Adjoint Folds and Unfolds

Or: Scything through the thicket of morphisms

Ralf Hinze

Computing Laboratory, University of Oxford
Wolfson Building, Parks Road, Oxford, OX1 3QD, England
ralf.hinze@comlab.ox.ac.uk
http://www.comlab.ox.ac.uk/ralf.hinze/

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. Fundamental laws such as fusion codify basic optimisation principles. However, most, if not all, programs require some tweaking to be given the form of an (un-) fold, and thus make them amenable to formal manipulation. In this paper, we remedy the situation by introducing adjoint folds and unfolds. We demonstrate that most programs are already of the required form and thus are directly amenable to manipulation. Central to the development is the categorical notion of an adjunction, which links adjoint (un-) folds to standard (un-) folds. We discuss a number of adjunctions and show that they are directly relevant to programming.

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 rules and optimisation rules 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 doesn't 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 the 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 can't claim originality for the idea: Bird and Paterson [1] used the approach to demonstrate that their generalised folds are uniquely defined. The purpose of the present paper 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 this approach. Specifically, the main contributions of this paper are the following:

- we introduce folds and unfolds as solutions of so-called Mendler-style equations (Mendler-style folds have been studied before [2], but we believe that they deserve to be better known);
- we argue that termination and productivity can be captured semantically using naturality;
- we show that by choosing suitable base categories mutually recursive types and parametric types 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 show their relevance to programming;
- finally, we provide a new proof of type fusion.

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 paper sufficiently self-contained, explaining the more advanced concepts as we go along. Some knowledge of the functional programming language Haskell [3] is useful, as the formal development is paralleled by a series of programming examples.

Outline The rest of the paper 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, if not most, Haskell functions fall under this umbrella. Finally, Section 5 reviews related work and Section 6 concludes.

2 Notation

We let $\mathbb{C}$, $\mathbb{D}$ and $\mathbb{E}$ range over categories. By abuse of notation $\mathbb{C}$ also denotes the class of objects: we write $A \in \mathbb{C}$ to express that A is an object of $\mathbb{C}$. The class of arrows from $A \in \mathbb{C}$ to $B \in \mathbb{C}$ is denoted $\mathbb{C}(A, B)$. If $\mathbb{C}$ is obvious from the context, we abbreviate $f \in \mathbb{C}(A, B)$ by $f : A \to B$. The latter notation is used in particular for total functions (arrows in **Set**) and functors (arrows in **Cat**). Furthermore, we let A, B, X and Y range over objects, F, G, H, I, L and R over functors, and α and β over natural transformations. Let $F, G : \mathbb{C} \to \mathbb{D}$ be two parallel functors, $\mathbb{D}^{\mathbb{C}}(F, G)$ denotes the class of natural transformations from the functor F to G, that is, $\alpha \in \mathbb{D}^{\mathbb{C}}(F, G)$ if and only if $\forall X : \mathbb{C} \,.\, \alpha\, X \in \mathbb{D}(F\, X, G\, X)$. If $\mathbb{C}$ and $\mathbb{D}$ are obvious from the context, we abbreviate $\alpha \in \mathbb{D}^{\mathbb{C}}(F, G)$ by $\alpha : F \dot{\to} G$. The inverse of an isomorphism is denoted ϕ°.

Partial applications of operators are often written as 'categorical sections', where $-$ marks the first and $=$ the optional second argument. As an example, $- * 2$ denotes the doubling function and $- * =$ multiplication. Another example is the so-called *hom-functor* $\mathbb{C}(-, =) : \mathbb{C}^{\mathsf{op}} \times \mathbb{C} \to \mathbf{Set}$, whose action on arrows is given by $\mathbb{C}(f, g)\, h = g \cdot h \cdot f$.

The formal development is complemented by a series of Haskell programs. Unfortunately, Haskell's lexical and syntactic conventions deviate somewhat from standard mathematical practise. 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've taken the liberty to typeset '::' as ':'.

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.

Haskell example 1. The datatype *Stack* models stacks of natural numbers.

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

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.

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

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

Haskell example 2. The datatype *Sequ* captures infinite sequences of natural numbers.

$$\mathbf{data}\, \mathit{Sequ} = \mathit{Next}\,(\mathit{Nat}, \mathit{Sequ})$$

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

$$\begin{aligned} &\mathit{from} : \mathit{Nat} \to \mathit{Sequ} \\ &\mathit{from} \quad n \quad = \mathit{Next}\,(n, \mathit{from}\,(n+1)) \end{aligned}$$

The function 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's customary to rephrase this problem 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 x. A fixed-point of the base function is then a solution of the recursion equation and vice versa.

Consider the type equation defining *Stack*. The base function, or rather, *base functor* of *Stack* is given by

$$\begin{aligned} &\mathbf{data}\, \mathfrak{Stack}\; \mathit{stack} = \mathfrak{Empty} \mid \mathfrak{Push}\,(\mathit{Nat}, \mathit{stack}) \\ &\mathbf{instance}\, \mathit{Functor}\, \mathfrak{Stack}\, \mathbf{where} \\ &\quad \mathit{fmap}\, f\, \mathfrak{Empty} \qquad\quad = \mathfrak{Empty} \\ &\quad \mathit{fmap}\, f\, (\mathfrak{Push}\,(n, s)) = \mathfrak{Push}\,(n, f\, s). \end{aligned}$$

The type argument of $\mathfrak{Stack}$ marks the recursive component.

All the functors underlying datatype declarations (sums of products) have two extremal fixed points: the *initial F-algebra* $\langle \mu F,\ \mathit{in} \rangle$ and the *final F-coalgebra* $\langle \nu F,\ \mathit{out} \rangle$, where $F : \mathbb{C} \to \mathbb{C}$ is the functor in question. (The proof that these fixed points exist is beyond the scope of this paper.) Very briefly, an F-algebra is a pair $\langle A,\ h \rangle$ consisting of an object $A \in \mathbb{C}$ and an arrow $h \in \mathbb{C}(F\, A, A)$. Likewise, an F-coalgebra is a pair $\langle A,\ h \rangle$ consisting of an object $A \in \mathbb{C}$ and an arrow $h \in \mathbb{C}(A, F\, A)$. (By abuse of language, we shall use the term (co-) algebra also for the components of the pair.) The objects μF and νF are the actual fixed points of the functor F: we have $F\,(\mu F) \cong \mu F$ and $F\,(\nu F) \cong \nu F$. The isomorphisms are witnessed by the arrows $\mathit{in} : F\,(\mu F) \cong \mu F$ and $\mathit{out} : \nu F \cong F\,(\nu F)$.

Some languages such as Charity [4] or Coq [5] 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 $\mathbf{Cpo}_{\perp}$, the category of complete partial orders and strict continuous functions, initial algebras and final coalgebras actually coincide [6]. By contrast, in **Set** elements of an inductive type are finite, whereas elements of a co-inductive type are potentially infinite. Operationally, an element of an inductive type is constructed in a finite number of steps, whereas an element of a coinductive type is deconstructed in a finite number of steps.

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 the initial algebra of *Sequ*'s base functor is the empty set.

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

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

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, for emphasis we keep them separate. □

Turning to the function *total*, let us first adapt the definition to the new 'two-level type' $\mu\mathfrak{Stack}$. (The term is due to Sheard [7]; one level describes the structure of the data, the other level ties the recursive knot.)

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

Now, if we abstract away from the recursive call, we obtain a non-recursive base function of type $(\mu\mathfrak{Stack} \to \mathit{Nat}) \to (\mu\mathfrak{Stack} \to \mathit{Nat})$. Functions of this type possibly have many fixed points — consider as an extreme example the identity function, which has an infinite number of fixed points. Interestingly, the problem disappears into thin air, if we additionally remove the constructor *In*.

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

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*

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

is well-defined and furthermore that the equation has exactly one solution.

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

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

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

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

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

$$x \cdot in = \Psi\, x, \qquad \text{and dually} \qquad out \cdot x = \Psi\, x, \tag{1}$$

where the unknown x has type $\mathbb{C}(\mu F, A)$ on the left and $\mathbb{C}(A, \nu F)$ on the right. Arrows defined by equations of this form are known as *Mendler-style folds and unfolds* [8]. We shall henceforth 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 an (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 $F : \mathbb{C} \to \mathbb{C}$ be some endofunctor. An *initial fixed-point equation* in the unknown $x \in \mathbb{C}(\mu F, A)$ has the syntactic form

$$x \cdot in = \Psi\, x, \tag{2}$$

where the base function Ψ has type

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

The polymorphic type of Ψ ensures *termination*: the first argument of Ψ, the recursive call of x, can only be applied to proper sub-terms of x's argument — recall that the type argument of F marks the recursive components. The naturality condition can be seen as the semantic counterpart of the *guarded-by-destructors* condition [9]. This becomes more visible, if we move the isomorphism $in : F\,(\mu F) \cong \mu F$ to the right-hand side: $x = \Psi\, x \cdot in^{\circ}$. Here in° is the destructor that guards the recursive calls.

While the definition of *length* fits nicely into the framework above, the following program doesn't.

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

$$\begin{aligned}
&\mathbf{data}\; Nat = Z \mid S\; Nat\\
&fac : Nat \;\to Nat\\
&fac \;\; Z \qquad = 1\\
&fac \;\; (S\, n) \;=\; S\, n * fac\, n
\end{aligned}$$

First of all, let us split *Nat* into two levels.

$$\begin{aligned}
&\mathbf{type}\; Nat = \mu\mathfrak{Nat}\\
&\mathbf{data}\; \mathfrak{Nat}\; nat = \mathfrak{Z} \mid \mathfrak{S}\; nat\\
&\mathbf{instance}\; Functor\; \mathfrak{Nat}\; \mathbf{where}\\
&\quad fmap\, f\; \mathfrak{Z} \qquad = \mathfrak{Z}\\
&\quad fmap\, f\; (\mathfrak{S}\, n) = \mathfrak{S}\, (f\, n)
\end{aligned}$$

The implementation of factorial is clearly terminating. However, the associated base function

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

lacks naturality. In a sense, $\mathfrak{fac}$'s type is too concrete, as it reveals that the recursive call takes a natural number. An adversary can make use of this information turning the terminating program into a non-terminating one:

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

We will get back to this example in Section 4.5. □

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 2 has a *unique solution*. When working in **Set** this result implies that the equation admits a solution that is indeed a total function. On the other hand, if the underlying category is $\mathbf{Cpo}_\perp$, then the solution is a continuous function that doesn't necessarily terminate for all its inputs, since initial algebras in $\mathbf{Cpo}_\perp$ possibly contain infinite elements.

Turning to the proof of uniqueness, let us first spell out the naturality property underlying Ψ's type: if $h \in \mathbb{C}(X_1, X_2)$, then $\mathbb{C}(F\,h, id) \cdot \Psi = \Psi \cdot \mathbb{C}(h, id)$. Recalling that $\mathbb{C}(f, g)\,h = g \cdot h \cdot f$, this unfolds to

$$\Psi\,(f \cdot h) = \Psi\,f \cdot F\,h, \tag{3}$$

for all arrows $f \in \mathbb{C}(X_2, A)$. This property implies, in particular, that Ψ is completely determined by its image of id as $\Psi\,h = \Psi\,id \cdot F\,h$. Moreover, the type of Ψ is isomorphic to $\mathbb{C}(F\,A, A)$, the type of F-algebras.

With hindsight, we generalise the above statement slightly. Let $F : \mathbb{D} \to \mathbb{C}$ be an arbitrary functor, then

$$\phi : \forall A\,B\,.\,\mathbb{C}(F\,A, B) \cong (\forall X{:}\mathbb{D}\,.\,\mathbb{D}(X, A) \to \mathbb{C}(F\,X, B)). \tag{4}$$

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

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

which is (4) in abstract clothing. Let us explicate the proof of (4). The functions witnessing the isomorphism are

$$\phi\,f = \lambda\,\kappa\,.\,f \cdot F\,\kappa \qquad \text{and} \qquad \phi^\circ\,\Psi = \Psi\,id.$$

It is easy to see that ϕ° is the left-inverse of ϕ.

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

For the opposite direction, we have to make use of the naturality property (3). (The naturality property is the same for the more general setting.)

$$\begin{aligned}
& \phi\,(\phi^{\circ}\,\Psi) \\
= \quad & \{ \text{ definition of } \phi^{\circ} \text{ and definition of } \phi \ \} \\
& \lambda\,\kappa\,.\,\Psi\, id \cdot F\,\kappa \\
= \quad & \{ \text{ naturality of } \Psi \ \} \\
& \lambda\,\kappa\,.\,\Psi\,(id \cdot \kappa) \\
= \quad & \{ \text{ identity and extensionality } \} \\
& \Psi
\end{aligned}$$

We are finally in a position to prove that Equation (2) has a *unique* solution: we show that x is a solution if and only if x is a standard fold.

$$\begin{aligned}
& x \cdot in = \Psi\, x \\
\iff \quad & \{ \text{ isomorphism } \} \\
& x \cdot in = \phi\,(\phi^{\circ}\,\Psi)\, x \\
\iff \quad & \{ \text{ definition of } \phi \text{ and definition of } \phi^{\circ} \ \} \\
& x \cdot in = \Psi\, id \cdot F\, x \\
\iff \quad & \{ \text{ initial algebras } \} \\
& x = (\!|\Psi\, id|\!)
\end{aligned}$$

The proof only requires that the initial F-algebra exists in $\mathbb{C}$.

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 $x \in \mathbb{C}(A, \nu F)$ has the syntactic form

$$out \cdot x = \Psi\, x, \qquad (5)$$

where the base function Ψ has type

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

The polymorphic type of Ψ ensures *productivity*: every recursive call is guarded by a constructor. The naturality condition captures the *guarded-by-constructors* condition [9]. This can be seen more clearly, if we move the isomorphism $out : \nu F \cong F\,(\nu F)$ to the right-hand side: $x = Out^{\circ} \cdot \Psi\, x$. Here Out° is the constructor that guards the recursive calls.

The type of Ψ is isomorphic to $\mathbb{C}(A, F\,A)$, the type of F-coalgebras. More generally, let $F : \mathbb{D} \to \mathbb{C}$, then

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

Again, this is an instance of the Yoneda lemma: now $H = \mathbb{C}(A, F\,-)$ is a covariant functor $H : \mathbb{C} \to \mathbf{Set}$ and

$$\forall H\, B\, .\, H\,B \cong (\mathbb{D}(B, -) \mathrel{\dot{\to}} H).$$

Finally, the functions witnessing the isomorphism are

$$\phi\, f = \lambda\,\kappa\, .\, F\,\kappa \cdot f \qquad \text{and} \qquad \phi^{\circ}\, \Psi = \Psi\, id.$$

In the following two 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.

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

data *Tree* = *Node Nat Trees*
data *Trees* = *Nil* | *Cons* (*Tree*, *Trees*)

Functions that consume a tree or a list of trees are typically defined by mutual value recursion.

$$\begin{aligned}
&\mathit{flattena} : \mathit{Tree} \to \mathit{Stack}\\
&\mathit{flattena}\ \ (\mathit{Node}\ n\ \mathit{ts}) = \mathit{Push}\,(n, \mathit{flattens}\ \mathit{ts})\\
&\mathit{flattens} : \mathit{Trees} \to \mathit{Stack}\\
&\mathit{flattens}\ \ (\mathit{Nil}) = \mathit{Empty}\\
&\mathit{flattens}\ \ (\mathit{Cons}\,(t, \mathit{ts})) = \mathit{stack}\,(\mathit{flattena}\ t, \mathit{flattens}\ \mathit{ts})
\end{aligned}$$

The helper function *stack* concatenates two stacks, see Example 11. □

Can we fit the above definitions into the framework of the previous section? Perhaps surprisingly, the answer is 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 $A_1 \in \mathbb{C}_1$ and $A_2 \in \mathbb{C}_2$; 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

$f_1 \in \mathbb{C}_1(A_1, B_1)$ and $f_2 \in \mathbb{C}_2(A_2, B_2)$. Identity and composition are defined componentwise:

$$id = \langle id, id \rangle \qquad \text{and} \qquad \langle f_1, f_2 \rangle \cdot \langle g_1, g_2 \rangle = \langle f_1 \cdot g_1, f_2 \cdot g_2 \rangle. \tag{7}$$

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

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

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

The Haskell types are given by projections: $\mathit{Tree} = \mathit{Outl}\,(\mu F)$ and $\mathit{Trees} = \mathit{Outr}\,(\mu F)$. The functions *flattena* and *flattens* are handled accordingly, we bundle them to an arrow

$$\mathit{flatten} \in (\mathbb{C} \times \mathbb{C})(\mu F, \Delta \mathit{Stack}),$$

where $\Delta : \mathbb{C} \to \mathbb{C} \times \mathbb{C}$ is the *diagonal functor* defined by $\Delta A = \langle A, A \rangle$ and $\Delta f = \langle f, f \rangle$. The Haskell functions are then given by projections: $\mathit{flattena} = \mathit{Outl}\,\mathit{flatten}$ and $\mathit{flattens} = \mathit{Outr}\,\mathit{flatten}$.

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 $\mathbb{D}$.

$$
\begin{array}{ll}
& x \cdot in = \Psi\, x \\
\iff & \{ \text{ surjective pairing: } f = \langle \mathit{Outl}\, f, \mathit{Outr}\, f \rangle \ \} \\
& \langle \mathit{Outl}\, x, \mathit{Outr}\, x \rangle \cdot \langle \mathit{Outl}\, in, \mathit{Outr}\, in \rangle = \Psi\, \langle \mathit{Outl}\, x, \mathit{Outr}\, x \rangle \\
\iff & \{ \text{ set } x_1 = \mathit{Outl}\, x,\ x_2 = \mathit{Outr}\, x \text{ and } in_1 = \mathit{Outl}\, in,\ in_2 = \mathit{Outr}\, in \ \} \\
& \langle x_1, x_2 \rangle \cdot \langle in_1, in_2 \rangle = \Psi\, \langle x_1, x_2 \rangle \\
\iff & \{ \text{ definition of composition } \} \\
& \langle x_1 \cdot in_1, x_2 \cdot in_2 \rangle = \Psi\, \langle x_1, x_2 \rangle \\
\iff & \{ \text{ surjective pairing: } f = \langle \mathit{Outl}\, f, \mathit{Outr}\, f \rangle \ \} \\
& \langle x_1 \cdot in_1, x_2 \cdot in_2 \rangle = \langle \mathit{Outl}\,(\Psi\, \langle x_1, x_2 \rangle), \mathit{Outr}\,(\Psi\, \langle x_1, x_2 \rangle) \rangle \\
\iff & \{ \text{ equality of functions } \} \\
& x_1 \cdot in_1 = (\mathit{Outl} \cdot \Psi)\, \langle x_1, x_2 \rangle \quad \text{and} \quad x_2 \cdot in_2 = (\mathit{Outr} \cdot \Psi)\, \langle x_1, x_2 \rangle \\
\iff & \{ \text{ set } \Psi_1 = \mathit{Outl} \cdot \Psi \text{ and } \Psi_2 = \mathit{Outr} \cdot \Psi \ \} \\
& x_1 \cdot in_1 = \Psi_1\, \langle x_1, x_2 \rangle \quad \text{and} \quad x_2 \cdot in_2 = \Psi_2\, \langle x_1, x_2 \rangle
\end{array}
$$

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's a simple exercise to bring the equations of Example 4 into this form.

Haskell definition 2. Mutually recursive datatypes can be modelled as follows.

$$\mathbf{newtype}\, \mu_1\, f_1\, f_2 = \mathit{In}_1\, \{ in_1^{\circ} : f_1\, (\mu_1\, f_1\, f_2)\, (\mu_2\, f_1\, f_2) \}$$
$$\mathbf{newtype}\, \mu_2\, f_1\, f_2 = \mathit{In}_2\, \{ in_2^{\circ} : f_2\, (\mu_1\, f_1\, f_2)\, (\mu_2\, f_1\, f_2) \}$$

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

Haskell example 5. The base functors of *Tree* and *Trees* are

$$\mathbf{data}\ \mathfrak{Tree}\ tree\ trees = \mathfrak{Node}\ Nat\ trees$$

$$\mathbf{data}\ \mathfrak{Trees}\ tree\ trees = \mathfrak{Nil} \mid \mathfrak{Cons}\ tree\ trees.$$

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

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

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

$$\begin{aligned}
&flattena : \mu_1\, \mathfrak{Tree}\, \mathfrak{Trees} \to Stack\\
&flattena\ (In_1\, x) = \mathfrak{flattena}\,(flattena, flattens)\, x\\
&flattens : \mu_2\, \mathfrak{Tree}\, \mathfrak{Trees} \to Stack\\
&flattens\ (In_2\, x) = \mathfrak{flattens}\,(flattena, flattens)\, x
\end{aligned}$$

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

Perhaps surprisingly, 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.

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

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

In Haskell datatypes can be parametrised by types.

Haskell example 6. The type of perfectly balanced, binary leaf trees, perfect trees for short, is given by

$$\begin{aligned}
&\mathbf{data}\ Perfect\ a = Zero\ a \mid Succ\,(Perfect\,(a, a))\\
&\mathbf{instance}\ Functor\ Perfect\ \mathbf{where}\\
&\quad fmap\, f\,(Zero\, x) = Zero\,(f\, x)\\
&\quad fmap\, f\,(Succ\, x) = Succ\,(fmap\,(f \times f)\, x).\\
&(f \times g)\,(a, b) = (f\ a, g\ b)
\end{aligned}$$

The type 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.

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

The function *size* calculates the size of a perfect tree, making good use of the balance condition. The definition requires *polymorphic recursion* [11], as the recursive call has type $\mathit{Perfect}\,(a, a) \to \mathit{Nat}$, 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 $F : \mathbb{C} \to \mathbb{D}$; an arrow of $\mathbb{D}^{\mathbb{C}}(F, G)$ is a natural transformation $\alpha : F \dot{\to} G$. (As an aside, $\mathbb{D}^{\mathbb{C}}$ is the exponential in **Cat**.)

Now, the base functor underlying *Perfect* is an endofunctor over a functor category:

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

The higher-order functor F sends a functor to a functor. Since its fixed point $\mathit{Perfect} = \mu F$ 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

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

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

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

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

To capture the fact that μf and νf are functors whenever f is a higher-order functor, we need an extension of the Haskell 98 class system.

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

The declarations use a so-called *polymorphic predicate* [12], 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 98 [13], but the resulting code is somewhat clumsy. □

category	initial fixed-point equation $x \cdot \mathit{in} = \Psi\, x$	final fixed-point equation $\mathit{out} \cdot x = \Psi\, x$
Set	inductive type standard fold	coinductive type standard unfold
Cpo	—	continuous coalgebra (domain) continuous unfold (F locally continuous in $\mathbf{Cpo}_\perp$)
$\mathbf{Cpo}_\perp$	continuous algebra (domain) strict continuous fold	continuous coalgebra (domain) strict continuous unfold
	(F locally continuous in $\mathbf{Cpo}_\perp$, $\mu F \cong \nu F$)	
$\mathbb{C} \times \mathbb{D}$	mutually recusive 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

Table 1. Initial algebras and final coalgebras in different categories.

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

$$\mathbf{data}\, \mathfrak{Perfect}\, \mathit{perfect}\; a = \mathfrak{Zero}\; a \mid \mathfrak{Succ}\, (\mathit{perfect}\, (a, a))$$

$$\begin{aligned}
&\mathbf{instance}\, (\mathit{Functor}\; x) \Rightarrow \mathit{Functor}\, (\mathfrak{Perfect}\; x)\; \mathbf{where}\\
&\quad \mathit{fmap}\; f\; (\mathfrak{Zero}\; x) \;= \mathfrak{Zero}\, (f\; x)\\
&\quad \mathit{fmap}\; f\; (\mathfrak{Succ}\; x) = \mathfrak{Succ}\, (\mathit{fmap}\, (f \times f)\; x)
\end{aligned}$$

Accordingly, the base function of *size* is a higher-order natural transformation that takes natural transformations to natural transformations.

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

$$\begin{aligned}
&\mathit{size} : \forall a\, .\, \mu\mathfrak{Perfect}\; a \to \mathit{Nat}\\
&\mathit{size} \quad (\mathit{In}\; x) \quad = \mathfrak{size}\; \mathit{size}\; x
\end{aligned}$$

The resulting equation fits the pattern of an initial fixed-point equation. Consequently, it has a unique solution. □

The bottom line is that no new theory is needed to deal with parametric datatypes and polymorphic functions over them.

Table 1 summarises our findings so far.

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 append in the introduction. Here is another example along those lines.

Haskell example 8. The function *shunt* pushes the elements of the first onto the second stack.

$$\begin{array}{lll} \mathit{shunt} : (\mu\mathfrak{Stack}, & \mathit{Stack}) & \to \mathit{Stack} \\ \mathit{shunt}\ \ (\mathit{In}\,\mathfrak{Empty}, & y) & = y \\ \mathit{shunt}\ \ (\mathit{In}\,(\mathfrak{Push}\,(a, x)), y) & & = \mathit{shunt}\,(x, \mathit{In}\,(\mathfrak{Push}\,(a, y))) \end{array}$$

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

Haskell example 9. The functions *nats* and *squares* generate the sequence of natural numbers interleaved with the sequence of squares.

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

The two definitions are not instances of final fixed-point equations, as the functions are mutually recursive, but the datatype isn't. □

In Example 8 the element of the initial algebra is embedded in a context. The central idea of this paper is to model this context by a functor, generalising fixed-point equations to

$$x \cdot L\,\mathit{in} = \Psi\,x, \qquad \text{and dually} \qquad R\,\mathit{out} \cdot x = \Psi\,x, \tag{8}$$

where the unknown x has type $\mathbb{C}(L\,(\mu F), A)$ on the left and $\mathbb{C}(A, R\,(\nu F))$ on the right. The functor L models the context of μF, in the case of *shunt*, $L = - \times \mathit{Stack}$. Dually, R allows x to return an element of νF embedded in a context. Section 4.5 discusses a suitable choice for R in Example 9. Of course, we can't use any plain, old functor for L and R; for reasons to become clear later on, we require them to be adjoint: $L \dashv R$.

Let $\mathbb{C}$ and $\mathbb{D}$ be categories. The functors L and R

$$\mathbb{C} \underset{R}{\overset{L}{\underset{\perp}{\rightleftarrows}}} \mathbb{D}$$

are *adjoint* if and only if there is a bijection

$$\phi : \forall A\, B\, .\, \mathbb{C}(L\, A, B) \cong \mathbb{D}(A, R\, B),$$

that is natural both in A and B. The isomorphism ϕ is sometimes called the *adjoint transposition*.

The adjoint transposition allows us to eliminate L in the source and R in the target of an arrow, which is the key for showing that generalised fixed-point equations (8) 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 $L \dashv R$ be an adjoint pair of functors $L : \mathbb{D} \to \mathbb{C}$ and $R : \mathbb{C} \to \mathbb{D}$ and let $F : \mathbb{D} \to \mathbb{D}$ be some endofunctor. An *adjoint initial fixed-point equation* in the unknown $x \in \mathbb{C}(L\,(\mu F), A)$ has the syntactic form

$$x \cdot L\, in = \Psi\, x, \tag{9}$$

where the base function Ψ has type

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

The unique solution of (9) is called an *adjoint fold*.

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

$$\phi\,(f \cdot L\, h) = \phi\, f \cdot h. \tag{10}$$

We reason as follows.

$$\begin{aligned}
& x \cdot L\, in = \Psi\, x \\
\iff & \quad \{ \text{ adjunction } \} \\
& \phi\,(x \cdot L\, in) = \phi\,(\Psi\, x) \\
\iff & \quad \{ \text{ naturality of } \phi \ \} \\
& \phi\, x \cdot in = \phi\,(\Psi\, x) \\
\iff & \quad \{ \text{ adjunction } \} \\
& \phi\, x \cdot in = (\phi \cdot \Psi \cdot \phi^{\circ})\,(\phi\, x) \\
\iff & \quad \{ \text{ Section 3.1 } \} \\
& \phi\, x = (\!|(\phi \cdot \Psi \cdot \phi^{\circ})\; id|\!) \\
\iff & \quad \{ \text{ adjunction } \} \\
& x = \phi^{\circ}\, (\!|(\phi \cdot \Psi \cdot \phi^{\circ})\; id|\!)
\end{aligned}$$

In three simple steps we have transformed the adjoint fold $x \in \mathbb{C}(L\,(\mu F), A)$ into the standard fold $\phi\, x \in \mathbb{D}(\mu F, R\, A)$ and, furthermore, the adjoint base function $\Psi : \forall X \,.\, \mathbb{C}(L\,X, A) \to \mathbb{C}(L\,(F\,X), A)$ into the standard base function $(\phi \cdot \Psi \cdot \phi^{\circ}) : \forall X \,.\, \mathbb{D}(X, R\,A) \to \mathbb{D}(F\,X, R\,A)$. We have shown in Section 3.1 that the resulting equation has a unique solution. The arrow $\phi\,x$ is called the *transpose* of x.

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 $x \in \mathbb{D}(\mu F, R\,A)$ has the syntactic form

$$R\,out \cdot x = \Psi\,x, \tag{11}$$

where the base function Ψ has type

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

The unique solution of (11) is called an *adjoint unfold*.

The proof of uniqueness relies on the fact that the inverse ϕ° of the adjoint transposition is natural in B: $\mathbb{C}(id, h) \cdot \phi^{\circ} = \phi^{\circ} \cdot \mathbb{D}(id, R\,h)$, that is,

$$\phi^{\circ}\,(R\,h \cdot f) = h \cdot \phi^{\circ}\,f. \tag{12}$$

We leave it to the reader to fill in the details.

4.3 Identity: $Id \dashv Id$

The simplest example of an adjunction is $Id \dashv Id$, which shows that adjoint fixed-point equations (8) subsume fixed-point equations (1).

In the following sections we explore more interesting examples of adjunctions. Each section is structured as follows: we introduce an adjunction, specialise Equations (8) to the adjoint functors, and then provide some Haskell examples that fit the pattern.

4.4 Currying: $- \times X \dashv -^{X}$

The best-known example of an adjunction is perhaps 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.

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

The object B^{X} is the *exponential* of X and 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. In the case of **Set**, the isomorphisms are given by

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

Let's specialise the adjoint equations to $L = - \times X$ and $R = -^{X}$.

$$\begin{aligned} & x \cdot L\,in = \Psi\,x \\ \Longleftrightarrow \quad & \{ \text{ definition of } L \ \} \\ & x \cdot (in \times id) = \Psi\,x \\ \Longleftrightarrow \quad & \{ \text{ pointwise } \} \\ & x\,(in\,a, c) = \Psi\,x\,(a, c) \end{aligned}
\qquad
\begin{aligned} & R\,out \cdot x = \Psi\,x \\ \Longleftrightarrow \quad & \{ \text{ definition of } R \ \} \\ & (out\cdot) \cdot x = \Psi\,x \\ \Longleftrightarrow \quad & \{ \text{ pointwise } \} \\ & out\,(x\;a\;c) = \Psi\,x\;a\;c \end{aligned}$$

The adjoint fold takes two arguments, an element of an initial algebra and a second argument (often an accumulator), both of which are available on the right-hand side. The transposed fold is then a higher-order function that yields an exponential. Dually, a curried unfold is transformed into an uncurried unfold.

Haskell example 10. To turn the definition of *shunt* 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. The adjoint functors are $L = - \times Stack$ and $R = -^{Stack}$.

$$\begin{aligned}
&\mathfrak{shunt} : \forall x\,.\,(L\,x \to Stack) \to (L\,(\mathfrak{Stack}\,x) && \to Stack) \\
&\mathfrak{shunt} \quad\; shunt \quad\; (\mathfrak{Empty}, \quad y) && = y \\
&\mathfrak{shunt} \quad\; shunt \quad\; (\mathfrak{Push}\,(a, x), y) && = shunt\,(x, In\,(\mathfrak{Push}\,(a, y)))
\end{aligned}$$

$$\begin{aligned}
&shunt : L\,(\mu\mathfrak{Stack}) \to Stack \\
&shunt \quad (In\,x, y) \quad = \mathfrak{shunt}\,shunt\,(x, y)
\end{aligned}$$

The definition of *shunt* matches exactly the scheme for adjoint initial fixed-point equations. The transposed fold, $\phi\,shunt$,

$$\begin{aligned}
&shunt' : \mu\mathfrak{Stack} && \to R\,Stack \\
&shunt' \quad (In\,\mathfrak{Empty}) && = \lambda y \to y \\
&shunt' \quad (In\,(\mathfrak{Push}\,(a, x))) && = \lambda y \to shunt'\,x\,(In\,(\mathfrak{Push}\,(a, y)))
\end{aligned}$$

is the curried variant of *shunt*. □

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

Haskell example 11. The type of lists is given as the initial algebra of a higher-order base functor of kind $(\star \to \star) \to (\star \to \star)$.

$$\begin{aligned}
&\mathbf{data}\,\mathfrak{List}\,list\;a = \mathfrak{Nil} \mid \mathfrak{Cons}\,(a, list\;a) \\
&\mathbf{instance}\,(Functor\;list) \Rightarrow Functor\,(\mathfrak{List}\,list)\,\mathbf{where} \\
&\quad fmap\,f\,\mathfrak{Nil} \qquad\qquad\; = \mathfrak{Nil} \\
&\quad fmap\,f\,(\mathfrak{Cons}\,(a, x)) = \mathfrak{Cons}\,(f\;a, fmap\,f\;x)
\end{aligned}$$

Lists generalise stacks, sequences of natural numbers, to an arbitrary element type. The function *append* concatenates two lists.

$$\begin{aligned}
&append : \forall a\,.\,(\mu\mathfrak{List}\,a, && List\;a) \to List\;a \\
&append \quad (In\,\mathfrak{Nil}, && y) \quad = y \\
&append \quad (In\,(\mathfrak{Cons}\,(a, x)), y) && \quad\;\; = In\,(\mathfrak{Cons}\,(a, append\,(x, y)))
\end{aligned}$$

Concatenation generalises the function *stack* (not shown) to sequences of an arbitrary element type. □

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

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

All that is left to do is to find the right adjoint of the lifted product $- \mathbin{\dot{\times}} H$. (One could be led to think that $F \mathbin{\dot{\times}} H \mathbin{\dot{\to}} G \cong F \mathbin{\dot{\to}} (H \mathbin{\dot{\to}} G)$, but this doesn't make any sense as $H \mathbin{\dot{\to}} G$ is not a functor. Also, lifting exponentials pointwise $G^H\, A = (G\, A)^{H\, A}$ doesn't work, because the data doesn't define a functor as the exponential is contravariant in its first argument.) For simplicity, let us assume that the functor category is $\mathbf{Set}^{\mathbb{C}}$ so that $G^H : \mathbb{C} \to \mathbf{Set}$. We reason as follows:

$$\begin{aligned}
& G^H\, A \\
\cong\ & \quad \{\ \text{Yoneda lemma}\ \} \\
& \mathbb{C}(A, -) \mathbin{\dot{\to}} G^H \\
\cong\ & \quad \{\ \text{requirement: } - \mathbin{\dot{\times}} H \dashv -^H\ \} \\
& \mathbb{C}(A, -) \mathbin{\dot{\times}} H \mathbin{\dot{\to}} G \\
\cong\ & \quad \{\ \text{natural transformation}\ \} \\
& \forall X{:}\mathbb{C}\ .\ \mathbb{C}(A, X) \times H\, X \to G\, X \\
\cong\ & \quad \{\ - \times X \dashv -^X\ \} \\
& \forall X{:}\mathbb{C}\ .\ \mathbb{C}(A, X) \to (G\, X)^{H\, X}.
\end{aligned}$$

If we set $G^H\, A = \forall X{:}\mathbb{C}\ .\ \mathbb{C}(A, X) \to (G\, X)^{H\, X}$ and $G^H\, f = \Lambda\, X\ .\ \mathbb{C}(f, id) \to id$, then $- \mathbin{\dot{\times}} H \dashv -^H$. It is instructive to spell out the transpositions:

$$\begin{aligned}
\phi\, \sigma\ &= \Lambda\, A\ .\ \lambda\, s\ .\ \Lambda\, X\ .\ \lambda\, \kappa\ .\ \lambda\, t\ .\ \sigma\, X\, (F\, \kappa\, s, t) \\
\phi^{\circ}\, \tau &= \Lambda\, A\ .\ \lambda\, (s, t)\ .\ \tau\, A\, s\, A\, id\, t.
\end{aligned}$$

Haskell definition 4. The definition of exponentials goes beyond Haskell 98, as it requires rank-2 types (the data constructor *Exp* has a rank-2 type).

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

Morally, h and g are functors, as well. However, their mapping functions are not needed to define the *Exp h g* instance of *Functor*. □

Haskell example 12. Returning to Example 11, we may conclude that the defining equation of *append* has a unique solution. Its transpose of type $List \mathbin{\dot{\to}} List^{List}$ is interesting as it combines *append* with *fmap*:

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

For clarity, we have inlined the definition of *Exp List List*. □

4.5 Mutual value recursion: $(+) \dashv \boldsymbol{\Delta} \dashv (\times)$

The functions *nats* and *squares* introduced in Example 9 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 ones. Nonetheless, the correspondence suggests to view *nats* and *squares* as a single arrow in a product category.

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

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 one 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

$$\phi : \forall A\, B \,.\, (A_1 + A_2 \to B) \cong (A_1 \to B) \times (A_2 \to B).$$

The adjunction captures the observation that we can represent a pair of functions to the same codomain by a single function from the coproduct of the domains. The adjoint transpositions are given by

$$\phi\, f = \langle f \cdot inl, f \cdot inr \rangle \qquad \text{and} \qquad \phi^{\circ}\, \langle f_1, f_2 \rangle = f_1 \triangledown f_2.$$

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

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

$$
\begin{aligned}
& \Delta out \cdot x = \Psi\, x \\
\iff & \quad \{ \text{ definition of } \Delta \} \\
& \langle out, out \rangle \cdot x = \Psi\, x \\
\iff & \quad \{ \text{ surjective pairing: } f = \langle Outl\, f, Outr\, f \rangle \} \\
& \langle out, out \rangle \cdot \langle Outl\, x, Outr\, x \rangle = \Psi\, \langle Outl\, x, Outr\, x \rangle \\
\iff & \quad \{ \text{ set } x_1 = Outl\, x \text{ and } x_2 = Outr\, x \} \\
& \langle out, out \rangle \cdot \langle x_1, x_2 \rangle = \Psi\, \langle x_1, x_2 \rangle \\
\iff & \quad \{ \text{ definition of composition } \} \\
& \langle out \cdot x_1, out \cdot x_2 \rangle = \Psi\, \langle x_1, x_2 \rangle \\
\iff & \quad \{ \text{ surjective pairing: } f = \langle Outl\, f, Outr\, f \rangle \} \\
& \langle out \cdot x_1, out \cdot x_2 \rangle = \langle Outl\, (\Psi\, \langle x_1, x_2 \rangle), Outr\, (\Psi\, \langle x_1, x_2 \rangle) \rangle \\
\iff & \quad \{ \text{ equality of functions } \} \\
& out \cdot x_1 = (Outl \cdot \Psi)\, \langle x_1, x_2 \rangle \quad \text{and} \quad out \cdot x_2 = (Outr \cdot \Psi)\, \langle x_1, x_2 \rangle \\
\iff & \quad \{ \text{ set } \Psi_1 = Outl \cdot \Psi \text{ and } \Psi_2 = Outr \cdot \Psi \} \\
& out \cdot x_1 = \Psi_1\, \langle x_1, x_2 \rangle \quad \text{and} \quad out \cdot x_2 = \Psi_2\, \langle x_1, x_2 \rangle
\end{aligned}
$$

The resulting equations are similar to those of Section 3.3, except that now the destructor *out* is the same in both equations.

Haskell example 13. Continuing Haskell Example 9, the base functions of *nats* and *squares* are given by

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

The recursion equations

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

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

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

The datatype *Either* defined **data** $Either\ a\ b = Left\ a \mid Right\ b$ is Haskell's coproduct. □

Turning 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

$$\phi : \forall A\, B\ .\ (A \to B_1) \times (A \to B_2) \cong (A \to B_1 \times B_2).$$

We can represent a pair of functions with the same domain by a single function to the product of the codomains. The bijection is witnessed by

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

Specialising the adjoint initial fixed-point equation yields

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

Haskell example 14. Perhaps surprisingly, we can use mutual value recursion to fit the definition of factorial, see Example 3, into the framework. The definition of *fac* has the form of a *paramorphism* [14], as the argument that drives the recursion is not only 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.

$$\begin{aligned}
&\mathit{fac} : \mu\mathfrak{Nat} \to \mathit{Nat}\\
&\mathit{fac}\ (\mathit{In}\ \mathfrak{Z}) = 1\\
&\mathit{fac}\ (\mathit{In}\ (\mathfrak{S}\ n)) = \mathit{In}\ (\mathfrak{S}\ (\mathit{id}\ n)) * \mathit{fac}\ n
\end{aligned}$$

$$\begin{aligned}
&\mathit{id} : \mu\mathfrak{Nat} \to \mathit{Nat}\\
&\mathit{id}\ (\mathit{In}\ \mathfrak{Z}) = \mathit{In}\ \mathfrak{Z}\\
&\mathit{id}\ (\mathit{In}\ (\mathfrak{S}\ n)) = \mathit{In}\ (\mathfrak{S}\ (\mathit{id}\ n))
\end{aligned}$$

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

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

$$\begin{aligned}
&\mathfrak{id} : \forall x\ .\ (x \to \mathit{Nat}, x \to \mathit{Nat}) \to (\mathfrak{Nat}\ x \to \mathit{Nat})\\
&\mathfrak{id}\ (\mathit{fac}, \mathit{id})\ (\mathfrak{Z}) = \mathit{In}\ \mathfrak{Z}\\
&\mathfrak{id}\ (\mathit{fac}, \mathit{id})\ (\mathfrak{S}\ n) = \mathit{In}\ (\mathfrak{S}\ (\mathit{id}\ n))
\end{aligned}$$

The transposed fold has type $\mu\mathfrak{Nat} \to \mathit{Nat} \times \mathit{Nat}$ and corresponds to the usual encoding of paramorphisms as folds (using tupling).

As an aside, the trick doesn't work for the 'base function' $\mathfrak{bogus}$, as the resulting function still lacks naturality. □

Haskell example 15. Incidentally, we can employ a similar approach to also fit the Fibonacci function into the framework.

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

The definition is sometimes characterised as a *histomorphism* [15] because in the third equation *fib* depends on two previous values, rather than only one. Now, setting $\mathit{fib}'\ n = \mathit{fib}\ (S\ n)$, 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, [16].)

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

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

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 [17]. □

The double adjunction $(+) \dashv \Delta \dashv (\times)$ actually 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.

4.6 Mutual value recursion: $\sum i \in \mathbb{I} \dashv \boldsymbol{\Delta} \dashv \prod i \in \mathbb{I}$

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). 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: $\mathbb{C}^{\mathbf{2}}$, where $\mathbf{2}$ is some two-element set. To be able to deal with an arbitrary number of functions we simply generalise from $\mathbf{2}$ to an arbitrary index set.

A set forms a so-called *discrete category*: the objects are the elements of the set and the only arrows are the identities. 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: $A \in \mathbb{C}^{\mathbb{I}}$ if and only if $\forall i \in \mathbb{I} \,.\, A_i \in \mathbb{C}$ and $f \in \mathbb{C}^{\mathbb{I}}(A, B)$ if and only if $\forall i \in \mathbb{I} \,.\, f_i \in \mathbb{C}(A_i, B_i)$.

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).

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

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

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

It is worth singling out a special case of the construction that we shall need later on. First of all, note that

$$\mathbb{C}^{\mathbb{I}}(\Delta X, \Delta Y) \cong \mathbb{I} \to \mathbb{C}(X, Y)$$

Consequently, if the summands of the sum and the factors of the product are the same, $A = \Delta X$ and $B = \Delta Y$, we obtain another adjoint situation:

$$\forall X\ Y \,.\, \mathbb{C}(\textstyle\sum \mathbb{I} \,.\, X, Y) \cong \mathbb{I} \to \mathbb{C}(X, Y) \cong \mathbb{C}(X, \textstyle\prod \mathbb{I} \,.\, Y). \qquad (13)$$

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

4.7 Type application: $\boldsymbol{Lsh_X} \dashv (- \boldsymbol{X}) \dashv \boldsymbol{Rsh_X}$

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.

Haskell example 16. The function *sum* sums a list of natural numbers.

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

□

The definition of *sum* looks suspiciously like a fold, but it isn't, as it does not have the right type. The corresponding function on perfect trees doesn't even resemble a fold.

Haskell example 17. The function *sump* sums a perfect tree of natural numbers.

$$
\begin{aligned}
&sump : \mu\mathfrak{Perfect}\,Nat && \to Nat \\
&sump \quad (In\,(\mathfrak{Zero}\,x)) && = x \\
&sump \quad (In\,(\mathfrak{Succ}\,x)) && = sump\,(fmap\; plus\; x)
\end{aligned}
$$

Here, *plus* is the uncurried variant of addition: $plus\,(a, b) = a + b$. Note that the recursive call is not applied to a subterm of $\mathfrak{Succ}\,x$. In fact, it can't, as x has type $Perfect\,(Nat, Nat)$. (As an aside, this definition requires the functor instance for μ, see 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 $X \in \mathbb{D}$ define $T_X : \mathbb{C}^{\mathbb{D}} \to \mathbb{C}$ by $T_X\,F = F\,X$ and $T_X\,\alpha = \alpha\,X$. (The natural transformation α is applied to the object X. In Haskell this type application is invisible, which is why we can't see that *sum* isn't a standard fold.) It is easy to show that this data defines a functor: $T_X\,id = id\,X = id_X$ and $T_X\,(\alpha \cdot \beta) = (\alpha \cdot \beta)\,X = \alpha\,X \cdot \beta\,X = T_X\,\alpha \cdot T_X\,\beta$. Using T_X we can assign *sum* the type $T_{Nat}\,List \to Nat$. All that is left to do is to check whether T_X is part of an adjunction. It turns out that T_X has, in fact, both a left and a right adjoint. We choose to derive the left adjoint.

$$
\begin{aligned}
& \mathbb{C}(A, T_X\,B) \\
\cong\quad & \{\ \text{definition of } T_X\ \} \\
& \mathbb{C}(A, B\,X) \\
\cong\quad & \{\ \text{Yoneda (6)}\ \} \\
& \forall Y{:}\mathbb{D}\,.\,\mathbb{D}(X, Y) \to \mathbb{C}(A, B\,Y) \\
\cong\quad & \{\ \text{definition of a copower: } \mathbb{I} \to \mathbb{C}(X, Y) \cong \mathbb{C}(\textstyle\sum \mathbb{I}\,.\,X, Y)\ \} \\
& \forall Y{:}\mathbb{D}\,.\,\mathbb{C}(\textstyle\sum \mathbb{D}(X, Y)\,.\,A, B\,Y) \\
\cong\quad & \{\ \text{define } Lsh_X\,A = \Lambda\,Y{:}\mathbb{D}\,.\,\textstyle\sum \mathbb{D}(X, Y)\,.\,A\ \} \\
& \forall Y{:}\mathbb{D}\,.\,\mathbb{C}(Lsh_X\,A\,Y, B\,Y) \\
\cong\quad & \{\ \text{natural transformation}\ \} \\
& Lsh_X\,A \stackrel{.}{\to} B
\end{aligned}
$$

We call Lsh_X the *left shift* of X, for want of a better name. Dually, the right adjoint is $Rsh_X\ B = \Lambda\ Y{:}\mathbb{D}\ .\ \prod \mathbb{D}(Y, X)\ .\ B$, the *right shift* of X. Recall that in **Set**, the copower $\sum \mathbb{I}\ .\ A$ is the cartesian product $\mathbb{I} \times A$ and the power $\prod \mathbb{I}\ .\ A$ is the set of functions $\mathbb{I} \to A$. This correspondence suggests the following Haskell implementation. However, it is important to note that $\mathbb{I}$ is a set, not an object.

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

$$
\begin{array}{l}
\mathbf{newtype}\ Lsh_x\ a\ y = Lsh\ (x \to y, a) \\
\mathbf{instance}\ Functor\ (Lsh_x\ a)\ \mathbf{where} \\
\quad fmap\ f\ (Lsh\ (\kappa, a)) = Lsh\ (f \cdot \kappa, a) \\
\mathbf{newtype}\ Rsh_x\ b\ y = Rsh\ ((y \to x) \to b) \\
\mathbf{instance}\ Functor\ (Rsh_x\ b)\ \mathbf{where} \\
\quad fmap\ f\ (Rsh\ g) \quad = Rsh\ (\lambda\kappa \to g\ (\kappa \cdot f))
\end{array}
$$

The functor $Rsh_x\ b$ implements a continuation type — often, but not necessarily the types x and b are identical. □

As usual, let's specialise the adjoint equations.

$$
\begin{array}{ll}
\quad x \cdot T_X\ in = \Psi\ x & \quad T_X\ out \cdot x = \Psi\ x \\
\iff \quad \{\ \text{definition of } T_X\ \} & \iff \quad \{\ \text{definition of } T_X\ \} \\
\quad x \cdot in\ X = \Psi\ x & \quad out\ X \cdot x = \Psi\ x
\end{array}
$$

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

Haskell example 18. The base function of $sump$ is given by

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

The definition requires the $\mathfrak{Perfect}$ functor instance, which in turn induces the $Functor\ x$ context. The transpose of $sump$ is a fold that returns a higher-order function.

$$
\begin{array}{llll}
sump' : \forall x\ .\ Perfect\ x \to (x \to Nat) \to Nat & & & \\
sump' \quad (Zero\ x) & = \lambda\kappa & \to \kappa\ x \\
sump' \quad (Succ\ x) & = \lambda\kappa & \to sump'\ x\ (plus \cdot (\kappa \times \kappa))
\end{array}
$$

For clarity, we have inlined the definition of $Rsh_{Nat}\ Nat$ and slightly optimised the result. Quite interestingly, the transformation turns a *generalised fold* in the sense of Bird and Paterson [1] into an *efficient generalised fold* in the sense of Hinze [18]. Both versions have a linear running time, but $sump'$ avoids the repeated invocations of the mapping function ($fmap\ plus$). □

4.8 Type composition: $Lan_I \dashv (- \cdot I) \dashv Ran_I$

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.

Haskell example 19. The function *concat* generalises the binary function *append* to a list of lists.

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

□

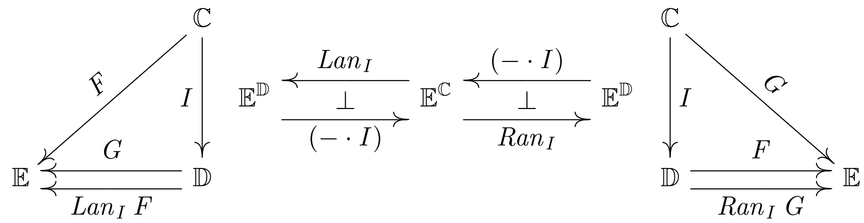
The definition has the structure of an ordinary fold, but again the type is not quite right: we need a natural transformation of type $\mu\mathfrak{List} \dot{\to} G$, but *concat* has type $\mu\mathfrak{List} \cdot List \dot{\to} List$. Can we fit the definition into the framework of adjoint equations? 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: $(- \cdot List)\,(\mu\mathfrak{List}) \dot{\to} List$. (We interpret $List \cdot List$ as $(- \cdot List)\ List$ rather than $(List \cdot -)\ List$ because the outer list, written $\mu\mathfrak{List}$ for emphasis, drives the recursion.)

Given a functor $I : \mathbb{C} \to \mathbb{D}$, define the higher-order functor $Pre_I : \mathbb{E}^{\mathbb{D}} \to \mathbb{E}^{\mathbb{C}}$ by $Pre_I\ F = F \cdot I$ and $Pre_I\ \alpha = \alpha \cdot I$. (The natural transformation α is composed with the functor I. In Haskell, type composition is invisible. Again, this is why the definition of *concat* looks like a fold, but it isn't.) As usual, we should make sure that the data actually defines a functor: $Pre_I\ id_F = id_F \cdot I = id_{F \cdot I}$ and $Pre_I\ (\alpha \cdot \beta) = (\alpha \cdot \beta) \cdot I = (\alpha \cdot I) \cdot (\beta \cdot I) = Pre_I\ \alpha \cdot Pre_I\ \beta$. Using the higher-order functor we can assign *concat* the type $Pre_{List}\ (\mu\mathfrak{List}) \dot{\to} List$. As a second step, we have to construct the right adjoint of the higher-order functor. Similar to the situation of the previous section, Pre_I has both a left and a right adjoint. For variety, we derive the latter.

$$\begin{aligned} & F \cdot I \dot{\to} G \\ \cong\ & \{ \text{ natural transformation as an end } \} \\ & \forall A{:}\mathbb{C}\,.\,\mathbb{E}(F\,(I\,A), G\,A) \\ \cong\ & \{ \text{ Yoneda (4) } \} \\ & \forall A{:}\mathbb{C}\,.\,\forall X{:}\mathbb{D}\,.\,\mathbb{D}(X, I\,A) \to \mathbb{E}(F\,X, G\,A) \\ \cong\ & \{ \text{ definition of power: } \mathbb{I} \to \mathbb{C}(A,B) \cong \mathbb{C}(A, \textstyle\prod \mathbb{I}\,.\,B) \} \\ & \forall A{:}\mathbb{C}\,.\,\forall X{:}\mathbb{D}\,.\,\mathbb{E}(F\,X, \textstyle\prod \mathbb{D}(X, I\,A)\,.\,G\,A) \\ \cong\ & \{ \text{ interchange of quantifiers } \} \\ & \forall X{:}\mathbb{D}\,.\,\forall A{:}\mathbb{C}\,.\,\mathbb{E}(F\,X, \textstyle\prod \mathbb{D}(X, I\,A)\,.\,G\,A) \\ \cong\ & \{ \text{ the functor } \mathbb{E}(F\,X, -) \text{ preserves ends } \} \\ & \forall X{:}\mathbb{D}\,.\,\mathbb{E}(F\,X, \forall A{:}\mathbb{C}\,.\,\textstyle\prod \mathbb{D}(X, I\,A)\,.\,G\,A) \end{aligned}$$

$$\begin{aligned}
&\cong \quad \{ \text{ define } Ran_I\ G = \Lambda\, X{:}\,\mathbb{D}\ .\ \forall A{:}\,\mathbb{C}\ .\ \textstyle\prod \mathbb{D}(X, I\ A)\ .\ G\ A\ \} \\
&\quad \forall X{:}\,\mathbb{D}\ .\ \mathbb{E}(F\ X, Ran_I\ G\ X) \\
&\cong \quad \{ \text{ natural transformation as an end } \} \\
&\quad F \dot{\to} Ran_I\ G
\end{aligned}$$

The functor $Ran_I\ G$ is called the *right Kan extension* of G along I. (If we view $I : \mathbb{C} \to \mathbb{D}$ as an inclusion functor, then $Ran_I\ G : \mathbb{D} \to \mathbb{E}$ extends $G : \mathbb{C} \to \mathbb{E}$ to the whole of $\mathbb{D}$.) Dually, the left adjoint is called the *left Kan extension* and is defined $Lan_I\ F = \Lambda\, X{:}\,\mathbb{D}\ .\ \exists\, A{:}\,\mathbb{C}\ .\ \sum \mathbb{D}(I\ A, X)\ .\ F\ A$. The universally quantified object in the definition of Ran_I is a so-called *end*, which corresponds to a polymorphic type in Haskell. We refer the interested reader to Mac Lane's textbook [19] for further information. Dually, 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.



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

$$\begin{aligned}
&\mathbf{newtype}\ Ran_i\ g\ x = Ran\ \{ ran^{\circ} : \forall a\ .\ (x \to i\ a) \to g\ a \} \\
&\mathbf{instance}\ Functor\ (Ran_i\ g)\ \mathbf{where} \\
&\quad fmap\ f\ (Ran\ h) = Ran\ (\lambda\kappa \to h\ (\kappa \cdot f))
\end{aligned}$$

The type $Ran_i\ g$ can be seen as a *generalised continuation type* — often, but not necessarily the type constructors i and g are identical. Morally, i and g are functors. However, their mapping functions are not needed to define the $Ran_i\ g$ instance of *Functor*. Hence, we omit the $(Functor\ i, Functor\ g)$ context. The adjoint transpositions are defined

$$\begin{aligned}
&\phi_{Ran} : \forall i\ f\ g\ .\ (Functor\ f) \Rightarrow (\forall x\ .\ f\ (i\ x) \to g\ x) \to (\forall x\ .\ f\ x \to Ran_i\ g\ x) \\
&\phi_{Ran}\ \alpha = \lambda s \to Ran\ (\lambda\kappa \to \alpha\ (fmap\ \kappa\ s)) \\
&\phi^{\circ}_{Ran} : \forall i\ f\ g\ .\ (\forall x\ .\ f\ x \to Ran_i\ g\ x) \to (\forall x\ .\ f\ (i\ x) \to g\ x) \\
&\phi^{\circ}_{Ran}\ \beta = \lambda s \to ran^{\circ}\ (\beta\ s)\ id.
\end{aligned}$$

Again, we omit *Functor* contexts that are not needed.

Turning to the definition of the left Kan extension we require another extension of the Haskell 98 type system: existential types.

$$\begin{aligned}
&\mathbf{data}\ Lan_i\ f\ x = \forall a\ .\ Lan\ (i\ a \to x, f\ a) \\
&\mathbf{instance}\ Functor\ (Lan_i\ f)\ \mathbf{where} \\
&\quad fmap\ f\ (Lan\ (\kappa, s)) = Lan\ (f \cdot \kappa, s)
\end{aligned}$$

The existential quantifier is written as a universal quantifier *in front of* the data constructor *Lan*. Ideally, Lan_I should be given by a **newtype** declaration, but **newtype** constructors must not have an existential context. For similar reasons, we cannot use a deconstructor, that is, a selector function lan°. The type $Lan_i\, f$ can be seen as a *generalised abstract data type*: $f\ a$ is the internal state and $i\ a \to x$ the observer function — again, the type constructors i and g are likely to be identical. The adjoint transpositions are given by

$$\begin{aligned}
&\phi_{Lan} : \forall i\, f\ g\ .\ (Functor\ g) \Rightarrow (\forall x\ .\ f\ x \to g\,(i\ x)) \to (\forall x\ .\ Lan_i\, f\ x \to g\ x)\\
&\phi_{Lan}\ \alpha = \lambda(Lan\,(\kappa, s)) \to fmap\ \kappa\,(\alpha\ s)\\
&\phi^{\circ}_{Lan} : \forall i\, f\ g\ .\ (\forall x\ .\ Lan_i\, f\ x \to g\ x) \to (\forall x\ .\ f\ x \to g\,(i\ x))\\
&\phi^{\circ}_{Lan}\ \beta = \lambda s \to \beta\,(Lan\,(id, s))
\end{aligned}$$

□

Again, let us specialise the adjoint equations.

$$\begin{array}{ll}
& x \cdot Pre_I\ in = \Psi\ x\\
\iff & \{\ \text{definition of } Pre_I\ \}\\
& x \cdot (in \cdot I) = \Psi\ x\\
\iff & \{\ \text{pointwise}\ \}\\
& x\ A\,(in\,(I\ A)\ s) = \Psi\ x\ A\ s
\end{array}
\qquad
\begin{array}{ll}
& Pre_I\ out \cdot x = \Psi\ x\\
\iff & \{\ \text{definition of } Pre_I\ \}\\
& (out \cdot I) \cdot x = \Psi\ x\\
\iff & \{\ \text{pointwise}\ \}\\
& out\,(I\ A)\,(x\ A\ s) = \Psi\ x\ A\ s
\end{array}$$

Note that '$\cdot$' in the original equation 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 type arguments. The usual caveat applies when reading the equations as Haskell definitions: as type application is invisible, the derived equation is indistinguishable from the orginal one.

Haskell example 20. Continuing Haskell Example 19, the base function of *concat* is straightforward, except perhaps for the types.

$$\begin{aligned}
&\mathfrak{concat} : \forall x\ .\ (\forall a\ .\ x\,(List\ a) \to List\ a) \to\\
&\qquad\qquad (\forall a\ .\ \mathfrak{List}\ x\,(List\ a) \to List\ a)\\
&\mathfrak{concat}\quad concat\ \ (\mathfrak{Nil}) \qquad\quad = In\ \mathfrak{Nil}\\
&\mathfrak{concat}\quad concat\ \ (\mathfrak{Cons}\,(a, x)) \ \ = append\,(a, concat\ x)
\end{aligned}$$

The base function is a higher-order natural transformation. The transpose of *concat* is quite revealing. First of all, its type is

$$\phi\ concat : List \stackrel{.}{\to} Ran_{List}\ List \cong \forall a\ .\ List\ a \to \forall b\ .\ (a \to List\ b) \to List\ b.$$

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

$$\begin{aligned}
&concat' : \forall a\ b\ .\ \mu\mathfrak{List}\ a \to (a \to List\ b) \to List\ b\\
&concat' \qquad x \qquad = \lambda\kappa \qquad\quad \to concat\,(fmap\ \kappa\ x)
\end{aligned}$$

For clarity, we have inlined $Ran_{List}\ List$. □

adjunction	initial fixed-point equation	final fixed-point equation
$L \dashv R$	$x \cdot L\,in = \Psi\,x$ $\phi\,x \cdot in = (\phi \cdot \Psi \cdot \phi^{\circ})\,(\phi\,x)$	$R\,out \cdot x = \Psi\,x$ $out \cdot \phi^{\circ}\,x = (\phi^{\circ} \cdot \Psi \cdot \phi)\,(\phi^{\circ}\,x)$
$Id \dashv Id$	standard fold standard fold	standard unfold standard unfold
$(- \times X) \dashv (-^{X})$	parametrised fold fold returning an exponential	curried unfold unfold from a pair
$(+) \dashv \Delta$	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
$Lsh_X \dashv (-\,X)$	—	monomorphic unfold unfold from a left shift
$(-\,X) \dashv Rsh_X$	monomorphic fold fold to a right shift	—
$Lan_I \dashv (- \cdot I)$	—	polymorphic unfold unfold from a left Kan extension
$(- \cdot I) \dashv Ran_I$	polymorphic fold fold to a right Kan extension	—

Table 2. Adjunctions and types of recursion.

Kan extensions generalise the constructions of the previous section: we have $Lsh_A\,B \cong Lan_{(K\,A)}\,(K\,B)$ and $Rsh_A\,B \cong Ran_{(K\,A)}\,(K\,B)$, where K is the constant functor. The double adjunction $Lsh_X \dashv (-\,X) \dashv Rsh_X$ is implied by $Lan_I \dashv (- \cdot I) \dashv Ran_I$. Here is the proof for the right adjoint:

$$
\begin{aligned}
& F\,A \to B \\
\cong\;& \{ \text{ arrows as natural transformations } \} \\
& F \cdot K\,A \mathrel{\dot{\to}} K\,B \\
\cong\;& \{ (- \cdot I) \dashv Ran_I \} \\
& F \mathrel{\dot{\to}} Ran_{K\,A}\,(K\,B) \\
\cong\;& \{ Ran_{K\,A}\,(K\,B) \cong Rsh_A\,B \} \\
& F \mathrel{\dot{\to}} Rsh_A\,B .
\end{aligned}
$$

Table 2 summarises our findings.

5 Related work

Building on the work of Hagino [20], Malcolm [21] and many others, Bird and de Moor gave a comprehensive account of the “Algebra of Programming” in their seminal textbook [22]. While the work was well received and highly appraised in general, it also received some criticism. Poll and Thompson write in an otherwise positive review [23]:

> 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

$$x \cdot (in \times id) = h \cdot G\,x \cdot \phi,$$

where ϕ is some natural transformation and h an arbitrary arrow. 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 [1]. In order to show that generalised folds are uniquely defined, they discuss conditions to ensure that the more general equation $x \cdot L\,in = \Psi\,x$, 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. Somewhat ironically, the rest of the paper, which is concerned with folds for nested datatypes, doesn't build upon this elegant approach. Also, they don't consider (adjoint) unfolds. Nonetheless, Bird and Paterson deserve most of the credit for their fundamental insight, so three cheers to them!

An alternative, type-theoretic approach to (co-) inductive types was proposed by Mendler [8]. 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. Our fixed-point equations capture this approach. 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 have shown that Mendler-style folds can be based on difunctors [2]. It remains to be seen whether adjoint folds can also be generalised in this direction. Abel, Matthes and Uustalu extended Mendler-style folds to higher kinds [24]. Among other things, they demonstrate that suitable extensions of Girard's system F^{ω} retain the strong normalisation property.

There is a large body of work on 'morphisms'. Building on the notions of functors and natural transformations Malcolm generalised the Bird-Meertens formalism to arbitrary datatypes [21]. 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, Fokkinga and Paterson to the category **Cpo** [25]. The latter paper also introduced the now famous terms *catamorphism* and *anamorphism* (for folds and unfolds), along with the banana and lens brackets ($(\!|-|\!)$ and $[\!(-)\!]$). The notion of a *paramorphism* was introduced by Meertens [14]. Roughly speaking, paramorphisms generalise primitive recursion to arbitrary datatypes. Their duals, *apomorphisms*, were only studied later by Vene and Uustalu [26]. (While initial algebras have been the subject of intensive research, final coalgebras have received less attention — they are certainly under-appreciated [27].) An alternative solution to the '*append*-problem' was proposed by Pardo [28]: he introduces *folds with parameters* and uses them to implement *generic accumulations*. His accumulations subsume Gibbons' *downwards accumulations* [29].

The discovery of nested datatypes and their expressive power [10, 30, 31] 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 [1] 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 [18]. The approach being governed by pragmatic concerns was put on a firm theoretical footing by Martin, Gibbons and Baley [32] — 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 an adjoint fold — was later re-discovered by Johann and Ghani [33].

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.) It remains to be seen whether more exotic species such as *histomorphisms* or *futomorphisms* [15] are also subsumed by the framework. (It does work for the simple example of Fibonacci.)

6 Conclusion

I had the idea for this paper when I re-read "Generalised folds for nested datatypes" by Bird and Paterson [1]. 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. Termination and productivity are then 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 etc — and allows the scheme to be viewed as an instance of an adjoint (un-) fold.

Of course, the investigation of adjoint (un-) folds is not complete; it has barely begun. For one thing, it remains to develop the calculational properties of adjoint (un-) folds. Their definition

$$x = (\!|\Psi|\!)_L \iff x \cdot L\, in = \Psi\, x$$
$$x = [\!(\Psi)\!]_R \iff R\, out \cdot x = \Psi\, x$$

gives rise to the usual reflection, computation and fusion laws. In addition, one might hope for elegant laws manipulating the underlying adjoint functors. For another thing, it will be interesting to see whether other members of the morphism zoo can be fitted into the framework.

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 Nicolas Wu for carefully proof-reading (an even longer) draft of the manuscript and for suggesting numerous improvements regarding both contents and style (in fact, too numerous to be incorporated into the paper). Many thanks also go the members of the Algebra of Programming group at Oxford for several fruitful discussions on adjoint functors and for suggesting the subtitle. I owe a particular debt of gratitude to Thorsten Altenkirch for showing me how to derive the right adjoint of $- \dot{\times} H$ (at that time, I couldn't 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). Finally, I would like to thank Richard Bird for general advice and for encouraging (even pushing me) to write things up.

References

1. Bird, R., Paterson, R.: Generalised folds for nested datatypes. Formal Aspects of Computing **11**(2) (1999) 200–222
2. Uustalu, T., Vene, V.: Coding recursion a la Mendler (extended abstract). In Jeuring, J., ed.: Proceedings of the 2nd Workshop on Generic Programming, Ponte de Lima, Portugal. (July 2000) 69–85 The proceedings appeared as a technical report of Universiteit Utrecht, UU-CS-2000-19.
3. Peyton Jones, S.: Haskell 98 Language and Libraries. Cambridge University Press (2003)
4. Cockett, R., Fukushima, T.: About Charity. Yellow Series Report 92/480/18, Dept. of Computer Science, Univ. of Calgary (June 1992)
5. The Coq Development Team: The Coq proof assistant reference manual `http://coq.inria.fr`.

6. Fokkinga, M.M., Meijer, E.: Program calculation properties of continuous algebras. Technical Report CS-R9104, Centre of Mathematics and Computer Science, CWI, Amsterdam (January 1991)
7. Sheard, T., Pasalic, T.: Two-level types and parameterized modules. J. Functional Programming **14**(5) (September 2004) 547–587
8. Mendler, N.P.: Inductive types and type constraints in the second-order lambda calculus. Annals of Pure and Applied Logic **51**(1–2) (1991) 159–172
9. Giménez, E.: Codifying guarded definitions with recursive schemes. In Dybjer, P., Nordström, B., Smith, J.M., eds.: Types for Proofs and Programs, International Workshop TYPES'94, Båstad, Sweden, June 6-10, 1994, Selected Papers. Volume 996 of Lecture Notes in Computer Science., Springer-Verlag (1995) 39–59
10. Bird, R., Meertens, L.: Nested datatypes. In Jeuring, J., ed.: Fourth International Conference on Mathematics of Program Construction, MPC'98, Marstrand, Sweden. Volume 1422 of Lecture Notes in Computer Science., Springer-Verlag (June 1998) 52–67
11. Mycroft, A.: Polymorphic type schemes and recursive definitions. In Paul, M., Robinet, B., eds.: Proceedings of the International Symposium on Programming, 6th Colloquium, Toulouse, France. Volume 167 of Lecture Notes in Computer Science. (1984) 217–228
12. Hinze, R., Peyton Jones, S.: Derivable type classes. In Hutton, G., ed.: Proceedings of the 2000 ACM SIGPLAN Haskell Workshop. Volume 41(1) of Electronic Notes in Theoretical Computer Science., Elsevier Science (August 2001) 5–35 The preliminary proceedings appeared as a University of Nottingham technical report.
13. Trifonov, V.: Simulating quantified class constraints. In: Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell, New York, NY, USA, ACM (2003) 98–102
14. Meertens, L.: Paramorphisms. Formal Aspects of Computing **4** (1992) 413–424
15. Uustalu, T., Vene, V.: Primitive (co)recursion and course-of-value (co)iteration, categorically. Informatica, Lith. Acad. Sci. **10**(1) (1999) 5–26
16. Hinze, R.: Functional Pearl: Streams and unique fixed points. In Thiemann, P., ed.: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08), ACM Press (September 2008) 189–200
17. Bird, R.: Introduction to Functional Programming using Haskell. 2nd edn. Prentice Hall Europe, London (1998)
18. Hinze, R.: Efficient generalized folds. In Jeuring, J., ed.: Proceedings of the 2nd Workshop on Generic Programming, Ponte de Lima, Portugal. (July 2000) 1–16 The proceedings appeared as a technical report of Universiteit Utrecht, UU-CS-2000-19.
19. MacLane, S.: Categories for the Working Mathematician. 2nd edn. Graduate Texts in Mathematics. Springer-Verlag, Berlin (1998)
20. Hagino, T.: A typed lambda calculus with categorical type constructors. In Pitt, D., Poigne, A., Rydeheard, D., eds.: Category Theory and Computer Science. (1987) LNCS 283.
21. Malcolm, G.: Data structures and program transformation. Science of Computer Programming **14**(2–3) (1990) 255–280
22. Bird, R., de Moor, O.: Algebra of Programming. Prentice Hall Europe, London (1997)
23. Poll, E., Thompson, S.: Book review: "the algebra of programming". J. Functional Programming **9**(3) (May 1999) 347–354
24. Abel, A., Matthes, R., Uustalu, T.: Iteration and coiteration schemes for higher-order and nested datatypes. Theoretical Computer Science **333**(1-2) (2005) 3–66

25. Meijer, E., Fokkinga, M., Paterson, R.: Functional programming with bananas, lenses, envelopes and barbed wire. In: 5th ACM Conference on Functional Programming Languages and Computer Architecture, FPCA'91, Cambridge, MA, USA. Volume 523 of Lecture Notes in Computer Science., Springer-Verlag (1991) 124–144
26. Vene, V., Uustalu, T.: Functional programming with apomorphisms (corecursion). Proceedings of the Estonian Academy of Sciences: Physics, Mathematics **47**(3) (1998) 147–161
27. Gibbons, J., Jones, G.: The under-appreciated unfold. In Felleisen, M., Hudak, P., Queinnec, C., eds.: Proceedings of the third ACM SIGPLAN international conference on Functional programming, ACM Press (1998) 273–279
28. Pardo, A.: Generic accumulations. In Gibbons, J., Jeuring, J., eds.: Proceedings of the IFIP TC2 Working Conference on Generic Programming, Schloss Dagstuhl. Volume 243., Kluwer Academic Publishers (July 2002) 49–78
29. Gibbons, J.: Generic downwards accumulations. Sci. Comput. Program. **37**(1-3) (2000) 37–65
30. Connelly, R.H., Morris, F.L.: A generalization of the trie data structure. Mathematical Structures in Computer Science **5**(3) (September 1995) 381–418
31. Okasaki, C.: Catenable double-ended queues. In: Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming, Amsterdam, The Netherlands (June 1997) 66–74 *ACM SIGPLAN Notices,* 32(8), August 1997.
32. Martin, C., Gibbons, J., Bayley, I.: Disciplined, efficient, generalised folds for nested datatypes. Formal Aspects of Computing **16**(1) (April 2004) 19–35
33. Johann, P., Ghani, N.: Typed lambda calculi and applications, 8th international conference, TLCA 2007, Paris, France, June 26-28, 2007, proceedings. Volume 4583 of Lecture Notes in Computer Science., Springer-Verlag (2007) 207–222