) + \mathit{height}\, u \quad .
\end{aligned}$$

Note that $\mathit{height}\,(\mathit{cat}\,(\mathit{In}\, s, u)) = \mathit{height}\,(\mathit{In}\, s) + \mathit{height}\, u$ is the pointwise version of $\mathit{height} \cdot \mathit{cat} \cdot \mathsf{L}\, \mathit{in} = \mathit{plus} \cdot (\mathit{height} \times \mathit{height}) \cdot \mathsf{L}\, \mathit{in}$. Likewise, **case**-fusion is the pointwise variant of join-fusion (21), $k \cdot (g_1 \triangledown g_2) = k \cdot g_1 \triangledown k \cdot g_2$.

The proof is short and sweet—every step is more or less forced. Furthermore, the central step, the application of the monoidal laws, stands out clearly. Along the way we have re-discovered the function *height2*. It also served as the link in the original proof, which established $\mathit{height} \cdot \mathit{cat} = \mathit{height2} = \mathit{plus} \cdot (\mathit{height} \times \mathit{height})$ in two steps. The new proof format merges the two separate proofs into one. □

On the face of it the proof above is tantalisingly close to a conventional inductive proof, with *in* marking the induction argument and $\propto$ marking the application of the induction hypothesis. Indeed, in the case of **Set** a unique fixed-point proof can be easily converted into an inductive proof. (The converse is true if the inductive proof establishes the equality of two functions: $\forall x\ .\ f\, x = g\, x$.) However, the unique fixed-point principle is agnostic of the underlying category and furthermore it works equally well for coinductive types.

Exercise 52. Show that $(\mathit{Stack}, \mathit{Empty}, \diamond)$ is a monoid:

$$\begin{aligned} \mathit{Empty} \diamond s = s = s \diamond \mathit{Empty} \ \ , \\ (s \diamond t) \diamond u = s \diamond (t \diamond u) \ \ , \end{aligned}$$

or, written in a point-free style,

$$\begin{aligned} \mathit{cat} \cdot (\mathit{empty} \vartriangle \mathit{id}) = \mathit{id} = \mathit{cat} \cdot (\mathit{id} \vartriangle \mathit{empty}) \ \ , \\ \mathit{cat} \cdot (\mathit{cat} \times \mathit{id}) = \mathit{cat} \cdot (\mathit{id} \times \mathit{cat}) \cdot \mathit{assocr} \ \ , \end{aligned}$$

where $\mathit{assocr} : (A \times B) \times C \cong A \times (B \times C)$ is the standard isomorphism between nested products. □

4 Further Reading

This section provides some background on our subject, including references for further reading. A more in-depth appreciation of related work can be found in the article "Adjoint Folds and Unfolds—An Extended Study" [11].

Category theory. The categorical trinity—category, functor and natural transformation—was discovered by Eilenberg and Mac Lane. The first treatment of categories in their own right appeared in 1945 [34]—the paper investigates the notion of a natural isomorphism and is well worth reading. The definitive reference for category theory is Mac Lane's masterpiece [8]. Introductory textbooks to category theory include Awodey [35], Barr and Wells [36] and Pierce [37].

The notion of an adjunction was introduced by Daniel Kan in 1958 [7]. Rydeheard [38] illustrates the concept using the free construction of a monoid as a running example, solving some of our exercises. (In the 1980s and 1990s there was a series of conferences on "Category Theory and Computer Science"; this paper appeared as a tutorial contribution to the first incarnation, which was called "Category Theory and Computer Programming".) Spivey [39] explored the categorical background of Bird's theory of lists [40, 41], later known as the Bird-Meertens formalism. Our calculational treatment of adjunctions is inspired by Fokkinga and Meertens' paper [42].

Recursion schemes. There is a large body of work on recursion schemes or 'morphisms'. Utilising the categorical notions of functors and natural transformations, Malcolm [43] generalised the Bird-Meertens formalism to arbitrary datatypes. His work assumed **Set** as the underlying category and was adapted by Meijer et al. [5] to the category **Cpo**. The latter paper also popularised the now famous terms *catamorphism* and *anamorphism* (for folds and unfolds), along with the banana and lens brackets ($(\!|-|\!)$ and $[\![-]\!]$). Fokkinga [44] captured mutually recursive functions by *mutumorphisms*. An alternative solution to the '*append*-problem' was proposed by Pardo [45]: he introduces *folds with parameters* and uses them to implement *generic accumulations*. Building on the work of Hagino [46], Malcolm [43] and many others, Bird and de Moor gave a comprehensive account of the "Algebra of Programming" in their seminal textbook [9]. The textbook puts a particular emphasis on a relational approach to program construction—moving from the concept of a category to the richer structure of an allegory.

The discovery of nested datatypes and their expressive power [24, 47, 48] led to a flurry of research. Standard folds on nested datatypes, which are natural transformations by construction, were perceived as not being expressive enough. The paper "Generalised folds for nested datatypes" by Bird and Paterson [31] addressed the problem by adding extra parameters to folds leading to the notion of a *generalised fold*. The second part of these lecture notes is, in fact, based on their work. In order to show that generalised folds are uniquely defined, they discuss conditions to ensure that the more general equation $x \cdot \mathsf{L}\, in = \Psi\, x$, our adjoint initial fixed-point equation, uniquely defines x. Two solutions are provided to this problem, the second of which requires L to have a right adjoint. They also show that the right Kan extension is the right adjoint of pre-composition.

An alternative, type-theoretic approach to (co)inductive types was proposed by Mendler [21]. His induction combinators R^{μ} and S^{ν} map a base function to its unique fixed point. Strong normalisation is guaranteed by the polymorphic type of the base function. The first categorical justification of Mendler-style recursion was given by De Bruin [49].

Other recursion schemes. We have shown that many recursion schemes fall under the umbrella of adjoint (un)folds. However, we cannot reasonably expect that adjoint (un)folds subsume all existing species of morphisms. For instance, a largely orthogonal extension of standard folds are *recursion schemes from comonads* [50, 51]. Very briefly, given a comonad N and a distributive law $\alpha : \mathsf{F} \circ \mathsf{N} \dot{\to} \mathsf{N} \circ \mathsf{F}$, we can define an arrow $f = (\!|\mathsf{N}\, in \cdot \alpha|\!) : \mu\mathsf{F} \to \mathsf{N}\,(\mu\mathsf{F})$ that fans out a data structure. Then the equation in the unknown $x : \mu\mathsf{F} \to A$,

$$x \cdot in = a \cdot \mathsf{F}\,(\mathsf{N}\, x \cdot f) \enspace ,$$

has a unique solution for every algebra $a : \mathsf{F}\,(\mathsf{N}\, A) \to A$. This scheme includes so-called histomorphisms as a special case (the Fibonacci function is an example of a histomorphism).

We have noted that initial algebras and final coalgebras are different entities. The fact that $\mu\mathsf{F}$ and $\nu\mathsf{F}$ are not compatible in general has the unfortunate consequence that we cannot freely combine folds (consumers) and unfolds (producers). A way out of this dilemma is to use hylomorphisms based on recursive coalgebras as a structured recursion scheme [52]. Very briefly, a coalgebra $\langle C,\, c\rangle$ is called *recursive* if for *every* algebra $\langle A,\, a\rangle$ the equation in the unknown $x : \mathscr{C}(C, A)$,

$$x = a \cdot \mathsf{G}\, x \cdot c \enspace , \tag{109}$$

has a *unique* solution. The equation captures the *divide-and-conquer* pattern of computation: a problem is divided into sub-problems (c), the sub-problems are solved recursively ($\mathsf{G}\, x$), and finally the sub-solutions are combined into a single solution (a). The uniquely defined arrow x is called a *hylomorphism.* Hylomorphisms are more expressive than adjoint folds. The added expressive power comes at a price, however. Hylomorphisms sometimes suffer from the practical problem that a suitable control functor (G above) is hard to find, see [11] for a more in-depth comparison.

Type fusion. The initial algebra approach to the semantics of datatypes originates in the work of Lambek [13] on fixed points in categories. Lambek suggests that lattice theory provides a fruitful source of inspiration for results in category theory. This viewpoint was taken up by Backhouse [53], who generalised a number of lattice-theoretic fixed point rules to category theory. One important law is *type fusion*, which allows us to fuse an application of a functor with an initial algebra to form another initial algebra

$$\mathsf{L}\,(\mu\mathsf{F}) \cong \mu\mathsf{G} \quad \Longleftarrow \quad \mathsf{L}\circ\mathsf{F} \cong \mathsf{G}\circ\mathsf{L} \ .$$

The witnesses of the isomorphism $\mathsf{L}\,(\mu\mathsf{F}) \cong \mu\mathsf{G}$ can be defined as solutions of (adjoint) fixed point equations. Using type fusion one can show, for instance,

$$\mu\mathfrak{List}\,Nat \cong \mu\mathfrak{Stack} \ ,$$

which allows us to relate the functions *total* and *sums*. The paper [11] contains many more examples and also shows the intimate link between adjoint (un)folds and type fusion.

5 Conclusion

Adjunctions have proved to be one of the most important ideas in category theory, predominantly due to their ubiquity. Many mathematical constructions turn out to be adjoint functors that form adjunctions, with Mac Lane [8, p.vii] famously saying, "Adjoint functors arise everywhere."

The same is probably true of computing science. Every fundamental type or type constructor—initial object, final object, sum, product, exponential, free algebra, cofree coalgebra—arises out of an adjunction. An adjunction features an amazing economy of expression, combining introduction-, elimination, β- and η-rules in a single statement. Indeed, suitable categories of discourse can be defined just using adjunctions, for example, a category is called cartesian closed if the following adjunctions exist: $\Delta \dashv 1$, $\Delta \dashv \times$, and $- \times X \dashv (-)^X$ for each choice of X.

Adjoint folds and unfolds strike a fine balance between expressiveness and ease of use. We have shown that many Haskell functions fit under this banner. The mechanics are straightforward: given a (co)recursive function, we abstract away from the recursive calls, additionally removing occurrences of *in* and *out* that guard those calls. In **Set** termination and productivity are ensured by a naturality condition on the resulting base function. The categorical concept of an adjunction plays a central role in this development. In a sense, each adjunction captures a different recursion scheme—accumulating parameters, mutual recursion, polymorphic recursion on nested datatypes and so forth—and allows the scheme to be viewed as an instance of an adjoint (un)fold.

Acknowledgements. I am grateful to Nate Foster, Jeremy Gibbons, José Pedro Magalhães and Nicolas Wu for proof-reading several drafts of these lecture notes. They spotted numerous errors and suggested many improvements regarding style and presentation.

References

1. Jones, M.P.: A system of constructor classes: overloading and implicit higher-order polymorphism. Journal of Functional Programming **5**(1) (January 1995) 1–35
2. Wadler, P.: Theorems for free! In: The Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA'89), London, UK, Addison-Wesley Publishing Company (September 1989) 347–359
3. Wadler, P.: Comprehending monads. In: Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice, ACM Press (June 1990) 61–78
4. McBride, C., Paterson, R.: Functional Pearl: Applicative programming with effects. Journal of Functional Programming **18**(1) (2008) 1–13
5. Meijer, E., Fokkinga, M., Paterson, R.: Functional programming with bananas, lenses, envelopes and barbed wire. In Hughes, J., ed.: 5th ACM Conference on Functional Programming Languages and Computer Architecture, FPCA'91, Cambridge, MA, USA. Volume 523 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (1991) 124–144
6. Gibbons, J., Jones, G.: The under-appreciated unfold. In Felleisen, M., Hudak, P., Queinnec, C., eds.: Proceedings of the third ACM SIGPLAN international conference on Functional programming, ACM Press (1998) 273–279
7. Kan, D.M.: Adjoint functors. Transactions of the American Mathematical Society **87**(2) (1958) 294–329
8. Mac Lane, S.: Categories for the Working Mathematician. 2nd edn. Graduate Texts in Mathematics. Springer-Verlag, Berlin (1998)
9. Bird, R., De Moor, O.: Algebra of Programming. Prentice Hall Europe, London (1997)
10. Hinze, R.: A category theory primer (2010) `www.cs.ox.ac.uk/ralf.hinze/SSGIP10/Notes.pdf`.
11. Hinze, R.: Adjoint folds and unfolds—an extended study. Science of Computer Programming (2011) to appear.
12. Mulry, P.: Strong monads, algebras and fixed points. In Fourman, M., Johnstone, P., Pitts, A., eds.: Applications of Categories in Computer Science. Cambridge University Press (1992)
13. Lambek, J.: A fixpoint theorem for complete categories. Math. Zeitschr. **103** (1968) 151–161
14. Hinze, R., James, D.W.H.: Reason isomorphically! In Oliveira, B.C., Zalewski, M., eds.: Proceedings of the 6th ACM SIGPLAN workshop on Generic programming (WGP '10), New York, NY, USA, ACM (September 2010) 85–96
15. Peyton Jones, S.: Haskell 98 Language and Libraries. Cambridge University Press (2003)
16. Cockett, R., Fukushima, T.: About Charity. Yellow Series Report 92/480/18, Dept. of Computer Science, Univ. of Calgary (June 1992)

17. The Coq Development Team: The Coq proof assistant reference manual (2010) `http://coq.inria.fr`.
18. Goguen, J.A., Thatcher, J.W., Wagner, E.G., Wright, J.B.: Initial algebra semantics and continuous algebras. Journal of the ACM **24**(1) (January 1977) 68–95
19. Smyth, M.B., Plotkin, G.D.: The category-theoretic solution of recursive domain equations. SIAM Journal on Computing **11**(4) (1982) 761–783
20. Sheard, T., Pasalic, T.: Two-level types and parameterized modules. Journal of Functional Programming **14**(5) (September 2004) 547–587
21. Mendler, N.P.: Inductive types and type constraints in the second-order lambda calculus. Annals of Pure and Applied Logic **51**(1–2) (1991) 159–172
22. Giménez, E.: Codifying guarded definitions with recursive schemes. In Dybjer, P., Nordström, B., Smith, J.M., eds.: Types for Proofs and Programs, International Workshop TYPES'94, Båstad, Sweden, June 6-10, 1994, Selected Papers. Volume 996 of Lecture Notes in Computer Science., Springer-Verlag (1995) 39–59
23. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press (1998)
24. Bird, R., Meertens, L.: Nested datatypes. In Jeuring, J., ed.: Fourth International Conference on Mathematics of Program Construction, MPC'98, Marstrand, Sweden. Volume 1422 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (June 1998) 52–67
25. Mycroft, A.: Polymorphic type schemes and recursive definitions. In Paul, M., Robinet, B., eds.: Proceedings of the International Symposium on Programming, 6th Colloquium, Toulouse, France. Volume 167 of Lecture Notes in Computer Science. (1984) 217–228
26. Hinze, R., Peyton Jones, S.: Derivable type classes. In Hutton, G., ed.: Proceedings of the 2000 ACM SIGPLAN Haskell Workshop. Volume 41(1) of Electronic Notes in Theoretical Computer Science., Elsevier Science (August 2001) 5–35 The preliminary proceedings appeared as a University of Nottingham technical report.
27. Trifonov, V.: Simulating quantified class constraints. In: Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell, New York, NY, USA, ACM (2003) 98–102
28. Lämmel, R., Peyton Jones, S.: Scrap your boilerplate with class: extensible generic functions. In Pierce, B., ed.: Proceedings of the 2005 International Conference on Functional Programming, Tallinn, Estonia, September 26–28, 2005. (September 2005)
29. Meertens, L.: Paramorphisms. Formal Aspects of Computing **4** (1992) 413–424
30. Backhouse, R., Bijsterveld, M., Van Geldrop, R., Van der Woude, J.: Category theory as coherently constructive lattice theory (2003) Working Document, available from `http://www.cs.nott.ac.uk/~rcb/MPC/CatTheory.ps.gz`.
31. Bird, R., Paterson, R.: Generalised folds for nested datatypes. Formal Aspects of Computing **11**(2) (1999) 200–222
32. Hinze, R.: Efficient generalized folds. In Jeuring, J., ed.: Proceedings of the second Workshop on Generic Programming. (2000) 1–16 The proceedings appeared as a technical report of Universiteit Utrecht, UU-CS-2000-19.
33. Hinze, R.: Kan extensions for program optimisation—*Or:* art and dan explain an old trick. In Gibbons, J., Nogueira, P., eds.: 11th International Conference on Mathematics of Program Construction (MPC '12). Volume 7342 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2012) 324–362
34. Eilenberg, S., MacLane, S.: General theory of natural equivalences. Transactions of the American Mathematical Society **58**(2) (September 1945) 231–294

35. Awodey, S.: Category Theory. 2nd edn. Oxford University Press (2010)
36. Barr, M., Wells, C.: Category Theory for Computing Science. 3rd edn. Les Publications CRM, Montréal (1999) The book is available from Centre de recherches mathématiques `http://crm.umontreal.ca/`.
37. Pierce, B.C.: Basic Category Theory for Computer Scientists. The MIT Press (1991)
38. Rydeheard, D.: Adjunctions. In Pitt, D., Abramsky, S., Poigné, A., Rydeheard, D., eds.: Category Theory and Computer Science. Volume 240 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (1986) 51–57
39. Spivey, M.: A categorical approach to the theory of lists. In Van de Snepscheut, J., ed.: Mathematics of Program Construction. Volume 375 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (1989) 399–408
40. Bird, R.: An introduction to the theory of lists. In Broy, M., ed.: Proceedings of the NATO Advanced Study Institute on Logic of programming and calculi of discrete design, Marktoberdorf, Germany, Springer Berlin / Heidelberg (1987) 5–42
41. Bird, R.: A calculus of functions for program derivation. Technical Report PRG-64, Programming Research Group, Oxford University Computing Laboratory (1987)
42. Fokkinga, M.M., Meertens, L.: Adjunctions. Technical Report Memoranda Inf 94-31, University of Twente, Enschede, Netherlands (June 1994)
43. Malcolm, G.: Data structures and program transformation. Science of Computer Programming **14**(2–3) (1990) 255–280
44. Fokkinga, M.M.: Law and Order in Algorithmics. PhD thesis, University of Twente (February 1992)
45. Pardo, A.: Generic accumulations. In Gibbons, J., Jeuring, J., eds.: Proceedings of the IFIP TC2 Working Conference on Generic Programming, Schloss Dagstuhl. Volume 243., Kluwer Academic Publishers (July 2002) 49–78
46. Hagino, T.: A typed lambda calculus with categorical type constructors. In Pitt, D., Poigne, A., Rydeheard, D., eds.: Category Theory and Computer Science. Volume 283 of Lecture Notes in Computer Science. (1987)
47. Connelly, R.H., Morris, F.L.: A generalization of the trie data structure. Mathematical Structures in Computer Science **5**(3) (September 1995) 381–418
48. Okasaki, C.: Catenable double-ended queues. In: Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming, Amsterdam, The Netherlands (June 1997) 66–74 *ACM SIGPLAN Notices,* 32(8), August 1997.
49. De Bruin, P.J.: Inductive types in constructive languages. PhD thesis, University of Groningen (1995)
50. Uustalu, T., Vene, V., Pardo, A.: Recursion schemes from comonads. Nordic J. of Computing **8** (September 2001) 366–390
51. Bartels, F.: Generalised coinduction. Math. Struct. in Comp. Science **13** (2003) 321—-348
52. Capretta, V., Uustalu, T., Vene, V.: Recursive coalgebras from comonads. Information and Computation **204**(4) (2006) 437–468
53. Backhouse, R., Bijsterveld, M., Van Geldrop, R., Van der Woude, J.: Categorical fixed point calculus. In Pitt, D., Rydeheard, D.E., Johnstone, P., eds.: Proceedings of the 6th International Conference on Category Theory and Computer Science (CTCS '95), Cambridge, UK. Volume 953 of Lecture Notes in Computer Science., Springer-Verlag (August 1995) 159–179