Science of Computer Programming 78 (2013) 2108–2159

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Adjoint folds and unfolds—An extended study

Ralf Hinze

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

ARTICLE INFO

Article history:
Received 21 January 2011
Accepted 21 August 2011
Available online 14 August 2012

Keywords:
Initial algebra
Fold
Final coalgebra
Unfold
Adjunction
Kan extension
Fusion
Haskell

ABSTRACT

Folds and unfolds are at the heart of the algebra of programming. They allow the cognoscenti to derive and manipulate programs rigorously and effectively. However, most, if not all, programs require some tweaking to be given the form of an (un)fold. In this article, we remedy the situation by introducing adjoint (un)folds. We demonstrate that most programs are already of the required form and thus are directly amenable to formal manipulation. Central to the development is the categorical notion of an adjunction, which links adjoint (un)folds to standard (un)folds. We discuss a multitude of basic adjunctions and ways of combining adjunctions and show that they are directly relevant to programming. Furthermore, we develop the calculational properties of adjoint (un)folds, providing several fusion laws, which codify basic optimisation principles. We give a novel proof of type fusion based on adjoint folds and discuss several applications—type fusion states conditions for fusing a left adjoint with an initial algebra to form another initial algebra. The formal development is complemented by a series of examples in Haskell.

1. Introduction

One Ring to rule them all, One Ring to find them,
One Ring to bring them all and in the darkness bind them
The Lord of the Rings—J. R. R. Tolkien.

Effective calculations are likely to be based on a few fundamental principles. The theory of initial datatypes aspires to play that rôle when it comes to calculating programs. And indeed, a single combining form and a single proof principle rule them all: programs are expressed as folds, program calculations are 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 alongside 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.

We offer a solution to this problem in the form of adjoint folds and unfolds. 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 though: 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 to be expected, this dualises to unfolds. I cannot claim originality

E-mail address: ralf.hinze@cs.ox.ac.uk.
URL: http://www.cs.ox.ac.uk/ralf.hinze/.

0167-6423/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2012.07.011

for the idea: Bird and Paterson [11] used the approach to demonstrate that their generalised folds are uniquely defined. The purpose of the present article is to show that the idea is more profound and more far-reaching. In a sense, we turn a proof technique into a definitional principle and explore the consequences and opportunities of doing this. Specifically, the main contributions of this article are the following:

- we introduce folds and unfolds as solutions of so-called Mendler-style equations—Mendler-style folds have been studied before [70], but we believe that they deserve to be better known;
- we show that by choosing suitable base categories mutually recursive types, parametric types and generalised algebraic datatypes are subsumed by the framework;
- we generalise Mendler-style equations to adjoint equations and demonstrate that many programs are of the required form;
- we conduct a systematic study of adjunctions and ways of combining adjunctions and show their relevance to programming;
- we develop the calculational properties of adjoint folds and unfolds;
- we employ the laws to provide a novel proof of type fusion, fusing a left adjoint with an initial algebra to form another initial algebra, and discuss several applications—type fusion has been described before [5], but again we believe that it deserves to be better known;
- finally, we relate adjoint folds and unfolds to other recursion schemes, most notably generalised iteration [53] and hylomorphisms based on recursive coalgebras [12].

We largely follow a deductive approach: simple (co)recursive programs are naturally captured as solutions of Mendler-style equations; adjoint equations generalise them in a straightforward way. Furthermore, we emphasise duality throughout by developing adjoint folds and unfolds in tandem.

Prerequisites. A basic knowledge of category theory is assumed, along the lines of the categorical trinity: categories, functors and natural transformations. I have made some effort to keep the article sufficiently self-contained, explaining the more advanced concepts as we go along. Most of the category-theoretic results can be found in the textbook by Mac Lane [49]—whenever possible I point to the relevant definitions and theorems. In a sense, the purpose of this article is to demonstrate that the results are highly relevant to programming. Some knowledge of the functional programming language Haskell [60] is useful, as the formal development is parallelled by a series of programming examples.

Outline. The rest of the article is structured as follows. Section 2 introduces some notation, serving mainly as a handy reference. Section 3 reviews conventional folds and unfolds. We take a somewhat non-standard approach and introduce them as solutions of Mendler-style equations. Section 4 generalises these equations to adjoint equations and demonstrates that many Haskell functions fall under this umbrella. Central to the development is the categorical notion of an adjunction. Section 5 introduces a multitude of basic adjunctions and Section 6 looks at different ways of combining adjunctions. Section 7 develops the calculational properties of adjoint folds and unfolds. Like their vanilla counterparts, they enjoy reflection, computation and a variety of fusion laws. We then lift fusion to the realm of objects and functors in Section 8, where type fusion allows us to fuse an application of a left adjoint with an initial algebra to form another initial algebra. Each of the adjunctions introduced in Section 5 provides an interesting instance of type fusion. We discuss type firstification, type specialisation, tabulation and several others. Section 9 relates adjoint (un)folds to other prominent recursion schemes: hylomorphisms based on recursive coalgebras and cohylomorphisms based on corecursive algebras. Finally, Section 10 reviews related work and Section 11 concludes.

The article is based on the papers "Adjoint Folds and Unfolds, Or: Scything through the Thicket of Morphisms" presented at MPC 2010 and "Type Fusion" presented at AMAST 2010. The material has been thoroughly revised and extended. Most notably, Sections 3.5, 4.3, 5.2, 5.4, 5.9, 5.10, 6, 7, 8.1, 8.3, 8.4 and 9 are new.

2. Notation

We let $\mathbb{C}$, $\mathbb{D}$ etc. range over categories. By abuse of notation $\mathbb{C}$ also denotes the class of objects: $A : \mathbb{C}$ expresses that A is an object of $\mathbb{C}$. We let A, B etc. range over objects. The class of arrows from $A : \mathbb{C}$ to $B : \mathbb{C}$ is denoted $\mathbb{C}(A, B)$. If the category $\mathbb{C}$ is obvious from the context, we abbreviate $f : \mathbb{C}(A, B)$ by $f : A \to B$ or by $f : B \leftarrow A$. The arrow notation is used in particular for total functions, arrows in **Set**, and functors, arrows in **Cat**. We let f, g etc. range over arrows. Sometimes, we abbreviate the identity id_A by A. The inverse of an isomorphism f is denoted f°. We also write $f : A \cong B : f^{\circ}$ to express that $f : A \to B$ and $f^{\circ} : A \leftarrow B$ witness the isomorphism $A \cong B$.

Partial applications of functors are often written using 'categorical dummies', where $-$ marks the first and $=$ the optional second argument. As an example, $- \times A$ denotes the functor which maps X to $X \times A$ and f to $f \times A$. Another example is the so-called *hom-functor* $\mathbb{C}(-, =) : \mathbb{C}^{\mathrm{op}} \times \mathbb{C} \to$ **Set**, whose action on arrows is given by

$$\mathbb{C}(f, g)\, h = g \cdot h \cdot f. \tag{1}$$

We also use 'big' lambda notation to denote functors: $- \times A$ is alternatively written $\Lambda X \;.\; X \times A$. Very briefly, lambda terms can be interpreted in a cartesian closed category; the category **Cat** of small categories and functors is cartesian closed [49, p. 98]. We let F, G etc. and $\mathfrak{F}$, $\mathfrak{G}$ etc. range over functors.

Let $\mathsf{F}, \mathsf{G} : \mathbb{C} \to \mathbb{D}$ be two functors. The class of natural transformations from F to G is denoted $\mathsf{Nat}\,(\mathsf{F}, \mathsf{G})$. We usually abbreviate $\alpha : \mathsf{Nat}\,(\mathsf{F}, \mathsf{G})$ by $\alpha : \mathsf{F} \stackrel{.}{\to} \mathsf{G}$. In line with the big lambda notation for functors, we also use $\alpha : \forall X \,.\, E_1 \to E_2$ as a shorthand for $(\Lambda X \,.\, E_1) \stackrel{.}{\to} (\Lambda X \,.\, E_2)$. This notation is particularly convenient if we do not wish to name the two functors. Furthermore, we write $\alpha : \mathsf{F} \cong \mathsf{G}$ or $\alpha : \forall X \,.\, E_1 \cong E_2$, if α is a natural isomorphism. We let α, β etc. range over natural transformations.

The formal development is complemented by a series of Haskell programs. Unfortunately, Haskell's lexical and syntactic conventions deviate somewhat from standard mathematical practice. In Haskell, type variables start with a lower-case letter (identifiers with an initial upper-case letter are reserved for type and data constructors). Lambda expressions such as $\lambda\, x \,.\, e$ are written $\lambda x \to e$. In the Haskell code, the conventions of the language are adhered to, with one notable exception: I have taken the liberty to typeset '::' as ':'—in Haskell, '::' is used to provide a type signature, while ':' is syntax for consing an element to a list, an operator I do not use in this article.

3. Fixed-point equations

To iterate is human, to recurse divine.
L. Peter Deutsch

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

Example 1. The datatype *Stack* models stacks of natural numbers.

```
data Stack = Empty | Push (Nat, Stack)
```

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

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

```
total : Stack          → Nat
total   (Empty)        = 0
total   (Push (n, s))  = n + total s
```

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

Example 2. The type *Sequ* captures infinite sequences of naturals.

```
data Sequ = Next (Nat, Sequ)
```

The function *from* constructs the infinite sequence of natural numbers, from the given argument onwards.

```
from : Nat → Sequ
from   n   = Next (n, from (n + 1))
```

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

Both the types, *Stack* and *Sequ*, and the functions, *total* and *from*, 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 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 *Stack*. Its base function or, rather, its *base functor* is given by

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

We adopt the convention that the base functor is named after the underlying type, using 𝔱𝔥𝔦𝔰 font for the former and *this* font for the latter. The type argument of 𝔖𝔱𝔞𝔠𝔨 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 categorical notation 𝔖𝔱𝔞𝔠𝔨 is written 𝔖𝔱𝔞𝔠𝔨 $S = 1 + Nat \times S$, where 1 is the final object, $+$ denotes the sum or coproduct, and $\times$ denotes the product. (Furthermore, we use 0 to denote the initial object.)

All functors underlying first-order datatype declarations (sums of products, no function types) have two extremal fixed points: the *initial* F-*algebra* $\langle \mu\mathsf{F},\ in\rangle$ and the *final* F-*coalgebra* $\langle \nu\mathsf{F},\ out\rangle$, where $\mathsf{F} : \mathbb{C} \to \mathbb{C}$ is the functor in question. The proof that these fixed points exist is beyond the scope of this article, but see, for instance, [48]. Briefly, an F-algebra is a pair $\langle A,\ a\rangle$ consisting of an object $A : \mathbb{C}$ and an arrow $a : \mathbb{C}(\mathsf{F}\,A, A)$. Likewise, an F-coalgebra is a pair $\langle C,\ c\rangle$ consisting of an object $C : \mathbb{C}$ and an arrow $c : \mathbb{C}(C, \mathsf{F}\,C)$. By abuse of language, we shall use the term (co)algebra also for the components of the pair.

The import of initiality is that there is a unique algebra homomorphism from the initial algebra $\langle \mu\mathsf{F},\ in\rangle$ to any algebra $\langle A,\ a\rangle$, where an *algebra homomorphism* between algebras $\langle A,\ a\rangle$ and $\langle B,\ b\rangle$ is an arrow $h : \mathbb{C}(A, B)$ such that $h \cdot a = b \cdot \mathsf{F}\,h$. This unique arrow, called *fold*, is written $(\!|f|\!)$—the algebra is enclosed in banana brackets. The *uniqueness property*, also called universal property or characterisation, is captured by the following equivalence.

$$x = (\!|a|\!) \iff x \cdot in = a \cdot \mathsf{F}\,x \tag{2}$$

Dually, a *coalgebra homomorphism* between coalgebras $\langle C,\ c\rangle$ and $\langle D,\ d\rangle$ is an arrow $h : \mathbb{C}(C, D)$ such that $c \cdot h = \mathsf{F}\,h \cdot d$. Finality means that there is a unique coalgebra homomorphism from any coalgebra $\langle C,\ c\rangle$ to the final coalgebra $\langle \nu\mathsf{F},\ out\rangle$. This unique arrow, called *unfold*, is written $[\!(f)\!]$—the coalgebra is enclosed in lens brackets. Again, the *uniqueness property* can be captured by an equivalence.

$$x = [\!(c)\!] \iff out \cdot x = \mathsf{F}\,x \cdot c$$

The objects $\mu\mathsf{F}$ and $\nu\mathsf{F}$ are indeed fixed points of the functor F: the two isomorphisms are witnessed by the arrows $in : \mathsf{F}\,(\mu\mathsf{F}) \cong \mu\mathsf{F} : (\!|\mathsf{F}\,in|\!)$ and $[\!(\mathsf{F}\,out)\!] : \mathsf{F}\,(\nu\mathsf{F}) \cong \nu\mathsf{F} : out$.

Some programming languages such as Charity [15] or Coq [66] 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 [28,65]—some background is provided at the end of this section. By 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 number of steps, whereas an element of a coinductive type allows any finite number of observations.

Turning to our running examples, we view *Stack* as an initial algebra—though inductive and coinductive stacks are both equally useful. For sequences only the coinductive reading makes sense, since in **Set** the initial algebra of *Sequ*'s base functor is the empty set.

Definition 1. In Haskell, initial algebras and final coalgebras can be defined as follows.

$$\textbf{newtype}\ \mu f = In\ \{in^\circ : f\ (\mu f)\}$$
$$\textbf{newtype}\ \nu f = Out^\circ\ \{out : f\ (\nu f)\}$$

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 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' $\mu\mathfrak{Stack}$. The term is due to [64]; one level describes the structure of the data, the other level ties the recursive knot.

$$\begin{aligned}
&total : \mu\mathfrak{Stack} \to Nat\\
&total\ \ (In\ (\mathfrak{Empty})) = 0\\
&total\ \ (In\ (\mathfrak{Push}\ (n, s))) = n + total\ s
\end{aligned}$$

Now, if we abstract away from the recursive call, we obtain a non-recursive base function of type $(\mu\mathfrak{Stack} \to Nat) \to (\mu\mathfrak{Stack} \to Nat)$. 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*.

$$\begin{aligned}
&\mathfrak{total} : \forall x\ .\ (x \to Nat) \to (\mathfrak{Stack}\,x \to Nat)\\
&\mathfrak{total}\ \ total\ \ (\mathfrak{Empty}) = 0\\
&\mathfrak{total}\ \ total\ \ (\mathfrak{Push}\ (n, s)) = n + total\ s
\end{aligned}$$

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*

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

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

Applying an analogous transformation to the type *Sequ* and the function *from* we obtain

$$\textbf{data}\ \mathfrak{Sequ}\ sequ = \mathfrak{Next}\ (Nat, sequ)$$

$$\begin{aligned}
&\mathfrak{from} : \forall x\ .\ (Nat \to x) \to (Nat \to \mathfrak{Sequ}\,x)\\
&\mathfrak{from}\ \ from\ \ n = \mathfrak{Next}\ (n, from\ (n+1))
\end{aligned}$$

$$\begin{aligned}
&from : Nat \to \nu\mathfrak{Sequ}\\
&from\ \ n = Out^\circ\ (\mathfrak{from}\ from\ n).
\end{aligned}$$

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 to consider fixed-point equations of the form

$$x \cdot in = \Psi\, x, \quad \text{and dually } out \cdot x = \Psi\, x, \tag{3}$$

where the unknown x has type $\mathbb{C}(\mu\mathsf{F}, A)$ on the left and $\mathbb{C}(A, \nu\mathsf{G})$ on the right. The Haskell definitions above are pointwise versions of these equations: $x\,(In\, a) = \Psi\, x\, a$ and $x\, a = Out^{\circ}\,(\Psi\, x\, a)$. Arrows defined by equations of this form are known as *Mendler-style folds and unfolds*, because they were originally introduced by Mendler [56] 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. *Initial fixed-point equations*

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

$$x \cdot in = \Psi\, x, \tag{4}$$

where the base function Ψ has type

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

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

The naturality condition can be seen as the semantic counterpart of the *guarded-by-destructors* condition [27]. 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 (4) 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. (We shall say more about termination in Section 9.)

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

Example 3. The naturality condition is sufficient but not necessary as the example of factorial demonstrates.

$$
\begin{array}{l}
\textbf{data}\; Nat = Z \mid S\, Nat \\[4pt]
fac : Nat \;\; \to Nat \\
fac \;\; (Z) \;\; = 1 \\
fac \;\; (S\, n) = S\, n * fac\, n
\end{array}
$$

Like for *total*, we split the datatype into two levels.

$$
\begin{array}{l}
\textbf{type}\; Nat = \mu\mathfrak{Nat} \\
\textbf{data}\; \mathfrak{Nat}\; nat = \mathfrak{Z} \mid \mathfrak{S}\; nat \\
\textbf{instance}\; Functor\; \mathfrak{Nat}\; \textbf{where} \\
\quad fmap\, f\; (\mathfrak{Z}) \;\;\; = \mathfrak{Z} \\
\quad fmap\, f\; (\mathfrak{S}\, n) = \mathfrak{S}\,(f\, n)
\end{array}
$$

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

$$
\begin{array}{llll}
\mathfrak{fac} : (Nat \to Nat) \to (\mathfrak{Nat}\, Nat \to Nat) & & & \\
\mathfrak{fac} \;\; fac & (\mathfrak{Z}) & = & 1 \\
\mathfrak{fac} \;\; fac & (\mathfrak{S}\, n) & = & In\,(\mathfrak{S}\, n) * fac\, n
\end{array}
$$

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

$$
\begin{array}{llll}
\mathfrak{bogus} : (Nat \to Nat) \to (\mathfrak{Nat}\, Nat \to Nat) & & & \\
\mathfrak{bogus} \;\; fac & (\mathfrak{Z}) & = & 1 \\
\mathfrak{bogus} \;\; fac & (\mathfrak{S}\, n) & = & n * fac\,(In\,(\mathfrak{S}\, n)).
\end{array}
$$

We will get back to this example in Section 5.5 (Example 20). □

Turning to the proof of uniqueness, let us spell out the naturality property of the base function Ψ. If $h : \mathbb{C}(X_1, X_2)$, then $\mathbb{C}(\mathsf{F}\,h, id) \cdot \Psi = \Psi \cdot \mathbb{C}(h, id)$. Using the definition of hom-functors (1), this unfolds to

$$\Psi\, f \cdot \mathsf{F}\,h = \Psi\,(f \cdot h), \tag{5}$$

for all arrows $f : \mathbb{C}(X_2, A)$. 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 Eq. (4) has a *unique* solution, we show that x is a solution if and only if x is a standard fold.

$$\begin{aligned} & x \cdot in = \Psi\,x \\ \iff & \quad \{\ \Psi \text{ is natural } (5)\ \} \\ & x \cdot in = \Psi\,id \cdot \mathsf{F}\,x \\ \iff & \quad \{\ \text{uniqueness property of standard folds } (2)\ \} \\ & x = (\!|\Psi\,id|\!) \end{aligned}$$

Overloading the banana brackets, the unique solution of $x \cdot in = \Psi\,x$ 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 $\mathbb{C}(\mathsf{F}\,A, A)$, the type of F-algebras. With hindsight, we generalise the isomorphism slightly. Let $\mathsf{F} : \mathbb{D} \to \mathbb{C}$ be an arbitrary functor, then

$$\gamma : \forall A, B\ .\ \mathbb{C}(\mathsf{F}\,A, B) \cong (\forall X : \mathbb{D}\ .\ \mathbb{D}(X, A) \to \mathbb{C}(\mathsf{F}\,X, B)). \tag{6}$$

Readers versed in category theory will notice that this bijection between arrows and natural transformations is an instance of the *Yoneda Lemma* [49, p. 61]. Let $\mathsf{H} = \mathbb{C}(\mathsf{F}\,-, B)$ be the contravariant functor $\mathsf{H} : \mathbb{D}^{op} \to \mathbf{Set}$ that maps an object $A : \mathbb{D}^{op}$ to the set of arrows $\mathbb{C}(\mathsf{F}\,A, B) : \mathbf{Set}$. The Yoneda Lemma states that this set is naturally isomorphic to a set of natural transformations:

$$\forall \mathsf{H}, A\ .\ \mathsf{H}\,A \cong (\mathbb{D}^{op}(A, -) \mathrel{\dot{\to}} \mathsf{H}), \tag{7}$$

which is (6) in abstract clothing. Let us explicate the proof of (6). The functions witnessing the isomorphism are

$$\gamma\, f = \lambda\, k\ .\ f \cdot \mathsf{F}\,k \quad \text{and} \quad \gamma^{\circ}\,\Psi = \Psi\,id. \tag{8}$$

The reader should convince herself that $\gamma\,f$ is a natural transformation of the required type. It is easy to see that γ° is the left-inverse of γ.

$$\begin{aligned} & \gamma^{\circ}\,(\gamma\,f) \\ = & \quad \{\ \text{definition of } \gamma \text{ and definition of } \gamma^{\circ}\ (8)\ \} \\ & f \cdot \mathsf{F}\,id \\ = & \quad \{\ \mathsf{F} \text{ functor and identity}\ \} \\ & f \end{aligned}$$

For the opposite direction, we have to make use of the naturality property (5). (Even though Ψ has now a slightly more general type, the naturality property is the same.)

$$\begin{aligned} & \gamma\,(\gamma^{\circ}\,\Psi) \\ = & \quad \{\ \text{definition of } \gamma^{\circ} \text{ and definition of } \gamma\ (8)\ \} \\ & \lambda\,k\ .\ \Psi\,id \cdot \mathsf{F}\,k \\ = & \quad \{\ \Psi \text{ is natural } (5)\ \} \\ & \lambda\,k\ .\ \Psi\,(id \cdot k) \\ = & \quad \{\ \text{identity and extensionality } (\Psi \text{ is a function})\ \} \\ & \Psi \end{aligned}$$

Remark 1. A special case of (6) is worth singling out: if $\mathsf{F} = \mathsf{Id}$, then $\gamma\,f$ is just post-composition: $\gamma\,f = \mathbb{C}(X, f) = (f \cdot -)$. (Recall that functors respect the types: if $f : \mathbb{C}(A, B)$, then $\mathbb{C}(X, f) : \mathbb{C}(X, A) \to \mathbb{C}(X, B)$. Furthermore, $\mathbb{C}(X, f)$ is natural in X.) □

To summarise, the type of base functions is isomorphic to the type of algebras. Consequently, Mendler-style folds and standard folds are related by $(\!|\Psi|\!) = (\!|\gamma^{\circ}\,\Psi|\!) = (\!|\Psi\,id|\!)$ and $(\!|\lambda\,x\ .\ a \cdot \mathsf{F}\,x|\!) = (\!|\gamma\,a|\!) = (\!|a|\!)$.

3.2. Final fixed-point equations

The development of the previous section dualises to final coalgebras. For reference, let us spell out the details.

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

$$out \cdot x = \Psi\, x, \qquad (9)$$

where the base function Ψ has type

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

Overloading the lens brackets, the unique solution of (9) is denoted $[\!(\Psi)\!]$.

In the category **Set**, the naturality condition captures the *guarded-by-constructors* condition [27] 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 an $\mathsf{G}\,(\nu\mathsf{G})$ 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 $\mathbb{C}(A, \mathsf{G}\,A)$, the type of G-coalgebras. More generally, let $\mathsf{G} : \mathbb{D} \to \mathbb{C}$, then

$$\gamma : \forall A, B\,.\, \mathbb{C}(A, \mathsf{G}\,B) \cong (\forall X : \mathbb{D}\,.\, \mathbb{D}(B, X) \to \mathbb{C}(A, \mathsf{G}\,X)). \qquad (10)$$

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

$$\forall \mathsf{H}, B\,.\, \mathsf{H}\,B \cong (\mathbb{D}(B, -) \overset{\cdot}{\to} \mathsf{H}). \qquad (11)$$

The functions witnessing the isomorphism are

$$\gamma\, f = \lambda\, k\,.\, \mathsf{G}\,k \cdot f \quad \text{and} \quad \gamma^{\circ}\,\Psi = \Psi\, id.$$

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

3.3. Mutual type recursion: $\mathbb{C} \times \mathbb{D}$

In Haskell, datatypes can be defined by mutual recursion.

Example 4. The type of multiway trees, also known as rose trees, is defined by mutual type recursion.

$$\begin{aligned} &\textbf{data}\ Rose\ = Node\,(Nat, Roses)\\ &\textbf{data}\ Roses = Nil \mid Cons\,(Rose, Roses)\end{aligned}$$

As function follows form, functions that consume a tree or a list of trees are typically defined by mutual value recursion.

$$\begin{aligned} &flattena : Rose \to Stack\\ &flattena\ \ (Node\,(n, ts)) = Push\,(n, flattens\, ts)\\ &flattens : Roses \to Stack\\ &flattens\ \ (Nil) = Empty\\ &flattens\ \ (Cons\,(t, ts)) = cat\,(flattena\, t, flattens\, ts)\end{aligned}$$

The helper function *cat*, defined in Example 9, concatenates two stacks. □

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.

Given two categories $\mathbb{C}_1$ and $\mathbb{C}_2$, the *product category* $\mathbb{C}_1 \times \mathbb{C}_2$ is constructed as follows: an object of $\mathbb{C}_1 \times \mathbb{C}_2$ is a pair $\langle A_1, A_2\rangle$ of objects $A_1 : \mathbb{C}_1$ and $A_2 : \mathbb{C}_2$; an arrow of $(\mathbb{C}_1 \times \mathbb{C}_2)(\langle A_1, A_2\rangle, \langle B_1, B_2\rangle)$ is a pair $\langle f_1, f_2\rangle$ of arrows $f_1 : \mathbb{C}_1(A_1, B_1)$ and $f_2 : \mathbb{C}_2(A_2, B_2)$. Identity and composition are defined component-wise:

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

The functor $\mathsf{Outl} : \mathbb{C}_1 \times \mathbb{C}_2 \to \mathbb{C}_1$, which projects onto the first category, is defined by $\mathsf{Outl}\ \langle A_1, A_2\rangle = A_1$ and $\mathsf{Outl}\ \langle f_1, f_2\rangle = f_1$, and, likewise, $\mathsf{Outr} : \mathbb{C}_1 \times \mathbb{C}_2 \to \mathbb{C}_2$. (As an aside, $\mathbb{C}_1 \times \mathbb{C}_2$ is the categorical product in **Cat**.)

Returning to Example 4, the base functor underlying *Rose* and *Roses* can be seen as an endofunctor over a product category:

$$\mathsf{F}\,\langle A, B\rangle = \langle Nat \times B,\ 1 + A \times B\rangle.$$

The Haskell types *Rose* and *Roses* are then the components of the fixed point $\mu\mathsf{F} = \langle Rose, Roses\rangle$. The functions *flattena* and *flattens* are handled accordingly: we bundle them to a single arrow

$$flatten = \langle flattena, flattens\rangle : (\mathbb{C} \times \mathbb{C})(\mu\mathsf{F}, \langle Stack, Stack\rangle).$$

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

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

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 4 into this form.

Definition 2. 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 5. The base functors of *Rose* and *Roses* are

$$
\begin{aligned}
&\textbf{data}\ \mathfrak{Rose}\ tree\ trees\ \ = \mathfrak{Node}\,(Nat, trees) \\
&\textbf{data}\ \mathfrak{Roses}\ tree\ trees = \mathfrak{Nil} \mid \mathfrak{Cons}\,(tree, trees).
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathfrak{flattena} : \forall x_1\, x_2\,.\,(x_1 \to Stack, x_2 \to Stack) \to (\mathfrak{Rose}\, x_1\, x_2 \to Stack) \\
&\mathfrak{flattena}\ (flattena, flattens)\ (\mathfrak{Node}\,(n, ts)) = Push\,(n, flattens\ ts) \\
&\mathfrak{flattens} : \forall x_1\, x_2\,.\,(x_1 \to Stack, x_2 \to Stack) \to (\mathfrak{Roses}\, x_1\, x_2 \to Stack) \\
&\mathfrak{flattens}\ (flattena, flattens)\ (\mathfrak{Nil}) = Empty \\
&\mathfrak{flattens}\ (flattena, flattens)\ (\mathfrak{Cons}\,(t, ts)) = cat\,(flattena\ t, flattens\ ts)
\end{aligned}
$$

The definitions of *flattena* and *flattens* match exactly the scheme above.

$$
\begin{aligned}
&flattena : \mu_1\, \mathfrak{Rose}\, \mathfrak{Roses} \to Stack \\
&flattena\ \ (In_1\, t) = \mathfrak{flattena}\,(flattena, flattens)\ t \\
&flattens : \mu_2\, \mathfrak{Rose}\, \mathfrak{Roses} \to Stack \\
&flattens\ \ (In_2\, ts) = \mathfrak{flattens}\,(flattena, flattens)\ ts
\end{aligned}
$$

Since the two equations are equivalent to an initial fixed-point equation in $\mathbb{C} \times \mathbb{C}$, 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.

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

3.4. Type functors: $\mathbb{D}^{\mathbb{C}}$

In Haskell, datatypes can be parametrised by types.

Example 6. The type of perfectly balanced, binary leaf trees [31], perfect trees for short, is given by

```
data Perfect a = Zero a | Succ (Perfect (a, a))
instance Functor Perfect where
  fmap f (Zero a) = Zero (f a)
  fmap f (Succ p) = Succ (fmap (f × f) p)
(f × g) (a, b) = (f a, g b).
```

The type Perfect is a so-called *nested datatype* [10] as the type argument is changed in the recursive call. The constructors represent the height of the tree: a perfect tree of height 0 is a leaf; a perfect tree of height $n + 1$ is a perfect tree of height n that contains pairs of elements.

```
size : ∀a . Perfect a → Nat
size       (Zero a)  = 1
size       (Succ p)  = 2 ∗ size p
```

The function *size* calculates the size of a perfect tree, making good use of the balance condition. The definition requires *polymorphic recursion* [57], as the recursive call has type Perfect $(a, a) \to Nat$, which is a substitution instance of the declared type. □

Can we fit the definitions above into the framework of Section 3.1? Again, the answer is yes. We only have to choose a suitable base category: this time, a functor category.

Given two categories $\mathbb{C}$ and $\mathbb{D}$, the *functor category* $\mathbb{D}^{\mathbb{C}}$ is constructed as follows: an object of $\mathbb{D}^{\mathbb{C}}$ is a functor $\mathsf{F} : \mathbb{C} \to \mathbb{D}$; an arrow of $\mathbb{D}^{\mathbb{C}}(\mathsf{F}, \mathsf{G})$ is a natural transformation $\alpha : \mathsf{F} \dot{\to} \mathsf{G}$. (As an aside, $\mathbb{D}^{\mathbb{C}}$ is the exponential in **Cat**.)

The base functor of Perfect is an endofunctor over a functor category:

$$\mathsf{F}\, P = \Lambda A \,.\, A + P\,(A \times A).$$

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

$$size : \mu\mathsf{F} \dot{\to} \mathsf{K}\, Nat,$$

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

Definition 3. The definition of second-order initial algebras and final coalgebras is identical to that of Definition 1, except for an additional type argument.

```
newtype μf a = In     {in°  : f (μf) a}
newtype νf a = Out° {out : f (νf) a}
```

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 [51].

```
instance (∀x . (Functor x) ⇒ Functor (f x)) ⇒ Functor (μf) where
  fmap f (In     s) = In     (fmap f s)
instance (∀x . (Functor x) ⇒ Functor (f x)) ⇒ Functor (νf) where
  fmap f (Out° s) = Out° (fmap f s)
```

The declarations use a so-called *polymorphic predicate* [38], 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 [68,51], but the resulting code is somewhat clumsy. Alternatively, one can use 'recursive dictionaries'

```
instance Functor (f (μf)) ⇒ Functor (μf) where
  fmap f (In     s) = In     (fmap f s)
instance Functor (f (νf)) ⇒ Functor (νf) where
  fmap f (Out° s) = Out° (fmap f s)
```

and rely on the compiler to tie the recursive knot [47]. □

Let us specialise fixed-point equations to functor categories.

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

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

Example 7. Continuing Example 6, the base functor of Perfect maps functors to functors: it has kind $(\star \to \star) \to (\star \to \star)$.

$$\begin{aligned}
&\textbf{data}\ \mathfrak{Perfect}\ perfect\ a = \mathfrak{Zero}\ a \mid \mathfrak{Succ}\ (perfect\ (a, a))\\
&\textbf{instance}\ (Functor\ x) \Rightarrow Functor\ (\mathfrak{Perfect}\ x)\ \textbf{where}\\
&\quad fmap\ f\ (\mathfrak{Zero}\ a) \ = \mathfrak{Zero}\ (f\ a)\\
&\quad fmap\ f\ (\mathfrak{Succ}\ p) = \mathfrak{Succ}\ (fmap\ (f \times f)\ p)
\end{aligned}$$

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.

$$\begin{aligned}
&\mathfrak{size} : \forall x\ .\ (\forall a\ .\ x\ a \to Nat) \to (\forall a\ .\ \mathfrak{Perfect}\ x\ a \to Nat)\\
&\mathfrak{size}\quad size\quad (\mathfrak{Zero}\ a) \quad = 1\\
&\mathfrak{size}\quad size\quad (\mathfrak{Succ}\ p) \quad = 2 * size\ p\\
&size : \forall a\ .\ \mu\mathfrak{Perfect}\ a \to Nat\\
&size\quad (In\ p) \quad = \mathfrak{size}\ size\ p
\end{aligned}$$

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

3.5. A special case: generalised algebraic datatypes: $\mathbb{C}^{\mathbb{I}}$

The following example illustrates a recent addition to Haskell: generalised algebraic datatypes [34,62]. In fact, the example goes slightly beyond what is expressible in Haskell, drawing inspiration from the functional programming language *Ω*mega [63].

Example 8. The datatype *Expr a* represents expressions of type *a*, where *a* ranges over types of kind *Type*. (Haskell's notion of kinds is somewhat restricted. The language distinguishes only between $\star$, the kind of inhabited types, and type functions. The ability to define new kinds is one of the major innovations of *Ω*mega.)

$$\begin{aligned}
&\textbf{kind}\ Type = \texttt{Bool} \mid \texttt{Nat} \mid \texttt{Expr}\ Type\\
&\textbf{data}\ Expr : Type \to \star\ \textbf{where}\\
&\quad Zero \quad : Expr\ \texttt{Nat}\\
&\quad Succ \quad : Expr\ \texttt{Nat} \to Expr\ \texttt{Nat}\\
&\quad IsZero : Expr\ \texttt{Nat} \to Expr\ \texttt{Bool}\\
&\quad If \quad\ \ : (Expr\ \texttt{Bool}, Expr\ a, Expr\ a) \to Expr\ a\\
&\quad Quote : Expr\ a \to Expr\ (\texttt{Expr}\ a)
\end{aligned}$$

The kind signature makes explicit that *Expr* is not a parametric datatype in the usual sense, that is, a container type. An element of type *Expr* `Bool` is an expression that when evaluated yields a truth value; it is not some container containing truth values. The argument `Bool` : *Type* is a so-called *type code*. It is not an inhabited type, rather, it represents one, namely, *Bool* : $\star$. For emphasis, the argument of *Expr* is sometimes called a *type index*.

The type function *decode* maps a type code to the type it represents.

$$\begin{aligned}
&decode : Type \quad \to \star\\
&decode\ \ (\texttt{Bool}) \quad = Bool\\
&decode\ \ (\texttt{Nat}) \quad\ = Nat\\
&decode\ \ (\texttt{Expr}\ t) = Expr\ t
\end{aligned}$$

The following function defines an interpreter for the expression language.

$$\begin{aligned}
&eval : Expr\ a \qquad\quad \to decode\ a\\
&eval\ \ (Zero) \qquad\quad\ = 0\\
&eval\ \ (Succ\ e) \qquad\ \ = eval\ e + 1\\
&eval\ \ (IsZero\ e) \qquad = eval\ e == 0\\
&eval\ \ (If\ (e_1, e_2, e_3)) = \textbf{if}\ eval\ e_1\ \textbf{then}\ eval\ e_2\ \textbf{else}\ eval\ e_3\\
&eval\ \ (Quote\ e) \qquad = e
\end{aligned}$$

The interpreter is noticeable in that it is *tag free*. If it receives a Boolean expression, then it returns a truth value. □

The definition of *eval* proceeds by straightforward structural recursion, but is it a fold? Again, the answer is yes, and again we only have to find a suitable base category. Clearly, *Expr* does not denote a functor of type $\mathbb{C} \to \mathbb{C}$ as this category models container types of kind $\star \to \star$. Rather, it is a functor of type $\mathbb{I} \to \mathbb{C}$ where $\mathbb{I}$ is some suitable 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* $\mathbb{C}^{\mathbb{I}}$, where $\mathbb{I}$ is some arbitrary index set, is a functor category from a discrete category: $A : \mathbb{C}^{\mathbb{I}}$ if and only if

Table 1
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
$\mathbb{C} \times \mathbb{D}$	Mutually recursive inductive types Mutually recursive folds	Mutually recursive coinductive types Mutually recursive unfolds
$\mathbb{D}^{\mathbb{C}}$	Inductive type functor Higher-order fold	Coinductive type functor Higher-order unfold
$\mathbb{C}^{\mathbb{I}}$	Indexed inductive type Indexed standard fold	Indexed coinductive type Indexed standard unfold

$\forall i \in \mathbb{I} \;.\; A_i : \mathbb{C}$ and $f : \mathbb{C}^{\mathbb{I}}(A, B)$ if and only if $\forall i \in \mathbb{I} \;.\; f_i : \mathbb{C}(A_i, B_i)$. In other words, generalised algebraic datatypes are a special case of parametric types and the results of the previous section are applicable.

In Example 8, the index set $\mathbb{I}$ is the set of all type codes—the initial algebra of $\mathsf{F}X = 1 + 1 + X$ in **Set**. Both *Expr* and *decode* denote functors of type $\mathbb{I} \to \mathbb{C}$, that is, $\mathbb{I}$-indexed families of objects. Finally, *eval* is a natural transformation of type $Expr \stackrel{.}{\to} decode$, that is, an $\mathbb{I}$-indexed family of arrows.

Table 1 summarises our findings so far. As a brief reminder, **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: $\mathbf{SCpo}(A_\bot, B) \cong \mathbf{Cpo}(A, B)$. (In other words, we have an adjunction $(-)_\bot \dashv \mathsf{Incl}$ 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).

4. Adjoint fixed-point equations

⟨...⟩, good general theory does not search for the maximum generality, but for the right generality.
Categories for the Working Mathematician—Saunders Mac Lane

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 9. The function *cat* concatenates two stacks.

$$\begin{array}{lll} cat : (Stack, & Stack) \to & Stack \\ cat \;\; (Empty, & ns) & = ns \\ cat \;\; (Push\,(m, ms), ns) & & = Push\,(m, cat\,(ms, ns)) \end{array}$$

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 10. The functions *nats* and *squares* generate the sequence of natural numbers interleaved with the sequence of squares.

$$\begin{array}{l} nats : Nat \to \nu\mathfrak{Sequ} \\ nats \;\; n \;\; = Out^{\circ}\,(\mathfrak{Next}\,(n, squares\, n)) \end{array}$$

$$\begin{array}{l} squares : Nat \to \nu\mathfrak{Sequ} \\ squares \;\; n \;\; = Out^{\circ}\,(\mathfrak{Next}\,(n * n, nats\,(n + 1))) \end{array}$$

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 9 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 $x \cdot (in \times id) = \Psi\, x$. The central idea of this article is to model this context by a functor, generalising fixed-point equations to

$$x \cdot \mathsf{L}\, in = \Psi\, x, \quad \text{and dually } \mathsf{R}\, out \cdot x = \Psi\, x, \tag{12}$$

where the unknown x has type $\mathbb{C}(\mathsf{L}\,(\mu\mathsf{F}), A)$ on the left and $\mathbb{C}(A, \mathsf{R}\,(\nu\mathsf{G}))$ 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 5.5 discusses a suitable choice for R in Example 10.

Of course, the functors L and R cannot be arbitrary. For instance, for $\mathsf{L} = \mathsf{K}\,A$ where $\mathsf{K} : \mathbb{C} \to \mathbb{C}^{\mathbb{D}}$ is the constant functor and $\Psi = id$, the equation $x \cdot \mathsf{L}\, in = \Psi\, x$ 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, a concept we introduce next.

Let $\mathbb{C}$ and $\mathbb{D}$ be categories. The functors L and R are *adjoint*, $\mathsf{L} \dashv \mathsf{R}$,

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

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

$$\phi : \mathbb{C}(\mathsf{L}\,A, B) \cong \mathbb{D}(A, \mathsf{R}\,B) : \phi^{\circ},$$

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 ϕ is called the *left adjunct* or *adjoint transposition*. Accordingly, ϕ° is called the *right adjunct*. We shall introduce more material about adjunctions as we go along. For a calculational introduction to adjunctions we refer the interested reader to the paper "Adjunctions" by Fokkinga and Meertens [22].

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 (12) have unique solutions. This is what we do next.

4.1. Adjoint initial fixed-point equations

One Size Fits All
Frank Zappa and The Mothers of Invention

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

$$x \cdot \mathsf{L}\, in = \Psi\, x, \tag{13}$$

where the base function Ψ has type

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

The unique solution of (13) is called an *adjoint fold*, denoted $(\!|\Psi|\!)_{\mathsf{L}}$.

The proof of uniqueness makes essential use of the fact that the left adjunct ϕ is natural in A, that is, $\mathbb{D}(h, id) \cdot \phi = \phi \cdot \mathbb{C}(\mathsf{L}\,h, id)$, which translates to

$$\phi\, f \cdot h = \phi\,(f \cdot \mathsf{L}\,h). \tag{14}$$

We reason as follows.

$$\begin{aligned}
& x \cdot \mathsf{L}\, in = \Psi\, x \\
\Longleftrightarrow\ & \quad \{\ \text{adjunction: } \phi \cdot \phi^{\circ} = id \text{ and } \phi^{\circ} \cdot \phi = id\ \} \\
& \phi\,(x \cdot \mathsf{L}\, in) = \phi\,(\Psi\, x) \\
\Longleftrightarrow\ & \quad \{\ \phi \text{ is natural } (14)\ \} \\
& \phi\, x \cdot in = \phi\,(\Psi\, x) \\
\Longleftrightarrow\ & \quad \{\ \text{adjunction: } \phi \cdot \phi^{\circ} = id \text{ and } \phi^{\circ} \cdot \phi = id\ \} \\
& \phi\, x \cdot in = (\phi \cdot \Psi \cdot \phi^{\circ})\,(\phi\, x) \\
\Longleftrightarrow\ & \quad \{\ \text{Section 3.1}\ \} \\
& \phi\, x = (\!|\phi \cdot \Psi \cdot \phi^{\circ}|\!) \\
\Longleftrightarrow\ & \quad \{\ \text{adjunction: } \phi \cdot \phi^{\circ} = id \text{ and } \phi^{\circ} \cdot \phi = id\ \} \\
& x = \phi^{\circ}\,(\!|\phi \cdot \Psi \cdot \phi^{\circ}|\!)
\end{aligned}$$

In three simple steps we have transformed the adjoint fold $x : \mathbb{C}(\mathsf{L}\,(\mu\mathsf{F}), A)$ into the standard fold $\phi\, x : \mathbb{D}(\mu\mathsf{F}, \mathsf{R}\,A)$ and, alongside, the adjoint base function $\Psi : \forall X\ .\ \mathbb{C}(\mathsf{L}\,X, A) \to \mathbb{C}(\mathsf{L}\,(\mathsf{F}\,X), A)$ into the standard base function $(\phi \cdot \Psi \cdot \phi^{\circ})$:

$\forall X\,.\,\mathbb{D}(X, \mathsf{R}\,A) \to \mathbb{D}(\mathsf{F}\,X, \mathsf{R}\,A)$. We have shown in Section 3.1 that the resulting equation has a unique solution. We call the standard fold $\phi\,x$ the *transpose* of x (usually named x'). To summarise,

$$(\!|\Psi|\!)_{\mathsf{L}} = \phi^{\circ}\,(\!|\phi \cdot \Psi \cdot \phi^{\circ}|\!) \quad \text{or, equivalently, } \phi\,(\!|\Psi|\!)_{\mathsf{L}} = (\!|\phi \cdot \Psi \cdot \phi^{\circ}|\!)\,.$$

Example 11. To turn the definition of *cat*, see Example 9, into the form of an adjoint equation, we follow the same steps as in Section 3. First, we determine the base function abstracting away from the recursive call, additionally removing *in*, and then we tie the recursive knot.

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

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

The defining equation fits the pattern of an adjoint initial fixed-point equation, $x \cdot (\mathit{in} \times \mathit{id}) = \Psi\,x$. It remains to check that $\mathsf{L} = - \times \mathit{Stack}$ has a right adjoint. It turns out that this is the case: the right adjoint is $\mathsf{R} = (-)^{\mathit{Stack}}$, the so-called exponential. We shall study this adjunction, famously known as the 'curry adjunction', in detail in Section 5.3. For now, we just record that *cat* is uniquely defined. □

4.2. Adjoint final fixed-point equations

Buy one get one free!

A common form of sales promotion (BOGOF).

Dually, an *adjoint final fixed-point equation* in the unknown $x : \mathbb{D}(A, \mathsf{R}\,(\nu\mathsf{G}))$ has the syntactic form

$$\mathsf{R}\,\mathit{out} \cdot x = \Psi\,x, \tag{15}$$

where the base function Ψ has type

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

The unique solution of (15) is called an *adjoint unfold*, denoted $[\![\Psi]\!]_{\mathsf{R}}$.

4.3. Algebraic adjoint folds

It is the pervading law of all things organic and inorganic,
⟨...⟩ That form ever follows function. This is the law.

Louis Henri Sullivan

This section contains advanced material, which you may want to skip on a first reading.

We have seen in Section 3.1 that Mendler-style folds, which take a base function as an argument, are equivalent to standard folds, which rely on the notion of an algebra.

$$\text{Mendler-style: } x \cdot \mathit{in} = \Psi\,x \quad \text{algebraic: } x \cdot \mathit{in} = a \cdot \mathsf{F}\,x$$

The adjoint folds introduced in Section 4.1 can be loosely characterised as 'Mendler-style adjoint folds'. There is also a corresponding notion of 'algebraic adjoint folds', called generalised iteration in [53], and, like their vanilla variants, the two schemes are inter-definable. A standard fold insists on the idea that the control structure of a function ever follows the structure of its input data. Mendler-style adjoint folds loosen this tight coupling. The control structure is given implicitly through the adjunction. An algebraic adjoint fold makes the control structure explicit by introducing a 'control functor' G.

$$\text{Mendler-style: } x \cdot \mathsf{L}\,\mathit{in} = \Psi\,x \quad \text{algebraic: } x \cdot \mathsf{L}\,\mathit{in} = a \cdot \mathsf{G}\,x \cdot \alpha$$

An algebraic adjoint fold $x : \mathbb{C}(\mathsf{L}\,(\mu\mathsf{F}), A)$ is the unique solution of the equation on the right. The recursive call structure is governed by the functor $\mathsf{G} : \mathbb{D} \to \mathbb{D}$. The diagram below displays the functors involved.

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

Apart from the G-algebra $a : \mathbb{C}(\mathsf{G}\,A, A)$ one further ingredient is required: a natural transformation $\alpha : \mathsf{L} \circ \mathsf{F} \dot{\to} \mathsf{G} \circ \mathsf{L}$ that serves as an impedance matcher, translating the data into the control structure. (The types of $x \cdot \mathsf{L}\,in : \mathbb{C}(\mathsf{L}\,(\mathsf{F}\,(\mu\mathsf{F})), A)$ and $a \cdot \mathsf{G}\,x : \mathbb{C}(\mathsf{G}\,(\mathsf{L}\,(\mu\mathsf{F})), A)$ dictate the type of α.) The diagram below visualises the type information.

$$\begin{tikzcd}
 & \mathsf{G}\,(\mathsf{L}\,(\mu\mathsf{F})) \arrow[dr, "\mathsf{G}\,x"] & \\
\mathsf{L}\,(\mathsf{F}\,(\mu\mathsf{F})) \arrow[ur, "\alpha\,(\mu\mathsf{F})"] \arrow[d, "\mathsf{L}\,in"'] & & \mathsf{G}\,A \arrow[d, "a"] \\
\mathsf{L}\,(\mu\mathsf{F}) \arrow[rr, "x"] & & A
\end{tikzcd}$$

As an aside, there is no requirement that α is an isomorphism. However, if it is one, then we have $\mathsf{L}\,(\mu\mathsf{F}) \cong \mu\mathsf{G}$—this special case is investigated in Section 8.

It is not hard to see that Mendler-style adjoint folds subsume algebraic adjoint folds. We only have to show that the base function $\Psi\,X\,x = a \cdot \mathsf{G}\,x \cdot \alpha\,X$ has the required naturality property. The easy proof is left to the reader.

The other way round—turning a Mendler-style into an algebraic adjoint fold—is more involved. It is useful to first review some basic facts about adjunctions. An adjunction can be defined in a variety of ways. Recall that the adjuncts $\phi : \mathbb{C}(\mathsf{L}\,A, B) \cong \mathbb{D}(A, \mathsf{R}\,B) : \phi^\circ$ have to be natural both in A and B. This implies that ϕ° and ϕ are uniquely defined by their images of the identity: $\epsilon = \phi^\circ\,id$ and $\eta = \phi\,id$. 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*

$$(\epsilon \circ \mathsf{L}) \cdot (\mathsf{L} \circ \eta) = id_{\mathsf{L}} \quad \text{and} \quad (\mathsf{R} \circ \epsilon) \cdot (\eta \circ \mathsf{R}) = id_{\mathsf{R}}, \tag{16}$$

where $\circ$ denotes horizontal composition of a natural transformation with a functor: $(\mathsf{F} \circ \alpha)\,A = \mathsf{F}\,(\alpha\,A)$ and $(\alpha \circ \mathsf{F})\,A = \alpha\,(\mathsf{F}\,A)$. (The reader should convince herself that $\mathsf{F} \circ \alpha$ and $\alpha \circ \mathsf{F}$ are again natural transformations.)

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

$$\begin{aligned}
\phi^\circ\,g &= \epsilon \cdot \mathsf{L}\,g & \epsilon &= \phi^\circ\,id & \mathsf{L}\,h &= \phi^\circ\,(\eta \cdot h) \\
\phi\,f &= \mathsf{R}\,f \cdot \eta & \eta &= \phi\,id & \mathsf{R}\,k &= \phi\,(k \cdot \epsilon).
\end{aligned} \tag{17}$$

Now, given a base function $\Psi : \mathbb{C}(\mathsf{L}\,-, A) \dot{\to} \mathbb{C}(\mathsf{L}\,(\mathsf{F}\,-), A)$ we have to construct a functor G, a natural transformation $\alpha : \mathsf{L} \circ \mathsf{F} \dot{\to} \mathsf{G} \circ \mathsf{L}$ and a G-algebra $a : \mathbb{C}(\mathsf{G}\,A, A)$. The first two pieces of data are induced by the adjunction: a canonical choice for the control structure is $\mathsf{G} = \mathsf{L} \circ \mathsf{F} \circ \mathsf{R}$. Using this definition, the type of α expands to $\mathsf{L} \circ \mathsf{F} \dot{\to} \mathsf{L} \circ \mathsf{F} \circ \mathsf{R} \circ \mathsf{L}$, which suggests to define $\alpha = \mathsf{L} \circ \mathsf{F} \circ \eta$. Finally, the G-algebra is derived from the base function: $a = \Psi\,\epsilon$ or, making the types explicit, $a = \Psi\,(\mathsf{R}\,A)\,(\epsilon\,A) : \mathbb{C}(\mathsf{L}\,(\mathsf{F}\,(\mathsf{R}\,A)), A)$. It remains to show that the Mendler-style adjoint fold and the derived algebraic adjoint fold define the same arrow, which is implied by the equality of the base functions $\Psi\,X\,x = a \cdot \mathsf{G}\,x \cdot \alpha\,X$.

$$\begin{aligned}
& a \cdot \mathsf{G}\,x \cdot \alpha\,X \\
=\;& \{\text{ definition of } a, \mathsf{G} \text{ and } \alpha \} \\
& \Psi\,(\mathsf{R}\,A)\,(\epsilon\,A) \cdot (\mathsf{L} \circ \mathsf{F} \circ \mathsf{R})\,x \cdot (\mathsf{L} \circ \mathsf{F} \circ \eta)\,X \\
=\;& \{\ \mathsf{L} \text{ and } \mathsf{F} \text{ functors} \} \\
& \Psi\,(\mathsf{R}\,A)\,(\epsilon\,A) \cdot \mathsf{L}\,(\mathsf{F}\,(\mathsf{R}\,x \cdot \eta\,X)) \\
=\;& \{\ \Psi \text{ is natural: } \Psi\,X_1\,f \cdot \mathsf{L}\,(\mathsf{F}\,h) = \Psi\,X_2\,(f \cdot \mathsf{L}\,h) \} \\
& \Psi\,X\,(\epsilon\,A \cdot \mathsf{L}\,(\mathsf{R}\,x \cdot \eta\,X)) \\
=\;& \{\ \mathsf{L} \text{ functor} \} \\
& \Psi\,X\,(\epsilon\,A \cdot \mathsf{L}\,(\mathsf{R}\,x) \cdot \mathsf{L}\,(\eta\,X)) \\
=\;& \{\ \epsilon \text{ is natural: } h \cdot \epsilon\,X_1 = \epsilon\,X_2 \cdot \mathsf{L}\,(\mathsf{R}\,h) \} \\
& \Psi\,X\,(x \cdot \epsilon\,(\mathsf{L}\,X) \cdot \mathsf{L}\,(\eta\,X)) \\
=\;& \{\text{ triangle identity } (16) \} \\
& \Psi\,X\,x
\end{aligned}$$

It is instructive to turn stack concatenation into an algebraic adjoint fold.

Example 12. The canonical control structure for *cat*, introduced in Example 9, is $\mathsf{G} = \mathsf{L} \circ \mathfrak{Stack} \circ \mathsf{R}$ where $\mathsf{L} = - \times Stack$ and $\mathsf{R} = (-)^{Stack}$. The counit of $\mathsf{L} \dashv \mathsf{R}$ is function application $apply : \forall X\ .\ X^{Stack} \times Stack \to X$ defined $apply\,(f, s) = f\,s$.

The unit *return* : $\forall X \,.\, X \to (X \times \mathit{Stack})^{\mathit{Stack}}$ is given by $\mathit{return}\, x = \lambda s \,.\, (x, s)$.[1] Consequently, the natural transformation $\mathsf{L} \circ \mathfrak{Stack} \circ \mathit{return} : \mathsf{L} \circ \mathfrak{Stack} \dot{\to} \mathsf{G} \circ \mathsf{L}$ unfolds to

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

For the algebra, we partially evaluate $\mathfrak{cat}$ *apply* obtaining

$$\begin{aligned}
&\mathit{cat}^{\triangledown} : \mathsf{L}\,(\mathfrak{Stack}\,(\mathsf{R}\, \mathit{Stack})) \to \mathit{Stack}\\
&\mathit{cat}^{\triangledown} \quad (\mathfrak{Empty}, \quad ns) && = ns\\
&\mathit{cat}^{\triangledown} \quad (\mathfrak{Push}\,(m, f), ns) && = \mathit{Push}\,(m, f\, ns).
\end{aligned}$$

The algebraic adjoint fold is then

$$\begin{aligned}
&\mathit{cat} : (\mathit{Stack}, \mathit{Stack}) \to \mathit{Stack}\\
&\mathit{cat} \quad (\mathit{In}\, ms, ns) \quad = (\mathit{cat}^{\triangledown} \cdot \mathit{fmap}\, \mathit{cat} \cdot \mathit{cat}^{\leftrightarrow})\,(ms, ns).
\end{aligned}$$

Here, *fmap* is the action of the functor G on arrows. The functioning of *cat* is not entirely obvious: the natural transformation $\mathit{cat}^{\leftrightarrow}$ turns the recursive component into a function (a closure at run-time), *fmap cat* post-composes this function with *cat*, and $\mathit{cat}^{\triangledown}$ finally invokes the resulting function. □

The example demonstrates that the functor $\mathsf{G} = \mathsf{L} \circ \mathsf{F} \circ \mathsf{R}$ is not necessarily the simplest or the most obvious control structure—the parameter of *cat* is propagated unchanged to the recursive call, which suggests that creating a closure may not be necessary. Indeed, often there is a simpler choice: some G' with $\alpha' : \mathsf{L} \circ \mathsf{F} \dot{\to} \mathsf{G}' \circ \mathsf{L}$. (Can you find such a structure in the case of *cat*?) This leads to a more general question: given some (simple) algebraic adjoint fold, $\Psi\, x = a' \cdot \mathsf{G}'\, x \cdot \alpha'\, X$, is there an easy way to show that it is equivalent to the canonical one given by the construction above? (Imagine embarking on a round-trip: we turn an algebraic adjoint fold into a Mendler-style one and then convert back.) Now, the algebra of the canonical fold is $\Psi\, \epsilon$, so all we have to do is to relate $\Psi\, \epsilon$ to a'. Let us calculate.

$$\begin{aligned}
& \Psi\,(\mathsf{R}\, A)\,(\epsilon\, A)\\
= {} & \quad \{\text{ definition of } \Psi \text{ }\}\\
& a' \cdot \mathsf{G}'\,(\epsilon\, A) \cdot \alpha'\,(\mathsf{R}\, A)\\
= {} & \quad \{\text{ horizontal and vertical composition of natural transformations }\}\\
& a' \cdot ((\mathsf{G}' \circ \epsilon) \cdot (\alpha' \circ \mathsf{R}))\, A\\
= {} & \quad \{\text{ define } \tau = (\mathsf{G}' \circ \epsilon) \cdot (\alpha' \circ \mathsf{R}) \text{ }\}\\
& a' \cdot \tau\, A
\end{aligned}$$

The natural transformation τ simplifies a G to a G' structure:

$$\tau = (\mathsf{G}' \circ \epsilon) \cdot (\alpha' \circ \mathsf{R}) : \mathsf{G} \dot{\to} \mathsf{G}'.$$

The calculation shows that we can factor the algebra $\Psi\, \epsilon$ into a (simple) algebra a' and an application of the transformation τ.

Example 13. For stack concatenation, see Example 12, we can easily calculate a simplified 'control functor' (recall that $\mathsf{L} = - \times \mathit{Stack}$).

$$\begin{aligned}
& \mathsf{L} \circ \mathfrak{Stack}\\
= {} & \quad \{\text{ definition of } \mathfrak{Stack} \text{ }\}\\
& \mathsf{L} \circ (1 + -) \circ (\mathit{Nat} \times -)\\
\cong {} & \quad \{\text{ distributivity: } (- \times A) \circ (1 + -) \cong (A + -) \circ (- \times A) \text{ }\}\\
& (\mathit{Stack} + -) \circ \mathsf{L} \circ (\mathit{Nat} \times -)\\
\cong {} & \quad \{\text{ associativity: } (- \times A) \circ (B \times -) \cong (B \times -) \circ (- \times A) \text{ }\}\\
& (\mathit{Stack} + -) \circ (\mathit{Nat} \times -) \circ \mathsf{L}
\end{aligned}$$

If we define

$$\textbf{data}\ \mathfrak{Stack}'\ \mathit{stack} = \mathfrak{Empty}'\ \mathit{Stack} \mid \mathfrak{Push}'\,(\mathit{Nat}, \mathit{stack})$$

[1] Readers familiar with monads will recognise *return* as the unit of the state monad. This is not a coincidence. Every adjunction $\mathsf{L} \dashv \mathsf{R}$ gives rise to a monad, $\mathsf{R} \circ \mathsf{L}$, and to a comonad, $\mathsf{L} \circ \mathsf{R}$. For $\mathsf{L} = - \times S$ and $\mathsf{R} = (-)^S$ we obtain the well-known state monad and the less well-known costate comonad.

Table 2
Different kinds of folds.

Argument		Adjoint functor: No	Adjoint functor: Yes
	Algebra	Standard fold $(\!|a|\!)$	Algebraic adjoint fold $(\!|\lambda\, x\,.\,a \cdot \mathsf{G}\,x \cdot \alpha|\!)_{\mathsf{L}}$
	Recursive call	Mendler-style fold $(\!|\Psi|\!)$	Mendler-style adjoint fold $(\!|\Psi|\!)_{\mathsf{L}}$

then the witness of the isomorphism above is given by

$$\begin{aligned} &scat^{\leftrightarrow} : \forall x\,.\,\mathsf{L}\,(\mathfrak{Stack}\,x) &&\to \mathfrak{Stack}'\,(\mathsf{L}\,x)\\ &scat^{\leftrightarrow}\quad (\mathfrak{Empty}, \quad ns) &&= \mathfrak{Empty}'\,ns\\ &scat^{\leftrightarrow}\quad (\mathfrak{Push}\,(m, ms), ns) &&= \mathfrak{Push}'\,(m, (ms, ns)).\end{aligned}$$

Consequently, the natural transformation $\tau : \mathsf{L} \circ \mathfrak{Stack} \circ \mathsf{R} \mathrel{\dot{\to}} \mathfrak{Stack}'$ is defined

$$\begin{aligned} &\tau : \forall x\,.\,\mathsf{L}\,(\mathfrak{Stack}\,(\mathsf{R}\,x)) &&\to \mathfrak{Stack}'\,x\\ &\tau \quad (\mathfrak{Empty}, \quad ns) &&= \mathfrak{Empty}'\,ns\\ &\tau \quad (\mathfrak{Push}\,(m, f), ns) &&= \mathfrak{Push}'\,(m, f\,ns).\end{aligned}$$

Given these prerequisites, we can simplify the algebra of Example 12.

$$\begin{aligned} &scat^{\triangledown} : \mathfrak{Stack}'\,Stack &&\to Stack\\ &scat^{\triangledown}\quad (\mathfrak{Empty}'\,ns) &&= ns\\ &scat^{\triangledown}\quad (\mathfrak{Push}'\,(n, ns)) &&= Push\,(n, ns)\end{aligned}$$

It is not hard to see that $cat^{\triangledown} = scat^{\triangledown} \cdot \tau\,Stack$ and consequently

$$(\!|\lambda\, x\,.\,cat^{\triangledown} \cdot (\mathsf{L} \circ \mathfrak{Stack} \circ \mathsf{R})\,x \cdot cat^{\leftrightarrow}|\!)_{\mathsf{L}} = (\!|\lambda\, x\,.\,scat^{\triangledown} \cdot \mathfrak{Stack}'\,x \cdot scat^{\leftrightarrow}|\!)_{\mathsf{L}}.$$

The control structure of the simplified fold is straightforward: $scat^{\leftrightarrow}$ propagates the parameter to the tail of the stack, $\mathfrak{Stack}'\,cat$ recursively applies cat, and $scat^{\triangledown}$ constructs the resulting stack. In essence, we have transformed an implementation of cat involving higher-order functions into a first-order one. □

Can we always simplify the control structure? This depends, of course, on the adjunction. For the left adjoint $- \times X$ the answer is yes, if the base functor is linear like $\mathfrak{Stack}$. Using a similar calculation as in Example 13, we can always find a G' such that $\mathsf{L} \circ \mathsf{F} \cong \mathsf{G}' \circ \mathsf{L}$ (Section 8.3). In this particular case, we have $\mathsf{L}\,(\mu\mathsf{F}) \cong \mu\mathsf{G}$ (Section 8), so the adjoint fold is a standard fold in disguise. The answer is no, if the base functor is non-linear as in the following example.

Example 14. The datatype *Tree* models binary leaf trees.

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

The function

$$\begin{aligned} &flattenCat : (Tree, Stack) &&\to Stack\\ &flattenCat \quad (Leaf\,n, ns) &&= Push\,(n, ns)\\ &flattenCat \quad (Fork\,(l, r), ns) &&= flattenCat\,(l, flattenCat\,(r, ns))\end{aligned}$$

places the elements of a leaf tree onto a given stack. □

Due to the nesting of the recursive calls, there is no simple control functor, or, at least, there is no obvious one.[2]

To summarise, Mendler-style adjoint folds arise naturally. Given a recursive definition, the base function is obtained by abstracting away from the recursive calls, additionally removing *in*—we will repeatedly illustrate this process in the next section. Using general properties of adjunctions, the underlying control structure can be carved out, leading to the notion of an algebraic adjoint fold. Table 2 lists the different kinds of folds.

As usual, the development nicely dualises to final coalgebras. The details are left to the reader.

5. Basic adjunctions

adjunction, n.
1. The joining on or adding of a thing or person (to another).
Oxford English Dictionary

[2] I posed the problem of finding a simple control functor to a few colleagues. In response to the challenge Venanzio Capretta suggested $\mathsf{G}\,X = Stack + X \times X^{Stack}$, which is a slight simplification of the canonical functor $(\mathsf{L} \circ \mathsf{F} \circ \mathsf{R})\,X \cong Stack + X \times X^{Stack} \times X^{Stack} \times Stack$.

5.1. *Identity:* $\mathsf{Id} \dashv \mathsf{Id}$

The simplest example of an adjunction is $\mathsf{Id} \dashv \mathsf{Id}$, which demonstrates that adjoint fixed-point equations (12) subsume fixed-point equations (3).

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

5.2. *Isomorphism and equivalence of categories*

Identity functors are a special case of invertible functors. A functor $\mathsf{H} : \mathbb{C} \to \mathbb{D}$ is an *isomorphism of categories* if there is a functor $\mathsf{H}^\circ : \mathbb{C} \leftarrow \mathbb{D}$ with $\mathsf{H}^\circ \circ \mathsf{H} = \mathsf{Id}$ and $\mathsf{Id} = \mathsf{H} \circ \mathsf{H}^\circ$. Then the inverse functors are adjoint, $\mathsf{H}^\circ \dashv \mathsf{H}$, with adjuncts

$$\phi f = \mathsf{H} f \quad \text{and} \quad \phi^\circ g = \mathsf{H}^\circ g.$$

By symmetry, we also have $\mathsf{H} \dashv \mathsf{H}^\circ$.

The requirement that $\mathsf{H}^\circ \circ \mathsf{H}$ *equals* Id is usually too stringent—in category theory most entities are defined only up to isomorphism. The following notion weakens equality to isomorphism. A functor $\mathsf{H} : \mathbb{C} \to \mathbb{D}$ is an *equivalence of categories* if there is a functor $\mathsf{H}' : \mathbb{C} \leftarrow \mathbb{D}$ with $\epsilon : \mathsf{H}' \circ \mathsf{H} \cong \mathsf{Id}$ and $\eta : \mathsf{Id} \cong \mathsf{H} \circ \mathsf{H}'$. Then the functors are adjoint, $\mathsf{H}' \dashv \mathsf{H}$.

$$\begin{tikzcd} \mathbb{C} \arrow[r, phantom, "\perp"] \arrow[r, leftarrow, bend left, "\mathsf{H}'"] \arrow[r, bend right, "\mathsf{H}"'] & \mathbb{D} \end{tikzcd}$$

The isomorphisms ϵ and η are the units of the adjunction (Section 4.3). Consequently, the adjuncts are given by

$$\phi f = \mathsf{H} f \cdot \eta \quad \text{and} \quad \phi^\circ g = \epsilon \cdot \mathsf{H}' g.$$

As before, we also have $\mathsf{H} \dashv \mathsf{H}'$. Isomorphism of categories is a special case of equivalence where ϵ and η are manifestly identities.

In the following sections we explore more interesting examples. Each section is structured as follows: we introduce an adjunction, specialise Eqs. (12) to the adjoint functors, and then provide some Haskell examples that fit the pattern.

5.3. *Currying:* $- \times X \dashv (-)^X$

Perhaps the best-known example of an adjunction is currying. In **Set**, a function of two arguments can be treated as a function of the first argument whose values are functions of the second argument. In general, we are seeking the right adjoint of pairing with a fixed object X:

$$\phi : \forall A, B \,.\, \mathbb{C}(A \times X, B) \cong \mathbb{C}(A, B^X).$$

The object B^X is called the *exponential* of X and B (also written $X \Rightarrow B$ or $[X \to B]$). In **Set**, B^X is the set of total functions from X to B. That this adjunction exists is one of the requirements for *cartesian closure* [46].

$$\begin{tikzcd} \mathbb{C} \arrow[r, phantom, "\perp"] \arrow[r, leftarrow, bend left, "- \times X"] \arrow[r, bend right, "(-)^X"'] & \mathbb{C} \end{tikzcd}$$

In the case of **Set**, the isomorphisms are given by

$$\phi f = \lambda\, a \,.\, \lambda\, x \,.\, f\,(a, x) \quad \text{and} \quad \phi^\circ g = \lambda\,(a, x) \,.\, g\, a\, x.$$

The adjuncts are also known as *curry* and *uncurry*, hence the name curry adjunction.

Let us specialise the adjoint equations to $\mathsf{L} = - \times X$ and $\mathsf{R} = (-)^X$ in **Set**.

$$\begin{array}{ll} & x \cdot \mathsf{L}\, in = \Psi\, x \\ \iff & \{ \text{definition of } \mathsf{L} \} \\ & x \cdot (in \times id) = \Psi\, x \\ \iff & \{ \text{extensionality} \} \\ & \forall a, c \,.\, x\,(in\, a, c) = \Psi\, x\,(a, c) \end{array} \qquad \begin{array}{ll} & \mathsf{R}\, out \cdot x = \Psi\, x \\ \iff & \{ \text{definition of } \mathsf{R} \} \\ & (out \cdot -) \cdot x = \Psi\, x \\ \iff & \{ \text{extensionality} \} \\ & \forall a, c \,.\, out\,(x\, a\, c) = \Psi\, x\, a\, c \end{array}$$

The adjoint fold takes two arguments, an element of an initial algebra and a second argument (often an accumulator, see Example 15), 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.

We have briefly discussed concatenation in Section 4 (Examples 9 and 11). Here is another example along those lines.

Example 15. The function *shunt* pushes the elements of the first onto the second stack.

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

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

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

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

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

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

The transposed fold, ϕ *shunt*,

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

is simply the curried variant of *shunt*. □

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

Example 16. The type of lists is given as the initial algebra of a higher-order base functor of kind $(\star \to \star) \to (\star \to \star)$.

$$\mathbf{data}\ \mathfrak{List}\,\mathit{list}\ a = \mathfrak{Nil} \mid \mathfrak{Cons}\ (a, \mathit{list}\ a)$$

Lists generalise stacks, sequences of natural numbers, to an arbitrary element type. Likewise, the function

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

generalises *cat* (Example 4) 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 make any sense as $\mathsf{H} \mathbin{\dot\to} \mathsf{G}$ is not a functor. Also, lifting exponentials pointwise $\mathsf{G}^{\mathsf{H}}\,A = (\mathsf{G}\,A)^{\mathsf{H}\,A}$ does not work, because 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 $\mathbf{Set}^{\mathbb{C}}$ so that $\mathsf{G}^{\mathsf{H}} : \mathbb{C} \to \mathbf{Set}$. (The category $\mathbf{Set}^{\mathbb{C}^{op}}$ of contravariant, set-valued functors and natural transformations is known as the category of *pre-sheaves*.) We reason as follows:

$$\begin{array}{ll} & \mathsf{G}^{\mathsf{H}}\,A \\ \cong & \{\ \text{Yoneda Lemma (7)}\ \} \\ & \mathbb{C}(A, -) \mathbin{\dot\to} \mathsf{G}^{\mathsf{H}} \\ \cong & \{\ \text{requirement: } - \mathbin{\dot\times} \mathsf{H} \dashv (-)^{\mathsf{H}}\ \} \\ & \mathbb{C}(A, -) \mathbin{\dot\times} \mathsf{H} \mathbin{\dot\to} \mathsf{G} \end{array}$$

The derivation suggests that the exponential of H and G is given by $\mathsf{G}^{\mathsf{H}}\,A = \mathbb{C}(A, -) \mathbin{\dot\times} \mathsf{H} \mathbin{\dot\to} \mathsf{G}$. 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 last step. We postpone a proof until Section 5.9, where we establish a more general result abstracting away from **Set**.

Definition 4. The definition of exponentials goes beyond Haskell 2010 [51], as it requires rank-2 types (the data constructor *Exp* has a rank-2 type).

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

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

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

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

Example 17. Continuing Example 16, we may conclude that the defining equation of *append* has a unique solution. Its transpose of type $\mathsf{List} \dot{\to} \mathsf{List}^{\mathsf{List}}$ is interesting as it combines *append* with *fmap*:

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

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

5.4. *A special case: simultaneous recursion:* $- \times \mu\mathsf{G} \dashv (-)^{\mu\mathsf{G}}$

Often a function recurses on two arguments simultaneously.

Example 18. The function *add*

$$\begin{aligned}
&\mathit{add} : (\mu\mathfrak{Stack}, \mu\mathfrak{Stack}) && \to \mathit{Stack}\\
&\mathit{add}\quad (\mathit{In}\,\mathfrak{Empty}, \mathit{ns}) && = \mathit{Empty}\\
&\mathit{add}\quad (\mathit{ms}, \mathit{In}\,\mathfrak{Empty}) && = \mathit{Empty}\\
&\mathit{add}\quad (\mathit{In}\,(\mathfrak{Push}\,(m, \mathit{ms})), \mathit{In}\,(\mathfrak{Push}\,(n, \mathit{ns}))) && = \mathit{Push}\,(m+n, \mathit{add}\,(\mathit{ms}, \mathit{ns}))
\end{aligned}$$

zips two stacks adding corresponding elements. □

The definition of *add* has the form of an adjoint equation: $x \cdot (\times)\,\mathit{in} = \Psi\,x$ (here *in* is the initial algebra of the product category $\mathbb{C} \times \mathbb{C}$). Unfortunately, the product functor $\times$ is *not* a left adjoint—it is a right adjoint (Section 5.5). So we have to start afresh. Abstracting away from the particulars of the motivating example, an equation for *simultaneous recursion* in the unknown $x : \mathbb{C}(\mu\mathsf{F} \times \mu\mathsf{G}, A)$ has the form

$$x \cdot (\mathit{in}_{\mathsf{F}} \times \mathit{in}_{\mathsf{G}}) = \Psi\,x, \tag{18}$$

where the base function Ψ has type

$$\Psi : \forall X, Y\,.\,\mathbb{C}(X \times Y, A) \to \mathbb{C}(\mathsf{F}\,X \times \mathsf{G}\,Y, A).$$

The two arguments of x are destructed in lock-step. Now, to ensure uniqueness of solutions it should be sufficient to focus on one argument and indeed:

$$\begin{aligned}
&\quad x \cdot (\mathit{in}_{\mathsf{F}} \times \mathit{in}_{\mathsf{G}}) = \Psi\,x\\
\iff &\quad \{\ \times \text{ functor and } \mathit{in}_{\mathsf{G}} \text{ isomorphism}\ \}\\
&\quad x \cdot (\mathit{in}_{\mathsf{F}} \times \mathit{id}) = \Psi\,x \cdot (\mathit{id} \times \mathit{in}^{\circ}_{\mathsf{G}}).
\end{aligned}$$

We obtain an equation of the form discussed in the previous section with the left adjoint $\mathsf{L} = - \times \mu\mathsf{G}$. It remains to check that the derived base function $\Psi'\,X\,x = \Psi\,\langle X,\ \mu\mathsf{G}\rangle\,x \cdot (\mathit{id} \times \mathit{in}^{\circ}_{\mathsf{G}})$ has the required naturality property:

$$\begin{aligned}
&\quad \Psi'\,X_1\,f \cdot (\mathsf{F}\,h \times \mathit{id})\\
= &\quad \{\ \text{definition of } \Psi'\ \}\\
&\quad \Psi\,\langle X_1,\ \mu\mathsf{G}\rangle\,f \cdot (\mathit{id} \times \mathit{in}^{\circ}_{\mathsf{G}}) \cdot (\mathsf{F}\,h \times \mathit{id})\\
= &\quad \{\ \times \text{ functor and identity}\ \}\\
&\quad \Psi\,\langle X_1,\ \mu\mathsf{G}\rangle\,f \cdot (\mathsf{F}\,h \times \mathsf{G}\,\mathit{id}) \cdot (\mathit{id} \times \mathit{in}^{\circ}_{\mathsf{G}})\\
= &\quad \{\ \Psi \text{ is natural: } \Psi\,\langle X_1,\ Y_1\rangle\,f \cdot (\mathsf{F}\,h \times \mathsf{G}\,k) = \Psi\,\langle X_2,\ Y_2\rangle\,(f \cdot (h \times k))\ \}\\
&\quad \Psi\,\langle X_2,\ \mu\mathsf{G}\rangle\,(f \cdot (h \times \mathit{id})) \cdot (\mathit{id} \times \mathit{in}^{\circ}_{\mathsf{G}})\\
= &\quad \{\ \text{definition of } \Psi'\ \}\\
&\quad \Psi'\,X_2\,(f \cdot (h \times \mathit{id})).
\end{aligned}$$

We have reduced the symmetric equation for simultaneous recursion (18) to an asymmetric adjoint fixed-point equation. Consequently, the arrow given by (18) and hence *add* are both well-defined.

5.5. Mutual value recursion: $(+) \dashv \Delta \dashv (\times)$

The functions *nats* and *squares* introduced in Example 10 are defined by mutual recursion. The program is similar to Example 4, which defines *flattena* and *flattens*, with the notable difference that only one datatype is involved, rather than a pair of mutually recursive datatypes. Nonetheless, the correspondence suggests to view *nats* and *squares* as a single arrow in a product category.

$$numbers : \langle Nat, Nat \rangle \to \Delta(\nu\mathfrak{Sequ})$$

Here $\Delta : \mathbb{C} \to \mathbb{C} \times \mathbb{C}$ is the *diagonal functor* defined by $\Delta A = \langle A, A \rangle$ and $\Delta f = \langle f, f \rangle$. According to the type, *numbers* is an adjoint unfold, provided the diagonal functor has a left adjoint. It turns out that Δ has both a left and a right adjoint. We discuss the left adjoint first.

The left adjoint of the diagonal functor is the *coproduct*.

$$\phi : \forall A, B \,.\, \mathbb{C}((+)\,A, B) \cong (\mathbb{C} \times \mathbb{C})(A, \Delta B)$$

Note that B is an object of $\mathbb{C}$ and A is an object of $\mathbb{C} \times \mathbb{C}$, that is, a pair of objects. Unrolling the definition of arrows in $\mathbb{C} \times \mathbb{C}$ we have

$$\phi : \forall A, B \,.\, \mathbb{C}(A_1 + A_2, B) \cong \mathbb{C}(A_1, B) \times \mathbb{C}(A_2, B).$$

The adjunction captures the observation that we can represent a pair of arrows to the same codomain by a single arrow from the coproduct of the domains. The adjuncts are given by ($\triangledown$ is case analysis)

$$\phi f = \langle f \cdot inl,\, f \cdot inr \rangle \quad \text{and} \quad \phi^\circ \langle g_1, g_2 \rangle = g_1 \triangledown g_2.$$

The reader is invited to verify that the two functions are indeed inverses.

Using a similar reasoning as in Section 3.3, we can unfold the adjoint final fixed-point equation specialised to the diagonal functor:

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

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 3.3, except that now the destructor *out* is the same in both equations.

Example 19. Continuing Example 10, the base functions of *nats* and *squares* are given by

$$\begin{array}{l}
\mathfrak{nats} : \forall x \,.\, (Nat \to x, Nat \to x) \to (Nat \to \mathfrak{Sequ}\, x)\\
\mathfrak{nats}\ (nats, squares)\ n = \mathfrak{Next}\,(n, squares\, n)\\
\mathfrak{squares} : \forall x \,.\, (Nat \to x, Nat \to x) \to (Nat \to \mathfrak{Sequ}\, x)\\
\mathfrak{squares}\ (nats, squares)\ n = \mathfrak{Next}\,(n * n, nats\,(n+1)).
\end{array}$$

The recursion equations

$$\begin{array}{l}
nats : Nat \to \nu\mathfrak{Sequ}\\
nats\ n = Out^\circ\,(\mathfrak{nats}\,(nats, squares)\, n)\\
squares : Nat \to \nu\mathfrak{Sequ}\\
squares\ n = Out^\circ\,(\mathfrak{squares}\,(nats, squares)\, n)
\end{array}$$

exactly fit the pattern above (if we move Out° to the left-hand side). Hence, both functions are uniquely defined. Their transpose, $\phi^\circ \langle nats, squares \rangle$, combines the two functions into a single one using a coproduct.

$$\begin{array}{l}
numbers : \mathsf{Either}\, Nat\, Nat \to \nu\mathfrak{Sequ}\\
numbers\ (Left\ n) = Out^\circ\,(\mathfrak{Next}\,(n, numbers\,(Right\, n)))\\
numbers\ (Right\, n) = Out^\circ\,(\mathfrak{Next}\,(n * n, numbers\,(Left\,(n+1))))
\end{array}$$

The predefined datatype Either given by **data** $\mathsf{Either}\, a\, b = Left\ a \mid Right\ b$ 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*.

$$\phi : \forall A, B \,.\, (\mathbb{C} \times \mathbb{C})(\Delta A, B) \cong \mathbb{C}(A, (\times)\, B)$$

Unrolling the definition of $\mathbb{C} \times \mathbb{C}$, we have

$$\phi : \forall A, B \,.\, \mathbb{C}(A, B_1) \times \mathbb{C}(A, B_2) \cong \mathbb{C}(A, B_1 \times B_2).$$

We can represent a pair of arrows with the same domain by a single arrow to the product of the codomains. The bijection is witnessed by ($\vartriangle$ is pairing)

$$\phi \langle f_1, f_2 \rangle = f_1 \vartriangle f_2 \quad \text{and} \quad \phi^\circ g = \langle outl \cdot g,\, outr \cdot g \rangle.$$

Specialising the adjoint initial fixed-point equation yields

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

To deal with mutually recursive functions Fokkinga [21] introduced the concept of a *mutumorphism*: a pair of functions $x_1 : \mathbb{C}(\mu\mathsf{F}, A_1)$ and $x_2 : \mathbb{C}(\mu\mathsf{F}, A_2)$ satisfying the equations

$$x_1 \cdot in = a_1 \cdot \mathsf{F}\,(x_1 \vartriangle x_2) \quad\text{and}\quad x_2 \cdot in = a_2 \cdot \mathsf{F}\,(x_1 \vartriangle x_2),$$

where $a_1 : \mathbb{C}(\mathsf{F}\,(A_1 \times A_2), A_1)$ and $a_2 : \mathbb{C}(\mathsf{F}\,(A_1 \times A_2), A_2)$ serve as 'algebras'. Fokkinga shows that the two equations are equivalent to the single equation $x_1 \vartriangle x_2 = (\!|a_1 \vartriangle a_2|\!)$. Since adjoint folds and mutumorphism both solve the problem of giving a semantics to functions defined by mutual recursion, they are surely related. Let us calculate.

$$\begin{aligned}
&x_1 \cdot in = a_1 \cdot \mathsf{F}\,(x_1 \vartriangle x_2) \quad\text{and}\quad x_2 \cdot in = a_2 \cdot \mathsf{F}\,(x_1 \vartriangle x_2)\\
\Longleftrightarrow\quad&\{\text{ equality of arrows and definition of composition in } \mathbb{C}\times\mathbb{C}\ \}\\
&\langle x_1, x_2\rangle \cdot \langle in, in\rangle = \langle a_1, a_2\rangle \cdot \langle \mathsf{F}\,(x_1 \vartriangle x_2), \mathsf{F}\,(x_1 \vartriangle x_2)\rangle\\
\Longleftrightarrow\quad&\{\text{ definition of } \Delta \text{ and definition of } \phi\ \}\\
&\langle x_1, x_2\rangle \cdot \Delta in = \langle a_1, a_2\rangle \cdot \Delta(\mathsf{F}\,(\phi\,\langle x_1, x_2\rangle))\\
\Longleftrightarrow\quad&\{\text{ set } x = \langle x_1, x_2\rangle \text{ and } a = \langle a_1, a_2\rangle\ \}\\
&x \cdot \Delta in = a \cdot \Delta(\mathsf{F}\,(\phi\,x))
\end{aligned}$$

Voilá. We obtain an adjoint fold with $\Psi\,x = a \cdot \Delta(\mathsf{F}\,(\phi\,x))$. It is actually an algebraic adjoint fold in disguise (Section 4.3)—we only have to massage the right-hand side. Since the steps are not specific to the adjunction at hand, we abstract away from Δ and '$\times$'. Let $x : \mathbb{C}(\mathsf{L}\,X, A)$, then

$$\begin{aligned}
&a \cdot \mathsf{L}\,(\mathsf{F}\,(\phi\,x))\\
=\quad&\{\text{ adjunction: } \phi\,(x : \mathbb{C}(\mathsf{L}\,A, B)) = \mathsf{R}\,x \cdot \eta\,A\ \}\\
&a \cdot \mathsf{L}\,(\mathsf{F}\,(\mathsf{R}\,x \cdot \eta\,X))\\
=\quad&\{\ \mathsf{L} \text{ and } \mathsf{F} \text{ functors and definition of horizontal composition }\}\\
&a \cdot (\mathsf{L}\circ\mathsf{F}\circ\mathsf{R})\,x \cdot (\mathsf{L}\circ\mathsf{F}\circ\eta)\,X.
\end{aligned}$$

Voilá again. An algebraic fold in canonical form emerges with $\mathsf{G} = \mathsf{L}\circ\mathsf{F}\circ\mathsf{R}$ and $\alpha = \mathsf{L}\circ\mathsf{F}\circ\eta$. Note that a is indeed an algebra, an algebra for the functor G! It is instructive to replay Fokkinga's proof of uniqueness in the abstract setting, reducing the algebraic adjoint fold to a standard fold.

$$\begin{aligned}
&x \cdot \mathsf{L}\,in = a \cdot \mathsf{L}\,(\mathsf{F}\,(\phi\,x))\\
\Longleftrightarrow\quad&\{\text{ adjunction: } \phi \cdot \phi^\circ = id \text{ and } \phi^\circ \cdot \phi = id\ \}\\
&\phi\,(x \cdot \mathsf{L}\,in) = \phi\,(a \cdot \mathsf{L}\,(\mathsf{F}\,(\phi\,x)))\\
\Longleftrightarrow\quad&\{\ \phi \text{ is natural (14) }\}\\
&\phi\,x \cdot in = \phi\,a \cdot \mathsf{F}\,(\phi\,x)\\
\Longleftrightarrow\quad&\{\text{ uniqueness property of standard folds (2) }\}\\
&\phi\,x = (\!|\phi\,a|\!)
\end{aligned}$$

We obtain an attractive formula, Fokkinga's mutu-Charn law in the abstract.

Fokkinga also observes that *paramorphisms* [54] can be seen as a special case of mutumorphisms.

Example 20. We can use mutual value recursion to fit the definition of factorial (Example 3) into the framework. The definition of *fac* has the form of a *paramorphism*, 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.

$$\begin{array}{lll}
fac : \mu\mathfrak{Nat} & \to & Nat\\
fac\ \ (In\,\mathfrak{Z}) & = & 1\\
fac\ \ (In\,(\mathfrak{S}\,n)) & = & In\,(\mathfrak{S}\,(id\,n)) * fac\,n
\end{array}
\qquad
\begin{array}{lll}
id : \mu\mathfrak{Nat} & \to & Nat\\
id\ \ (In\,\mathfrak{Z}) & = & In\,\mathfrak{Z}\\
id\ \ (In\,(\mathfrak{S}\,n)) & = & In\,(\mathfrak{S}\,(id\,n))
\end{array}$$

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

$$\begin{array}{llll}
\mathfrak{fac} : \forall x\,.\,(x \to Nat, x \to Nat) \to (\mathfrak{Nat}\,x \to Nat) & & &\\
\mathfrak{fac}\quad (fac,\quad id) & (\mathfrak{Z}) & = & 1\\
\mathfrak{fac}\quad (fac,\quad id) & (\mathfrak{S}\,n) & = & In\,(\mathfrak{S}\,(id\,n)) * fac\,n\\
\mathfrak{id} : \forall x\,.\,(x \to Nat, x \to Nat) \to (\mathfrak{Nat}\,x \to Nat) & & &\\
\mathfrak{id}\quad (fac,\quad id) & (\mathfrak{Z}) & = & In\,\mathfrak{Z}\\
\mathfrak{id}\quad (fac,\quad id) & (\mathfrak{S}\,n) & = & In\,(\mathfrak{S}\,(id\,n))
\end{array}$$

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. □

Example 21. Incidentally, we can employ a similar approach to also fit the Fibonacci function into the framework.

$$\begin{aligned}
&fib : Nat && \to Nat\\
&fib\ \ (Z) && = Z\\
&fib\ \ (S\,Z) && = S\,Z\\
&fib\ \ (S\,(S\,n)) && = fib\,n + fib\,(S\,n)
\end{aligned}$$

The definition is sometimes characterised as a *histomorphism* [69] because in the third equation *fib* depends on two previous values, rather than only one. Setting $fib'\ n = fib\ (S\,n)$, we can transform the nested recursion into a mutual recursion. Indeed, this is the usual approach taken when defining the stream of Fibonacci numbers, see, for example, [35].

$$\begin{aligned}
&fib : Nat && \to Nat && \qquad fib' : Nat && \to Nat\\
&fib\ \ (Z) && = Z && \qquad fib'\ \ (Z) && = S\,Z\\
&fib\ \ (S\,n) && = fib'\,n && \qquad fib'\ \ (S\,n) && = fib\,n + fib'\,n
\end{aligned}$$

We leave the details to the reader and only remark that the transposed fold corresponds to the usual linear-time implementation of Fibonacci, called *twofib* in [8]. □

The diagram below illustrates the double adjunction $(+) \dashv \Delta \dashv (\times)$.

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

Each double adjunction gives rise to four different schemes and transformations: two for initial and two for final fixed-point equations. We have discussed $(+) \dashv \Delta$ for unfolds and $\Delta \dashv (\times)$ for folds. Their 'inverses' are less useful: using $(+) \dashv \Delta$ we can transform an adjoint *fold* that works on a coproduct of mutually recursive datatypes into a standard fold over a product category (see Section 3.3). Dually, $\Delta \dashv (\times)$ enables us to transform an adjoint *unfold* that yields a product of mutually recursive datatypes into a standard unfold over a product category.

5.6. *Mutual value recursion: $\Sigma\, i \in \mathbb{I} \dashv \Delta \dashv \Pi\, i \in \mathbb{I}$*

In the previous section we have considered *two* functions defined by mutual recursion. It is straightforward to generalise the development to n mutually recursive functions (or, indeed, to an infinite number of functions). Likewise, we can generalise the Fibonacci example to histomorphisms that depend on a *fixed* number of previous values. The same reasoning applies to their duals, so-called *futumorphisms* [69].

Central to the previous undertaking was the notion of a product category. Now, the product category $\mathbb{C} \times \mathbb{C}$ can be regarded as a simple functor category: $\mathbb{C}^2$, where 2 is some two-element set. To be able to deal with an arbitrary number of functions we simply generalise from 2 to an arbitrary index set.

Recall that a set forms a discrete category (Section 3.5). The diagonal functor $\Delta : \mathbb{C} \to \mathbb{C}^{\mathbb{I}}$ now sends each index to the same object: $(\Delta A)_i = A$. Left and right adjoints of the diagonal functor generalise the constructions of the previous section. The left adjoint of the diagonal functor is a simple form of a *dependent sum* (also called a dependent product).

$$\forall A, B\ .\ \mathbb{C}(\Sigma\, i \in \mathbb{I}\ .\ A_i, B) \cong \mathbb{C}^{\mathbb{I}}(A, \Delta B)$$

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

$$\forall A, B\ .\ \mathbb{C}^{\mathbb{I}}(\Delta A, B) \cong \mathbb{C}(A, \Pi\, i \in \mathbb{I}\ .\ B_i)$$

The following diagram summarises the type information.

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

It is worth singling out a special case of the construction that we shall need later on. First of all, note that $\mathbb{C}^{\mathbb{I}}(\Delta X, \Delta Y) \cong \mathbb{I} \to \mathbb{C}(X, Y)$. 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:

$$\forall X, Y\ .\ \mathbb{C}(\Sigma\,\mathbb{I}\ .\ X, Y) \cong \mathbb{I} \to \mathbb{C}(X, Y) \cong \mathbb{C}(X, \Pi\,\mathbb{I}\ .\ Y). \tag{19}$$

Alternatively, we obtain the derived adjunction $\Sigma\,\mathbb{I} \dashv \Pi\,\mathbb{I}$ as the composition $(\Sigma\, i \in \mathbb{I}) \circ \Delta \dashv (\Pi\, i \in \mathbb{I}) \circ \Delta$ of $\Sigma\, i \in \mathbb{I} \dashv \Delta$ with $\Delta \dashv \Pi\, i \in \mathbb{I}$ (Section 6.1). The degenerated sum $\Sigma\,\mathbb{I}\ .\ A$ is also called a *copower*, sometimes written $\mathbb{I} \bullet A$. The degenerated product $\Pi\,\mathbb{I}\ .\ A$ is also called a *power*, sometimes written $A^{\mathbb{I}}$. In **Set**, we have $\Sigma\,\mathbb{I}\ .\ A = \mathbb{I} \times A$ and $\Pi\,\mathbb{I}\ .\ A = \mathbb{I} \to A$. (Hence, $\Sigma\,\mathbb{I} \dashv \Pi\,\mathbb{I}$ is essentially a variant of currying).

5.7. *Type application:* $\mathsf{Lsh}_X \dashv (-X) \dashv \mathsf{Rsh}_X$

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 22. The function *suml* defined

$$\begin{aligned} &suml : \mu\mathfrak{List}\,Nat && \to Nat \\ &suml \quad (In\,\mathfrak{Nil}) && = 0 \\ &suml \quad (In\,(\mathfrak{Cons}\,(a, as))) && = a + suml\,as \end{aligned}$$

sums a list of natural numbers. It is the adaptation of *total* (Example 1) to the type of parametric lists (we relate *total* and *suml* in Example 33). □

The definition of *suml* looks suspiciously like a fold, but it is not, as it does not have the right type. The corresponding function on perfect trees does not even resemble a fold.

Example 23. The function *sump* sums a perfect tree of naturals.

$$\begin{aligned} &sump : \mu\mathfrak{Perfect}\,Nat \to Nat \\ &sump \quad (In\,(\mathfrak{Zero}\,n)) = n \\ &sump \quad (In\,(\mathfrak{Succ}\,p)) = sump\,(fmap\,plus\,p) \quad \textbf{where} \quad plus\,(a, b) = a + b \end{aligned}$$

The recursive call of *sump* is not applied to a subterm of $\mathfrak{Succ}\,p$. In fact, it cannot, as p has type $\mathsf{Perfect}\,(Nat, Nat)$, not $\mathsf{Perfect}\,Nat$. As an aside, this definition requires the functor instance for μ (Definition 3). □

Perhaps surprisingly, the definitions above fit into the framework of adjoint fixed-point equations. We simply have to view type application as a functor: given $X : \mathbb{D}$ define $\mathsf{App}_X : \mathbb{C}^{\mathbb{D}} \to \mathbb{C}$ by $\mathsf{App}_X\,\mathsf{F} = \mathsf{F}\,X$ and $\mathsf{App}_X\,\alpha = \alpha\,X$. (The natural transformation α is applied to the object X. In Haskell this type application is invisible, which is why we cannot see that *suml* is not a standard fold.) It is easy to show that this data defines a functor: $\mathsf{App}_X\,id = id\,X = id_X$ and $\mathsf{App}_X\,(\alpha \cdot \beta) = (\alpha \cdot \beta)\,X = \alpha\,X \cdot \beta\,X = \mathsf{App}_X\,\alpha \cdot \mathsf{App}_X\,\beta$. Using App_X we can assign *suml* the type $\mathsf{App}_{Nat}\,(\mu\mathfrak{List}) \to Nat$. All that is left to do is to check whether App_X is part of an adjunction. It turns out that under some mild conditions (existence of copowers and powers) App_X has both a left and a right adjoint. We choose to derive the left adjoint.

$$\begin{aligned} & \mathbb{C}(A, \mathsf{App}_X\,\mathsf{B}) \\ \cong\ & \{ \text{definition of } \mathsf{App}_X \} \\ & \mathbb{C}(A, \mathsf{B}\,X) \\ \cong\ & \{ \text{Yoneda Lemma (10)} \} \\ & \forall Y : \mathbb{D} \,.\, \mathbb{D}(X, Y) \to \mathbb{C}(A, \mathsf{B}\,Y) \\ \cong\ & \{ \text{copower (19)}: \mathbb{C}(\Sigma\,\mathbb{I} \,.\, X, Y) \cong \mathbb{I} \to \mathbb{C}(X, Y) \} \\ & \forall Y : \mathbb{D} \,.\, \mathbb{C}(\Sigma\,\mathbb{D}(X, Y) \,.\, A, \mathsf{B}\,Y) \\ \cong\ & \{ \text{define } \mathsf{Lsh}_X\,A = \Lambda\,Y : \mathbb{D} \,.\, \Sigma\,\mathbb{D}(X, Y) \,.\, A \} \\ & \forall Y : \mathbb{D} \,.\, \mathbb{C}(\mathsf{Lsh}_X\,A\,Y, \mathsf{B}\,Y) \\ \cong\ & \{ \text{natural transformations} \} \\ & \mathsf{Lsh}_X\,A \mathrel{\dot{\to}} \mathsf{B} \end{aligned}$$

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 $\mathsf{Rsh}_X\,B = \Lambda\,Y : \mathbb{D} \,.\, \Pi\,\mathbb{D}(Y, X) \,.\, B$, the *right shift* of X. The following diagram summarises the type information.

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

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

Definition 5. The functors Lsh and Rsh can be defined as follows.

$$\begin{aligned} &\textbf{newtype}\ \mathsf{Lsh}_x\,a\,y = Lsh\,(x \to y, a) \\ &\textbf{instance}\ Functor\ (\mathsf{Lsh}_x\,a)\ \textbf{where} \\ &\quad fmap\,f\ (Lsh\,(k, a)) = Lsh\,(f \cdot k, a) \\ &\textbf{newtype}\ \mathsf{Rsh}_x\,b\,y = Rsh\,\{rsh^{\circ} : (y \to x) \to b\} \\ &\textbf{instance}\ Functor\ (\mathsf{Rsh}_x\,b)\ \textbf{where} \\ &\quad fmap\,f\ (Rsh\,g) \quad = Rsh\,(\lambda k \to g\,(k \cdot f)) \end{aligned}$$

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 necessarily 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). The adjuncts are defined

$$\begin{aligned}
&\phi_{\mathsf{Lsh}} &&: \forall x\,a\,b\,.\,(\forall y\,.\,\mathsf{Lsh}_x\,a\,y \to b\,y) \to (a \to b\,x)\\
&\phi_{\mathsf{Lsh}}\,\alpha &&= \lambda s \to \alpha\,(Lsh\,(id, s))\\
&\phi^{\circ}_{\mathsf{Lsh}} &&: \forall x\,a\,b\,.\,(Functor\,b) \Rightarrow (a \to b\,x) \to (\forall y\,.\,\mathsf{Lsh}_x\,a\,y \to b\,y)\\
&\phi^{\circ}_{\mathsf{Lsh}}\,g &&= \lambda(Lsh\,(k, s)) \to fmap\,k\,(g\,s)\\
&\phi_{\mathsf{Rsh}} &&: \forall x\,a\,b\,.\,(Functor\,a) \Rightarrow (a\,x \to b) \to (\forall y\,.\,a\,y \to \mathsf{Rsh}_x\,b\,y)\\
&\phi_{\mathsf{Rsh}}\,f &&= \lambda s \to Rsh\,(\lambda k \to f\,(fmap\,k\,s))\\
&\phi^{\circ}_{\mathsf{Rsh}} &&: \forall x\,a\,b\,.\,(\forall y\,.\,a\,y \to \mathsf{Rsh}_x\,b\,y) \to (a\,x \to b)\\
&\phi^{\circ}_{\mathsf{Rsh}}\,\beta &&= \lambda s \to rsh^{\circ}\,(\beta\,s)\,id.
\end{aligned}$$

Note that the adjuncts are also natural in x, the parameter of the adjunctions. □

As usual, let us specialise the adjoint equations.

$$\begin{aligned}
&x \cdot \mathsf{App}_X\,in = \Psi\,x &\qquad& \mathsf{App}_X\,out \cdot x = \Psi\,x\\
\iff\;&\{\text{ definition of } \mathsf{App}_X \} && \iff\;\{\text{ definition of } \mathsf{App}_X \}\\
&x \cdot in\,X = \Psi\,x && out\,X \cdot x = \Psi\,x
\end{aligned}$$

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

Example 24. Continuing Example 23, the base function of *sump* is

$$\begin{aligned}
&\mathfrak{sump} : \forall x\,.\,(Functor\,x) \Rightarrow (x\,Nat \to Nat) \to (\mathfrak{Perfect}\,x\,Nat \to Nat)\\
&\mathfrak{sump}\quad sump\quad (\mathfrak{Zero}\,n) \quad = n\\
&\mathfrak{sump}\quad sump\quad (\mathfrak{Succ}\,p) \quad = sump\,(fmap\,plus\,p).
\end{aligned}$$

The definition requires the $\mathfrak{Perfect}\,x$ functor instance, which in turn induces the *Functor x* context. The transpose of *sump* is a fold that returns a higher-order function: $sump' : \mathsf{Perfect} \mathrel{\dot{\to}} \mathsf{Rsh}_{Nat}\,Nat$. Let us derive its definition by simplifying the base function $\phi_{\mathsf{Rsh}} \cdot \mathfrak{sump} \cdot \phi^{\circ}_{\mathsf{Rsh}}$. **Case** $\mathfrak{Zero}\,n$**:**

$$\begin{aligned}
&\phi_{\mathsf{Rsh}}\,(\mathfrak{sump}\,(\phi^{\circ}_{\mathsf{Rsh}}\,sump'))\,(\mathfrak{Zero}\,n)\\
=\;&\{\text{ definition of } \phi_{\mathsf{Rsh}} \text{ (Definition 5) } \}\\
&\lambda\,k\,.\,\mathfrak{sump}\,(\phi^{\circ}_{\mathsf{Rsh}}\,sump')\,(\mathfrak{Perfect}\,k\,(\mathfrak{Zero}\,n))\\
=\;&\{\text{ definition of } \mathfrak{Perfect} \text{ (Example 7) and definition of } \mathfrak{sump} \}\\
&\lambda\,k\,.\,k\,n
\end{aligned}$$

For clarity, we have omitted the constructor *Rsh* and the destructor rsh°. **Case** $\mathfrak{Succ}\,p$**:**

$$\begin{aligned}
&\phi_{\mathsf{Rsh}}\,(\mathfrak{sump}\,(\phi^{\circ}_{\mathsf{Rsh}}\,sump'))\,(\mathfrak{Succ}\,p)\\
=\;&\{\text{ definition of } \phi_{\mathsf{Rsh}} \text{ (Definition 5) } \}\\
&\lambda\,k\,.\,\mathfrak{sump}\,(\phi^{\circ}_{\mathsf{Rsh}}\,sump')\,(\mathfrak{Perfect}\,k\,(\mathfrak{Succ}\,p))\\
=\;&\{\text{ definition of } \mathfrak{Perfect} \text{ (Example 7) and definition of } \mathfrak{sump} \}\\
&\lambda\,k\,.\,\phi^{\circ}_{\mathsf{Rsh}}\,sump'\,(\mathsf{Perfect}\,plus\,(\mathsf{Perfect}\,(k \times k)\,p))\\
=\;&\{\text{ Perfect functor } \}\\
&\lambda\,k\,.\,\phi^{\circ}_{\mathsf{Rsh}}\,sump'\,(\mathsf{Perfect}\,(plus \cdot (k \times k))\,p)\\
=\;&\{\text{ definition of } \phi^{\circ}_{\mathsf{Rsh}} \text{ (Definition 5) } \}\\
&\lambda\,k\,.\,sump'\,(\mathsf{Perfect}\,(plus \cdot (k \times k))\,p)\,id\\
=\;&\{\; sump' \text{ is natural: } sump'\,p\,(k \cdot h) = sump'\,(\mathsf{Perfect}\,h\,p)\,k \;\}\\
&\lambda\,k\,.\,sump'\,p\,(plus \cdot (k \times k))
\end{aligned}$$

The only non-trivial step is the last one where we use the fact that the argument $sump'$ itself is natural. Inlining the base function, we obtain

$$\begin{aligned}
&sump' : \forall x\,.\,\mathsf{Perfect}\,x \to (x \to Nat) \to Nat\\
&sump'\quad (Zero\,n) \;=\; \lambda k \;\to k\,n\\
&sump'\quad (Succ\,p) \;=\; \lambda k \;\to sump'\,p\,(plus \cdot (k \times k)).
\end{aligned}$$

Quite interestingly, the transformation turns a *generalised fold* in the sense of Bird and Paterson [11] into an *efficient generalised fold* in the sense of Hinze [30]. Both versions have a linear running time, but $sump'$ avoids the repeated invocations of the mapping function (*fmap plus*). □

5.8. *Type composition:* $\mathsf{Lan_J} \dashv (- \circ \mathsf{J}) \dashv \mathsf{Ran_J}$

Yes, we can.
Concession speech in the New Hampshire presidential primary—Barack Obama

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

Example 25. The function *concat* defined

$$\begin{aligned}
&\mathit{concat} : \forall a\,.\ \mu\mathfrak{List}\,(\mathsf{List}\,a) && \to \mathsf{List}\,a\\
&\mathit{concat}\quad (\mathit{In}\,\mathfrak{Nil}) && = \mathit{In}\,\mathfrak{Nil}\\
&\mathit{concat}\quad (\mathit{In}\,(\mathfrak{Cons}\,(l, ls))) && = \mathit{append}\,(l, \mathit{concat}\,ls)
\end{aligned}$$

generalises the binary function *append* to a list of lists. □

The definition has the structure of a standard fold, but again the type is not quite right: we need a natural transformation of type $\mu\mathfrak{List} \dot{\to} \mathsf{G}$, but *concat* has type $\mu\mathfrak{List} \circ \mathsf{List} \dot{\to} \mathsf{List}$. 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. To this end, we view pre-composition as a functor: $(- \circ \mathsf{List})\ (\mu\mathfrak{List}) \dot{\to} \mathsf{List}$. We interpret $\mathsf{List} \circ \mathsf{List}$ as $(- \circ \mathsf{List})\ \mathsf{List}$ rather than $(\mathsf{List} \circ -)\ \mathsf{List}$ because the outer list, written $\mu\mathfrak{List}$ for emphasis, drives the recursion.

Given a functor $\mathsf{J} : \mathbb{C} \to \mathbb{D}$, define the higher-order functor $\mathsf{Pre_J} : \mathbb{E}^{\mathbb{D}} \to \mathbb{E}^{\mathbb{C}}$ by $\mathsf{Pre_J}\,\mathsf{F} = \mathsf{F} \circ \mathsf{J}$ and $\mathsf{Pre_J}\,\alpha = \alpha \circ \mathsf{J}$, where the horizontal composition of a natural transformation and a functor is defined $(\alpha \circ \mathsf{J})\,X = \alpha\,(\mathsf{J}\,X)$. (In Haskell, this composition is invisible. Again, this is why the definition of *concat* looks like a fold, but it is not.) As usual, we should make sure that the data actually defines a functor: $\mathsf{Pre_J}\,id_{\mathsf{F}} = id_{\mathsf{F}} \circ \mathsf{J} = id_{\mathsf{F} \circ \mathsf{J}}$ and $\mathsf{Pre_J}\,(\alpha \cdot \beta) = (\alpha \cdot \beta) \circ \mathsf{J} = (\alpha \circ \mathsf{J}) \cdot (\beta \circ \mathsf{J}) = \mathsf{Pre_J}\,\alpha \cdot \mathsf{Pre_J}\,\beta$. (The calculations make use of Godement's rules [6], which relate different types of composites.) Using the higher-order functor we can assign *concat* the type $\mathsf{Pre_{List}}\,(\mu\mathfrak{List}) \dot{\to} \mathsf{List}$. 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 $\mathsf{Pre_J}$ has both a left and a right adjoint. For variety, we derive the latter.

$$\begin{aligned}
&\quad \mathsf{F} \circ \mathsf{J} \dot{\to} \mathsf{G}\\
\cong\ &\quad \{\text{ natural transformation as an end [49, p. 223] }\}\\
&\quad \forall Y : \mathbb{C}\,.\ \mathbb{E}(\mathsf{F}\,(\mathsf{J}\,Y), \mathsf{G}\,Y)\\
\cong\ &\quad \{\text{ Yoneda Lemma (6) }\}\\
&\quad \forall Y : \mathbb{C}\,.\ \forall X : \mathbb{D}\,.\ \mathbb{D}(X, \mathsf{J}\,Y) \to \mathbb{E}(\mathsf{F}\,X, \mathsf{G}\,Y)\\
\cong\ &\quad \{\text{ power (19): } \mathbb{I} \to \mathbb{C}(Y, B) \cong \mathbb{C}(Y, \Pi\,\mathbb{I}\,.\,B) \}\\
&\quad \forall Y : \mathbb{C}\,.\ \forall X : \mathbb{D}\,.\ \mathbb{E}(\mathsf{F}\,X, \Pi\,\mathbb{D}(X, \mathsf{J}\,Y)\ .\ \mathsf{G}\,Y)\\
\cong\ &\quad \{\text{ interchange of quantifiers [49, p. 231f] }\}\\
&\quad \forall X : \mathbb{D}\,.\ \forall Y : \mathbb{C}\,.\ \mathbb{E}(\mathsf{F}\,X, \Pi\,\mathbb{D}(X, \mathsf{J}\,Y)\ .\ \mathsf{G}\,Y)\\
\cong\ &\quad \{\text{ the hom-functor } \mathbb{E}(A, -) \text{ preserves ends [49, p. 225] }\}\\
&\quad \forall X : \mathbb{D}\,.\ \mathbb{E}(\mathsf{F}\,X, \forall Y : \mathbb{C}\,.\ \Pi\,\mathbb{D}(X, \mathsf{J}\,Y)\ .\ \mathsf{G}\,Y)\\
\cong\ &\quad \{\text{ define } \mathsf{Ran_J}\,\mathsf{G} = \Lambda\,X : \mathbb{D}\,.\ \forall Y : \mathbb{C}\,.\ \Pi\,\mathbb{D}(X, \mathsf{J}\,Y)\ .\ \mathsf{G}\,Y \}\\
&\quad \forall X : \mathbb{D}\,.\ \mathbb{E}(\mathsf{F}\,X, \mathsf{Ran_J}\,\mathsf{G}\,X)\\
\cong\ &\quad \{\text{ natural transformation as an end [49, p. 223] }\}\\
&\quad \mathsf{F} \dot{\to} \mathsf{Ran_J}\,\mathsf{G}
\end{aligned}$$

Since each step is natural in F and G, the composite isomorphism is also natural in F and G. The functor $\mathsf{Ran_J}\,\mathsf{G}$ is called the *right Kan extension* of G along J. (If we view $\mathsf{J} : \mathbb{C} \to \mathbb{D}$ as an inclusion functor, then $\mathsf{Ran_J}\,\mathsf{G} : \mathbb{D} \to \mathbb{E}$ extends $\mathsf{G} : \mathbb{C} \to \mathbb{E}$ to the whole of $\mathbb{D}$.) The universally quantified object in the definition of $\mathsf{Ran_J}$ 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 notation above, in particular, as it blends with the notation for natural transformations. And indeed, natural transformations are an example of an end: $\mathbb{D}^{\mathbb{C}}(\mathsf{F}, \mathsf{G}) = \forall X : \mathbb{C}\,.\ \mathbb{D}(\mathsf{F}\,X, \mathsf{G}\,X)$. We refer the interested reader to [49] for further details.

Dually, the left adjoint of $\mathsf{Pre_J}$ is called the *left Kan extension* and is defined $\mathsf{Lan_J}\,\mathsf{F} = \Lambda\,X : \mathbb{D}\,.\ \exists Y : \mathbb{C}\,.\ \Sigma\,\mathbb{D}(\mathsf{J}\,Y, X)\ .\ \mathsf{F}\,Y$. 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.

$$\begin{tikzcd}
 & \mathbb{C} \arrow[ld, "\mathsf{F}"'] \arrow[d, "\mathsf{J}"] \\
\mathbb{E} & \mathbb{D} \arrow[l, shift right, "\mathsf{G}"'] \arrow[l, shift left, "\mathsf{Lan_J\,F}"]
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathbb{E}^{\mathbb{D}} \arrow[r, bend right, "- \circ \mathsf{J}"'] \arrow[r, phantom, "\perp"] & \mathbb{E}^{\mathbb{C}} \arrow[l, bend right, "\mathsf{Lan_J}"'] \arrow[r, bend right, "\mathsf{Ran_J}"'] \arrow[r, phantom, "\perp"] & \mathbb{E}^{\mathbb{D}} \arrow[l, bend right, "- \circ \mathsf{J}"']
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathbb{C} \arrow[d, "\mathsf{J}"'] \arrow[rd, "\mathsf{G}"] & \\
\mathbb{D} \arrow[r, shift left, "\mathsf{F}"] \arrow[r, shift right, "\mathsf{Ran_J\,G}"'] & \mathbb{E}
\end{tikzcd}$$

Definition 6. Like Exp, the definition of the right Kan extension requires rank-2 types (the data constructor *Ran* has a rank-2 type).

$$\begin{array}{l}\textbf{newtype}\ \mathsf{Ran}_j\, g\, x = \mathit{Ran}\,\{\mathit{ran}^\circ : \forall a\,.\,(x \to j\,a) \to g\,a\}\\ \textbf{instance}\ \mathit{Functor}\ (\mathsf{Ran}_j\, g)\ \textbf{where}\\ \quad \mathit{fmap}\, f\ (\mathit{Ran}\, h) = \mathit{Ran}\,(\lambda k \to h\,(k \cdot f))\end{array}$$

The type $\mathsf{Ran}_j\, g$ can be seen as a *generalised continuation type*—often, but not necessarily the type constructors j and g are identical ($\mathsf{Ran}_\mathsf{J}\,\mathsf{J}$ is known as the *codensity monad*). 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

$$\begin{array}{lcl}\phi_{\mathsf{Ran}} & : & \forall j\, f\, g\,.\,(\mathit{Functor}\, f) \Rightarrow (\forall x\,.\,f\,(j\,x) \to g\,x) \to (\forall x\,.\,f\,x \to \mathsf{Ran}_j\, g\, x)\\ \phi_{\mathsf{Ran}}\,\alpha & = & \lambda s \to \mathit{Ran}\,(\lambda k \to \alpha\,(\mathit{fmap}\, k\, s))\\ \phi^\circ_{\mathsf{Ran}} & : & \forall j\, f\, g\,.\,(\forall x\,.\,f\,x \to \mathsf{Ran}_j\, g\, x) \to (\forall x\,.\,f\,(j\,x) \to g\,x)\\ \phi^\circ_{\mathsf{Ran}}\,\beta & = & \lambda s \to \mathit{ran}^\circ\,(\beta\, s)\,\mathit{id}.\end{array}$$

Note that the adjuncts are also natural in j, the parameter of the adjunction.

Turning to the definition of the left Kan extension we require another extension of the Haskell 2010 type system [51]: existential types.

$$\begin{array}{l}\textbf{data}\ \mathsf{Lan}_j\, f\, x = \forall a\,.\,\mathit{Lan}\,(j\,a \to x, f\,a)\\ \textbf{instance}\ \mathit{Functor}\ (\mathsf{Lan}_j\, f)\ \textbf{where}\\ \quad \mathit{fmap}\, f\ (\mathit{Lan}\,(k, s)) = \mathit{Lan}\,(f \cdot k, s).\end{array}$$

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. 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 datatype*: $f\,a$ 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

$$\begin{array}{lcl}\phi_{\mathsf{Lan}} & : & \forall j\, f\, g\,.\,(\forall x\,.\,\mathsf{Lan}_j\, f\, x \to g\,x) \to (\forall x\,.\,f\,x \to g\,(j\,x))\\ \phi_{\mathsf{Lan}}\,\alpha & = & \lambda s \to \alpha\,(\mathit{Lan}\,(\mathit{id}, s))\\ \phi^\circ_{\mathsf{Lan}} & : & \forall j\, f\, g\,.\,(\mathit{Functor}\, g) \Rightarrow (\forall x\,.\,f\,x \to g\,(j\,x)) \to (\forall x\,.\,\mathsf{Lan}_j\, f\, x \to g\,x)\\ \phi^\circ_{\mathsf{Lan}}\,\beta & = & \lambda(\mathit{Lan}\,(k, s)) \to \mathit{fmap}\, k\,(\beta\, s).\end{array}$$

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

As usual, let us specialise the adjoint equations.

$$\begin{array}{lll}& x \cdot \mathsf{Pre}_\mathsf{J}\,\mathit{in} = \Psi\, x & \\ \iff & \{\text{ definition of } \mathsf{Pre}_\mathsf{J}\ \} & \\ & x \cdot (\mathit{in} \circ \mathsf{J}) = \Psi\, x & \\ \iff & \{\text{ extensionality }\} & \\ & \forall A\,.\,\forall s\,.\,x\,A\,(\mathit{in}\,(\mathsf{J}\,A)\,s) = \Psi\, x\, A\, s &\end{array} \qquad \begin{array}{lll}& \mathsf{Pre}_\mathsf{J}\,\mathit{out} \cdot x = \Psi\, x & \\ \iff & \{\text{ definition of } \mathsf{Pre}_\mathsf{J}\ \} & \\ & (\mathit{out} \circ \mathsf{J}) \cdot x = \Psi\, x & \\ \iff & \{\text{ extensionality }\} & \\ & \forall A\,.\,\forall s\,.\,\mathit{out}\,(\mathsf{J}\,A)\,(x\,A\,s) = \Psi\, x\, A\, s &\end{array}$$

Note that '$\cdot$' in the original equations denotes the vertical composition of natural transformations: $(\alpha \cdot \beta)\,X = \alpha\,X \cdot \beta\,X$. Also note that the natural transformations x and in are applied to different types. 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 26. Continuing Example 25, the base function of *concat* is straightforward, except perhaps for the types.

$$\begin{array}{llll}\mathfrak{concat} : \forall x\,.\,(\forall a\,.\,x\,(\mathsf{List}\,a) \to \mathsf{List}\,a) \to (\forall a\,.\,\mathfrak{List}\,x\,(\mathsf{List}\,a) \to \mathsf{List}\,a) &&&\\ \mathfrak{concat} & \mathit{concat} & (\mathfrak{Nil}) & = \mathit{In}\,\mathfrak{Nil}\\ \mathfrak{concat} & \mathit{concat} & (\mathfrak{Cons}\,(l, ls)) & = \mathit{append}\,(l, \mathit{concat}\, ls)\end{array}$$

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

$$\mathit{concat}' : \mathsf{List} \mathrel{\dot{\to}} \mathsf{Ran}_{\mathsf{List}}\,\mathsf{List} \cong \forall a\,.\,\mathsf{List}\,a \to \forall b\,.\,(a \to \mathsf{List}\,b) \to \mathsf{List}\,b.$$

The type suggests that concat' is the bind of the list monad, written $\gg\!=$ in Haskell, and this is indeed the case!

$$\begin{array}{llll}\mathit{concat}' : \forall a\, b\,.\,\mu\mathfrak{List}\,a \to (a \to \mathsf{List}\,b) \to \mathsf{List}\,b &&&\\ \mathit{concat}' & \mathit{as} & = \lambda k & \to \mathit{concat}\,(\mathit{fmap}\, k\, \mathit{as})\end{array}$$

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

Given some type signature, it is not always straightforward to read off the pre-composed functor J as the following example demonstrates.

Example 27. The parametric datatype of streams

$$\textbf{data}\ \mathsf{Stream}\ a = \mathit{Link}\ (a, \mathsf{Stream}\ a)$$

generalises the type *Sequ* of infinite sequences of naturals (Example 2). The function

$$\begin{array}{lll} \mathit{zip} : \forall a\, b\,.\ (\mathsf{Stream}\ a,\ \mathsf{Stream}\ b) & \to & \mathsf{Stream}\ (a, b) \\ \mathit{zip}\qquad (\mathit{Link}\ (a, s), \mathit{Link}\ (b, t)) & = & \mathit{Link}\ ((a, b), \mathit{zip}\ (s, t)) \end{array}$$

makes use of the added flexibility, turning a pair of streams into a stream of pairs. Unlike the previous example, *zip* is polymorphic in *two* type variables. Categorically speaking, it is a natural transformation between functors from a product category to some other category:

$$\mathit{zip} : (\times) \circ (\mathsf{Stream} \times \mathsf{Stream}) \dot{\to} \mathsf{Stream} \circ (\times).$$

The type possibly looks a bit confusing with three occurrences of $\times$. The product in $\mathsf{Stream} \times \mathsf{Stream}$ is the product in **Cat**. Specifically, it is the arrow part of the functor $(\times) : \mathbf{Cat} \times \mathbf{Cat} \to \mathbf{Cat}$ defined component-wise: $(\mathsf{F} \times \mathsf{G})\ \langle A,\ B\rangle = \langle \mathsf{F}\,A,\ \mathsf{G}\,B\rangle$ and $(\mathsf{F}\times\mathsf{G})\ \langle f,\ g\rangle = \langle \mathsf{F}\,f,\ \mathsf{G}\,g\rangle$. The other two occurrences of $\times$ denote the product in the ambient category $(\times) : \mathbb{C}\times\mathbb{C} \to \mathbb{C}$. The signature suggests that *zip* is an adjoint unfold, whose transpose has type

$$\mathit{zip}' : \mathsf{Lan}_{\times}\ ((\times) \circ (\mathsf{Stream} \times \mathsf{Stream})) \dot{\to} \mathsf{Stream}.$$

Interestingly, if we unravel the definitions, we find that *zip′* corresponds to the function *zipWith* defined

$$\begin{array}{lll} \mathit{zipWith} : \forall a\, b\, c\,.\ ((a, b) \to c, \mathsf{Stream}\ a,\ \mathsf{Stream}\ b) & \to & \mathsf{Stream}\ c \\ \mathit{zipWith}\qquad (f,\qquad \mathit{Link}\ (a, s), \mathit{Link}\ (b, t)) & = & \mathit{Link}\ (f\ (a, b), \mathit{zipWith}\ (f, s, t)). \end{array}$$

We have turned the existential type into a universal type, as Haskell does not support 'free-flowing' existentials. Unfortunately, we cannot use Lan to implement *zip′* directly, as the kind system insists that the type index of Lan has kind $\star \to \star$, but $\times$ has kind $\star \to \star \to \star$. □

Kan extensions generalise the constructions of the previous section: If the category $\mathbb{C}$ is non-empty ($\mathbb{C} \neq \mathbf{0}$), 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:

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

Since adjoints are unique up to isomorphism, we have $\mathsf{Ran}_{\mathsf{K}\,A} \circ \mathsf{K} \cong \mathsf{Rsh}_A$.

5.9. *Currying continued:* $- \dot{\times}\ \mathsf{H} \dashv (-)^{\mathsf{H}}$

The Kan extension along the identity functor is the identity:

$$\begin{array}{ll} & \mathsf{F} \dot{\to} \mathsf{Id}\ \mathsf{G} \\ \cong & \{\ \text{identity}\ \} \\ & \mathsf{F} \circ \mathsf{Id} \dot{\to} \mathsf{G} \\ \cong & \{\ \text{right Kan extension: } (- \circ \mathsf{Id}) \dashv \mathsf{Ran}_{\mathsf{Id}}\ \} \\ & \mathsf{F} \dot{\to} \mathsf{Ran}_{\mathsf{Id}}\ \mathsf{G}. \end{array}$$

Since every step is natural in F and G, we may conclude $\mathsf{Ran}_{\mathsf{Id}} \cong \mathsf{Id}$—this is an application of the principle of indirect proof, see, for instance, [37]. Dually, we have $\mathsf{Lan}_{\mathsf{Id}} \cong \mathsf{Id}$. The identities look innocent enough, but there is an interesting cross-connection to the Yoneda Lemma. Let $\mathsf{H} : \mathbb{C} \to \mathbb{D}$ be some functor. If we unfold the definitions of Ran and Lan, we obtain the isomorphisms

$$\mathsf{H}\,A \cong \forall Y : \mathbb{C}\ .\ \Pi\,\mathbb{C}(A, Y)\ .\ \mathsf{H}\,Y, \qquad (20)$$

$$\mathsf{H}\,A \cong \exists Y : \mathbb{C}\ .\ \Sigma\,\mathbb{C}(Y, A)\ .\ \mathsf{H}\,Y, \qquad (21)$$

both of which are natural in H and A. The first identity can be seen as a generalisation of the Yoneda Lemma: if $\mathsf{H} : \mathbb{C} \to \mathbf{Set}$ is set-valued, then the powers are given by function spaces and the end reduces to a set of natural transformations. (All the necessary powers exist if $\mathbb{C}$ has small hom-sets.) In other words, (20) specialises to (11). The second identity dualises the first and is known as the *density formula* (or as the co-Yoneda Lemma). Using the density formula we can finally derive the right adjoint of the lifted product $- \mathbin{\dot\times} \mathsf{H}$. (In Section 5.3, we motivated the definition of the exponential G^{H} for the special case of set-valued functors.) We calculate

$$\begin{aligned}
& \mathbb{D}^{\mathbb{C}}(\mathsf{F} \mathbin{\dot\times} \mathsf{H}, \mathsf{G}) \\
\cong\ & \quad \{ \text{ natural transformation as an end [49, p. 223] } \} \\
& \forall Y : \mathbb{C} \,.\, \mathbb{D}((\mathsf{F} \mathbin{\dot\times} \mathsf{H})\, Y, \mathsf{G}\, Y) \\
\cong\ & \quad \{ \text{ definition of } \dot\times \ \} \\
& \forall Y : \mathbb{C} \,.\, \mathbb{D}(\mathsf{F}\, Y \times \mathsf{H}\, Y, \mathsf{G}\, Y) \\
\cong\ & \quad \{ \text{ assumption: } \mathbb{D} \text{ is cartesian closed } - \times X \dashv (-)^X \ \} \\
& \forall Y : \mathbb{C} \,.\, \mathbb{D}(\mathsf{F}\, Y, (\mathsf{G}\, Y)^{\mathsf{H}\, Y}) \\
\cong\ & \quad \{ \text{ density formula (21) } \} \\
& \forall Y : \mathbb{C} \,.\, \mathbb{D}(\exists X : \mathbb{C} \,.\, \Sigma\ \mathbb{C}(X, Y) \,.\, \mathsf{F}\, X, (\mathsf{G}\, Y)^{\mathsf{H}\, Y}) \\
\cong\ & \quad \{ \text{ the hom-functor } \mathbb{D}(-, B) \text{ reverses ends [49, p. 225] } \} \\
& \forall Y : \mathbb{C} \,.\, \forall X : \mathbb{C} \,.\, \mathbb{D}(\Sigma\ \mathbb{C}(X, Y) \,.\, \mathsf{F}\, X, (\mathsf{G}\, Y)^{\mathsf{H}\, Y}) \\
\cong\ & \quad \{ \text{ interchange of quantifiers [49, p. 231f] } \} \\
& \forall X : \mathbb{C} \,.\, \forall Y : \mathbb{C} \,.\, \mathbb{D}(\Sigma\ \mathbb{C}(X, Y) \,.\, \mathsf{F}\, X, (\mathsf{G}\, Y)^{\mathsf{H}\, Y}) \\
\cong\ & \quad \{ \ \Sigma\, \mathbb{I} \dashv \Pi\, \mathbb{I} \text{ (19) } \} \\
& \forall X : \mathbb{C} \,.\, \forall Y : \mathbb{C} \,.\, \mathbb{D}(\mathsf{F}\, X, \Pi\ \mathbb{C}(X, Y) \,.\, (\mathsf{G}\, Y)^{\mathsf{H}\, Y}) \\
\cong\ & \quad \{ \text{ the hom-functor } \mathbb{D}(A, -) \text{ preserves ends [49, p. 225] } \} \\
& \forall X : \mathbb{C} \,.\, \mathbb{D}(\mathsf{F}\, X, \forall Y : \mathbb{C} \,.\, \Pi\ \mathbb{C}(X, Y) \,.\, (\mathsf{G}\, Y)^{\mathsf{H}\, Y}) \\
\cong\ & \quad \{ \text{ define } \mathsf{G}^{\mathsf{H}}\, A = \forall X : \mathbb{C} \,.\, \Pi\ \mathbb{C}(A, X) \,.\, (\mathsf{G}\, X)^{\mathsf{H}\, X} \ \} \\
& \forall X : \mathbb{C} \,.\, \mathbb{D}(\mathsf{F}\, X, \mathsf{G}^{\mathsf{H}}\, X) \\
\cong\ & \quad \{ \text{ natural transformation as an end [49, p. 223] } \} \\
& \mathbb{D}^{\mathbb{C}}(\mathsf{F}, \mathsf{G}^{\mathsf{H}}).
\end{aligned}$$

The definition of G^{H} is interesting as it combines ends, powers and exponentials in a single formula. The calculation shows that $\mathbb{D}^{\mathbb{C}}$ is cartesian closed if $\mathbb{D}$ is cartesian closed and the necessary ends and powers exist.

5.10. *Swapping arguments:* $(X^{(-)})^{op} \dashv X^{(-)}$

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. Infinite sequences and functions over the natural numbers are in one-to-one correspondence. The functions *tabulate* and *lookup* witness the isomorphism.

$$\begin{aligned}
& \mathit{tabulate} : (\mathit{Nat} \to \mathit{Nat}) \to \mathit{Sequ} \\
& \mathit{tabulate}\ \ f = \mathit{Next}\ (f\ Z, \mathit{tabulate}\ (f \cdot S)) \\
& \mathit{lookup} : \mathit{Sequ} \to (\mathit{Nat} \to \mathit{Nat}) \\
& \mathit{lookup}\ \ (\mathit{Next}\ (v, vs))\ (Z) = v \\
& \mathit{lookup}\ \ (\mathit{Next}\ (v, vs))\ (S\, n) = \mathit{lookup}\ vs\ n
\end{aligned}$$

The first isomorphism tabulates a given function, producing a stream of its values. Its inverse looks up a natural number at a given position. □

Tabulation is a standard unfold, but what about *lookup*? Its type involves exponentials: $\mathit{lookup} : \mathbb{C}(\mathit{Sequ}, \mathit{Nat}^{\mu\mathfrak{Nat}})$. However, the curry adjunction $- \times X \dashv (-)^X$ of Section 5.3 is not applicable here, as the right adjoint fixes the source object. We need its counterpart, the functor $X^{(-)} : \mathbb{C}^{\mathrm{op}} \to \mathbb{C}$, which fixes the target object. Since this functor is contravariant, the type of *lookup* is actually $\mathbb{C}^{\mathrm{op}}(Nat^{(-)}\,(\mu\mathfrak{Nat}), Sequ)$, which suggests that the arrow is an adjoint fold! The functor $X^{(-)}$ is interesting as it is self-adjoint:

$$\begin{tikzcd}\mathbb{C}^{\mathrm{op}} \arrow[r, phantom, "\perp"] & \mathbb{C}. \arrow[l, bend right, "(X^{(-)})^{\mathrm{op}}"'] \arrow[l, leftarrow, bend left, "X^{(-)}"]\end{tikzcd}$$

Briefly, the opposite category $\mathbb{C}^{\mathrm{op}}$ has the same objects as $\mathbb{C}$; the arrows of $\mathbb{C}^{\mathrm{op}}$ are in one-to-one correspondence to the arrows in $\mathbb{C}$, that is, $f^{\mathrm{op}} : \mathbb{C}^{\mathrm{op}}(A, B)$ if and only if $f : \mathbb{C}(B, A)$. The operation $(-)^{\mathrm{op}}$ can be extended to a covariant functor $(-)^{\mathrm{op}} : \mathbf{Cat} \to \mathbf{Cat}$, whose arrow part is defined $\mathsf{F}^{\mathrm{op}} A = \mathsf{F} A$ and $\mathsf{F}^{\mathrm{op}} f^{\mathrm{op}} = (\mathsf{F} f)^{\mathrm{op}}$. The adjunction $(X^{(-)})^{\mathrm{op}} \dashv X^{(-)}$ is a consequence of currying:

$$\begin{aligned}
&\mathbb{C}^{\mathrm{op}}(X^A, B)\\
\cong\;&\{\text{ opposite category }\}\\
&\mathbb{C}(B, X^A)\\
\cong\;&\{\text{ currying: } - \times X \dashv (-)^X \}\\
&\mathbb{C}(B \times A, X)\\
\cong\;&\{\ \times \text{ is commutative }\}\\
&\mathbb{C}(A \times B, X)\\
\cong\;&\{\text{ currying: } - \times X \dashv (-)^X \}\\
&\mathbb{C}(A, X^B).
\end{aligned}$$

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

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

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: $\mathbb{C}(A, \mathsf{R}\,(\nu\mathsf{F}))$. Since $\mathsf{R} = X^{(-)}$ is contravariant, the final coalgebra in $\mathbb{C}^{\mathrm{op}}$ is the initial algebra in $\mathbb{C}$. Since furthermore $X^{(-)}$ is self-adjoint, we obtain the type of an adjoint fold: $\mathbb{C}(A, \mathsf{L}\,(\mu\mathsf{F})) = \mathbb{C}^{\mathrm{op}}(\mathsf{L}\,(\mu\mathsf{F}), A)$.

It may seem overkill to model *lookup* as an adjoint fold—its transposed fold simply swaps the two arguments. However, this approach will pay dividends later, when we show that *tabulate* and *lookup* are actually inverses (Section 8.7). Besides, $X^{(-)}$ is interesting in its own right, as it is the first adjunction that involves contravariant functors.

Table 3 summarises the basic adjunctions introduced in this section.

6. Combining adjunctions

What is a combinator library? ⟨. . .⟩ the key idea is this: a combinator library offers functions (the combinators) that combine functions together to make bigger functions.

A History of Haskell: Being Lazy With Class—Paul Hudak et al

In the previous section we have discussed a variety of basic adjunctions. In this section we look at ways of combining these adjunctions to form more complex ones. This will allow us to fit even more definitions under the umbrella of adjoint equations.

6.1. Composition of adjoints

Like functors, adjunctions can be composed. Given adjunctions $\mathsf{L}_1 \dashv \mathsf{R}_1$ and $\mathsf{L}_2 \dashv \mathsf{R}_2$, their *composition* yields an adjunction $\mathsf{L}_2 \circ \mathsf{L}_1 \dashv \mathsf{R}_1 \circ \mathsf{R}_2$.

$$\begin{tikzcd}\mathbb{C} \arrow[r, phantom, "\perp"] & \mathbb{D} \arrow[l, bend right, "\mathsf{L}_2"'] \arrow[l, leftarrow, bend left, "\mathsf{R}_2"] \arrow[r, phantom, "\perp"] & \mathbb{E} \arrow[l, bend right, "\mathsf{L}_1"'] \arrow[l, leftarrow, bend left, "\mathsf{R}_1"] & \text{then} & \mathbb{C} \arrow[r, phantom, "\perp"] & \mathbb{E} \arrow[l, bend right, "\mathsf{L}_2 \circ \mathsf{L}_1"'] \arrow[l, leftarrow, bend left, "\mathsf{R}_1 \circ \mathsf{R}_2"]\end{tikzcd}$$

Table 3
Adjunctions and types of recursion.

Adjunction	Initial fixed-point equation	Final fixed-point equation
$\mathsf{L} \dashv \mathsf{R}$	$x \cdot \mathsf{L}\, in = \Psi\, x$ $\phi\, x \cdot in = (\phi \cdot \Psi \cdot \phi^{\circ})\, (\phi\, x)$	$\mathsf{R}\, out \cdot x = \Psi\, x$ $out \cdot \phi^{\circ}\, x = (\phi^{\circ} \cdot \Psi \cdot \phi)\, (\phi^{\circ}\, x)$
$\mathsf{Id} \dashv \mathsf{Id}$	Standard fold Standard fold	Standard unfold Standard unfold
$- \times X \dashv (-)^X$	Parametrised fold Fold to an exponential	Curried unfold Unfold from a product
$- \times \mu\mathsf{G} \dashv (-)^{\mu\mathsf{G}}$	Simultaneous recursion Fold to an exponential	–
$(X^{(-)})^{\mathsf{op}} \dashv X^{(-)}$	Swapped curried fold Fold to an exponential	
$(+) \dashv \Delta$	Recursion from a coproduct of mutually recursive types Mutual value recursion on mutually recursive types	Mutual value recursion Single recursion from a coproduct domain
$\Delta \dashv (\times)$	Mutual value recursion Single recursion to a product domain	Recursion to a product of mutually recursive types Mutual value recursion on mutually recursive types
$\mathsf{Lsh}_X \dashv (-X)$	–	Monomorphic unfold Unfold from a left shift
$(-X) \dashv \mathsf{Rsh}_X$	Monomorphic fold Fold to a right shift	–
$\mathsf{Lan}_\mathsf{J} \dashv (- \circ \mathsf{J})$	–	Polymorphic unfold Unfold from a left Kan extension
$(- \circ \mathsf{J}) \dashv \mathsf{Ran}_\mathsf{J}$	Polymorphic fold Fold to a right Kan extension	–

Observe that the right adjoints are composed in the reverse order. To establish the bijection $\mathbb{C}(\mathsf{L}_2\, (\mathsf{L}_1\, A), B) \cong \mathbb{E}(A, \mathsf{R}_1\, (\mathsf{R}_2\, B))$, we simply compose the two adjuncts.

$$\begin{aligned}
&\mathbb{C}(\mathsf{L}_2\, (\mathsf{L}_1\, A), B)\\
\cong\;&\quad \{\text{ assumption: } \mathsf{L}_2 \dashv \mathsf{R}_2 \}\\
&\mathbb{D}(\mathsf{L}_1\, A, \mathsf{R}_2\, B)\\
\cong\;&\quad \{\text{ assumption: } \mathsf{L}_1 \dashv \mathsf{R}_1 \}\\
&\mathbb{E}(A, \mathsf{R}_1\, (\mathsf{R}_2\, B))
\end{aligned}$$

Each step is natural in A and B and consequently also the composite isomorphism. The adjuncts are defined

$$\begin{aligned}
\phi\, \langle A,\ B\rangle &= \phi_1\, \langle A,\ \mathsf{R}_2\, B\rangle \cdot \phi_2\, \langle \mathsf{L}_1\, A,\ B\rangle,\\
\phi^{\circ}\, \langle A,\ B\rangle &= \phi_2^{\circ}\, \langle \mathsf{L}_1\, A,\ B\rangle \cdot \phi_1^{\circ}\, \langle A,\ \mathsf{R}_2\, B\rangle.
\end{aligned}$$

As an example, the double adjunction $(+) \dashv \Delta \dashv (\times)$ implies the adjunction $(+) \circ \Delta \dashv (\times) \circ \Delta$, that is, $A + A \to B \cong A \to B \times B$ (see also Section 5.6). Note that $\mathsf{L} \dashv \mathsf{M} \dashv \mathsf{R}$ induces the adjunction $\mathsf{L} \circ \mathsf{M} \dashv \mathsf{R} \circ \mathsf{M}$, *not* $\mathsf{L} \dashv \mathsf{R}$, even though the notation may seem to suggest this.

Many adjoint folds and unfolds arise as compositions of adjunctions. For reasons of space, we refrain from providing detailed Haskell examples. Instead, we consider some example type signatures and show how to read them as types of adjoint folds, $\mathsf{L}\, (\mu\mathsf{F}) \to B$. Let F be some first-order and let H be some higher-order functor. Then

$$\begin{aligned}
(\mu\mathsf{F} \times A_1) \times A_2 \to B &= ((- \times A_2) \circ (- \times A_1))\, (\mu\mathsf{F}) \to B\\
\mu\mathsf{H}\, T \times A \to B &= ((- \times A) \circ (- T))\, (\mu\mathsf{H}) \to B\\
\forall X\ .\ \mu\mathsf{H}\, (\mathsf{J}_1\, (\mathsf{J}_2\, X)) \to \mathsf{B}\, X &= ((- \circ \mathsf{J}_2) \circ (- \circ \mathsf{J}_1))\, (\mu\mathsf{H}) \stackrel{\cdot}{\to} \mathsf{B}.
\end{aligned}$$

Often, there is a choice as to how we view a type. In the first example, since the cartesian product is associative, we can alternatively use

$$\mu\mathsf{F} \times (A_1 \times A_2) \to B \quad = (- \times (A_1 \times A_2))\, (\mu\mathsf{F}) \to B.$$

The transpose is now a simple curried fold. Previously, it was doubly curried. Likewise, since composition is associative, we can view the third type in a different way:

$$\forall X \,.\, \mu\mathsf{H}\,(\mathsf{J}_1\,(\mathsf{J}_2\,X)) \to \mathsf{B}\,X = (- \circ (\mathsf{J}_1 \circ \mathsf{J}_2))\,(\mu\mathsf{H}) \mathrel{\dot{\to}} \mathsf{B}.$$

The original transpose uses a nested Kan extension; this one manages with a single Kan.

In fact, compositions of adjunctions can often be simplified. The adjunctions of Section 5 that have a parameter satisfy the following identities:

$$\begin{aligned}
(- \times Y) \circ (- \times X) &\cong (- \times X \times Y) \\
(-)^X \circ (-)^Y &\cong (-)^{X \times Y} \\
(-\,Y) \circ (-\,X) &\cong \mathsf{Uncurry} \circ (-\,\langle X,\ Y\rangle) \\
\mathsf{Lsh}_X \circ \mathsf{Lsh}_Y &\cong \mathsf{Curry} \circ \mathsf{Lsh}_{\langle X,\,Y\rangle} \\
\mathsf{Rsh}_X \circ \mathsf{Rsh}_Y &\cong \mathsf{Curry} \circ \mathsf{Rsh}_{\langle X,\,Y\rangle} \\
(- \circ \mathsf{J}) \circ (- \circ \mathsf{H}) &\cong (- \circ \mathsf{H} \circ \mathsf{J}) \\
\mathsf{Lan}_\mathsf{J} \circ \mathsf{Lan}_\mathsf{H} &\cong \mathsf{Lan}_{\mathsf{J} \circ \mathsf{H}} \\
\mathsf{Ran}_\mathsf{J} \circ \mathsf{Ran}_\mathsf{H} &\cong \mathsf{Ran}_{\mathsf{J} \circ \mathsf{H}},
\end{aligned}$$

where $\mathsf{Curry} : \mathbb{C} \times \mathbb{D} \to \mathbb{E} \cong \mathbb{C} \to \mathbb{E}^{\mathbb{D}} : \mathsf{Uncurry}$ is the isomorphism of the curry adjunction in **Cat**. We show the last property and leave the others as exercises.

$$\begin{aligned}
& \mathsf{F} \mathrel{\dot{\to}} (\mathsf{Ran}_\mathsf{J} \circ \mathsf{Ran}_\mathsf{H})\,\mathsf{G} \\
= \quad & \{\text{ definition of functor composition }\} \\
& \mathsf{F} \mathrel{\dot{\to}} \mathsf{Ran}_\mathsf{J}\,(\mathsf{Ran}_\mathsf{H}\,\mathsf{G}) \\
\cong \quad & \{\text{ right Kan extension: } (- \circ \mathsf{J}) \dashv \mathsf{Ran}_\mathsf{J} \} \\
& \mathsf{F} \circ \mathsf{J} \mathrel{\dot{\to}} \mathsf{Ran}_\mathsf{H}\,\mathsf{G} \\
\cong \quad & \{\text{ right Kan extension: } (- \circ \mathsf{H}) \dashv \mathsf{Ran}_\mathsf{H} \} \\
& \mathsf{F} \circ \mathsf{J} \circ \mathsf{H} \mathrel{\dot{\to}} \mathsf{G} \\
\cong \quad & \{\text{ right Kan extension: } (- \circ \mathsf{J} \circ \mathsf{H}) \dashv \mathsf{Ran}_{\mathsf{J} \circ \mathsf{H}} \} \\
& \mathsf{F} \mathrel{\dot{\to}} \mathsf{Ran}_{\mathsf{J} \circ \mathsf{H}}\,\mathsf{G}
\end{aligned}$$

Since every step is natural in F and G, the claim follows.

6.2. Product of adjoints

In Section 3.3 we have seen that we can view arrows over datatypes defined by mutual recursion as a fold in a product category. Now assume that the defining equations already involve adjoint functors:

$$x_1 \cdot \mathsf{L}_1\,in_1 = \Psi_1\,\langle x_1,\ x_2\rangle \quad \text{and} \quad x_2 \cdot \mathsf{L}_2\,in_2 = \Psi_2\,\langle x_1,\ x_2\rangle.$$

To model this situation, we define the product of adjunctions.

Given adjunctions $\mathsf{L}_1 \dashv \mathsf{R}_1$ and $\mathsf{L}_2 \dashv \mathsf{R}_2$, the *product*[3] $\mathsf{L}_1 \times \mathsf{L}_2 \dashv \mathsf{R}_1 \times \mathsf{R}_2$ is an adjunction between product categories.

$$\begin{tikzcd}
\mathbb{C}_1 \arrow[r, bend right, "\mathsf{R}_1"'] \arrow[r, phantom, "\perp"] & \mathbb{D}_1 \arrow[l, bend right, "\mathsf{L}_1"']
\end{tikzcd} \quad \text{and} \quad
\begin{tikzcd}
\mathbb{C}_2 \arrow[r, bend right, "\mathsf{R}_2"'] \arrow[r, phantom, "\perp"] & \mathbb{D}_2 \arrow[l, bend right, "\mathsf{L}_2"']
\end{tikzcd} \quad \text{then} \quad
\begin{tikzcd}
\mathbb{C}_1 \times \mathbb{C}_2 \arrow[r, bend right, "\mathsf{R}_1 \times \mathsf{R}_2"'] \arrow[r, phantom, "\perp"] & \mathbb{D}_1 \times \mathbb{D}_2 \arrow[l, bend right, "\mathsf{L}_1 \times \mathsf{L}_2"']
\end{tikzcd}$$

Recall that the product $(- \times =) : \mathbf{Cat} \times \mathbf{Cat} \to \mathbf{Cat}$ is itself a functor whose action on arrows, that is, functors is defined $(\mathsf{F} \times \mathsf{G})\,\langle A,\ B\rangle = \langle \mathsf{F}\,A,\ \mathsf{G}\,B\rangle$ and $(\mathsf{F} \times \mathsf{G})\,\langle f,\ g\rangle = \langle \mathsf{F}\,f,\ \mathsf{G}\,g\rangle$.

The bijection is easy to establish:

$$\begin{aligned}
& (\mathbb{C}_1 \times \mathbb{C}_2)((\mathsf{L}_1 \times \mathsf{L}_2)\,\langle A_1,\ A_2\rangle,\ \langle B_1,\ B_2\rangle) \\
\cong \quad & \{\text{ definition of } \mathsf{F} \times \mathsf{G} \text{ and definition of } \mathbb{C} \times \mathbb{D} \} \\
& \mathbb{C}_1(\mathsf{L}_1\,A_1, B_1) \times \mathbb{C}_2(\mathsf{L}_2\,A_2, B_2) \\
\cong \quad & \{\text{ assumptions: } \mathsf{L}_1 \dashv \mathsf{R}_1 \text{ and } \mathsf{L}_2 \dashv \mathsf{R}_2 \} \\
& \mathbb{D}_1(A_1, \mathsf{R}_1\,B_1) \times \mathbb{D}_2(A_2, \mathsf{R}_2\,B_2) \\
\cong \quad & \{\text{ definition of } \mathbb{C} \times \mathbb{D} \text{ and definition of } \mathsf{F} \times \mathsf{G} \} \\
& (\mathbb{D}_1 \times \mathbb{D}_2)(\langle A_1,\ A_2\rangle,\ (\mathsf{R}_1 \times \mathsf{R}_2)\,\langle B_1,\ B_2\rangle).
\end{aligned}$$

[3] Small categories and adjunctions form a category called **Adj**. The product of adjunctions is, however, not a categorical product in **Adj**, since in general the projection functors Outl and Outr are not part of an adjoint situation.

Each isomorphism is natural in $\langle A_1, A_2\rangle$ and $\langle B_1, B_2\rangle$ and hence also their composition. The adjuncts are given by

$$\phi\,\langle f, g\rangle = \langle \phi_1 f, \phi_2 g\rangle \quad\text{and}\quad \phi^\circ\,\langle f, g\rangle = \langle \phi_1^\circ f, \phi_2^\circ g\rangle.$$

Returning to the original problem, the required adjunction is simply $\mathsf{L}_1 \times \mathsf{L}_2$. Assuming the usual abbreviations (see Section 3.3), the adjoint fixed-point equation unfolds to

$$x \cdot (\mathsf{L}_1 \times \mathsf{L}_2)\, in = \Psi\, x \iff \begin{array}{l} x_1 \cdot \mathsf{L}_1\, in_1 = \Psi_1\,\langle x_1, x_2\rangle \\ x_2 \cdot \mathsf{L}_2\, in_2 = \Psi_2\,\langle x_1, x_2\rangle. \end{array}$$

As a slight variation consider two arrows that recurse over *one* datatype, rather than over a pair of mutually recursive datatypes:

$$x \cdot ((\mathsf{L}_1 \times \mathsf{L}_2) \circ \Delta)\, in = \Psi\, x \iff \begin{array}{l} x_1 \cdot \mathsf{L}_1\, in = \Psi_1\,\langle x_1, x_2\rangle \\ x_2 \cdot \mathsf{L}_2\, in = \Psi_2\,\langle x_1, x_2\rangle. \end{array}$$

The adjunction is now given as the composition of $\Delta \dashv (\times)$ with a product of adjunctions: $(\mathsf{L}_1 \times \mathsf{L}_2) \circ \Delta \dashv (\times) \circ (\mathsf{R}_1 \times \mathsf{R}_2)$. Note that L_1 and L_2 need not be the same functor—more often than not one of the functors is Id.

6.3. Post-composition

In Section 5.3 we have discussed adjoint folds of type $\mu\mathsf{F} \times A \to B$ and $\mu\mathsf{H} \mathbin{\dot\times} \mathsf{A} \mathbin{\dot\to} \mathsf{B}$. In Section 6.1 we have considered a variation of the theme: an arrow of type $\mu\mathsf{H}\,T \times A \to B$. Here is another twist: a polymorphic function whose accumulating argument is monomorphic, $\forall X\ .\ \mu\mathsf{H}X \times A \to \mathsf{B}X$. One way to deal with this example is to lift the 'curry adjunction' $- \times X \dashv (-)^X$ into a higher realm, to a functor category.

Let $\mathsf{F} : \mathbb{C} \to \mathbb{D}$ be some functor. We have alluded on various occasions to the fact that post-composition $\mathsf{F} \circ - : \mathbb{C}^{\mathbb{E}} \to \mathbb{D}^{\mathbb{E}}$ is itself functorial: the action on arrows, that is, natural transformations is defined $(\mathsf{F} \circ \alpha)\, A = \mathsf{F}\,(\alpha\, A)$.

Now, if $\mathsf{L} \dashv \mathsf{R}$ is an adjunction then $\mathsf{L} \circ - \dashv \mathsf{R} \circ -$ is an adjunction, as well.

$$\begin{tikzcd} \mathbb{C} \arrow[r, bend right, "\mathsf{R}"'] \arrow[r, phantom, "\perp"] & \mathbb{D} \arrow[l, bend right, "\mathsf{L}"'] \end{tikzcd} \quad\text{then}\quad \begin{tikzcd} \mathbb{C}^{\mathbb{E}} \arrow[r, bend right, "\mathsf{R} \circ -"'] \arrow[r, phantom, "\perp"] & \mathbb{D}^{\mathbb{E}} \arrow[l, bend right, "\mathsf{L} \circ -"'] \end{tikzcd}$$

(The implication can, in fact, be strengthened to an equivalence. We leave the details to the reader.) The proof proceeds as follows.

$$\begin{array}{ll} & \mathbb{C}^{\mathbb{E}}(\mathsf{L} \circ \mathsf{A}, \mathsf{B}) \\ \cong & \{\ \text{natural transformation as an end [49, p. 223]}\ \} \\ & \forall X : \mathbb{E}\ .\ \mathbb{C}(\mathsf{L}\,(\mathsf{A}X), \mathsf{B}X) \\ \cong & \{\ \text{assumption: } \mathsf{L} \dashv \mathsf{R}\ \} \\ & \forall X : \mathbb{E}\ .\ \mathbb{D}(\mathsf{A}X, \mathsf{R}\,(\mathsf{B}X)) \\ \cong & \{\ \text{natural transformation as an end [49, p. 223]}\ \} \\ & \mathbb{D}^{\mathbb{E}}(\mathsf{A}, \mathsf{R} \circ \mathsf{B}) \end{array}$$

The resulting adjuncts can be best expressed in terms of the units of the underlying adjunction $\mathsf{L} \dashv \mathsf{R}$—note the similarity to Eqs. (17).

$$\begin{aligned} \phi\,(\alpha : \mathsf{L} \circ \mathsf{A} \mathbin{\dot\to} \mathsf{B}) &= (\mathsf{R} \circ \alpha) \cdot (\eta \circ \mathsf{A}) \\ \phi^\circ\,(\beta : \mathsf{A} \mathbin{\dot\to} \mathsf{R} \circ \mathsf{B}) &= (\epsilon \circ \mathsf{B}) \cdot (\mathsf{L} \circ \beta) \end{aligned}$$

(As an aside, the units of $\mathsf{L} \circ - \dashv \mathsf{R} \circ -$ are simply $\epsilon \circ -$ and $\eta \circ -$.)

Using lifting we can solve the problem stated in the introduction to this section. To model polymorphic folds that have a monomorphic accumulating argument, we lift currying:

$$\forall X\ .\ \mu\mathsf{H}X \times A \to \mathsf{B}X \qquad \cong ((- \times A) \circ -)\,(\mu\mathsf{H}) \mathbin{\dot\to} \mathsf{B}.$$

Alternatively, we can replace A by $\mathsf{K}AX$ and then lift the product.

$$\forall X\ .\ \mu\mathsf{H}X \times \mathsf{K}AX \to \mathsf{B}X \cong (- \mathbin{\dot\times} \mathsf{K}\,Y)\,(\mu\mathsf{H}) \mathbin{\dot\to} \mathsf{B}$$

The latter construction is, however, more involved as it builds on exponentials in functor categories.

Table 4
Combining adjunctions.

$\mathsf{L}_1 \dashv \mathsf{R}_1$ and $\mathsf{L}_2 \dashv \mathsf{R}_2$	
Composition	$\mathsf{L}_2 \circ \mathsf{L}_1 \dashv \mathsf{R}_1 \circ \mathsf{R}_2$
Product	$\mathsf{L}_1 \times \mathsf{L}_2 \dashv \mathsf{R}_1 \times \mathsf{R}_2$
Post-composition	$\mathsf{L}_1 \circ - \dashv \mathsf{R}_1 \circ -$
Pre-composition	$- \circ \mathsf{R}_1 \dashv - \circ \mathsf{L}_1$

6.4. Pre-composition

The development of the previous section nicely dualises: post-composition is pre-composition in the opposite category. For reasons to be become clear in a moment, let us spell out the details.

Let $\mathsf{F} : \mathbb{C} \to \mathbb{D}$ be some functor. Pre-composition $- \circ \mathsf{F} : \mathbb{E}^{\mathbb{D}} \to \mathbb{E}^{\mathbb{C}}$ is itself functorial: the action on arrows is defined $(\alpha \circ \mathsf{F})\,A = \alpha\,(\mathsf{F}\,A)$.

Now, if $\mathsf{L} \dashv \mathsf{R}$ is an adjunction then $- \circ \mathsf{R} \dashv - \circ \mathsf{L}$ is an adjunction, as well.

$$\begin{tikzcd} \mathbb{C} \arrow[r, bend right, "\mathsf{R}"'] \arrow[r, phantom, "\perp"] & \mathbb{D} \arrow[l, bend right, "\mathsf{L}"'] \end{tikzcd} \quad \text{then} \quad \begin{tikzcd} \mathbb{E}^{\mathbb{D}} \arrow[r, bend right, "- \circ \mathsf{L}"'] \arrow[r, phantom, "\perp"] & \mathbb{E}^{\mathbb{C}} \arrow[l, bend right, "- \circ \mathsf{R}"'] \end{tikzcd}$$

The adjuncts are defined

$$\phi\,(\alpha : \mathsf{A} \circ \mathsf{R} \dot{\to} \mathsf{B}) = (\alpha \circ \mathsf{L}) \cdot (\mathsf{A} \circ \eta),$$
$$\phi^{\circ}\,(\beta : \mathsf{A} \dot{\to} \mathsf{B} \circ \mathsf{L}) = (\mathsf{B} \circ \epsilon) \cdot (\beta \circ \mathsf{R}).$$

Observe that in the lifted adjunction the left and the right adjoint are swapped. For instance, we have $- \circ (\times) \dashv - \circ \Delta$ since $\Delta \dashv (\times)$. The lifted adjunction is useful for massaging polymorphic folds whose type involves two type variables (see also Example 27).

$$\forall X_1, X_2 \;.\; \mu\mathsf{H}\,(X_1 \times X_2) \to X_1 = (- \circ (\times))\,(\mu\mathsf{H}) \dot{\to} \mathsf{Outl}$$

The transpose is a fold of type $\mu\mathsf{H} \dot{\to} (- \circ \Delta)\,\mathsf{Outl} = \mu\mathsf{H} \dot{\to} \mathsf{Id}$.

Something interesting has happened. If L has a right adjoint R, then the higher-order functor $- \circ \mathsf{L}$ has two left adjoints: $- \circ \mathsf{R}$ and the left Kan extension along L (Section 5.8). Since left adjoints are unique up to isomorphism, we can immediately conclude that

$$- \circ \mathsf{R} \cong \mathsf{Lan}_{\mathsf{L}}. \tag{22}$$

Two consequences of this fact are worth recording.

$$\mathsf{Lan}_{\mathsf{L}}\,\mathsf{Id} \cong \mathsf{R}$$
$$\mathsf{Lan}_{\mathsf{L}}\,(\mathsf{G} \circ \mathsf{F}) \cong \mathsf{G} \circ \mathsf{Lan}_{\mathsf{L}}\,\mathsf{F}$$

The right adjoint itself can be expressed as a Kan extension. Furthermore, the left Kan extension along L is preserved by any functor. Turning to the proofs, the first law is a direct consequence of (22). For the second, we apply (22) twice.

$$\mathsf{Lan}_{\mathsf{L}}\,(\mathsf{G} \circ \mathsf{F}) \cong (\mathsf{G} \circ \mathsf{F}) \circ \mathsf{R} \cong \mathsf{G} \circ (\mathsf{F} \circ \mathsf{R}) \cong \mathsf{G} \circ \mathsf{Lan}_{\mathsf{L}}\,\mathsf{F}.$$

As an example, for $\mathsf{L} = - \times X$ we obtain a characterisation of the exponential:

$$A^X = (-)^X A \cong \mathsf{Lan}_{-\times X}\,\mathsf{Id}\,A = \exists Y : \mathbb{C} \;.\; \Sigma\,\mathbb{C}(Y \times X, A) \;.\; Y.$$

Using coends and copowers, the internal hom-functor $(=)^{(-)}$ can be expressed in terms of the external hom-functor $\mathbb{C}(-, =)$.

To summarise, adjunctions and Kan extensions are intimately linked. In fact, one can show that L has a right adjoint if and only if the left Kan extension $\mathsf{Lan}_{\mathsf{L}}\,\mathsf{Id}$ exists and is preserved by L [49, Theorem X.7.2].

Table 4 summarises the constructions introduced in this section.

7. Calculational properties

Calculemus Igitur
Lambert Meertens

In this section we develop the calculational properties of adjoint (un)folds. We shall concentrate on initial algebras and folds in the main thrust of the section. For reference, Table 5 lists the dual laws for final coalgebras and unfolds.

7.1. Uniqueness property

The fact that an adjoint initial fixed-point equation has a unique solution can be captured by the following equivalence, the *uniqueness property*.

$$x = (\!|\Psi|\!)_{\mathsf{L}} \iff x \cdot \mathsf{L}\, in = \Psi\, x \qquad (23)$$

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

$$(\!|\Psi|\!)_{\mathsf{L}} \cdot \mathsf{L}\, in = \Psi\, (\!|\Psi|\!)_{\mathsf{L}} \qquad (24)$$

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*.

$$(\!|\Psi|\!)_{\mathsf{L}} = id \iff \Psi\, id = \mathsf{L}\, in \qquad (25)$$

As an application of these identities, let us prove the banana-split law [9], a simple optimisation which replaces a double tree traversal by a single one. The law is traditionally given in terms of standard folds. However, it can be readily ported to adjoint folds.

$$(\!|\Phi|\!)_{\mathsf{L}} \vartriangle (\!|\Psi|\!)_{\mathsf{L}} = (\!|\Phi \otimes \Psi|\!)_{\mathsf{L}} \qquad (26)$$
$$\textbf{where}\quad (\Phi \otimes \Psi)\, x = \Phi\, (outl \cdot x) \vartriangle \Psi\, (outr \cdot x) \qquad (27)$$

The double traversal on the left is optimised 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'.) It is worth pointing out that the definition of $\otimes$ neither mentions the base functor F nor the adjoint functor L—in a sense, the base functions are hiding unnecessary details. Indeed, the proof of (26) is shorter than the one for folds given in the aforementioned textbook. We appeal to the uniqueness property (23); the obligation is discharged as follows.

$$\begin{aligned}
& ((\!|\Phi|\!)_{\mathsf{L}} \vartriangle (\!|\Psi|\!)_{\mathsf{L}}) \cdot \mathsf{L}\, in \\
= \;& \{ \text{split-fusion: } (f \vartriangle g) \cdot h = f \cdot h \vartriangle g \cdot h \} \\
& (\!|\Phi|\!)_{\mathsf{L}} \cdot \mathsf{L}\, in \vartriangle (\!|\Psi|\!)_{\mathsf{L}} \cdot \mathsf{L}\, in \\
= \;& \{ \text{fold-computation (24)} \} \\
& \Phi\, (\!|\Phi|\!)_{\mathsf{L}} \vartriangle \Psi\, (\!|\Psi|\!)_{\mathsf{L}} \\
= \;& \{ \text{split-computation: } outl \cdot (f \vartriangle g) = f \text{ and } outr \cdot (f \vartriangle g) = g \} \\
& \Phi\, (outl \cdot ((\!|\Phi|\!)_{\mathsf{L}} \vartriangle (\!|\Psi|\!)_{\mathsf{L}})) \vartriangle \Psi\, (outr \cdot ((\!|\Phi|\!)_{\mathsf{L}} \vartriangle (\!|\Psi|\!)_{\mathsf{L}})) \\
= \;& \{ \text{definition of } \otimes \text{ (27)} \} \\
& (\Phi \otimes \Psi)\, ((\!|\Phi|\!)_{\mathsf{L}} \vartriangle (\!|\Psi|\!)_{\mathsf{L}})
\end{aligned}$$

The type of an adjoint fold $\mathbb{C}(\mathsf{L}\,(\mu\mathsf{F}), A)$ involves three ingredients: the result type *A*, the adjoint functor L and the base functor F. Correspondingly, there are three fusion laws that allow us to fuse a context with a fold to form another fold: *vanilla fusion* for a context that manipulates *A* (Section 7.2), *conjugate fusion* for a context that modifies L (Section 7.3) and finally *base functor fusion* for a context that changes F (Section 7.5). In fact, there is a fourth law that generalises the first two (Section 7.4), but we are skipping ahead.

7.2. Fusion

The *fusion law* states a condition for fusing an arrow $h : \mathbb{C}(A, B)$ with an adjoint fold $(\!|\Phi|\!)_{\mathsf{L}} : \mathbb{C}(\mathsf{L}\,(\mu\mathsf{F}), A)$ to form another adjoint fold $(\!|\Psi|\!)_{\mathsf{L}} : \mathbb{C}(\mathsf{L}\,(\mu\mathsf{F}), B)$. The condition can be easily calculated.

$$\begin{aligned}
& h \cdot (\!|\Phi|\!)_{\mathsf{L}} = (\!|\Psi|\!)_{\mathsf{L}} \\
\iff \;& \{ \text{uniqueness property (23)} \} \\
& h \cdot (\!|\Phi|\!)_{\mathsf{L}} \cdot \mathsf{L}\, in = \Psi\, (h \cdot (\!|\Phi|\!)_{\mathsf{L}}) \\
\iff \;& \{ \text{computation (24)} \} \\
& h \cdot \Phi\, (\!|\Phi|\!)_{\mathsf{L}} = \Psi\, (h \cdot (\!|\Phi|\!)_{\mathsf{L}}) \\
\Longleftarrow \;& \{ \text{abstracting away from } (\!|\Phi|\!)_{\mathsf{L}} \} \\
& \forall f\, .\; h \cdot \Phi\, f = \Psi\, (h \cdot f)
\end{aligned}$$

Consequently,

$$h \cdot (\!|\Phi|\!)_{\mathsf{L}} = (\!|\Psi|\!)_{\mathsf{L}} \Longleftarrow \forall f\, .\; h \cdot \Phi\, f = \Psi\, (h \cdot f). \qquad (28)$$

Like for banana-split, the fusion condition $h \cdot \Phi\, f = \Psi\,(h \cdot f)$ neither mentions the base functor F nor the adjoint functor L, which makes the law easy to use. Let us illustrate fusion by giving a second proof of the banana-split law (26). The first proof invoked the uniqueness property of folds and then the fusion law for split. Alternatively, we can first invoke the uniqueness property of split

$$g = f_1 \vartriangle f_2 \iff \mathit{outl} \cdot g = f_1 \wedge \mathit{outr} \cdot g = f_2,$$

and then discharge the obligation using the fusion law for folds. (We only prove the first conjunct, the proof of the second proceeds analogously.)

$$\begin{array}{ll}
& \mathit{outl} \cdot (\!|\Phi \otimes \Psi|\!)_{\mathsf{L}} = (\!|\Phi|\!)_{\mathsf{L}} \\
\Longleftarrow & \{\ \text{fold-fusion (28)}\ \} \\
& \forall f\ .\ \mathit{outl} \cdot (\Phi \otimes \Psi)\, f = \Phi\,(\mathit{outl} \cdot f) \\
\iff & \{\ \text{definition of} \otimes \text{(27)}\ \} \\
& \forall f\ .\ \mathit{outl} \cdot (\Phi\,(\mathit{outl} \cdot f) \vartriangle \Psi\,(\mathit{outr} \cdot f)) = \Phi\,(\mathit{outl} \cdot f) \\
\iff & \{\ \text{split-computation: } \mathit{outl} \cdot (f \vartriangle g) = f\ \} \\
& \forall f\ .\ \Phi\,(\mathit{outl} \cdot f) = \Phi\,(\mathit{outl} \cdot f)
\end{array}$$

Example 29. The function *height* determines the height of a stack.

$$\begin{array}{lll}
\mathit{height} : \mathit{Stack} & \rightarrow & \mathit{Nat} \\
\mathit{height}\ \ (\mathit{Empty}) & = & 0 \\
\mathit{height}\ \ (\mathit{Push}\,(n, s)) & = & 1 + \mathit{height}\ s
\end{array}$$

Let us show that *height* is a monoid homomorphism from the stack monoid to the monoid of natural numbers with addition, $\mathit{height} : (\mathit{Stack}, \mathit{Empty}, \diamond) \rightarrow (\mathit{Nat}, 0, +)$:

$$\mathit{height}\ (\mathit{Empty}) = 0,$$
$$\mathit{height}\ (x \diamond y) = \mathit{height}\ x + \mathit{height}\ y,$$

or, written in a point-free style,

$$\mathit{height} \cdot \mathit{empty} = \mathit{zero}, \qquad (29)$$
$$\mathit{height} \cdot \mathit{cat} = \mathit{plus} \cdot (\mathit{height} \times \mathit{height}). \qquad (30)$$

Here *zero* is the constant arrow that yields 0, *empty* is the constant arrow that yields *Empty*, and, finally, $\diamond$ and $+$ are *cat* and *plus* written infix. (Example 30 demonstrates that stacks with concatenation indeed form a monoid.) The first condition (29) is an immediate consequence of *height*'s definition. Regarding the second condition (30), there is no obvious attacking zone, 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 its adjoint fixed-point equation.

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 11. Fusion (28) immediately gives us

$$\forall \mathit{cat}\ .\ \mathit{height} \cdot \mathfrak{cat}\ \mathit{cat} = \mathfrak{height2}\,(\mathit{height} \cdot \mathit{cat}), \qquad (31)$$

from which we can easily synthesise a definition of $\mathfrak{height2}$:

$$\begin{array}{llrcl}
\mathfrak{height2} : \forall x\ .\ (\mathsf{L}\,x \rightarrow \mathit{Nat}) \rightarrow & (\mathsf{L}\,(\mathfrak{Stack}\,x) & & \rightarrow & \mathit{Nat}) \\
\mathfrak{height2} \quad \mathit{height2} & (\mathfrak{Empty}, & y) & = & \mathit{height}\ y \\
\mathfrak{height2} \quad \mathit{height2} & (\mathfrak{Push}\,(a, x), & y) & = & 1 + \mathit{height2}\,(x, y).
\end{array}$$

For the second step, we have to show

$$\mathit{plus} \cdot (\mathit{height} \times \mathit{height}) = (\!|\mathfrak{height2}|\!)_{\mathsf{L}}.$$

Appealing to uniqueness (23), we are left with the proof obligation

$$\mathit{plus} \cdot (\mathit{height} \times \mathit{height}) \cdot \mathsf{L}\,\mathit{in} = \mathfrak{height2}\,(\mathit{plus} \cdot (\mathit{height} \times \mathit{height})),$$

which is straightforward to discharge. $\square$

7.3. Conjugate fusion

Turning to the second fusion law, *conjugate fusion* allows us to fuse an adjoint fold $(\!|\Psi|\!)_{\mathsf{L}} : \mathbb{C}(\mathsf{L}\,(\mu\mathsf{F}), A)$ with a natural transformation $\sigma : \mathsf{L}' \dot{\to} \mathsf{L}$ to form another adjoint fold $(\!|\Psi'|\!)_{\mathsf{L}'} : \mathbb{C}(\mathsf{L}'\,(\mu\mathsf{F}), A)$. Why the name 'conjugate fusion'? Like a natural transformation relates two functors, a conjugate pair of natural transformations relates two adjunctions. Very briefly, given two adjunctions $\mathsf{L} \dashv \mathsf{R}$ and $\mathsf{L}' \dashv \mathsf{R}'$, the natural transformations $\sigma : \mathsf{L}' \dot{\to} \mathsf{L}$ and $\tau : \mathsf{R} \dot{\to} \mathsf{R}'$ are said to be *conjugate*, when the diagram

$$\begin{tikzcd}
\mathbb{C}(\mathsf{L}\,A, B) \arrow[r, phantom, "\cong"] \arrow[d, "\mathbb{C}(\sigma\,A,\,B)"'] & \mathbb{D}(A, \mathsf{R}\,B) \arrow[d, "\mathbb{D}(A,\,\tau\,B)"] \\
\mathbb{C}(\mathsf{L}'\,A, B) \arrow[r, phantom, "\cong"] & \mathbb{D}(A, \mathsf{R}'\,B)
\end{tikzcd}$$

commutes for all objects $A : \mathbb{D}$ and $B : \mathbb{C}$. One component of a conjugate pair uniquely determines the other: given a natural transformation $\sigma : \mathsf{L}' \dot{\to} \mathsf{L}$, there is a *unique* $\tau : \mathsf{R} \dot{\to} \mathsf{R}'$ so that σ and τ are conjugate, and vice versa.

Like fusion, conjugate fusion is subject to a condition:

$$\begin{aligned}
& (\!|\Psi|\!)_{\mathsf{L}} \cdot \sigma = (\!|\Psi'|\!)_{\mathsf{L}'} \\
\iff\; & \{\text{ uniqueness property (23) }\} \\
& (\!|\Psi|\!)_{\mathsf{L}} \cdot \sigma \cdot \mathsf{L}'\,in = \Psi'\,((\!|\Psi|\!)_{\mathsf{L}} \cdot \sigma) \\
\iff\; & \{\ \sigma \text{ is natural: } \mathsf{L}\,h \cdot \sigma = \sigma \cdot \mathsf{L}'\,h\ \} \\
& (\!|\Psi|\!)_{\mathsf{L}} \cdot \mathsf{L}\,in \cdot \sigma = \Psi'\,((\!|\Psi|\!)_{\mathsf{L}} \cdot \sigma) \\
\iff\; & \{\text{ computation (24) }\} \\
& \Psi\,(\!|\Psi|\!)_{\mathsf{L}} \cdot \sigma = \Psi'\,((\!|\Psi|\!)_{\mathsf{L}} \cdot \sigma) \\
\Longleftarrow\; & \{\text{ abstracting away from } (\!|\Psi|\!)_{\mathsf{L}}\ \} \\
& \forall f\ .\ \Psi\,f \cdot \sigma = \Psi'\,(f \cdot \sigma).
\end{aligned}$$

Consequently,

$$(\!|\Psi|\!)_{\mathsf{L}} \cdot \sigma = (\!|\Psi'|\!)_{\mathsf{L}'} \quad\Longleftarrow\quad \forall f\ .\ \Psi\,f \cdot \sigma = \Psi'\,(f \cdot \sigma). \tag{32}$$

Example 30. Conjugate fusion is more widely applicable than one might initially think. Let us demonstrate that (*Stack*, *Empty*, $\diamond$) is a monoid:

$$\begin{aligned}
& Empty \diamond s = s = s \diamond Empty, \\
& (s \diamond t) \diamond u = s \diamond (t \diamond u),
\end{aligned}$$

or, written in a point-free style,

$$\begin{aligned}
& cat \cdot (empty \vartriangle id) = id = cat \cdot (id \vartriangle empty), \\
& cat \cdot (cat \times id) = cat \cdot (id \times cat) \cdot assocr.
\end{aligned} \tag{33}$$

Here $assocr : (A \times B) \times C \cong A \times (B \times C)$ is the standard isomorphism between nested products. That *Empty* is left-neutral is a direct consequence of *cat*'s definition. Regarding right-neutrality, the crucial observation is that $id \vartriangle empty$ is a natural transformation of type $\mathsf{Id} \dot{\to} \mathsf{L}$ where $\mathsf{L} = - \times Stack$. We reason

$$\begin{aligned}
& cat \cdot (id \vartriangle empty) = id \\
\iff\; & \{\text{ definition of } cat \text{ and reflection (25) }\} \\
& (\!|\mathfrak{cat}|\!)_{\mathsf{L}} \cdot (id \vartriangle empty) = (\!|\lambda\,x\ .\ in \cdot \mathfrak{Stack}\,x|\!)_{\mathsf{Id}} \\
\iff\; & \{\text{ conjugate fusion (32): } id \vartriangle empty : \mathsf{Id} \dot{\to} \mathsf{L}\ \} \\
& \forall cat\ .\ \mathfrak{cat}\,cat \cdot (id \vartriangle empty) = in \cdot \mathfrak{Stack}\,(cat \cdot (id \vartriangle empty)).
\end{aligned}$$

The final obligation is straightforward to discharge.

To establish the associativity of *cat*, we adopt the same strategy as in Example 29: we show that both sides of (33) are equal to an adjoint fold. Starting with the right-hand side, we aim to derive a base function $\mathfrak{cat3}$ satisfying

$$(\!|\mathfrak{cat}|\!)_{\mathsf{L}} \cdot \sigma = (\!|\mathfrak{cat3}|\!)_{\mathsf{L}\circ\mathsf{L}} \quad \textbf{where} \quad \sigma = (id \times cat) \cdot assocr. \tag{34}$$

Again, conjugate fusion is applicable: σ is a natural transformation of type $\mathsf{L} \circ \mathsf{L} \dot{\to} \mathsf{L}$—the nesting of products translates into a composition of functors. Applying (32), we obtain

$$\forall cat\ .\ \mathfrak{cat}\,cat \cdot \sigma = \mathfrak{cat3}\,(cat \cdot \sigma), \tag{35}$$

from which we can synthesise the following definition of $\mathfrak{cat3}$:

$$\begin{aligned}
&\mathfrak{cat3} : \forall x \,.\, (\mathsf{L}\,(\mathsf{L}\,x) \to \mathit{Stack}) \to (\mathsf{L}\,(\mathsf{L}\,(\mathfrak{Stack}\,x)) && \to \mathit{Stack})\\
&\mathfrak{cat3} \quad \mathit{cat3} \quad ((\mathfrak{Empty}, y), z) && = \mathit{cat}\,(y, z)\\
&\mathfrak{cat3} \quad \mathit{cat3} \quad ((\mathfrak{Push}\,(a, x), y), z) && = \mathit{Push}\,(a, \mathit{cat3}\,((x, y), z)).
\end{aligned}$$

It remains to show that

$$\mathit{cat} \cdot (\mathit{cat} \times \mathit{id}) = (\!|\mathfrak{cat3}|\!)_{\mathsf{L}\circ\mathsf{L}},$$

which is left as an exercise to the reader. □

7.4. General fusion

The two previous laws are instances of a more general identity that manipulates arrows of type $\mathbb{C}(\mathsf{L}\,X, Y)$. Specifically, let $\alpha : \forall X : \mathbb{D}\,.\,\mathbb{C}(\mathsf{L}\,X, A) \to \mathbb{C}'(\mathsf{L}'\,X, A')$ be a natural transformation. Then

$$\alpha\,(\!|\Psi|\!)_{\mathsf{L}} = (\!|\Psi'|\!)_{\mathsf{L}'} \iff\!\!\!\Leftarrow \quad \alpha \cdot \Psi = \Psi' \cdot \alpha. \tag{36}$$

The correctness of the *general fusion* law rests on the naturality of α, that is, $\mathbb{C}'(\mathsf{L}'\,h, A') \cdot \alpha = \alpha \cdot \mathbb{C}(\mathsf{L}\,h, A)$, which unfolds to

$$\alpha\,f \cdot \mathsf{L}'\,h = \alpha\,(f \cdot \mathsf{L}\,h). \tag{37}$$

The straightforward proof of (36) follows the structure of the previous two, albeit on a higher level of abstraction.

$$\begin{aligned}
& \alpha\,(\!|\Psi|\!)_{\mathsf{L}} = (\!|\Psi'|\!)_{\mathsf{L}'}\\
\iff & \quad \{\text{ uniqueness property (23) }\}\\
& \alpha\,(\!|\Psi|\!)_{\mathsf{L}} \cdot \mathsf{L}'\,\mathit{in} = \Psi'\,(\alpha\,(\!|\Psi|\!)_{\mathsf{L}})\\
\iff & \quad \{\text{ naturality of } \alpha \text{ (37) }\}\\
& \alpha\,((\!|\Psi|\!)_{\mathsf{L}} \cdot \mathsf{L}\,\mathit{in}) = \Psi'\,(\alpha\,(\!|\Psi|\!)_{\mathsf{L}})\\
\iff & \quad \{\text{ computation (24) }\}\\
& \alpha\,(\Psi\,(\!|\Psi|\!)_{\mathsf{L}}) = \Psi'\,(\alpha\,(\!|\Psi|\!)_{\mathsf{L}})\\
\Longleftarrow & \quad \{\text{ abstracting away from } (\!|\Psi|\!)_{\mathsf{L}} \}\\
& \alpha \cdot \Psi = \Psi' \cdot \alpha
\end{aligned}$$

To see that fusion is an instance of (36), recall that $\mathbb{C}(\mathsf{L}\,X, h)$, that is, post-composing the arrow $h : \mathbb{C}(A, A')$, is a transformation of type $\mathbb{C}(\mathsf{L}\,X, A) \to \mathbb{C}(\mathsf{L}\,X, A')$ natural in X (Remark 1). The antecedent of (36) then specialises to the fusion condition:

$$\begin{aligned}
& \alpha \cdot \Psi = \Psi' \cdot \alpha\\
\iff & \quad \{\text{ set } \alpha\,A = \mathbb{C}(\mathsf{L}\,A, h) \}\\
& \mathbb{C}(\mathsf{L}\,(\mathsf{F}\,X), h) \cdot \Psi = \Psi' \cdot \mathbb{C}(\mathsf{L}\,X, h)\\
\iff & \quad \{\text{ definition of the hom-functor (1) }\}\\
& (h \cdot -) \cdot \Psi = \Psi' \cdot (h \cdot -)\\
\iff & \quad \{\text{ extensionality }\}\\
& \forall f\,.\, h \cdot \Psi\,f = \Psi'\,(h \cdot f).
\end{aligned}$$

Likewise, pre-composing a natural transformation $\mathsf{L}' \overset{\cdot}{\to} \mathsf{L}$ is a transformation of type $\mathbb{C}(\mathsf{L}\,X, A) \to \mathbb{C}(\mathsf{L}'\,X, A)$ natural in X. Consequently, conjugate fusion is an instance of (36), as well.

The antecedent of (36) is easy to satisfy if α has a left-inverse. In this case $\alpha \cdot \Psi = \Psi' \cdot \alpha \Longleftarrow \alpha \cdot \Psi \cdot \alpha^{\circ} = \Psi'$ and consequently $\alpha\,(\!|\Psi|\!)_{\mathsf{L}} = (\!|\alpha \cdot \Psi \cdot \alpha^{\circ}|\!)_{\mathsf{L}'}$. Now, a prominent example of a natural isomorphism between hom-sets is an adjunct, and indeed, the main result of Section 4.1, that an adjoint fold can be reduced to a standard fold $\phi\,(\!|\Psi|\!)_{\mathsf{L}} = (\!|\phi \cdot \Psi \cdot \phi^{\circ}|\!)_{\mathsf{Id}}$, is an instance of (36). In fact, we can generalise the identity slightly: let $\phi_1 : \mathsf{L}_1 \dashv \mathsf{R}_1$ and $\phi_2 : \mathsf{L}_2 \dashv \mathsf{R}_2$ be two adjunctions, then

$$\phi_1\,(\!|\Psi|\!)_{\mathsf{L}_1\circ\mathsf{L}_2} = (\!|\phi_1 \cdot \Psi \cdot \phi_1^{\circ}|\!)_{\mathsf{L}_2}.$$

Here ϕ_1 is used as a transformation of type $\mathbb{C}(\mathsf{L}_1\,(\mathsf{L}_2\,A), B) \to \mathbb{D}(\mathsf{L}_2\,A, \mathsf{R}_1\,B)$ that is natural in A.

7.5. Base functor fusion

In order to formulate the last fusion law, base functor fusion, we have to turn μ into a higher-order functor of type $\mathbb{C}^{\mathbb{C}} \to \mathbb{C}$. The object part of this functor maps a functor to its initial algebra. (This is a bit loose as this is only well defined for functors that have an initial algebra.) The arrow part maps a natural transformation $\alpha : \mathsf{F} \dot{\to} \mathsf{G}$ to an arrow $\mu\alpha : \mathbb{C}(\mu\mathsf{F}, \mu\mathsf{G})$. It is defined as

$$\mu\alpha = (\!|in \cdot \alpha\,(\mu\mathsf{G})|\!)\,. \tag{38}$$

Definition 7. In Haskell, the functorial action of μ on arrows can be programmed using explicit recursion.

$$\begin{aligned}
&\mu : (\mathit{Functor}\, f) \Rightarrow (\forall x\,.\, f\,x \to g\,x) \to (\mu f \to \mu g)\\
&\mu \quad \alpha \quad = \mathit{In} \cdot \alpha \cdot \mathit{fmap}\,(\mu\,\alpha) \cdot in^{\circ}
\end{aligned}$$

As α is natural, it can be invoked either after or before the recursive call of μ, resulting in different type constraints (*Functor f* or *Functor g*). Quite arbitrarily, we have picked the first choice. □

Using μ we can express the *base functor fusion law*: let $\alpha : \mathsf{F} \dot{\to} \mathsf{G}$ be a natural transformation – we might call α a 'base changer' – then

$$(\!|\Psi|\!)_{\mathsf{L}} \cdot \mathsf{L}\,(\mu\alpha) = (\!|\lambda\, x\,.\,\Psi\, x \cdot \mathsf{L}\,\alpha|\!)_{\mathsf{L}}. \tag{39}$$

The law states that an adjoint fold $(\!|\Psi|\!)_{\mathsf{L}} : \mathbb{C}(\mathsf{L}\,(\mu\mathsf{G}), A)$ can be fused with an arrow $\mathsf{L}\,(\mu\alpha) : \mathbb{C}(\mathsf{L}\,(\mu\mathsf{F}), \mathsf{L}\,(\mu\mathsf{G}))$ to form another adjoint fold $(\!|\lambda\, x\,.\,\Psi\, x \cdot \mathsf{L}\,\alpha|\!)_{\mathsf{L}} : \mathbb{C}(\mathsf{L}\,(\mu\mathsf{F}), A)$.

The proof of the fusion law relies on the fact that the initial algebra $in : \mathbb{C}(\mathsf{F}\,(\mu\mathsf{F}), \mu\mathsf{F})$ is natural in the base functor F—the algebra *in* is an example of a so-called *higher-order natural transformation* [26].

$$\mu\alpha \cdot in_{\mathsf{F}} = in_{\mathsf{G}} \cdot \alpha\,(\mu\mathsf{G}) \cdot \mathsf{F}\,(\mu\alpha). \tag{40}$$

Here $\lambda\,\alpha\,.\,\alpha\,(\mu\mathsf{G}) \cdot \mathsf{F}\,(\mu\alpha) = \lambda\,\alpha\,.\,\mathsf{G}\,(\mu\alpha) \cdot \alpha\,(\mu\mathsf{F})$ is the arrow part of the higher-order functor $\Lambda\,\mathsf{F}\,.\,\mathsf{F}\,(\mu\mathsf{F})$. The naturality property of *in* (40) is an immediate consequence of the computation law for standard folds.

$$\begin{aligned}
&\mu\alpha \cdot in_{\mathsf{F}}\\
=\;&\{\text{ definition of } \mu \text{ (38) }\}\\
&(\!|in_{\mathsf{G}} \cdot \alpha\,(\mu\mathsf{G})|\!) \cdot in_{\mathsf{F}}\\
=\;&\{\text{ computation law for standard folds: } (\!|f|\!) \cdot in = f \cdot \mathsf{F}\,(\!|f|\!) \text{ }\}\\
&in_{\mathsf{G}} \cdot \alpha\,(\mu\mathsf{G}) \cdot \mathsf{F}\,(\!|in_{\mathsf{G}} \cdot \alpha\,(\mu\mathsf{G})|\!)\\
=\;&\{\text{ definition of } \mu \text{ (38) }\}\\
&in_{\mathsf{G}} \cdot \alpha\,(\mu\mathsf{G}) \cdot \mathsf{F}\,(\mu\alpha)
\end{aligned}$$

For the proof of base functor fusion (39), we appeal to the uniqueness property (23) and reason

$$\begin{aligned}
&(\!|\Psi|\!)_{\mathsf{L}} \cdot \mathsf{L}\,(\mu\alpha) \cdot \mathsf{L}\,in_{\mathsf{F}}\\
=\;&\{\text{ L functor }\}\\
&(\!|\Psi|\!)_{\mathsf{L}} \cdot \mathsf{L}\,(\mu\alpha \cdot in_{\mathsf{F}})\\
=\;&\{\text{ } in \text{ is natural (40) }\}\\
&(\!|\Psi|\!)_{\mathsf{L}} \cdot \mathsf{L}\,(in_{\mathsf{G}} \cdot \alpha\,(\mu\mathsf{G}) \cdot \mathsf{F}\,(\mu\alpha))\\
=\;&\{\text{ } \alpha \text{ is natural: } \mathsf{G}\,h \cdot \alpha\,A = \alpha\,B \cdot \mathsf{F}\,h \text{ }\}\\
&(\!|\Psi|\!)_{\mathsf{L}} \cdot \mathsf{L}\,(in_{\mathsf{G}} \cdot \mathsf{G}\,(\mu\alpha) \cdot \alpha\,(\mu\mathsf{F}))\\
=\;&\{\text{ L functor }\}\\
&(\!|\Psi|\!)_{\mathsf{L}} \cdot \mathsf{L}\,in_{\mathsf{G}} \cdot \mathsf{L}\,(\mathsf{G}\,(\mu\alpha)) \cdot \mathsf{L}\,(\alpha\,(\mu\mathsf{F}))\\
=\;&\{\text{ computation (24) }\}\\
&\Psi\,(\!|\Psi|\!)_{\mathsf{L}} \cdot \mathsf{L}\,(\mathsf{G}\,(\mu\alpha)) \cdot \mathsf{L}\,(\alpha\,(\mu\mathsf{F}))\\
=\;&\{\text{ } \Psi \text{ is natural: } \Psi\,f \cdot \mathsf{L}\,(\mathsf{G}\,h) = \Psi\,(f \cdot \mathsf{L}\,h) \text{ }\}\\
&\Psi\,((\!|\Psi|\!)_{\mathsf{L}} \cdot \mathsf{L}\,(\mu\alpha)) \cdot \mathsf{L}\,(\alpha\,(\mu\mathsf{F})).
\end{aligned}$$

Example 31. The applicability of base functor fusion very much depends on the shape of the base functor, as it determines which functions can be expressed as arrows of the form $\mu\alpha$. In the case of stacks, we can essentially express *map*-like functions.

$$\begin{aligned}
&\mathit{map} : (\mathit{Nat} \to \mathit{Nat}) \to (\mu\mathfrak{Stack} \to \mu\mathfrak{Stack})\\
&\mathit{map}\;\; f \quad = \mu\,(\mathfrak{map}\, f)
\end{aligned}$$

$$\begin{aligned}
&\mathfrak{map} : \forall x\,.\,(\mathit{Nat} \to \mathit{Nat}) \to (\mathfrak{Stack}\, x \to \mathfrak{Stack}\, x)\\
&\mathfrak{map} \quad f \quad (\mathfrak{Empty}) \quad = \mathfrak{Empty}\\
&\mathfrak{map} \quad f \quad (\mathfrak{Push}\,(n, s)) = \mathfrak{Push}\,(f\,n, s)
\end{aligned}$$

Table 5
Calculational properties of unfolds.

Uniqueness property:

$$x = [\![\Psi]\!]_{\mathsf{R}} \iff \mathsf{R}\, out \cdot x = \Psi\, x.$$

Computation rule:

$$\mathsf{R}\, out \cdot [\![\Psi]\!]_{\mathsf{R}} = \Psi\, [\![\Psi]\!]_{\mathsf{R}}.$$

Reflection:

$$[\![\Psi]\!]_{\mathsf{R}} = id \iff \Psi\, id = \mathsf{R}\, out.$$

Fusion: Let $h : \mathbb{D}(B, A)$, then

$$[\![\Phi]\!]_{\mathsf{R}} \cdot h = [\![\Psi]\!]_{\mathsf{R}} \Longleftarrow \forall x\,.\, \Phi\, x \cdot h = \Psi\, (x \cdot h).$$

Conjugate fusion: Let $\tau : \mathsf{R} \dot{\to} \mathsf{R}'$, then

$$\tau \cdot [\![\Psi]\!]_{\mathsf{R}} = [\![\Psi']\!]_{\mathsf{R}'} \Longleftarrow \forall x\,.\, \tau \cdot \Psi\, x = \Psi'\, (\tau \cdot x).$$

General fusion: Let $\alpha : \forall X : \mathbb{C}\,.\, \mathbb{D}(A, \mathsf{R}\, X) \to \mathbb{D}'(A', \mathsf{R}'\, X)$, then

$$\alpha\, [\![\Psi]\!]_{\mathsf{R}} = [\![\Psi']\!]_{\mathsf{R}'} \Longleftarrow \alpha \cdot \Psi = \Psi' \cdot \alpha.$$

Base functor fusion: Let $\alpha : \mathsf{G} \dot{\to} \mathsf{F}$, then

$$\mathsf{R}\, (\nu\alpha) \cdot [\![\Psi]\!]_{\mathsf{R}} = [\![\lambda\, x\,.\, \mathsf{R}\, \alpha \cdot \Psi\, x]\!]_{\mathsf{R}},$$

where $\nu\alpha = [\![\alpha\, (\nu\mathsf{G}) \cdot out]\!]$.

Note that $\mathfrak{map}\, f : \mathfrak{Stack} \dot{\to} \mathfrak{Stack}$ has the required naturality property. Since μ is a functor and since $\mathfrak{map}\, id = id$ and $\mathfrak{map}\, (f \cdot g) = \mathfrak{map}\, f \cdot \mathfrak{map}\, g$, we can immediately conclude that

$$\begin{aligned} map\, id &= id, \\ map\, (f \cdot g) &= map\, f \cdot map\, g. \end{aligned}$$

(If we generalise *Stack* to List, then $\mathfrak{map}$ generalises to the arrow part of $\mathfrak{List}$ and *map* to the arrow part of List.) Using base functor fusion we can shift an application of *map f* into an adjoint fold, for example

$$(\!|\mathfrak{cat}|\!)_{\mathsf{L}} \cdot \mathsf{L}\, (map\, f) = (\!|\lambda\, cat\,.\, \mathfrak{cat}\, cat \cdot \mathsf{L}\, (\mathfrak{map}\, f)|\!)_{\mathsf{L}}.$$

Now, if we augment $\mathfrak{Stack}$ by a $\mathfrak{Skip}$ constructor (we are drawing inspiration from stream fusion [17] here)

$$\textbf{data}\ \mathfrak{Stack}\ stack = \mathfrak{Empty} \mid \mathfrak{Skip}\ stack \mid \mathfrak{Push}\ (Nat, stack),$$

then we can also express filter-like functions as arrows of the form $\mu\alpha$.

$$\begin{aligned} &filter : (Nat \to Bool) \to (\mu\mathfrak{Stack} \to \mu\mathfrak{Stack}) \\ &filter\ \ p = \mu\, (\mathfrak{filter}\, p) \end{aligned}$$

$$\begin{aligned} &\mathfrak{filter} : \forall x\,.\, (Nat \to Bool) \to (\mathfrak{Stack}\, x \to \mathfrak{Stack}\, x) \\ &\mathfrak{filter}\ \ p\ \ (\mathfrak{Empty}) = \mathfrak{Empty} \\ &\mathfrak{filter}\ \ p\ \ (\mathfrak{Skip}\, s) = \mathfrak{Skip}\, s \\ &\mathfrak{filter}\ \ p\ \ (\mathfrak{Push}\, (n, s)) = \textbf{if}\ p\, n\ \textbf{then}\ \mathfrak{Push}\, (n, s)\ \textbf{else}\ \mathfrak{Skip}\, s \end{aligned}$$

Let $true\, x = True$ and $(p \wedge q)\, x = p\, x \wedge q\, x$. Since $\mathfrak{filter}\, true = id$ and $\mathfrak{filter}\, (p \wedge q) = \mathfrak{filter}\, p \cdot \mathfrak{filter}\, q$, we can immediately conclude that

$$\begin{aligned} filter\, true &= id, \\ filter\, (p \wedge q) &= filter\, p \cdot filter\, q. \end{aligned}$$

Filtering distributes over concatenation:

$$filter\, p\, (s \diamond t) = filter\, p\, s \diamond filter\, p\, t.$$

The proof, left as an exercise to the reader, involves all three types of fusion: vanilla, conjugate and base functor fusion. □

So far we have been occupied with folds. Table 5 lists the dual laws for unfolds.

8. Type fusion

In the previous section we have discussed a variety of laws for fusing an adjoint (un)fold with a context. In this section we climb up one step on the abstraction ladder and lift fusion to the realm of objects and functors. *Type fusion* allows us to fuse an application of a functor with an initial algebra to form another initial algebra: $\mathsf{L}\,(\mu\mathsf{F}) \cong \mu\mathsf{G}$. Like general fusion (36), type fusion is subject to a condition: $\mathsf{L} \circ \mathsf{F} \cong \mathsf{G} \circ \mathsf{L}$. The isomorphism allows us to push the functor L into the fixed point: $\mathsf{L}\,(\mu\mathsf{F}) \cong \mathsf{L}\,(\mathsf{F}\,(\mu\mathsf{F})) \cong \mathsf{G}\,(\mathsf{L}\,(\mu\mathsf{F}))$. This simple calculation shows that $\mathsf{L}\,(\mu\mathsf{F})$ is a fixed point of G. If L is furthermore a left adjoint, then $\mathsf{L}\,(\mu\mathsf{F})$ is even the least fixed point:

Let $\mathbb{C}$ and $\mathbb{D}$ be categories, let $\mathsf{L} \dashv \mathsf{R}$ be an adjoint pair of functors $\mathsf{L} : \mathbb{C} \leftarrow \mathbb{D}$ and $\mathsf{R} : \mathbb{C} \rightarrow \mathbb{D}$, and let $\mathsf{F} : \mathbb{D} \rightarrow \mathbb{D}$ and $\mathsf{G} : \mathbb{C} \rightarrow \mathbb{C}$ be two endofunctors. Then

$$\mathsf{L}\,(\mu\mathsf{F}) \cong \mu\mathsf{G} \quad\Longleftarrow\quad \mathsf{L}\circ\mathsf{F} \cong \mathsf{G}\circ\mathsf{L}, \tag{41}$$

$$\nu\mathsf{F} \cong \mathsf{R}\,(\nu\mathsf{G}) \quad\Longleftarrow\quad \mathsf{F}\circ\mathsf{R} \cong \mathsf{R}\circ\mathsf{G}. \tag{42}$$

We show type fusion for initial algebras (41), the corresponding statement for final coalgebras follows by duality. Assuming $\mathfrak{s} : \mathsf{L}\circ\mathsf{F} \cong \mathsf{G}\circ\mathsf{L} : \mathfrak{s}^\circ$, the witnesses of the isomorphism $\mathfrak{f} : \mathsf{L}\,(\mu\mathsf{F}) \cong \mu\mathsf{G} : \mathfrak{f}^\circ$ are given as solutions of (adjoint) fixed point equations:

$$\mathfrak{f}\cdot\mathsf{L}\,\mathit{in} = \mathit{in}\cdot\mathsf{G}\,\mathfrak{f}\cdot\mathfrak{s} \quad\text{and}\quad \mathfrak{f}^\circ\cdot\mathit{in} = \mathsf{L}\,\mathit{in}\cdot\mathfrak{s}^\circ\cdot\mathsf{G}\,\mathfrak{f}^\circ. \tag{43}$$

Note that $\mathfrak{f}$ is an algebraic adjoint fold, $\mathfrak{f} = \llparenthesis\lambda\,x\,.\,\mathit{in}\cdot\mathsf{G}\,x\cdot\mathfrak{s}\rrparenthesis_{\mathsf{L}}$ (Section 4.3), while $\mathfrak{f}^\circ$ is a standard fold, $\mathfrak{f}^\circ = \llparenthesis\lambda\,x\,.\,\mathsf{L}\,\mathit{in}\cdot\mathfrak{s}^\circ\cdot\mathsf{G}\,x\rrparenthesis_{\mathsf{Id}} = \llparenthesis\mathsf{L}\,\mathit{in}\cdot\mathfrak{s}^\circ\rrparenthesis$. The diagram below summarises the type information.

$$\begin{tikzcd}
& \mathsf{G}\,(\mathsf{L}\,(\mu\mathsf{F})) \arrow[dl, shift left, "\mathfrak{s}^\circ"] \arrow[dr, shift left, "\mathsf{G}\,\mathfrak{f}"] & \\
\mathsf{L}\,(\mathsf{F}\,(\mu\mathsf{F})) \arrow[ur, shift left, "\mathfrak{s}"] \arrow[d, "\mathsf{L}\,\mathit{in}"', "\mathsf{L}\,\mathit{in}"] & & \mathsf{G}\,(\mu\mathsf{G}) \arrow[ul, shift left, "\mathsf{G}\,\mathfrak{f}^\circ"] \arrow[d, "\mathit{in}"', "\mathit{in}"] \\
\mathsf{L}\,(\mu\mathsf{F}) \arrow[rr, shift right, "\mathfrak{f}"'] & & \mu\mathsf{G} \arrow[ll, shift right, "\mathfrak{f}^\circ"']
\end{tikzcd}$$

The identities $\mathfrak{f}\cdot\mathfrak{f}^\circ = \mathit{id}$ and $\mathfrak{f}^\circ\cdot\mathfrak{f} = \mathit{id}$ are instances of a more general result which we state first.

Now that we have related fixed-points of functors, the next step is to relate the corresponding (adjoint) (un)folds. The isomorphisms $\mathfrak{f}$ and $\mathfrak{f}^\circ$ can be seen as *representation changers*: the nested type $\mathsf{L}\,(\mu\mathsf{F})$ is represented by the initial algebra $\mu\mathsf{G}$. The following identities capture the effect of the representation change on arrows: algebraic adjoint folds become standard folds!

$$\llparenthesis\lambda\,x\,.\,a\cdot\mathsf{G}\,x\cdot\mathfrak{s}\rrparenthesis_{\mathsf{L}} = \llparenthesis a\rrparenthesis\cdot\mathfrak{f} \quad\text{and}\quad \llparenthesis\lambda\,x\,.\,a\cdot\mathsf{G}\,x\cdot\mathfrak{s}\rrparenthesis_{\mathsf{L}}\cdot\mathfrak{f}^\circ = \llparenthesis a\rrparenthesis, \tag{44}$$

$$\llbracket\lambda\,x\,.\,\mathfrak{s}\cdot\mathsf{F}\,x\cdot c\rrbracket_{\mathsf{R}} = \mathfrak{f}\cdot\llbracket c\rrbracket \quad\text{and}\quad \llbracket\lambda\,x\,.\,\mathfrak{s}\cdot\mathsf{F}\,x\cdot c\rrbracket_{\mathsf{R}}\cdot\mathfrak{f}^\circ = \llbracket c\rrbracket.$$

Again, we show the statement only for initial algebras (44). Since $\mathfrak{f}$ changes both the adjoint functor and the base functor, neither conjugate nor base functor fusion is applicable. Instead, we appeal to the uniqueness property (23).

$$\begin{aligned}
&\llparenthesis a\rrparenthesis\cdot\mathfrak{f} = \llparenthesis\lambda\,x\,.\,a\cdot\mathsf{G}\,x\cdot\mathfrak{s}\rrparenthesis_{\mathsf{L}}\\
\Longleftrightarrow\quad&\{\text{ uniqueness property (23) }\}\\
&\llparenthesis a\rrparenthesis\cdot\mathfrak{f}\cdot\mathsf{L}\,\mathit{in} = a\cdot\mathsf{G}\,(\llparenthesis a\rrparenthesis\cdot\mathfrak{f})\cdot\mathfrak{s}
\end{aligned}$$

We argue

$$\begin{aligned}
&\llparenthesis a\rrparenthesis\cdot\mathfrak{f}\cdot\mathsf{L}\,\mathit{in}\\
=\quad&\{\text{ definition of }\mathfrak{f}\text{ (43) }\}\\
&\llparenthesis a\rrparenthesis\cdot\mathit{in}\cdot\mathsf{G}\,\mathfrak{f}\cdot\mathfrak{s}\\
=\quad&\{\text{ computation law for standard folds: }\llparenthesis a\rrparenthesis\cdot\mathit{in} = a\cdot\mathsf{G}\,\llparenthesis a\rrparenthesis\ \}\\
&a\cdot\mathsf{G}\,\llparenthesis a\rrparenthesis\cdot\mathsf{G}\,\mathfrak{f}\cdot\mathfrak{s}\\
=\quad&\{\ \mathsf{G}\text{ functor }\}\\
&a\cdot\mathsf{G}\,(\llparenthesis a\rrparenthesis\cdot\mathfrak{f})\cdot\mathfrak{s}.
\end{aligned}$$

The proof of the second part is nicely symmetric.

$$\begin{aligned}
&\llparenthesis\lambda\,x\,.\,a\cdot\mathsf{G}\,x\cdot\mathfrak{s}\rrparenthesis_{\mathsf{L}}\cdot\mathfrak{f}^\circ = \llparenthesis a\rrparenthesis\\
\Longleftrightarrow\quad&\{\text{ uniqueness property of standard folds (2) }\}\\
&\llparenthesis\lambda\,x\,.\,a\cdot\mathsf{G}\,x\cdot\mathfrak{s}\rrparenthesis_{\mathsf{L}}\cdot\mathfrak{f}^\circ\cdot\mathit{in} = a\cdot\mathsf{G}\,(\llparenthesis\lambda\,x\,.\,a\cdot\mathsf{G}\,x\cdot\mathfrak{s}\rrparenthesis_{\mathsf{L}}\cdot\mathfrak{f}^\circ)
\end{aligned}$$

The proof is

$$\begin{aligned}
& (\!|\lambda\, x\, .\, a \cdot \mathsf{G}\, x \cdot \mathfrak{s}|\!)_{\mathsf{L}} \cdot \mathfrak{f}^{\circ} \cdot in \\
= \quad & \{\text{ definition of } \mathfrak{f}^{\circ}\ (43)\ \} \\
& (\!|\lambda\, x\, .\, a \cdot \mathsf{G}\, x \cdot \mathfrak{s}|\!)_{\mathsf{L}} \cdot \mathsf{L}\, in \cdot \mathfrak{s}^{\circ} \cdot \mathsf{G}\, \mathfrak{f}^{\circ} \\
= \quad & \{\text{ computation } (24)\ \} \\
& a \cdot \mathsf{G}\, (\!|\lambda\, x\, .\, a \cdot \mathsf{G}\, x \cdot \mathfrak{s}|\!)_{\mathsf{L}} \cdot \mathfrak{s} \cdot \mathfrak{s}^{\circ} \cdot \mathsf{G}\, \mathfrak{f}^{\circ} \\
= \quad & \{\text{ assumption: } \mathfrak{s} \cdot \mathfrak{s}^{\circ} = id\ \} \\
& a \cdot \mathsf{G}\, (\!|\lambda\, x\, .\, a \cdot \mathsf{G}\, x \cdot \mathfrak{s}|\!)_{\mathsf{L}} \cdot \mathsf{G}\, \mathfrak{f}^{\circ} \\
= \quad & \{\ \mathsf{G} \text{ functor }\} \\
& a \cdot \mathsf{G}\, ((\!|\lambda\, x\, .\, a \cdot \mathsf{G}\, x \cdot \mathfrak{s}|\!)_{\mathsf{L}} \cdot \mathfrak{f}^{\circ}).
\end{aligned}$$

Interestingly, to convert back and fro we only need one direction of the assumption $\mathsf{L} \circ \mathsf{F} \cong \mathsf{G} \circ \mathsf{L}$.

We are now in a position to show that $\mathfrak{f}$ and $\mathfrak{f}^{\circ}$ are actually inverses. $\mathfrak{f} \cdot \mathfrak{f}^{\circ} = id_{\mu\mathsf{G}}$: We reason as follows.

$$\begin{aligned}
& (\!|\lambda\, x\, .\, in \cdot \mathsf{G}\, x \cdot \mathfrak{s}|\!)_{\mathsf{L}} \cdot \mathfrak{f}^{\circ} \\
= \quad & \{\text{ representation change } (44)\ \} \\
& (\!|in|\!) \\
= \quad & \{\text{ reflection for standard folds: } (\!|in|\!) = id\ \} \\
& id
\end{aligned}$$

$\mathfrak{f}^{\circ} \cdot \mathfrak{f} = id_{\mathsf{L}\,(\mu\mathsf{F})}$: For the reverse direction, we argue

$$\begin{aligned}
& (\!|\mathsf{L}\, in \cdot \mathfrak{s}^{\circ}|\!) \cdot \mathfrak{f} \\
= \quad & \{\text{ representation change } (44)\ \} \\
& (\!|\lambda\, x\, .\, \mathsf{L}\, in \cdot \mathfrak{s}^{\circ} \cdot \mathsf{G}\, x \cdot \mathfrak{s}|\!)_{\mathsf{L}} \\
= \quad & \{\text{ claim: see below }\} \\
& id.
\end{aligned}$$

The claim amounts to a special *reflection law* which is worth making explicit:

$$(\!|\lambda\, x\, .\, \mathsf{L}\, in \cdot \mathfrak{s}^{\circ} \cdot \mathsf{G}\, x \cdot \mathfrak{s}|\!)_{\mathsf{L}} = id. \tag{45}$$

For the proof we invoke reflection (25) and discharge the obligation using the assumption $\mathfrak{s}^{\circ} \cdot \mathfrak{s} = id$.

In Section 5 we have introduced a smörgåsbord of adjunctions. In the remainder of this section we instantiate type fusion to the various adjunctions—sometimes with surprising results.

8.1. *Identity:* $\mathsf{Id} \dashv \mathsf{Id}$

For the identity adjunction $\mathsf{Id} \dashv \mathsf{Id}$ type fusion simplifies to $\mu\mathsf{F} \cong \mu\mathsf{G} \Longleftarrow \mathsf{F} \cong \mathsf{G}$ and $\nu\mathsf{F} \cong \nu\mathsf{G} \Longleftarrow \mathsf{F} \cong \mathsf{G}$. In words, the functors $\mu, \nu : \mathbb{C}^{\mathbb{C}} \to \mathbb{C}$ preserve isomorphisms. This is, in fact, true of every functor. The representation changers $\mathfrak{f}$ and $\mathfrak{f}^{\circ}$ simplify to functor applications, $\mathfrak{f} = \mu\mathfrak{s}$ and $\mathfrak{f}^{\circ} = \mu\mathfrak{s}^{\circ}$, and likewise for ν. We can use this instance of type fusion to show, for example, that cons and snoc lists are isomorphic [5].

8.2. *Isomorphism and equivalence of categories*

If the functor R is an equivalence of categories with unit $\eta : \mathsf{Id} \cong \mathsf{R} \circ \mathsf{L}$, then $\mathsf{L} \circ \mathsf{F} \circ \eta : \mathsf{L} \circ \mathsf{F} \cong \mathsf{L} \circ \mathsf{F} \circ \mathsf{R} \circ \mathsf{L}$. Consequently, type fusion implies

$$\mathsf{L}\,(\mu\mathsf{F}) \cong \mu(\mathsf{L} \circ \mathsf{F} \circ \mathsf{R}), \quad \text{and dually} \quad \nu(\mathsf{R} \circ \mathsf{G} \circ \mathsf{L}) \cong \mathsf{R}\,(\nu\mathsf{G}).$$

Instead of forming the initial algebra in $\mathbb{D}$ and mapping the result to $\mathbb{C}$, we can form the initial algebra in $\mathbb{C}$ by 'going round in a circle'. The functor $\mathsf{L} \circ \mathsf{F} \circ \mathsf{R}$ was called the canonical control functor in Section 4.3.

8.3. *Currying:* $- \times X \dashv (-)^X$

Example 32. Concatenation was one of our motivating examples for adjoint folds. Somewhat ironically, using the representation changers of type fusion we can implement stack concatenation in terms of a standard fold. We have laid the groundwork in Example 13, where we calculated a functor $\mathfrak{Stack}'$ satisfying $\mathsf{L} \circ \mathfrak{Stack} \cong \mathfrak{Stack}' \circ \mathsf{L}$ (the natural transformation $\mathfrak{s}$ was called $scat^{\leftrightarrow}$ there). For this isomorphism the representation changer $\mathfrak{f} = (\!|\lambda\, x\ .\ in \cdot \mathfrak{Stack}'\, x \cdot scat^{\leftrightarrow}|\!)_{\mathsf{L}}$ unfolds to

$$\begin{array}{lll} \mathfrak{f} : \mathsf{L}\,(\mu\mathfrak{Stack}) & \to & \mu\mathfrak{Stack}' \\ \mathfrak{f}\quad (In\ \mathfrak{Empty}, t) & = & In\,(\mathfrak{Empty}'\ t) \\ \mathfrak{f}\quad (In\,(\mathfrak{Push}\,(n, s)), t) & = & In\,(\mathfrak{Push}'\,(n, \mathfrak{f}\,(s, t))). \end{array}$$

In Example 13 we also formulated stack concatenation as an algebraic adjoint fold: $cat = (\!|\lambda\, x\ .\ scat^{\triangledown} \cdot \mathfrak{Stack}'\, x \cdot scat^{\leftrightarrow}|\!)_{\mathsf{L}}$. Relation (44) immediately gives us $cat = (\!|scat^{\triangledown}|\!) \cdot \mathfrak{f}$. □

Let $\mathsf{L} = - \times X$. Using similar calculations as in Example 13 we can fuse all *linear datatypes*, whose base functors are polynomials of degree at most 1. The reasoning is based on basic properties of $+$ and $\times$:

$$\begin{array}{lllllll} (A+B)\times X & \cong & A\times X + B\times X & \qquad & \mathsf{L}\,(A+B) & \cong & \mathsf{L}\,A + \mathsf{L}\,B \\ (A\times B)\times X & \cong & A\times(B\times X) & & \mathsf{L}\,(A\times B) & \cong & A\times \mathsf{L}\,B \\ (A\times B)\times X & \cong & (A\times X)\times B & & \mathsf{L}\,(A\times B) & \cong & \mathsf{L}\,A\times B. \end{array}$$

The following table relates base functors:

$$\begin{array}{lllllll} \mathsf{F}\,Y & = & Y & \qquad & \mathsf{G}\,Z & = & Z \\ \mathsf{F}\,Y & = & C & & \mathsf{G}\,Z & = & \mathsf{L}\,C \\ \mathsf{F}\,Y & = & \mathsf{F}_1\,Y + \mathsf{F}_2\,Y & & \mathsf{G}\,Z & = & \mathsf{G}_1\,Z + \mathsf{G}_2\,Z \\ \mathsf{F}\,Y & = & C\times \mathsf{F}_1\,Y & & \mathsf{G}\,Z & = & C\times \mathsf{G}_1\,Z \\ \mathsf{F}\,Y & = & \mathsf{F}_1\,Y\times C & & \mathsf{G}\,Z & = & \mathsf{G}_1\,Z\times C. \end{array}$$

Each row in the table satisfies $\mathsf{L} \circ \mathsf{F} \cong \mathsf{G} \circ \mathsf{L}$ if $\mathsf{L} \circ \mathsf{F}_i \cong \mathsf{G}_i \circ \mathsf{L}$. Note that the F functors are necessarily polynomials of degree at most 1, as the combining forms for products ensure that one component is a constant type.

8.4. *Mutual value recursion:* $(+) \dashv \Delta \dashv (\times)$

The unit of the adjunction $\Delta \dashv (\times)$ is the so-called *diagonal arrow* $\delta = id \vartriangle id$. Recall that the unit is a natural transformation $\delta : \mathsf{Id} \dot{\to} (\times) \circ \Delta$. In this particular case the naturality property unfolds to $\delta \cdot h = (h \times h) \cdot \delta$. Now, the product of categories is itself a categorical product. Its unit is the diagonal functor, which consequently satisfies

$$\Delta \circ \mathsf{H} = (\mathsf{H} \times \mathsf{H}) \circ \Delta. \tag{46}$$

Thus, the precondition of type fusion is trivially satisfied and we may immediately conclude that

$$\Delta(\mu\mathsf{F}) \cong \mu(\mathsf{F}\times\mathsf{F}), \quad \text{and dually} \quad \nu(\mathsf{G}\times\mathsf{G}) \cong \Delta(\nu\mathsf{G}).$$

In fact, the statement can be strengthened: $\langle \mu\mathsf{F}, \mu\mathsf{G}\rangle \cong \mu(\mathsf{F}\times\mathsf{G})$ and likewise for final coalgebras. So a pair of two independent recursive datatypes is an extreme case of a pair of datatypes defined by mutual recursion. Consequently, adjoint folds of type $\Delta(\mu\mathsf{F}) \to A$ are really standard folds of type $\mu(\mathsf{F}\times\mathsf{F}) \to A$ in disguise! Since the natural transformations $\mathfrak{s}$ and $\mathfrak{s}^{\circ}$ are identities – the naturality property (46) is an equality, not an isomorphism – the conversion functions simplify somewhat: $\mathfrak{f} = (\!|\lambda\, x\ .\ in \cdot (\mathsf{F}\times\mathsf{F})\, x|\!)_{\Delta}$ and $\mathfrak{f}^{\circ} = (\!|\lambda\, x\ .\ \Delta in \cdot (\mathsf{F}\times\mathsf{F})\, x|\!)_{\mathsf{Id}} = (\!|\Delta in|\!)$. The special reflection law (45) can be simplified accordingly:

$$(\!|\lambda\,\langle x_1, x_2\rangle\ .\ \langle in \cdot \mathsf{F}\,x_1,\ in \cdot \mathsf{F}\,x_2\rangle|\!)_{\Delta} = id.$$

We obtain the mutu-ID law of [21].

8.5. *Type firstification:* $\mathsf{Lsh}_X \dashv (-X) \dashv \mathsf{Rsh}_X$

We have encountered two ways of defining sequences of natural numbers: *Stack* (Example 1) and $\mathsf{List}\,Nat$ (Example 16). We can use type fusion to show that these types are actually isomorphic. The transformation of $\mathsf{List}\,Nat$ into *Stack* can be seen as an instance of λ-*dropping* [18] or *firstification* [39] on the type level: a fixed point of a higher-order functor is reduced to a fixed-point of a first-order functor.

Example 33. Let us show that $\mathsf{List}\,Nat \cong Stack$. The underlying adjunction is type application $\mathsf{App}_{Nat} \dashv \mathsf{Rsh}_{Nat}$. Type fusion is directly applicable and we are left with showing the precondition $\mathsf{App}_{Nat} \circ \mathfrak{List} \cong \mathfrak{Stack} \circ \mathsf{App}_{Nat}$.

$$\begin{array}{ll} & (-\,Nat) \circ \mathfrak{List} \\ = & \{\ \text{composition of functors}\ \} \\ & \Lambda X\ .\ \mathfrak{List}\,X\ Nat \\ = & \{\ \text{definition of } \mathfrak{List}\ \} \\ & \Lambda X\ .\ 1 + Nat \times X\ Nat \\ = & \{\ \text{definition of } \mathfrak{Stack}\ \} \\ & \Lambda X\ .\ \mathfrak{Stack}\,(X\ Nat) \\ = & \{\ \text{composition of functors}\ \} \\ & \mathfrak{Stack} \circ (-\,Nat) \end{array}$$

Since every step is an equality, the isomorphisms are actually identities. (However, in Haskell a **data** declaration introduces a new type distinct from any other type, so the isomorphisms have to rename $\mathfrak{Nil}$ to $\mathfrak{Empty}$, $\mathfrak{Cons}$ to $\mathfrak{Push}$ and vice versa.)

Next, consider the functions *total* (Example 1) and *suml* (Example 22). To relate the two variants of computing the total, we first reformulate relation (44) for Mendler-style folds:

$$(\!|\lambda\, x \,.\, \Psi\, x \cdot \mathfrak{s}|\!)_{\mathsf{L}} = (\!|\lambda\, x \,.\, \Psi\, id \cdot \mathsf{G}\, x \cdot \mathfrak{s}|\!)_{\mathsf{L}} = (\!|\Psi\, id|\!) \cdot \mathfrak{f} = (\!|\Psi|\!) \cdot \mathfrak{f}.$$

Thus, $suml = total \cdot \mathfrak{f}$ if the base functions are related by $\mathfrak{suml}\, x = \mathfrak{total}\, x \cdot \mathfrak{s}$. In words, we can transform one base function into the other just by renaming the constructors. The proof is entirely straightforward. $\square$

Transforming a higher-order fixed point into a first-order fixed point works for so-called *regular datatypes*. The type of lists (Example 16) is regular; the type of perfect trees (Example 6) is not because the recursive call of $\mathsf{Perfect}$ involves a change of argument. Firstification is not applicable, as there is no first-order base functor $\mathfrak{Base}$ such that $\mathsf{App}_X \cdot \mathfrak{Perfect} = \mathfrak{Base} \cdot \mathsf{App}_X$. The class of regular datatypes is usually defined syntactically. Drawing from the development above, we can provide an alternative semantic characterisation.

Definition 8. Let $\mathsf{H} : \mathbb{C}^{\mathbb{D}} \to \mathbb{C}^{\mathbb{D}}$ be a higher-order functor. The parametric datatype $\mu\mathsf{H} : \mathbb{D} \to \mathbb{C}$ is *regular* if and only if there exists a functor $\mathsf{G} : \mathbb{D} \to \mathbb{C}^{\mathbb{C}}$ such that $\mathsf{App}_A \circ \mathsf{H} \cong \mathsf{G}\, A \circ \mathsf{App}_A$ for all objects $A : \mathbb{D}$. Then $(\mu\mathsf{H})\, A \cong \mu(\mathsf{G}\, A)$. $\square$

The regularity condition unfolds to $\mathsf{H}\, \mathsf{F}\, A \cong \mathsf{G}\, A\, (\mathsf{F}\, A)$, which makes explicit that all occurrences of 'the recursive call' F are applied to the same argument A. For lists, the required functor G is $\Lambda\, A \,.\, \Lambda\, B \,.\, 1 + A \times B$.

8.6. *Type specialisation:* $\mathsf{Lan}_{\mathsf{J}} \dashv (- \circ \mathsf{J}) \dashv \mathsf{Ran}_{\mathsf{J}}$

Type application is a special case of functor composition (cf Section 5.8). Likewise, firstification can be seen as an instance of type specialisation, where a nesting of types is fused to a single type that allows for a more compact and space-efficient representation.

Example 34. Lists of optional values, $\mathsf{List} \circ \mathsf{Maybe}$, where Maybe is given by

$$\textbf{data}\ \mathsf{Maybe}\ a = \mathit{Nothing} \mid \mathit{Just}\ a,$$

can be represented more compactly using

$$\textbf{data}\ \mathsf{Maybes}\ a = \mathit{Done} \mid \mathit{Skip}\ (\mathsf{Maybes}\ a) \mid \mathit{Yield}\ (a, \mathsf{Maybes}\ a).$$

Assuming that the constructor application $C\ (v_1, \ldots, v_i)$ requires $i + 1$ cells of storage, the compact representation saves $2n$ cells for a list of length n.

Let us show that $\mathsf{List} \circ \mathsf{Maybe} \cong \mathsf{Maybes}$. The underlying adjunction is pre-composition $\mathsf{Pre}_{\mathsf{Maybe}} \dashv \mathsf{Ran}_{\mathsf{Maybe}}$. Applying type fusion we have to demonstrate that $\mathsf{Pre}_{\mathsf{Maybe}} \circ \mathfrak{List} \cong \mathfrak{Maybes} \circ \mathsf{Pre}_{\mathsf{Maybe}}$.

$$\begin{aligned}
& \mathfrak{List}\, X \circ \mathsf{Maybe} \\
= \quad & \{\text{ composition of functors and definition of } \mathfrak{List} \} \\
& \Lambda\, A \,.\, 1 + \mathsf{Maybe}\, A \times X\, (\mathsf{Maybe}\, A) \\
= \quad & \{\text{ definition of } \mathsf{Maybe} \} \\
& \Lambda\, A \,.\, 1 + (1 + A) \times X\, (\mathsf{Maybe}\, A) \\
\cong \quad & \{ \times \text{ distributes over } + \text{ and } 1 \times B \cong B \} \\
& \Lambda\, A \,.\, 1 + X\, (\mathsf{Maybe}\, A) + A \times X\, (\mathsf{Maybe}\, A) \\
= \quad & \{\text{ composition of functors and definition of } \mathfrak{Maybes} \} \\
& \mathfrak{Maybes}\, (X \circ \mathsf{Maybe})
\end{aligned}$$

The central step is the application of distributivity, $(A + B) \times C \cong A \times C + B \times C$, which turns the nested type on the left into a 'flat' sum, which can be represented space-efficiently in Haskell—$\mathfrak{s}$'s definition makes this explicit.

$$\begin{aligned}
&\mathfrak{s} : \forall x \,.\, \forall a \,.\, \mathfrak{List}\, x\, (\mathsf{Maybe}\, a) && \to \mathfrak{Maybes}\, (x \circ \mathsf{Maybe})\, a \\
&\mathfrak{s}\quad (\mathfrak{Nil}) && = \mathfrak{Done} \\
&\mathfrak{s}\quad (\mathfrak{Cons}\, (\mathit{Nothing}, x)) && = \mathfrak{Skip}\, x \\
&\mathfrak{s}\quad (\mathfrak{Cons}\, (\mathit{Just}\, a, \quad x)) && = \mathfrak{Yield}\, (a, x)
\end{aligned}$$

The function $\mathfrak{s}$ is a natural transformation, whose components are again natural transformations, hence the nesting of universal quantifiers. $\square$

8.7. Tabulation: $(X^{(-)})^{op} \dashv X^{(-)}$

We have noted in Example 28 that functions over the natural numbers and infinite sequences are in one-to-one correspondence.

$$Nat^{\mu\mathfrak{Nat}} \cong \nu\mathfrak{Sequ} \tag{47}$$

A sequence can be seen as a *tabulation* of a function over the naturals. In general, a type T is said to tabulate functions from a type K if $V^K \cong T$ for some type V—a Table represents a function mapping Keys to Values. Perhaps surprisingly, the isomorphism above is an instance of type fusion, as well. The left adjoint is exponentiation $\mathsf{Exp}_V : \mathbb{C}^{op} \to \mathbb{C}$ defined $\mathsf{Exp}_V = V^{(-)}$ (Section 5.10). This instance is quite intriguing as the adjoint functors $(\mathsf{Exp}_V)^{op} \dashv \mathsf{Exp}_V$ are contravariant. Consequently, $\mathfrak{f}$ and $\mathfrak{s}$ live in the opposite category $\mathbb{C}^{op}$. Moreover, Exp_V maps an initial algebra to a final coalgebra! Formulated in terms of arrows in $\mathbb{C}$ rather than in $\mathbb{C}^{op}$, type fusion takes the form

$$\mathfrak{f} : \nu\mathsf{G} \cong \mathsf{Exp}_V\,(\mu\mathsf{F}) \quad \Longleftarrow \quad \mathfrak{s} : \mathsf{G} \circ \mathsf{Exp}_V \cong \mathsf{Exp}_V \circ \mathsf{F},$$

and the isomorphisms $\mathfrak{f}$ and $\mathfrak{f}^{\circ}$ are defined

$$\mathsf{Exp}_V\,in \cdot \mathfrak{f} = \mathfrak{s} \cdot \mathsf{G}\,\mathfrak{f} \cdot out \quad \text{and} \quad out \cdot \mathfrak{f}^{\circ} = \mathsf{G}\,\mathfrak{f}^{\circ} \cdot \mathfrak{s}^{\circ} \cdot \mathsf{Exp}_V\,in.$$

The arrow $\mathfrak{f} : \nu\mathsf{G} \to \mathsf{Exp}_V\,(\mu\mathsf{F})$ is a curried look-up function that maps a table to an exponential, which in turn maps a key, an element of $\mu\mathsf{F}$, to the corresponding value recorded in the table. The arrow $\mathfrak{f}^{\circ} : \mathsf{Exp}_V\,(\mu\mathsf{F}) \dot{\to} \nu\mathsf{G}$ tabulates a given exponential. Tabulation is a standard unfold, whereas look-up is an adjoint fold. Let us specialise the defining equations of $\mathfrak{f}$ and $\mathfrak{f}^{\circ}$ to the category **Set**. For $\mathfrak{f}$ alias *lookup*, we obtain

$$lookup\,(out^{\circ}\,t)\,(in\,i) = \mathfrak{s}\,(\mathsf{G}\,lookup\,t)\,i. \tag{48}$$

Both the table and the index are destructed, the look-up function is recursively applied, and the transformation $\mathfrak{s}$ then takes care of the non-recursive look-up. For $\mathfrak{f}^{\circ}$ alias *tabulate*, we obtain

$$tabulate\,f = out^{\circ}\,(\mathsf{G}\,tabulate\,(\mathfrak{s}^{\circ}\,(f \cdot in))). \tag{49}$$

The to-be-tabulated function f is pre-composed with in, $\mathfrak{s}^{\circ}$ then arranges the non-recursive tabulation, and finally *tabulate* is recursively applied.

Example 35. If we instantiate F and G to $\mathfrak{Nat}$ and $\mathfrak{Sequ}$, we obtain the functions given in Example 28. It is instructive to go through the exercise of deriving the isomorphisms in the framework of type fusion. The base isomorphism $\mathfrak{Sequ} \circ \mathsf{Exp}_{Nat} \cong \mathsf{Exp}_{Nat} \circ \mathfrak{Nat}$ is a simple consequence of the laws of exponentials.

$$\begin{aligned}
& \mathfrak{Sequ} \circ \mathsf{Exp}_{Nat} \\
= \quad & \{ \text{definition of } \mathfrak{Sequ} \text{ and definition of } \mathsf{Exp}_{Nat} \} \\
& \Lambda X\,.\,Nat \times Nat^{X} \\
\cong \quad & \{ \text{laws of exponentials} \} \\
& \Lambda X\,.\,Nat^{1+X} \\
= \quad & \{ \text{definition of } \mathfrak{Nat} \text{ and definition of } \mathsf{Exp}_{Nat} \} \\
& \mathsf{Exp}_{Nat} \circ \mathfrak{Nat}
\end{aligned}$$

The natural isomorphism is witnessed by

$$\begin{aligned}
& \mathfrak{s} : \forall x\,.\,\mathfrak{Sequ}\,(\mathsf{Exp}_{Nat}\,x) \to \mathsf{Exp}_{Nat}\,(\mathfrak{Nat}\,x) \\
& \mathfrak{s} \quad (\mathfrak{Next}\,(v, t)) \quad (\mathfrak{Z}) \quad = v \\
& \mathfrak{s} \quad (\mathfrak{Next}\,(v, t)) \quad (\mathfrak{S}\,n) = t\,n.
\end{aligned}$$

Inlining $\mathfrak{s}$ into Eq. (48) yields the look-up function:

$$\begin{aligned}
& lookup : \nu\mathfrak{Sequ} \to \mathsf{Exp}_{Nat}\,(\mu\mathfrak{Nat}) \\
& lookup \quad (Out^{\circ}\,(\mathfrak{Next}\,(v, t)))\,(In\,\mathfrak{Z}) \quad = v \\
& lookup \quad (Out^{\circ}\,(\mathfrak{Next}\,(v, t)))\,(In\,(\mathfrak{S}\,n)) = lookup\,t\,n.
\end{aligned}$$

The inverse of $\mathfrak{s}$ is defined

$$\begin{aligned}
& \mathfrak{s}^{\circ} : \forall x\,.\,\mathsf{Exp}_{Nat}\,(\mathfrak{Nat}\,x) \to \mathfrak{Sequ}\,(\mathsf{Exp}_{Nat}\,x) \\
& \mathfrak{s}^{\circ} \quad f \quad = \mathfrak{Next}\,(f\,\mathfrak{Z}, f \cdot \mathfrak{S}).
\end{aligned}$$

If we inline $\mathfrak{s}^{\circ}$ into Eq. (49), we obtain

$$\begin{aligned}
& tabulate : \mathsf{Exp}_{Nat}\,(\mu\mathfrak{Nat}) \to \nu\mathfrak{Sequ} \\
& tabulate \quad f \quad = Out^{\circ}\,(\mathfrak{Next}\,(f\,(In\,\mathfrak{Z}), tabulate\,(f \cdot In \cdot \mathfrak{S}))).
\end{aligned}$$

Voilá. We have derived the definitions of Example 28 written in terms of two-level types. Moreover, *lookup* and *tabulate* are inverses by construction. □

8.8. *Tabulation revisited:* $\mathsf{Exp} \dashv \mathsf{Sel}$

The relation between natural numbers and infinite sequences (47) can be lifted to streams:

$$V^{\mu\mathfrak{Nat}} \cong \nu\mathfrak{Stream}\,V.$$

The isomorphism is, in fact, natural in V, so it is an isomorphism between functors:

$$(-)^{\mu\mathfrak{Nat}} \cong \nu\mathfrak{Stream}. \qquad (50)$$

We obtain another adjoint situation. The left adjoint is now a curried version of exponentiation: $\mathsf{Exp} : \mathbb{C} \to (\mathbb{C}^{\mathbb{C}})^{\mathsf{op}}$ with $\mathsf{Exp}\,K = \Lambda\,V\,.\,V^K$. Wow! Using the functor Exp, Eq. (50) can be rephrased as $\mathsf{Exp}\,(\mu\mathfrak{Nat}) \cong \nu\mathfrak{Stream}$. The right adjoint of Exp exists, if the underlying category has ends.

$$\begin{aligned}
& (\mathbb{C}^{\mathbb{C}})^{\mathsf{op}}(\mathsf{Exp}\,A, \mathsf{B}) \\
\cong\ & \quad \{ \text{opposite category} \} \\
& \mathbb{C}^{\mathbb{C}}(\mathsf{B}, \mathsf{Exp}\,A) \\
\cong\ & \quad \{ \text{natural transformation as an end [49, p. 223]} \} \\
& \forall X : \mathbb{C}\,.\,\mathbb{C}(\mathsf{B}\,X, \mathsf{Exp}\,A\,X) \\
=\ & \quad \{ \text{definition of } \mathsf{Exp} \} \\
& \forall X : \mathbb{C}\,.\,\mathbb{C}(\mathsf{B}\,X, X^A) \\
\cong\ & \quad \{ (X^{(-)})^{\mathsf{op}} \dashv X^{(-)} \} \\
& \forall X : \mathbb{C}\,.\,\mathbb{C}(A, X^{\mathsf{B}\,X}) \\
\cong\ & \quad \{ \text{the hom-functor } \mathbb{C}(A,-) \text{ preserves ends [49, p. 225]} \} \\
& \mathbb{C}(A, \forall X : \mathbb{C}\,.\,X^{\mathsf{B}\,X}) \\
\cong\ & \quad \{ \text{define } \mathsf{Sel}\,\mathsf{B} = \forall X : \mathbb{C}\,.\,X^{\mathsf{B}\,X} \} \\
& \mathbb{C}(A, \mathsf{Sel}\,\mathsf{B})
\end{aligned}$$

The derivation shows that the right adjoint of Exp is a higher-order functor that maps a functor B, a type of tables, to the type of *selectors* $\forall X : \mathbb{C}\,.\,X^{\mathsf{B}\,X}$, polymorphic functions that select some entry from a given table.

Example 36. The witnesses of (50) are lifted variants of look-up and tabulation defined in Example 35.

$$\begin{aligned}
& \mathit{lookup} : \forall v\,.\,\nu\mathfrak{Stream}\,v \to \mathsf{Exp}\,(\mu\mathfrak{Nat})\,v \\
& \mathit{lookup}\ (\mathit{Out}^{\circ}\,(\mathfrak{Link}\,(v,t)))\,(\mathit{In}\,\mathfrak{Z}) = v \\
& \mathit{lookup}\ (\mathit{Out}^{\circ}\,(\mathfrak{Link}\,(v,t)))\,(\mathit{In}\,(\mathfrak{S}\,n)) = \mathit{lookup}\,t\,n \\
& \mathit{tabulate} : \forall v\,.\,\mathsf{Exp}\,(\mu\mathfrak{Nat})\,v \to \nu\mathfrak{Stream}\,v \\
& \mathit{tabulate}\ f = \mathit{Out}^{\circ}\,(\mathfrak{Link}\,(f\,(\mathit{In}\,\mathfrak{Z}), \mathit{tabulate}\,(f \cdot \mathit{In} \cdot \mathfrak{S})))
\end{aligned}$$

Modulo the constructor names the definitions are, in fact, identical to those of Example 35. The important observation is that the laws of exponentials are natural in all the variables involved. In particular, $\mathfrak{s} : \mathfrak{Sequ} \circ \mathsf{Exp}_V \dot{\to} \mathsf{Exp}_V \circ \mathfrak{Nat}$ is natural in the type of elements V. □

Generalising the definition of Section 8.7, the functor T is said to tabulate functions from the type K if $\mathsf{Exp}\,K \cong \mathsf{T}$. Type fusion implies that the final coalgebra $\nu\mathsf{G}$ tabulates functions from the initial algebra $\mu\mathsf{F}$ if $\mathsf{Exp} \circ \mathsf{F} \cong \mathsf{G} \circ \mathsf{Exp}$. Can we always find a G for a given base functor F? Yes, if the base functor is a polynomial—this statement can be generalised [36] but the details are beyond the scope of this article. The functor G is induced by the laws of exponentials.

$$\begin{array}{rcl@{\qquad}rcl}
X^0 & \cong & 1 & \mathsf{Exp}\,0 & \cong & \mathsf{K}\,1 \\
X^1 & \cong & X & \mathsf{Exp}\,1 & \cong & \mathsf{Id} \\
X^{A+B} & \cong & X^A \times X^B & \mathsf{Exp}\,(A+B) & \cong & \mathsf{Exp}\,A \mathbin{\dot{\times}} \mathsf{Exp}\,B \\
X^{A\times B} & \cong & (X^B)^A & \mathsf{Exp}\,(A \times B) & \cong & \mathsf{Exp}\,A \circ \mathsf{Exp}\,B
\end{array}$$

Since the laws of exponentials are natural in the base type X, they can be re-formulated as isomorphisms between functors (laws on the right). Note that the tabulation of a product is given by a composition of functors. This shows that the generalisation of tabulation from objects (Section 8.7) to functors (this section) is needed to tabulate types that involve products! The following table extends the relation between key types and table types to polynomial base functors.

$$\begin{array}{lcl@{\qquad}lcl}
\mathsf{F}\,X & = & X & \mathsf{G}\,Y & = & Y \\
\mathsf{F}\,X & = & 0 & \mathsf{G}\,Y & = & \mathsf{K}\,1 \\
\mathsf{F}\,X & = & 1 & \mathsf{G}\,Y & = & \mathsf{Id} \\
\mathsf{F}\,X & = & \mathsf{F}_1\,X + \mathsf{F}_2\,X & \mathsf{G}\,Y & = & \mathsf{G}_1\,Y \mathbin{\dot{\times}} \mathsf{G}_2\,Y \\
\mathsf{F}\,X & = & \mathsf{F}_1\,X \times \mathsf{F}_2\,X & \mathsf{G}\,Y & = & \mathsf{G}_1\,Y \circ \mathsf{G}_2\,Y
\end{array}$$

For each row in the table we have $\mathsf{Exp} \circ \mathsf{F} \cong \mathsf{G} \circ \mathsf{Exp}$ if $\mathsf{Exp} \circ \mathsf{F}_i \cong \mathsf{G}_i \circ \mathsf{Exp}$. The following Haskell code provides an example where the extra generality is needed.

Example 37. An alternative representation of the natural numbers, Knuth's humongous numbers [43], is given by the datatype of binary trees.

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

The empty tree represents 0; if l represents a and r represents b, then $\mathit{Fork}\ (l, r)$ represents $2^a + b$. Functions from this funny number type are tabulated by the nested datatype

$$\mathbf{data}\ \mathit{Trie}\ v = \mathit{Map}\ (v, \mathit{Trie}\ (\mathit{Trie}\ v)).$$

The type is sometimes characterised as a *truly nested datatype* [1] as the recursive calls are nested. This is a consequence of the fourth law of exponentials: the base functor $\mathsf{F}\,X = 1 + (X \times X)$ of *Tree* induces the base functor $\mathsf{G}\,Y = \mathsf{Id} \mathbin{\dot\times} (Y \circ Y)$ of *Trie*. Look-up and tabulation are then defined

```
lookup : ∀v . Trie v →      Exp Tree v
lookup         (Map (v, t)) Leaf          = v
lookup         (Map (v, t)) (Fork (l, r)) = lookup (lookup t l) r
tabulate : ∀v . Exp Tree v → Trie v
tabulate        f             = Map (f Leaf, tabulate (λl → tabulate (λr → f (Fork (l, r))))).
```

Even though the recursive calls are strangely nested, *lookup* is an adjoint fold and *tabulate* is a standard unfold. Furthermore, they are inverses by construction. □

The development generalises the results of the previous section. The left adjoint Exp_V can, in fact, be seen as the result of composing exponentiation with type application: $\mathsf{Exp}_V = \mathsf{App}_V \circ \mathsf{Exp}$. Then $\mathsf{Exp}_{Nat}\ (\mu\mathfrak{Nat}) \cong \nu\mathfrak{Sequ}$ is given by applying type fusion twice:

$$\begin{aligned}
& \mathsf{Exp}_{Nat}\ (\mu\mathfrak{Nat}) \\
= \quad & \{\ \mathsf{Exp}_V = \mathsf{App}_V \circ \mathsf{Exp}\ \} \\
& \mathsf{App}_{Nat}\ (\mathsf{Exp}\ (\mu\mathfrak{Nat})) \\
\cong \quad & \{\ \text{type fusion: } \mathsf{Exp}\ (\mu\mathfrak{Nat}) \cong \nu\mathfrak{Stream}\ \} \\
& \mathsf{App}_{Nat}\ (\nu\mathfrak{Stream}) \\
\cong \quad & \{\ \text{type fusion: } \mathsf{App}_{Nat}\ (\nu\mathfrak{Stream}) \cong \nu\mathfrak{Sequ}\ \} \\
& \nu\mathfrak{Sequ}.
\end{aligned}$$

9. Recursive coalgebras and corecursive algebras

We have noted in Section 3 that initial algebras and final coalgebras are different entities. For instance, in **Set** the initial algebra of $\mathfrak{Stack}$ comprises only finite stacks, whereas the final coalgebra also contains infinite ones. The initial algebra can always be embedded in the final coalgebra, using either a fold or an unfold: $(\!|out^{\circ}|\!) = [\![in^{\circ}]\!] : \mu\mathsf{F} \to \nu\mathsf{F}$. The other way round has to be programmed explicitly, which typically involves imposing some depth bound to ensure termination (cf Example 28). [14] provides further instructive examples along those lines.

The fact that $\mu\mathsf{F}$ and $\nu\mathsf{F}$ are not compatible has the unfortunate consequence that we cannot freely combine folds (consumers) and unfolds (producers). There are at least two ways out of this dilemma: (1) we can work in a setting where the two fixed points coincide $\mu\mathsf{F} \cong \nu\mathsf{F}$, a so-called *algebraically compact category* [23]—Haskell's ambient category **SCpo** provides an example of such a setting; or (2) we can restrict unfolds to coalgebras which only produce values in $\mu\mathsf{F}$. An attractive way to achieve the latter is to use hylomorphisms based on recursive coalgebras as a structured recursion scheme [12]. Hylomorphisms are, in fact, interesting in their own right as they provide an alternative framework for the algebra of programming. We introduce the notion below and then relate hylomorphisms to adjoint folds.

A coalgebra $\langle C, c\rangle$ is called *recursive* if for *every* algebra $\langle A, a\rangle$ the equation in the unknown $x : \mathbb{C}(C, A)$,

$$x = a \cdot \mathsf{G}\,x \cdot c, \tag{51}$$

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* or *hylo* for short and is written $(\!|a \mid c|\!)_{\mathsf{G}}$.

The functor G captures the control structure as the examples below demonstrate. However, the definition of a hylomorphism does not assume that the initial G-algebra exists. The powerset functor, for instance, admits no fixed points, yet we may want to divide a problem into a *set* of sub-problems. If the initial algebra $\langle\mu\mathsf{F}, in\rangle$ exists, then $\langle\mu\mathsf{F}, in^{\circ}\rangle$ is isomorphic to the final recursive coalgebra and, furthermore, folds and 'recursive unfolds' emerge as special cases of hylos: $(\!|a|\!) = (\!|a \mid in^{\circ}|\!)$ and $[\![c]\!] = (\!|in \mid c|\!)$. It is important to note that here $[\![c]\!]$ is the unique coalgebra homomorphism to the final *recursive* coalgebra, that is, the final object in the full subcategory of *recursive* coalgebras and coalgebra homomorphisms.

Example 38. The functor Tail_a

$$\textbf{data}\ \mathsf{Tail}_a\ b = \mathit{Stop}\ a \mid \mathit{Loop}\ b$$

captures tail recursion: a loop either terminates producing an a value, or it goes through another iteration. Tail-recursive programs share the algebra

$$\begin{aligned}
&\mathit{continue} : \forall a\ .\ \mathsf{Tail}_a\ a && \to a\\
&\mathit{continue}\qquad (\mathit{Stop}\ a) && = a\\
&\mathit{continue}\qquad (\mathit{Loop}\ b) && = b,
\end{aligned}$$

which simply removes the constructors. The algebra is also known as the *codiagonal*, the counit of the adjunction $(+) \dashv \Delta$.

As a simple application, here is a tail-recursive variant of *total*,

$$\begin{aligned}
&\mathit{total}^{\curlywedge} : (\mathit{Stack}, \quad \mathit{Nat}) && \to \mathsf{Tail}_{\mathit{Nat}}\ (\mathit{Stack}, \mathit{Nat})\\
&\mathit{total}^{\curlywedge}\quad (\mathit{Empty}, \quad e) && = \mathit{Stop}\ e\\
&\mathit{total}^{\curlywedge}\quad (\mathit{Push}\ (n, s), e) && = \mathit{Loop}\ (s, e + n)
\end{aligned}$$

$$\begin{aligned}
&\mathit{total}\ :\ (\mathit{Stack}, \mathit{Nat}) \to \mathit{Nat}\\
&\mathit{total} = \mathit{continue} \cdot \mathit{fmap}\ \mathit{total} \cdot \mathit{total}^{\curlywedge}
\end{aligned}$$

which adds the elements of a stack using an accumulating parameter. □

Example 39. The function *cat* for appending two stacks (Example 9) has a simple control structure: if the first stack is empty, we stop; otherwise, we recurse memorising its topmost element. This motivates the 'control functor'

$$\textbf{data}\ \mathsf{Rec}_a\ b = \mathit{Stop}\ a \mid \mathit{Recurse}\ (\mathit{Nat}, b).$$

The functor is similar to Tail_a, except that now we also keep track of the stack elements. (They are pushed onto the recursion stack modelled by Rec_a.)

The coalgebra

$$\begin{aligned}
&\mathit{cat}^{\curlywedge} : (\mathit{Stack}, \quad \mathit{Stack}) && \to \mathsf{Rec}_{\mathit{Stack}}\ (\mathit{Stack}, \mathit{Stack})\\
&\mathit{cat}^{\curlywedge}\quad (\mathit{Empty}, \quad \mathit{ns}) && = \mathit{Stop}\ \mathit{ns}\\
&\mathit{cat}^{\curlywedge}\quad (\mathit{Push}\ (m, \mathit{ms}), \mathit{ns}) && = \mathit{Recurse}\ (m, (\mathit{ms}, \mathit{ns}))
\end{aligned}$$

models the divide step of *cat* and the algebra

$$\begin{aligned}
&\mathit{cat}^{\curlyvee} : \mathsf{Rec}_{\mathit{Stack}}\ \mathit{Stack} && \to \mathit{Stack}\\
&\mathit{cat}^{\curlyvee}\quad (\mathit{Stop}\ \mathit{ns}) && = \mathit{ns}\\
&\mathit{cat}^{\curlyvee}\quad (\mathit{Recurse}\ (n, \mathit{ns})) && = \mathit{Push}\ (n, \mathit{ns})
\end{aligned}$$

captures its conquer step. Finally, *cat*

$$\begin{aligned}
&\mathit{cat}\ :\ (\mathit{Stack}, \mathit{Stack}) \to \mathit{Stack}\\
&\mathit{cat} = \mathit{cat}^{\curlyvee} \cdot \mathit{fmap}\ \mathit{cat} \cdot \mathit{cat}^{\curlywedge}
\end{aligned}$$

is given by a hylo equation. □

Turning to the relationship between adjoint folds and hylos, we have seen in Section 4.3 that Mendler-style adjoint folds are equivalent to algebraic adjoint folds. The latter scheme is already very close to the hylo form:

$$\begin{aligned}
&\quad x \cdot \mathsf{L}\, \mathit{in} = a \cdot \mathsf{G}\, x \cdot \alpha\\
\iff &\quad \{\ \mathit{in}\ \text{isomorphism and}\ \mathsf{L}\ \text{functor}\ \}\\
&\quad x = a \cdot \mathsf{G}\, x \cdot \alpha \cdot \mathsf{L}\, \mathit{in}^{\circ}.
\end{aligned}$$

The equivalence shows that $\alpha \cdot \mathsf{L}\, \mathit{in}^{\circ}$ is a recursive coalgebra. Thus, adjoint folds are a special case of hylomorphisms:

$$(\!|\lambda x\ .\ a \cdot \mathsf{G}\, x \cdot \alpha|\!)_{\mathsf{L}} = (\!|a \mid \alpha \cdot \mathsf{L}\, \mathit{in}^{\circ}|\!)_{\mathsf{G}}.$$

In fact, one can show a more general result: the G-coalgebra $\alpha\ C \cdot \mathsf{L}\, c : \mathsf{L}\, C \to \mathsf{G}\, (\mathsf{L}\, C)$ is recursive if the F-coalgebra $c : C \to \mathsf{F}\, C$ is recursive [12, Proposition 12].

Example 40. In Example 13 we defined list concatenation as an algebraic adjoint fold, in Example 39 as a hylomorphism. Though we have approached the problem from different angles, the final result is the same: modulo constructor names $\mathfrak{Stack}' = \mathsf{Rec}_{\mathit{Stack}}$, $\mathit{scat}^{\triangledown} = \mathit{cat}^{\curlyvee}$ and $\mathit{scat}^{\leftrightarrow} \cdot \mathsf{L}\, \mathit{in}^{\circ} = \mathit{cat}^{\curlywedge}$. □

When we introduced initial fixed-point equations we argued that the naturality of the base function ensures termination. We do not have the same guarantees for hylomorphisms, as the following example demonstrates [2]. (We use Pseudo-Haskell code below, since the example is not expressible in Haskell). Define the **Set** functor

```
data Square x = Nothing | Just {(a₁, a₂) | a₁ ∈ x, a₂ ∈ x, a₁ ≠ a₂}
instance Functor Square where
  fmap f Nothing = Nothing
  fmap f (Just (x₁, x₂))
     | f x₁ ≠ f x₂ = Just (f x₁, f x₂)
     | otherwise   = Nothing,
```

which gives the square of x with the diagonal removed. The action of *Square* on arrows preserves the invariant, possibly changing *Just* to *Nothing*. Now, define the constant coalgebra $c : Bool \to Square\ Bool$ by $c\ b = Just\ (False, True)$. The coalgebra c is recursive as the equation $x = a \cdot fmap\ x \cdot c$ has the unique solution $x\ b = a\ Nothing$. (Since c is constant, x has to be constant, as well. For a constant x, the call *fmap x* always yields *Nothing*. Consequently, $x\ b$ equals $a\ Nothing$.) If executed, however, *fmap x* will issue two recursive calls, *x False* and *x True*. Adámek et al. [2] have shown that this problem disappears if one imposes additional restrictions on the category and on the control functor: for finitary **Set**-functors that preserve inverse images, termination is guaranteed. Polynomial functors satisfy these conditions.

While hylos are more expressive, they sometimes suffer from the practical problem that a suitable control functor is hard to find. For instance, the function *flattenCat* (Example 14) is obviously an adjoint fold; the underlying control functor is, however, less obvious.

To summarise, the hylo scheme makes the control structure explicit; the data structure is hidden in the coalgebra (input) or in the algebra (output). By contrast, adjoint folds are explicit about the (input) data structure; the control structure is induced by the adjunction.

As to be expected, the construction dualises. An algebra is called *corecursive* if the hylo equation (51) has a unique solution for *every* coalgebra. The uniquely defined arrow is called a *cohylomorphism*. If the final coalgebra $\langle \nu\mathsf{F},\ out\rangle$ exists, then $\langle \nu\mathsf{F},\ out^{\circ}\rangle$ is isomorphic to the initial corecursive algebra. In other words, cohylomorphisms offer an alternative approach for structured corecursion [13]. Like adjoint folds are related to hylomorphisms, adjoint unfolds are related to cohylomorphisms. The investigation of the latter species is, however, still in its infancy.

10. Related work

Author's Prayer:
Copy from one, it's plagiarism;
copy from two, it's research.
Wilson Mizner

Adjoint folds. Building on the work of Hagino [29], Malcolm [50] and many others, Bird and de Moor gave a comprehensive account of the "Algebra of Programming" in their seminal textbook [9]. While the work was well received and highly appraised in general, it also received some criticism. Poll and Thompson [61] write in an otherwise positive review:

> The disadvantage is that even simple programs like factorial require some manipulation to be given a catamorphic form, and a two-argument function like concat requires substantial machinery to put it in catamorphic form, and thus make it amenable to manipulation.

The term 'substantial machinery' refers to Section 3.5 of the textbook where Bird and de Moor address the problem of assigning a unique meaning to the defining equation of *append* (called *cat* in the textbook). In fact, they generalise the problem slightly, considering equations of the form

$$x \cdot (in \times id) = a \cdot \mathsf{G}\, x \cdot \alpha,$$

which we recognise as the definition of an algebraic adjoint fold. Clearly, their approach is subsumed by the framework of adjoint folds.

The seed for this framework was laid in Section 6 of the paper "Generalised folds for nested datatypes" by Bird and Paterson [11]. 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. Somewhat ironically, the rest of the paper, which is concerned with folds for nested datatypes, does not build upon this elegant approach. Also, they do not consider adjoint unfolds. Nonetheless, Bird and Paterson deserve most of the credit for their fundamental insight, so three cheers to them! (As an aside, the first proof method uses colimits and is strictly more powerful. It can be used, for instance, to give a semantics to functions such as *add* that are defined by simultaneous recursion over a pair of datatypes: $\times(\mu\mathsf{F}) \to A$. Since the product is not a left adjoint, the framework developed in this article is not directly applicable, but see Section 5.4.) Algebraic adjoint folds were introduced by Matthes and Uustalu [53] under the name *generalised iteration* (cf Section 4.3).

Mendler-style folds. An alternative, type-theoretic approach to (co)inductive types was proposed by Mendler [56]. 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 [19]. Interestingly, in contrast to traditional category-theoretic treatments of (co)inductive types there is no requirement that the underlying type constructor is a covariant functor. Indeed, Uustalu and Vene [70] have shown that Mendler-style folds can be based on difunctors. It remains to be seen whether adjoint folds can also be generalised in this direction. Abel et al. [1] extended Mendler-style folds to higher kinds in the setting of typed term rewriting. Among other things, they demonstrate that suitable extensions of Girard's system F^{ω} retain the strong normalisation property and they show how to transform generalised Mendler-style folds into standard ones.

Recursion schemes. There is a large body of work on 'morphisms'. Building on the notions of functors and natural transformations Malcolm [50] generalised the Bird–Meertens formalism to arbitrary datatypes. Incidentally, he also discussed how to model mutually recursive types, albeit in an ad-hoc manner. His work assumed **Set** as the underlying category and was adapted by Meijer et al. [55] 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 $[\!(-)\!]$). The term catamorphism was actually coined by Meertens, the notation $(\!|-|\!)$ is due to Malcolm, and the name banana bracket is attributed to van der Woude.

The notion of a *paramorphism* was introduced by Meertens [54]. Roughly speaking, paramorphisms generalise primitive recursion to arbitrary datatypes. Their duals, *apomorphisms*, were only studied later by Vene and Uustalu [72]. While initial algebras have been the subject of intensive research, final coalgebras have received less attention—they are certainly under-appreciated [25].

Fokkinga [21] captured mutually recursive functions by *mutumorphisms*, see Section 5.5. He also observed that Malcolm's *zygomorphisms* arise as a special case, where one function depends on the other, but not the other way round. (Paramorphisms further specialise zygomorphisms in that the independent function is the identity.) An alternative solution to the '*append*-problem' was proposed by Pardo [59]: he introduces *folds with parameters* and uses them to implement *generic accumulations*. His accumulations subsume Gibbons' *downwards accumulations* [24].

The discovery of nested datatypes and their expressive power [10,16,58] 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 aforementioned paper by Bird and Paterson [11] addressed the problem by adding extra parameters to folds leading to the notion of a *generalised fold*. The author identified a potential source of inefficiency – generalised folds make heavy use of mapping functions – and proposed *efficient generalised folds* as a cure [30]. The approach being governed by pragmatic concerns was put on a firm theoretical footing by Martin et al. [52] – rather imaginatively the resulting folds were called *disciplined, efficient, generalised folds*. The fact that standard folds are actually sufficient for practical purposes – every adjoint fold can be transformed into a standard fold – was later re-discovered by Johann and Ghani [41].

The insight that generalised algebraic datatypes can be modelled by initial algebras in categories of indexed objects and arrows is also due to Johann and Ghani [42].

We have shown that all of these different morphisms and (un)folds fall under the umbrella of adjoint (un)folds. (Paramorphisms and apomorphisms require a slight tweak though: the argument or result must be guarded by an invocation of the identity.) 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* [71,7]. 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 $x : \mu\mathsf{F} \to A$,

$$x \cdot in = a \cdot \mathsf{F}\,(\mathsf{N}\,x \cdot f),$$

has a unique solution for every algebra $a : \mathsf{F}\,(\mathsf{N}\,A) \to A$. This scheme includes zygomorphisms and histomorphisms as special cases. While adjoint folds subsume zygomorphisms, they only capture histomorphisms that depend on a fixed number of previous values (Section 5.6).

Just in case you were wondering, we have not discussed monadic folds [20], because they can be defined in terms of standard folds. Interestingly though, the general construction involves an adjunction between the category of algebras over the ambient category and the category of algebras over the Kleisli category. One of the referees of MPC 2010 suggested to add the worker/wrapper transformation [40] to our catalogue of techniques. We have resisted the temptation to do so, as the transformation deals with the orthogonal problem of changing the result type of a fold to improve performance. Finally, we have left the exploration of relational adjoint (un)folds to future work.

Type fusion. The initial algebra approach to the semantics of datatypes originates in the work of Lambek [45] 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 et al. [5], who generalised a number of lattice-theoretic fixed point rules to category theory, type fusion being one of them. (The paper contains no proofs; these are provided in an unpublished manuscript [4].) The rules are illustrated by deriving isomorphisms between list types (cons and snoc lists)—currying is the only adjunction considered.

Finite versions of memo tables are known as *tries* or *digital search trees*. Knuth [44] attributes the idea of a trie to Thue [67]. Connelly and Morris [16] formalised the concept of a trie in a categorical setting: they showed that a trie is a functor and that the corresponding look-up function is a natural transformation. The author gave a polytypic definition of memo tables using type-indexed datatypes [32,33], which Section 8.8 puts on a sound theoretical footing. The insight that a function from an inductive type is tabulated by a coinductive type is due to Altenkirch [3]. He also mentions type fusion as a way of proving tabulation correct, but does not spell out the details. (Altenkirch attributes the idea to Backhouse.)

11. Conclusion

I had the idea for this article when I re-read "Generalised folds for nested datatypes" by Bird and Paterson [11]. I needed to prove the uniqueness of a certain function and I recalled that the paper offered a general approach for doing this. After a while I began to realise that the approach was far more general than I and possibly also the authors initially realised.

Adjoint folds and unfolds strike a fine balance between expressiveness and ease of use. We have shown that many if not most Haskell functions fit under this umbrella. 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.

A final thought: most if not all constructions in category theory are parametric in the underlying category, resulting in a remarkable economy of expression. Perhaps, we should spend more time and effort into utilising this economy for programming. This possibly leads to a new style of programming, which could be loosely dubbed as *category-parametric programming*.

Acknowledgements

Thanks are due to Tom Harper and Daniel James for carefully proof-reading a draft of this article and for suggesting numerous improvements. I owe a particular debt of gratitude to Thorsten Altenkirch for showing me how to derive the right adjoint of $- \dot{\times} \mathsf{H}$ in $\mathbf{Set}^{\mathbb{C}}$ (at that time, I could not imagine that I would ever find an application for the construction) and to Roland Backhouse for introducing me to type fusion (at that time, I failed to see the beauty of adjunctions). Thanks are furthermore due to the anonymous referees of MPC 2010 who provided several references and some historical perspective. One referee pointed out that adjoint folds can also be introduced via a distributive law, which led me to include Section 4.3. The anonymous referees of this special issue deserve a big thank you for carefully and patiently proof-reading the long submission and for numerous suggestions for improvement. Finally, I would like to thank Richard Bird for general advice and for encouraging (even pushing me) to write things up.

References

[1] A. Abel, R. Matthes, T. Uustalu, Iteration and coiteration schemes for higher-order and nested datatypes, Theoret. Comput. Sci. 333 (1-2) (2005) 3–66.
[2] J. Adámek, D. Lücke, S. Milius, Recursive coalgebras of finitary functors, Theor. Inform. Appl. 41 (4) (2007) 447–462. URL: http://dx.doi.org/10.1051/ita:2007028.
[3] T. Altenkirch, Representations of first order function types as terminal coalgebras, in: Typed Lambda Calculi and Applications, TLCA 2001, in: Lecture Notes in Computer Science, vol. 2044, Springer, Berlin, Heidelberg, 2001, pp. 62–78.
[4] R. Backhouse, M. Bijsterveld, R. van Geldrop, J. van der Woude, Category theory as coherently constructive lattice theory, 1994. Available from http://www.cs.nott.ac.uk/~rcb/MPC/CatTheory.ps.gz.
[5] R. Backhouse, M. Bijsterveld, R. van Geldrop, J. van der Woude, Categorical fixed point calculus, in: D. Pitt, D.E. Rydeheard, P. Johnstone (Eds.), Proceedings of the 6th International Conference on Category Theory and Computer Science, CTCS'95, Cambridge, UK, in: Lecture Notes in Computer Science, vol. 953, Springer-Verlag, 1995, pp. 159–179.
[6] M. Barr, C. Wells, Category Theory for Computing Science, 3rd ed., Les Publications, CRM, Montréal, 1999, the book is available from Centre de recherches mathématiques http://crm.umontreal.ca/.
[7] F. Bartels, Generalised coinduction, Math. Structures Comput. Sci. 13 (2003) 321–348.
[8] R. Bird, Introduction to Functional Programming using Haskell, 2nd ed., Prentice Hall Europe, London, 1998.
[9] R. Bird, O. de Moor, Algebra of Programming, Prentice Hall Europe, London, 1997.
[10] R. Bird, L. Meertens, Nested datatypes, in: J. Jeuring (Ed.), Fourth International Conference on Mathematics of Program Construction, MPC'98, Marstrand, Sweden, in: Lecture Notes in Computer Science, vol. 1422, Springer, Berlin, Heidelberg, 1998, pp. 52–67.
[11] R. Bird, R. Paterson, Generalised folds for nested datatypes, Form. Asp. Comput. 11 (2) (1999) 200–222.
[12] V. Capretta, T. Uustalu, V. Vene, Recursive coalgebras from comonads, Inform. Comput. 204 (4) (2006) 437–468.
[13] V. Capretta, T. Uustalu, V. Vene, Corecursive algebras: A study of general structured corecursion, in: M. Oliveira, J. Woodcock (Eds.), Formal Methods: Foundations and Applications, in: Lecture Notes in Computer Science, vol. 5902, Springer, Berlin / Heidelberg, 2009, pp. 84–100. URL: http://dx.doi.org/10.1007/978-3-642-10452-7_7.
[14] R. Cockett, Draft: Charitable thoughts (class notes), 1996. Available at ftp://ftp.cpsc.ucalgary.ca/pub/projects/charity/literature/papers_and_reports/charitable.ps.
[15] R. Cockett, T. Fukushima, About Charity, Yellow Series Report 92/480/18, Dept. of Computer Science, Univ. of Calgary, June 1992.
[16] R.H. Connelly, F.L. Morris, A generalization of the trie data structure, Math. Structures Comput. Sci. 5 (3) (1995) 381–418. URL: http://dx.doi.org/10.1017/S0960129500000803.
[17] D. Coutts, R. Leshchinskiy, D. Stewart, Stream fusion: from lists to streams to nothing at all, in: N. Ramsey (Ed.), Proceedings of the 12th ACM SIGPLAN International Conference on Functional programming, ICFP'07, ACM, New York, NY, USA, 2007, pp. 315–326. URL: http://doi.acm.org/10.1145/1291151.1291199.

[18] O. Danvy, An extensional characterization of lambda-lifting and lambda-dropping, in: A. Middeldorp, T. Sato (Eds.), 4th Fuji International Symposium on Functional and Logic Programming, FLOPS'99, Tsukuba, Japan, in: Lecture Notes in Computer Science, vol. 1722, Springer, Berlin, Heidelberg, 1999, pp. 241–250.
[19] P.J. de Bruin, Inductive types in constructive languages, Ph.D. Thesis, University of Groningen. 1995.
[20] M. Fokkinga, Monadic maps and folds for arbitrary datatypes. Tech. Rep. Memoranda Informatica 94–28, University of Twente, June 1994.
[21] M.M. Fokkinga, Law and order in algorithmics, Ph.D. Thesis, University of Twente, February 1992.
[22] M.M. Fokkinga, L. Meertens, Adjunctions, Tech. Rep, Memoranda Inf 94-31, University of Twente, Enschede, Netherlands, June 1994.
[23] P. Freyd, Algebraically complete categories, in: A. Carboni, M. Pedicchio, G. Rosolini (Eds.), Category Theory, in: Lecture Notes in Mathematics, vol. 1488, Springer, Berlin, Heidelberg, 1991, pp. 95–104.
[24] J. Gibbons, Generic downwards accumulations, Sci. Comput. Program. 37 (1–3) (2000) 37–65.
[25] J. Gibbons, G. Jones, The under-appreciated unfold, in: M. Felleisen, P. Hudak, C. Queinnec (Eds.), Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming, ACM Press, 1998, pp. 273–279.
[26] J. Gibbons, R. Paterson, Parametric datatype-genericity, in: P. Jansson (Ed.), Proceedings of the 2009 ACM SIGPLAN Workshop on Generic Programming, ACM Press, 2009, pp. 85–93.
[27] E. Giménez, Codifying guarded definitions with recursive schemes, in: P. Dybjer, B. Nordström, J.M. Smith (Eds.), Types for Proofs and Programs, International Workshop TYPES'94, Båstad, Sweden, June 6–10, 1994, Selected Papers, in: Lecture Notes in Computer Science, vol. 996, Springer-Verlag, 1995, pp. 39–59.
[28] J.A. Goguen, J.W. Thatcher, E.G. Wagner, J.B. Wright, Initial algebra semantics and continuous algebras, J. ACM 24 (1) (1977) 68–95.
[29] T. Hagino, A typed lambda calculus with categorical type constructors, in: D. Pitt, A. Poigne, D. Rydeheard (Eds.), Category Theory and Computer Science, in: Lecture Notes in Computer Science, vol. 283, 1987.
[30] R. Hinze, Efficient generalized folds, in: J. Jeuring (Ed.), Proceedings of the second Workshop on Generic Programming, 2000, pp. 1–16. The proceedings appeared as a technical report of Universiteit Utrecht, UU-CS-2000-19.
[31] R. Hinze, Functional Pearl: perfect trees and bit-reversal permutations, J. Funct. Programming 10 (3) (2000) 305–317.
[32] R. Hinze, Generalizing generalized tries, J. Funct. Programming 10 (4) (2000) 327–351.
[33] R. Hinze, Memo functions, polytypically! in: J. Jeuring (Ed.), Proceedings of the second Workshop on Generic Programming, 2000, pp. 17–32. The proceedings appeared as a technical report of Universiteit Utrecht, UU-CS-2000-19.
[34] R. Hinze, Fun with phantom types, in: J. Gibbons, O. de Moor (Eds.), The Fun of Programming. Cornerstones of Computing, Palgrave Macmillan, 2003, pp. 245–262.
[35] R. Hinze, Concrete stream calculus—an extended study, J. Funct. Programming 20 (5–6) (2011) 463–535.
[36] R. Hinze, Type fusion, in: D. Pavlovic, M. Johnson (Eds.), Thirteenth International Conference on Algebraic Methodology And Software Technology, AMAST 2010, in: Lecture Notes in Computer Science, vol. 6486, Springer, Berlin, Heidelberg, 2011, pp. 92–110.
[37] R. Hinze, D.W.H. James, Reason isomorphically!, in: B.C. Oliveira, M. Zalewski (Eds.), Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming, WGP'10, ACM, New York, NY, USA, 2010, pp. 85–96.
[38] R. Hinze, S. Peyton Jones, Derivable type classes, in: G. Hutton (Ed.), Proceedings of the 2000 ACM SIGPLAN Haskell Workshop, in: Electronic Notes in Theoretical Computer Science, vol. 41(1), Elsevier Science, 2001, pp. 5–35. The preliminary proceedings appeared as a University of Nottingham technical report.
[39] J. Hughes, Type specialisation for the λ-calculus; or, A new paradigm for partial evaluation based on type inference, in: O. Danvy, R. Glück, P. Thiemann (Eds.), Partial Evaluation. Dagstuhl Castle, Germany, February 1996, in: Lecture Notes in Computer Science, vol. 1110, Springer-Verlag, 1996, pp. 183–215.
[40] G. Hutton, M. Jaskelioff, A. Gill, Factorising folds for faster functions, J. Funct. Programming 20 (2010) 353–373 (special issue 3–4) URL: http://dx.doi.org/10.1017/S0956796810000122.
[41] P. Johann, N. Ghani, Initial algebra semantics is enough!, in: S. Ronchi Della Rocca (Ed.), Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26–28, 2007, Proceedings, in: Lecture Notes in Computer Science, vol. 4583, Springer, Berlin, Heidelberg, 2007, pp. 207–222.
[42] P. Johann, N. Ghani, Foundations for structured programming with gadts, in: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'08, ACM, New York, NY, USA, 2008, pp. 297–308. URL: http://doi.acm.org/10.1145/1328438.1328475.
[43] D.E. Knuth, TCALC, 1994. http://www-cs-faculty.stanford.edu/~knuth/programs/tcalc.w.gz.
[44] D.E. Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching, 2nd ed., Addison-Wesley Publishing Company, 1998.
[45] J. Lambek, A fixpoint theorem for complete categories, Math. Zeitschr. 103 (1968) 151–161.
[46] J. Lambek, From lambda-calculus to cartesian closed categories, in: J. Seldin, J. Hindley (Eds.), To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Academic Press, 1980, pp. 376–402.
[47] R. Lämmel, S. Peyton Jones, 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, 2005.
[48] D.J. Lehmann, M.B. Smyth, Algebraic specification of data types: a synthetic approach, Math. Syst. Theory 14 (1981) 97–139.
[49] S. Mac Lane, Categories for the Working Mathematician, 2nd ed., in: Graduate Texts in Mathematics, Springer-Verlag, Berlin, 1998.
[50] G. Malcolm, Data structures and program transformation, Sci. Comput. Program. 14 (2–3) (1990) 255–280.
[51] Simon Marlow, Haskell 2010—Language Report, 2010. http://www.haskell.org/onlinereport/haskell2010.
[52] C. Martin, J. Gibbons, I. Bayley, Disciplined, efficient, generalised folds for nested datatypes, Form. Asp. Comput. 16 (1) (2004) 19–35.
[53] R. Matthes, T. Uustalu, Substitution in non-wellfounded syntax with variable binding, Theoret. Comput. Sci. 327 (1–2) (2004) 155–174.
[54] L. Meertens, Paramorphisms, Form. Asp. Comput. 4 (1992) 413–424.
[55] E. Meijer, M. Fokkinga, R. Paterson, Functional programming with bananas, lenses, envelopes and barbed wire, in: J. Hughes (Ed.), 5th ACM Conference on Functional Programming Languages and Computer Architecture, FPCA'91, Cambridge, MA, USA, in: Lecture Notes in Computer Science, vol. 523, Springer, Berlin, Heidelberg, 1991, pp. 124–144.
[56] N.P. Mendler, Inductive types and type constraints in the second-order lambda calculus, Ann. Pure Appl. Logic 51 (1–2) (1991) 159–172.
[57] A. Mycroft, Polymorphic type schemes and recursive definitions, in: M. Paul, B. Robinet (Eds.), Proceedings of the International Symposium on Programming, 6th Colloquium, Toulouse, France, in: Lecture Notes in Computer Science, vol. 167, 1984, pp. 217–228.
[58] C. Okasaki, Catenable double-ended queues. In: Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming. Amsterdam, The Netherlands, pp. 66–74, ACM SIGPLAN Notices, 32(8), August 1997, June 1997.
[59] A. Pardo, Generic accumulations, in: J. Gibbons, J. Jeuring (Eds.), Proceedings of the IFIP TC2 Working Conference on Generic Programming, Schloss Dagstuhl, vol. 243, Kluwer Academic Publishers, 2002, pp. 49–78.
[60] S. Peyton Jones, Haskell 98 Language and Libraries, Cambridge University Press, 2003.
[61] E. Poll, S. Thompson, Book review: the algebra of programming, J. Funct. Programming 9 (3) (1999) 347–354.
[62] T. Schrijvers, S. Peyton Jones, M. Sulzmann, D. Vytiniotis, Complete and decidable type inference for gadts, in: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming, ICFP'09, ACM, New York, NY, USA, 2009, pp. 341–352. http://doi.acm.org/10.1145/1596550.1596599.
[63] T. Sheard, N. Linger, Programming in ωmega, in: Z. Horváth, R. Plasmeijer, A. Soós, V. Zsók (Eds.), Central European Functional Programming School, in: Lecture Notes in Computer Science, vol. 5161, Springer, Berlin, Heidelberg, 2008, pp. 158–227. http://dx.doi.org/10.1007/978-3-540-88059-2_5.
[64] T. Sheard, T. Pasalic, Two-level types and parameterized modules, J. Funct. Programming 14 (5) (2004) 547–587.
[65] M.B. Smyth, G.D. Plotkin, The category-theoretic solution of recursive domain equations, SIAM J. Comput. 11 (4) (1982) 761–783.
[66] The Coq Development Team, 2010. The Coq proof assistant reference manual. http://coq.inria.fr.

[67] A. Thue, Über die gegenseitige Lage gleicher Teile gewisser Zeichenreihen, Skrifter udgivne af Videnskaps-Selskabet i Christiania, Mathematisk-Naturvidenskabelig Klasse 1, 1–67, 1912, reprinted in Thue's Selected Mathematical Papers (Oslo: Universitetsforlaget, 1977), 413–477.
[68] V. Trifonov, Simulating quantified class constraints, in: Haskell'03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell, ACM, New York, NY, USA, 2003, pp. 98–102.
[69] T. Uustalu, V. Vene, Primitive (co)recursion and course-of-value (co)iteration, categorically, Informatica, Lith. Acad. Sci. 10 (1) (1999) 5–26.
[70] T. Uustalu, V. Vene, Coding recursion a la Mendler (extended abstract). In: Jeuring, J. (Ed.), Proceedings of the 2nd Workshop on Generic Programming, Ponte de Lima, Portugal, pp. 69–85, The proceedings appeared as a technical report of Universiteit Utrecht, UU-CS-2000-19, 2000.
[71] T. Uustalu, V. Vene, A. Pardo, Recursion schemes from comonads, Nordic J. Comput. 8 (2001) 366–390.
[72] V. Vene, T. Uustalu, Functional programming with apomorphisms (corecursion), Proceedings of the Estonian Academy of Sciences: Physics, Mathematics 47 (3) (1998) 147–161.