

Type Fusion

Ralf Hinze

Computing Laboratory, University of Oxford
ralf.hinze@comlab.ox.ac.uk

Abstract

Fusion is an indispensable tool in the arsenal of techniques for program derivation. Less well-known, but equally valuable is type fusion, which states conditions for fusing an application of a functor with an initial algebra to form another initial algebra. We provide a novel proof of type fusion based on adjoint folds and discuss several applications: type firstification, type specialisation and tabulation.

1. Introduction

Fusion is an indispensable tool in the arsenal of techniques for program derivation and optimisation. The simplest instance of fusion states conditions for fusing an application of a function with a fixed point to form another fixed point:

$$\ell(\mu f) = \mu g \iff \ell \cdot f = g \cdot \ell, \quad (1)$$

where μ denotes the fixed point operator and f , g and ℓ are functions of suitable types. The usual mode of operation is from left to right: the two-stage process of forming the fixed point μf and then applying ℓ is optimised into the single operation of forming the fixed point μg . Applied from right to left, the law enables us to decompose a fixed point.

In this paper we discuss lifting fusion to the realm of types and type constructors. *Type fusion* takes the form

$$L(\mu F) \cong \mu G \iff L \cdot F \cong G \cdot L,$$

where F , G and L are functors between suitable categories and μF denotes the initial algebra of the endofunctor F . Similar to function fusion, type fusion allows us to fuse an application of a functor with an initial algebra to form another initial algebra. Type fusion has, however, one further prerequisite: L has to be a left adjoint. We show that this condition arises naturally as a consequence of defining the arrows witnessing the isomorphism.

Type fusion has been described before [3], but we believe that it deserves to be better known. We provide a novel

proof of type fusion based on adjoint folds [11], which gives a simple formula for the aforementioned isomorphisms. We illustrate the versatility of type fusion through a variety of applications relevant to programming:

- type firstification: a fixed point of a higher-order functor is transformed to a fixed-point of a first-order one;
- type specialisation: a nesting of types is fused into a single type that is more space-efficient;
- tabulation: functions from initial algebras can be memoised using final coalgebras. This example is intriguing as the left adjoint is contravariant and higher-order.

The rest of the paper is structured as follows. To keep the paper sufficiently self-contained, Section 2 reviews initial algebras and introduces adjoint folds. (The material is partly taken from [11].) The central theorem, type fusion, is given in Section 3. Sections 4, 5 and 6 discuss applications. Finally, Section 7 reviews related work.

2. Background

2.1. Initial algebras and final coalgebras

We assume cartesian closed categories \mathbb{C} , \mathbb{D} and \mathbb{E} that are ω -cocomplete and ω -complete. Furthermore, we confine ourselves to ω -cocontinuous and ω -continuous functors.

Let $F : \mathbb{C} \rightarrow \mathbb{C}$ be an endofunctor. An *F-algebra* is a pair $\langle A, f \rangle$ consisting of an object $A \in \mathbb{C}$ and an arrow $f \in \mathbb{C}(F A, A)$. An *F-homomorphism* between algebras $\langle A, f \rangle$ and $\langle B, g \rangle$ is an arrow $h \in \mathbb{C}(A, B)$ such that $h \cdot f = g \cdot F h$. Identity is an F-homomorphism and F-homomorphisms compose. Consequently, the data defines a category. If \mathbb{C} is ω -cocomplete and F is ω -cocontinuous, this category possesses an initial object, the so-called *initial F-algebra* $\langle \mu F, in \rangle$. The import of initiality is that there is a unique arrow from $\langle \mu F, in \rangle$ to any other F-algebra $\langle A, f \rangle$. This unique arrow is written $\langle f \rangle$ and is called a *fold*. Expressed in terms of the base category, it satisfies the following *universal property*.

$$x = \langle f \rangle \iff x \cdot in = f \cdot F x \quad (2)$$

The universal property has several important consequences. Setting $x := id$ and $f := in$, we obtain the *reflection law*: $\llbracket in \rrbracket = id$. Substituting the left-hand side into the right-hand side gives the *computation law*: $\llbracket f \rrbracket \cdot in = f \cdot F \llbracket f \rrbracket$. Finally and most importantly, it implies the *fusion law* for fusing an arrow with a fold to form another fold.

$$h \cdot \llbracket f \rrbracket = \llbracket g \rrbracket \iff h \cdot f = g \cdot F h \quad (3)$$

Initial algebras provide semantics for recursive datatypes as found, for instance, in Haskell [16]. The following example illustrates the approach. In fact, Haskell is expressive enough to replay the development within the language.

Example 2.1 Consider the datatype *Stack*, which models stacks of natural numbers.

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

The function *total*, which computes the sum of a stack of natural numbers, is a typical example of a *fold*.

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

Since Haskell features higher-kinded type constructors, initial algebras can be captured by a datatype declaration.

```
newtype μf = In { in° : f (μf) }
```

The definition uses Haskell's record syntax to introduce the destructor in° in addition to the constructor *In*. Using this definition, the type of stacks can be factored into a non-recursive *base functor* that describes the structure of the data and an application of μ :

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

Folds or catamorphisms can be defined generically, that is, for arbitrary base functors by taking the computation law as the defining equation.

$$\begin{aligned} \llbracket - \rrbracket &: (Functor f) \Rightarrow (f b \rightarrow b) \rightarrow (\mu f \rightarrow b) \\ \llbracket f \rrbracket &= f \cdot fmap \llbracket f \rrbracket \cdot in^\circ \end{aligned}$$

Similar to the development on the type level, the function *total* can now be factored into a non-recursive algebra and an application of fold:

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

For emphasis, base functors, algebras and, later, base functions are typeset in this font. \square

To understand concepts in category theory it is helpful to look at a particularly simple class of categories: *preorders*. Every preorder gives rise to a category whose objects are the elements of the preorder and whose arrows are given by the ordering relation. These categories are quite special as there is at most one arrow between two objects: $a \rightarrow b$ is inhabited if and only if $a \leq b$. A functor between two preorders is a *monotone function*, which is a mapping on objects that respects the underlying ordering: $a \leq b \implies f a \leq f b$. A natural transformation between two monotone functions corresponds to a point-wise ordering: $f \leq g \iff \forall x . f x \leq g x$. Throughout the paper we shall specialise the development to preorders. For type fusion, Section 3, we turn things upside down: we first develop the theory in the simple setting and then generalise to arbitrary categories.

Let P be a preorder and let $f : P \rightarrow P$ be a monotone function. An f -algebra is an element a with $f a \leq a$, a so-called *prefix point*. An initial f -algebra is the least prefix point of f . Since there is at most one arrow between two objects in a preorder, the theory of initial algebras simplifies considerably: all that matters is the type of an arrow. The type of *in* corresponds to the *fixed-point inclusion law*:

$$f (\mu f) \leq \mu f, \quad (4)$$

which expresses that μf is indeed a prefix point. The type of fold, $\llbracket f \rrbracket \in \mathbb{C}(\mu F, B) \iff f \in \mathbb{C}(F B, B)$, translates to the *fixed-point induction law*:

$$\mu f \leq b \iff f b \leq b. \quad (5)$$

It captures the property that μf is smaller than or equal to every other prefix point. To illustrate the laws, let us prove that μf is a fixed point of f . The inclusion law states that $f (\mu f) \leq \mu f$, for the other direction we reason

$$\begin{aligned} &\mu f \leq f (\mu f) \\ \iff &\{ \text{induction (5)} \} \\ &f (f (\mu f)) \leq f (\mu f) \\ \iff &\{ f \text{ monotone} \} \\ &f (\mu f) \leq \mu f. \end{aligned}$$

The proof involves the type of fold, the fact that f is a functor and the type of *in*. In other words, it can be seen as a typing derivation of the inverse of *in*:

$$\begin{aligned} &\llbracket F in \rrbracket \in \mathbb{C}(\mu F, F (\mu F)) \\ \iff &\{ \text{type of fold} \} \\ &F in \in \mathbb{C}(F (F (\mu F)), F (\mu F)) \\ \iff &\{ F \text{ functor} \} \\ &in \in \mathbb{C}(F (\mu F), \mu F). \end{aligned}$$

Summing up, to interpret a category-theoretic result in the setting of preorders, we only consider the types of the

arrows. Conversely, an order-theoretic proof can be seen as a typing derivation — we only have to provide witnesses for the types. Category theory has been characterised as *coherently constructive lattice theory* [2], and to generalise an order-theoretic result we additionally have to establish the required coherence conditions. Continuing the example, to prove $F(\mu F) \cong \mu F$ we must show that $in \cdot (F in) = id$,

$$\begin{aligned} & in \cdot (F in) = id \\ \iff & \{ \text{reflection} \} \\ & in \cdot (F in) = (in) \\ \iff & \{ \text{fusion (3)} \} \\ & in \cdot F in = in \cdot F in, \end{aligned}$$

and $(F in) \cdot in = id$, left as an exercise to the reader.

Finally, let us remark that the development nicely dualises to *F-coalgebras* and *unfolds*. The final *F-coalgebra* is denoted $\langle \nu F, out \rangle$.

2.2. Adjoint folds and unfolds

Folds and unfolds are at the heart of the algebra of programming. However, most programs require some tweaking to be given the form of a fold or an unfold, and thus make them amenable to formal manipulation.

Example 2.2 Consider the function *shunt*, which pushes the elements of the first onto the second stack.

$$\begin{aligned} shunt & : \mu \mathit{Stack} \times \mathit{Stack} \rightarrow \mathit{Stack} \\ shunt (In \ \mathit{Empty}, y) & = y \\ shunt (In (\mathit{Push}(a, x)), y) & = shunt(x, In(\mathit{Push}(a, y))) \end{aligned}$$

The function is not a fold, simply because it does not have the right type. \square

Practical considerations dictate the introduction of a more general (co-) recursion scheme, christened *adjoint folds and unfolds* [11] for reasons to become clear in a moment. The central idea is to allow the initial algebra or the final coalgebra to be embedded in a context, modelled by a functor (L and R). The *adjoint fold* $(\Psi)_L \in \mathbb{C}(L(\mu F), B)$ is the unique solution of the adjoint initial fixed-point equation

$$x \cdot L in = \Psi x, \quad (6)$$

where the so-called *base function* Ψ has type

$$\Psi : \forall X \in \mathbb{D} . \mathbb{C}(L X, B) \rightarrow \mathbb{C}(L(F X), B).$$

It is important that Ψ is natural in X . Loosely speaking, the polymorphic type ensures *termination* of $(\Psi)_L$. Dually, the *adjoint unfold* $(\Psi)_R \in \mathbb{C}(A, R(\nu F))$ is the unique solution of the adjoint final fixed-point equation

$$R out \cdot x = \Psi x, \quad (7)$$

where the base function Ψ has type

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

Again, the base function has to be natural in X . The type now ensures *productivity*.

Of course, the functors L and R can't be arbitrary. For instance, for $L := K A$ where $K : \mathbb{C} \rightarrow \mathbb{C}^{\mathbb{D}}$ is the constant functor and $\Psi := id : \mathbb{C}(A, B) \rightarrow \mathbb{C}(A, B)$, Equation (6) simplifies to the trivial $x = x$. One approach for ensuring uniqueness is to try to express x in terms of a standard fold. This is where adjunctions enter the scene. Briefly, let \mathbb{C} and \mathbb{D} be categories. The functors L and R are *adjoint*, $L \dashv R$,

$$\begin{array}{ccc} & L & \\ \mathbb{C} & \xleftarrow{\quad} & \mathbb{D} \\ & \perp & \\ & R & \xrightarrow{\quad} \end{array}$$

if and only if there is a bijection

$$\phi : \forall A B . \mathbb{C}(L A, B) \cong \mathbb{D}(A, R B) \quad (8)$$

that is natural both in A and B . The isomorphism ϕ is called the *adjoint transposition*. It allows us to trade L in the source for R in the target, which is the key for showing that adjoint fixed-point equations have unique solutions. The proof proceeds in two steps. We first establish the result for the adjunction $Id \dashv Id$ (Section 2.2.1) and then generalise to arbitrary adjunctions (Section 2.2.2).

2.2.1 Mendler-style fixed-point equations

For the identity adjunction, adjoint folds specialise to so-called *Mendler-style folds* [18]. The proof of uniqueness makes essential use of the fact that the type of the base function $\forall X \in \mathbb{D} . \mathbb{C}(X, B) \rightarrow \mathbb{C}(F X, B)$ is isomorphic to the type $\mathbb{C}(F B, B)$, the arrow part of an *F-algebra*.

$$\mathbb{C}(F B, B) \cong (\forall X \in \mathbb{C} . \mathbb{C}(X, B) \rightarrow \mathbb{C}(F X, B)) \quad (9)$$

Readers versed in category theory will notice that this bijection is an instance of the *Yoneda lemma*. The functions witnessing the isomorphism are

$$\psi f = \lambda \kappa . f \cdot F \kappa \quad \text{and} \quad \psi^\circ \Psi = \Psi id. \quad (10)$$

Using this prerequisite we can establish uniqueness by showing that x is a solution of $x \cdot in = \Psi x$ if and only if x is a certain standard fold:

$$\begin{aligned} & x \cdot in = \Psi x \\ \iff & \{ \text{isomorphism (9)} \} \\ & x \cdot in = \psi(\psi^\circ \Psi) x \\ \iff & \{ \text{definition of } \psi \text{ and definition of } \psi^\circ \text{ (10)} \} \\ & x \cdot in = \Psi id \cdot F x \\ \iff & \{ \text{universal property (2)} \} \\ & x = (\Psi id). \end{aligned}$$

Example 2.3 It is an easy exercise to rephrase *total*, see Example 2.1, as a Mandler-style fold. We obtain its base function simply by abstracting away from the recursive call.

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

In contrast to the previous definition of *total*, the right-hand sides are identical to those of *total*. The latter function is then given as the unique solution of the adjoint equation

$$\begin{aligned} \text{total} &: \mu \text{Stack} \rightarrow \text{Nat} \\ \text{total} \quad (\text{In } s) &= \text{total } \text{total } s, \end{aligned}$$

which is the point-wise variant of $\text{total} \cdot \text{in} = \text{total } \text{total}$. \square

2.2.2 Adjoint fixed-point equations

Let us generalise the proof to an arbitrary adjunction. Again, naturality plays a key rôle: the calculation that determines the unique solution of $x \cdot \text{L } \text{in} = \Psi x$ makes essential use of the fact that the transposition ϕ is natural in A .

$$\begin{aligned} x \cdot \text{L } \text{in} &= \Psi x \\ \iff & \quad \{ \text{adjunction (8)} \} \\ \phi(x \cdot \text{L } \text{in}) &= \phi(\Psi x) \\ \iff & \quad \{ \text{naturality of } \phi: \phi(f \cdot \text{L } h) = \phi f \cdot h \} \\ \phi x \cdot \text{in} &= \phi(\Psi x) \\ \iff & \quad \{ \text{adjunction (8)} \} \\ \phi x \cdot \text{in} &= (\phi \cdot \Psi \cdot \phi^\circ)(\phi x) \\ \iff & \quad \{ \text{Section 2.2.1} \} \\ \phi x &= ((\phi \cdot \Psi \cdot \phi^\circ) \text{ id}) \\ \iff & \quad \{ \text{adjunction (8)} \} \\ x &= \phi^\circ((\phi \cdot \Psi \cdot \phi^\circ) \text{ id}). \end{aligned}$$

Note that $\phi \cdot \Psi \cdot \phi^\circ : \forall X \in \mathbb{D} . \mathbb{D}(X, \text{R } B) \rightarrow \mathbb{D}(\text{F } X, \text{R } B)$ is a composition of natural transformations. The arrow $x = \phi^\circ((\phi \cdot \Psi \cdot \phi^\circ) \text{ id}) \in \mathbb{C}(\text{L }(\mu\text{F}), B)$ is the desired solution of the adjoint equation; the underlying fold $\phi x \in \mathbb{C}(\mu\text{F}, \text{R } B)$ is called the *transpose* of x .

Example 2.4 In the case of *shunt*, the adjoint functor is pairing: $\text{L } X = X \times \text{Stack}$ and $\text{L } f = f \times \text{id}_{\text{Stack}}$. Its right adjoint is exponentiation: $\text{R } Y = Y^{\text{Stack}}$ and $\text{R } f = f^{\text{id}_{\text{Stack}}}$. This adjunction captures *currying*: a function of two arguments can be treated as a function of the first argument whose values are functions of the second argument. To see that *shunt* is an adjoint fold we factor the definition into a non-recursive base function *shunt* that abstracts away from the recursive call and an adjoint equation

that ties the recursive knot.

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

The last equation is the point-wise variant of $\text{shunt} \cdot \text{L } \text{in} = \text{shunt } \text{shunt}$. The transposed fold is simply the curried variant of *shunt*. \square

Let us specialise the result to preorders. An adjunction is a pair of monotone functions $\ell : Q \rightarrow P$ and $r : P \rightarrow Q$ such that

$$\ell a \leq b \iff a \leq r b. \quad (11)$$

The type of the adjoint fold $(\Psi)_{\text{L}} \in \mathbb{C}(\text{L }(\mu\text{F}), B) \leftarrow \Psi \in \forall X \in \mathbb{D} . \mathbb{C}(\text{L } X, B) \rightarrow \mathbb{C}(\text{L }(\text{F } X), B)$ translates to the *adjoint induction law*.

$$\begin{aligned} \ell(\mu f) \leq b & \iff & (12) \\ \forall x \in Q . \ell x \leq b & \implies \ell(f x) \leq b \end{aligned}$$

As usual, the development dualises to final coalgebras. We leave the details to the reader.

3. Type fusion

Turning to the heart of the matter, the aim of this section is to lift the fusion law (1) to the realm of objects and functors.

$$\text{L }(\mu\text{F}) \cong \mu\text{G} \iff \text{L} \cdot \text{F} \cong \text{G} \cdot \text{L}$$

To this end we have to construct two arrows $\tau : \text{L }(\mu\text{F}) \rightarrow \mu\text{G}$ and $\tau^\circ : \mu\text{G} \rightarrow \text{L }(\mu\text{F})$ that are inverses. The type of τ° suggests that the arrow is an ordinary fold. In contrast, τ looks suspiciously like an adjoint fold. Consequently, we shall require that L has a right adjoint. The diagram below summarises the type information.

$$\mathbb{C} \begin{array}{c} \xleftarrow{\text{G}} \\ \xrightarrow{\text{G}} \end{array} \mathbb{C} \begin{array}{c} \xleftarrow{\text{L}} \\ \xrightarrow{\text{R}} \end{array} \mathbb{D} \begin{array}{c} \xleftarrow{\text{F}} \\ \xrightarrow{\text{F}} \end{array} \mathbb{D}$$

As a preparatory step, we establish type fusion in the setting of preorders. The proof of the equivalence $\ell(\mu f) \cong \mu g$ consists of two parts. We show first that $\mu g \leq \ell(\mu f) \iff g \cdot \ell \leq \ell \cdot f$ and second that $\ell(\mu f) \leq \mu g \iff \ell \cdot f \leq g \cdot \ell$.

$$\begin{aligned} \mu g \leq \ell(\mu f) \\ \iff & \quad \{ \text{induction (5)} \} \\ g(\ell(\mu f)) & \leq \ell(\mu f) \end{aligned}$$

$$\begin{aligned}
&\Leftarrow \{ \text{transitivity} \} \\
&g(\ell(\mu f)) \leq \ell(f(\mu f)) \text{ and } \ell(f(\mu f)) \leq \ell(\mu f) \\
&\Leftarrow \{ \text{assumption } g \cdot \ell \leq \ell \cdot f \} \\
&\ell(f(\mu f)) \leq \ell(\mu f) \\
&\Leftarrow \{ \ell \text{ monotone} \} \\
&f(\mu f) \leq \mu f
\end{aligned}$$

The last inequality is just inclusion (4). For the second part, we apply adjoint induction (12), which leaves us with the proof obligation $\forall x \in Q. \ell x \leq \mu g \implies \ell(f x) \leq \mu g$.

$$\begin{aligned}
&\ell(f x) \leq \mu g \\
&\Leftarrow \{ \text{transitivity} \} \\
&\ell(f x) \leq g(\ell x) \text{ and } g(\ell x) \leq \mu g \\
&\Leftarrow \{ \text{assumption } \ell \cdot f \leq g \cdot \ell \} \\
&g(\ell x) \leq \mu g \\
&\Leftarrow \{ \text{transitivity} \} \\
&g(\ell x) \leq g(\mu g) \text{ and } g(\mu g) \leq \mu g \\
&\Leftarrow \{ g \text{ monotone and inclusion (4)} \} \\
&\ell x \leq \mu g
\end{aligned}$$

In the previous section we have seen that an order-theoretic proof can be interpreted constructively as a typing derivation. The first proof above defines the arrow τ° . (The natural isomorphism witnessing $L \cdot F \cong G \cdot L$ is called *swap*.)

$$\begin{aligned}
&(\llbracket L \text{ in} \cdot \text{swap}^\circ \rrbracket) \in \mathbb{C}(\mu G, L(\mu F)) \\
&\Leftarrow \{ \text{type of fold} \} \\
&L \text{ in} \cdot \text{swap}^\circ \in \mathbb{C}(G(L(\mu F)), L(\mu F)) \\
&\Leftarrow \{ \text{composition} \} \\
&\text{swap}^\circ \in \mathbb{C}(G(L(\mu F)), L(F(\mu F))) \\
&\quad \text{and } L \text{ in} \in \mathbb{C}(L(F(\mu F)), L(\mu F)) \\
&\Leftarrow \{ \text{assumption } \text{swap}^\circ : G \cdot L \dot{\rightarrow} L \cdot F \} \\
&L \text{ in} \in \mathbb{C}(L(F(\mu F)), L(\mu F)) \\
&\Leftarrow \{ L \text{ functor} \} \\
&\text{in} \in \mathbb{D}(F(\mu F), \mu F)
\end{aligned}$$

Conversely, the arrow τ is the adjoint fold $(\llbracket \Psi \rrbracket)_L$ whose base function Ψ is given by the second proof.

$$\begin{aligned}
&\text{in} \cdot G x \cdot \text{swap} \in \mathbb{C}(L(F X), \mu G) \\
&\Leftarrow \{ \text{composition} \} \\
&\text{swap} \in \mathbb{C}(L(F X), G(L X)) \\
&\quad \text{and } \text{in} \cdot G x \in \mathbb{C}(G(L X), \mu G) \\
&\Leftarrow \{ \text{assumption } \text{swap} : L \cdot F \dot{\rightarrow} G \cdot L \} \\
&\text{in} \cdot G x \in \mathbb{C}(G(L X), \mu G) \\
&\Leftarrow \{ \text{composition} \} \\
&G x \in \mathbb{C}(G(L X), G(\mu G)) \text{ and } \text{in} \in \mathbb{C}(G(\mu G), \mu G)
\end{aligned}$$

$$\begin{aligned}
&\Leftarrow \{ G \text{ functor and type of } \text{in} \} \\
&x \in \mathbb{C}(L X, \mu G)
\end{aligned}$$

We may conclude that $\tau = (\llbracket \lambda x. \text{in} \cdot G x \cdot \text{swap} \rrbracket)_L$ and $\tau^\circ = (\llbracket L \text{ in} \cdot \text{swap}^\circ \rrbracket)$ are the desired arrows. All that remains is to establish that they are inverses.

Theorem 3.1 (Type fusion) *Let \mathbb{C} and \mathbb{D} be categories, let $L \dashv R$ be an adjoint pair of functors $L : \mathbb{D} \rightarrow \mathbb{C}$ and $R : \mathbb{C} \rightarrow \mathbb{D}$, and let $F : \mathbb{D} \rightarrow \mathbb{D}$ and $G : \mathbb{C} \rightarrow \mathbb{C}$ be two endofunctors. Then*

$$L(\mu F) \cong \mu G \quad \Leftarrow \quad L \cdot F \cong G \cdot L \quad (13)$$

$$\nu F \cong R(\nu G) \quad \Leftarrow \quad F \cdot R \cong R \cdot G. \quad (14)$$

Proof We show type fusion for initial algebras (13), the corresponding statement for final coalgebras (14) follows by duality. The isomorphisms τ and τ° are given as solutions of adjoint fixed point equations:

$$\tau \cdot L \text{ in} = \text{in} \cdot G \tau \cdot \text{swap} \quad \text{and} \quad \tau^\circ \cdot \text{in} = L \text{ in} \cdot \text{swap}^\circ \cdot G \tau^\circ.$$

Proof of $\tau \cdot \tau^\circ = id_{\mu G}$:

$$\begin{aligned}
&(\tau \cdot \tau^\circ) \cdot \text{in} \\
&= \{ \text{definition of } \tau^\circ \} \\
&\tau \cdot L \text{ in} \cdot \text{swap}^\circ \cdot G \tau^\circ \\
&= \{ \text{definition of } \tau \} \\
&\text{in} \cdot G \tau \cdot \text{swap} \cdot \text{swap}^\circ \cdot G \tau^\circ \\
&= \{ \text{inverses} \} \\
&\text{in} \cdot G \tau \cdot G \tau^\circ \\
&= \{ G \text{ functor} \} \\
&\text{in} \cdot G(\tau \cdot \tau^\circ)
\end{aligned}$$

The equation $x \cdot \text{in} = \text{in} \cdot G x$ has a unique solution — the base function $\Psi x = \text{in} \cdot G x$ possesses the polymorphic type $\forall X \in \mathbb{C}. \mathbb{C}(X, \mu G) \rightarrow \mathbb{C}(G X, \mu G)$. Since *id* is also a solution, the result follows. **Proof of $\tau^\circ \cdot \tau = id_{L(\mu F)}$:**

$$\begin{aligned}
&(\tau^\circ \cdot \tau) \cdot L \text{ in} \\
&= \{ \text{definition of } \tau \} \\
&\tau^\circ \cdot \text{in} \cdot G \tau \cdot \text{swap} \\
&= \{ \text{definition of } \tau^\circ \} \\
&L \text{ in} \cdot \text{swap}^\circ \cdot G \tau^\circ \cdot G \tau \cdot \text{swap} \\
&= \{ G \text{ functor} \} \\
&L \text{ in} \cdot \text{swap}^\circ \cdot G(\tau^\circ \cdot \tau) \cdot \text{swap}
\end{aligned}$$

Again, $x \cdot L \text{ in} = L \text{ in} \cdot \text{swap}^\circ \cdot G x \cdot \text{swap}$ enjoys a unique solution — the base function $\Psi x = L \text{ in} \cdot \text{swap}^\circ \cdot G x \cdot \text{swap}$ has type $\forall X \in \mathbb{D}. \mathbb{C}(L X, L(\mu F)) \rightarrow \mathbb{C}(L(F X), L(\mu F))$. And again, *id* is also solution, which implies the result. \square

4. Application: firstification

Abstraction by parametrisation is a central concept in programming. A program can be made more general by abstracting away from a constant. Likewise, a type can be generalised by abstracting away from a type constant.

Example 4.1 Recall the type of stacks of natural numbers.

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

The type of stack elements, *Nat*, is somewhat arbitrary. Abstracting away from it yields the type of parametric lists

```
data List a = Nil | Cons (a × List a).
```

To avoid name clashes, we have renamed data and type constructors. \square

The inverse of abstraction is application or instantiation. We regain the original concept by instantiating the parameter to the constant. Continuing Example 4.1, we expect that

$$\text{List Nat} \cong \text{Stack}. \quad (15)$$

The isomorphism is non-trivial, as both types are recursively defined. The transformation of *List Nat* into *Stack* can be seen as an instance of *firstification* [12] or *λ -dropping* [7] on the type level: a fixed point of a higher-order functor is reduced to a fixed-point of a first-order functor.

Perhaps surprisingly, we can fit the isomorphism into the framework of type fusion. To this end, we have to view type application as a functor: given $T \in \mathbb{D}$ define $App_T : \mathbb{C}^{\mathbb{D}} \rightarrow \mathbb{C}$ by $App_T F = F T$ and $App_T \alpha = \alpha T$. Using type application we can rephrase Equation (15) as $App_{Nat}(\mu\mathfrak{List}) \cong \mu\mathfrak{Stack}$, where \mathfrak{List} is the higher-order base functor of *List* defined

```
data List list a = Nil | Cons (a, list a).
```

In order to apply type fusion, we have to check that App_T is part of an adjunction. It turns out that it has both a left and a right adjoint. Thus, we can firstify both inductive and coinductive parametric types. We derive the right adjoint and leave the derivation of the left one to the reader.

$$\begin{aligned} & \mathbb{C}(App_T A, B) \\ \cong & \quad \{ \text{definition of } App_T \} \\ & \mathbb{C}(A T, B) \\ \cong & \quad \{ \text{Yoneda lemma} \} \\ & \forall X \in \mathbb{D} . \mathbb{D}(X, T) \rightarrow \mathbb{C}(A X, B) \\ \cong & \quad \{ \text{def. of a power: } \mathbb{I} \rightarrow \mathbb{C}(Y, Z) \cong \mathbb{C}(Y, \prod \mathbb{I} . Z) \} \\ & \forall X \in \mathbb{D} . \mathbb{C}(A X, \prod \mathbb{D}(X, T) . B) \end{aligned}$$

$$\begin{aligned} \cong & \quad \{ \text{define } Rsh_T B := \Lambda X : \mathbb{D} . \prod \mathbb{D}(X, T) . B \} \\ & \forall X \in \mathbb{D} . \mathbb{C}(A X, Rsh_T B X) \\ \cong & \quad \{ \text{natural transformation} \} \\ & \mathbb{C}^{\mathbb{D}}(A, Rsh_T B) \end{aligned}$$

Hence, the functor App_T has a right adjoint if the underlying category has powers. We call Rsh_T the *right shift* of T , for want of a better name. Dually, the left adjoint is $Lsh_T A = \Lambda X : \mathbb{D} . \sum \mathbb{D}(T, X) . A$, the *left shift* of T .

Consequently, Theorem 3.1 is applicable. Generalising the original problem, Equation (15), the second-order type μF and the first-order type μG are related by $App_T(\mu F) \cong \mu G$ if $App_T \cdot F \cong G \cdot App_T$. Despite the somewhat complicated type, the natural isomorphism *swap* is usually straightforward to define: it simply renames the constructors, as illustrated below.

Example 4.2 Let us show that $\text{List Nat} \cong \text{Stack}$. We have to discharge the obligation $App_{Nat} \cdot \mathfrak{List} \cong \mathfrak{Stack} \cdot App_{Nat}$.

$$\begin{aligned} & App_{Nat} \cdot \mathfrak{List} \\ \cong & \quad \{ \text{composition of functors and definition of } App \} \\ & \Lambda X . \mathfrak{List} X Nat \\ \cong & \quad \{ \text{definition of } \mathfrak{List} \} \\ & \Lambda X . 1 + Nat \times X Nat \\ \cong & \quad \{ \text{definition of } \mathfrak{Stack} \} \\ & \Lambda X . \mathfrak{Stack} (X Nat) \\ \cong & \quad \{ \text{composition of functors and definition of } App \} \\ & \mathfrak{Stack} \cdot App_{Nat} \end{aligned}$$

The proof above is entirely straightforward. For reference, let us spell out the isomorphisms:

$$\begin{aligned} swap : \forall x . \mathfrak{List} x Nat & \rightarrow \mathfrak{Stack} (x Nat) \\ swap \quad Nil & = \mathfrak{Empty} \\ swap \quad (\mathfrak{Cons} (n, x)) & = \mathfrak{Push} (n, x) \\ swap^\circ : \forall x . \mathfrak{Stack} (x Nat) & \rightarrow \mathfrak{List} x Nat \\ swap^\circ \quad \mathfrak{Empty} & = Nil \\ swap^\circ \quad (\mathfrak{Push} (n, x)) & = \mathfrak{Cons} (n, x). \end{aligned}$$

The transformations rename Nil to \mathfrak{Empty} and \mathfrak{Cons} to \mathfrak{Push} , and vice versa. Finally, τ and τ° implement Λ -lifting and Λ -dropping.

$$\begin{aligned} \Lambda\text{-lift} : \mu\mathfrak{Stack} & \rightarrow \mu\mathfrak{List} Nat \\ \Lambda\text{-lift} (In x) & = In (swap^\circ (fmap \Lambda\text{-lift} x)) \\ \Lambda\text{-drop} : \mu\mathfrak{List} Nat & \rightarrow \mu\mathfrak{Stack} \\ \Lambda\text{-drop} (In x) & = In (fmap \Lambda\text{-drop} (swap x)) \end{aligned}$$

Since type application is invisible in Haskell, the adjoint fold $\Lambda\text{-drop}$ deceptively resembles a standard fold. \square

Transforming a higher-order fixed point into a first-order fixed point works for so-called *regular datatypes*. The type of lists is regular; the type of perfect trees defined

data *Perfect* $a = \text{Zero } a \mid \text{Succ } (\text{Perfect } (a \times a))$

is not because the recursive call of *Perfect* involves a change of argument. Firstification is not applicable, as there is no first-order base functor \mathfrak{Base} such that $\text{App}_T \cdot \mathfrak{Perfect} = \mathfrak{Base} \cdot \text{App}_T$. The class of regular datatypes is usually defined syntactically. Drawing from the development above, we can provide an alternative semantic characterisation.

Definition 4.1 *Let $F : \mathbb{C}^{\mathbb{D}} \rightarrow \mathbb{C}^{\mathbb{D}}$ be a higher-order functor. The parametric datatype $\mu F : \mathbb{D} \rightarrow \mathbb{C}$ is regular if and only if there exists a functor $G : \mathbb{C} \rightarrow \mathbb{C}$ such that $\text{App}_T \cdot F \cong G \cdot \text{App}_T$ for all objects $T : \mathbb{D}$. \square*

The regularity condition can be simplified to $FHT = G(HT)$, which makes explicit that all occurrences of ‘the recursive call’ H are applied to the same argument.

5. Application: type specialisation

Firstification can be seen as an instance of type specialisation: a nesting of types is fused to a single type that allows for a more compact and space-efficient representation. Let us illustrate the idea by means of an example.

Example 5.1 Lists of optional values, $\text{List} \cdot \text{Maybe}$, where Maybe is given by

data *Maybe* $a = \text{Nothing} \mid \text{Just } a,$

can be represented more compactly using the tailor-made

data *Seq* $a = \text{Done} \mid \text{Skip } (\text{Seq } a) \mid \text{Yield } (a \times \text{Seq } a).$

Assuming that the constructor $C(v_1, \dots, v_n)$ requires $n+1$ cells of storage, the compact representation saves $2n$ cells for a list of length n . \square

The goal of this section is to prove that

$$\text{List} \cdot \text{Maybe} \cong \text{Seq}, \quad (16)$$

or, more generally, $\mu F \cdot I \cong \mu G$ for suitably related base functors F and G . The application of Section 4 is an instance of this problem as the relation $HA \cong B$ between objects can be lifted to a relation $H \cdot KA \cong KB$ between functors.

To fit Equation (16) under the umbrella of type fusion, we have to view pre-composition as a functor. Given a functor $I : \mathbb{C} \rightarrow \mathbb{D}$, define the higher-order functor $\text{Pre}_I : \mathbb{E}^{\mathbb{D}} \rightarrow \mathbb{E}^{\mathbb{C}}$ by $\text{Pre}_I F = F \cdot I$ and $\text{Pre}_I \omega = \omega \cdot I$. Using the functor we can rephrase Equation (16) as $\text{Pre}_{\text{Maybe}} (\mu \text{List}) \cong \mu \text{Seq}$.

Of course, we first have to construct the right adjoint of Pre_I . It turns out that this is a well-studied problem in category theory [15, X.3]. Similar to the situation of the previous section, Pre_I has both a left and a right adjoint. The derivation of the right adjoint generalises the one of Section 4.

$$\begin{aligned} & \mathbb{E}^{\mathbb{C}}(F \cdot I, G) \\ \cong & \quad \{ \text{natural transformation as an end} \} \\ & \forall A \in \mathbb{C} . \mathbb{E}(F(I A), G A) \\ \cong & \quad \{ \text{Yoneda (9)} \} \\ & \forall A \in \mathbb{C} . \forall X \in \mathbb{D} . \mathbb{D}(X, I A) \rightarrow \mathbb{E}(F X, G A) \\ \cong & \quad \{ \text{def. of power: } \mathbb{I} \rightarrow \mathbb{C}(A, B) \cong \mathbb{C}(A, \prod \mathbb{I} . B) \} \\ & \forall A \in \mathbb{C} . \forall X \in \mathbb{D} . \mathbb{E}(F X, \prod \mathbb{D}(X, I A) . G A) \\ \cong & \quad \{ \text{interchange of quantifiers} \} \\ & \forall X \in \mathbb{D} . \forall A \in \mathbb{C} . \mathbb{E}(F X, \prod \mathbb{D}(X, I A) . G A) \\ \cong & \quad \{ \text{the functor } \mathbb{E}(F X, -) \text{ preserves ends} \} \\ & \forall X \in \mathbb{D} . \mathbb{E}(F X, \forall A \in \mathbb{C} . \prod \mathbb{D}(X, I A) . G A) \\ \cong & \quad \{ \text{Ran}_I G := \Lambda X : \mathbb{D} . \forall A \in \mathbb{C} . \prod \mathbb{D}(X, I A) . G A \} \\ & \forall X \in \mathbb{D} . \mathbb{E}(F X, \text{Ran}_I G X) \\ \cong & \quad \{ \text{natural transformation as an end} \} \\ & \mathbb{E}^{\mathbb{D}}(F, \text{Ran}_I G) \end{aligned}$$

The functor $\text{Ran}_I G$ is known as the *right Kan extension* of G along I . (If we view $I : \mathbb{C} \rightarrow \mathbb{D}$ as an inclusion functor, then $\text{Ran}_I G : \mathbb{D} \rightarrow \mathbb{E}$ extends $G : \mathbb{C} \rightarrow \mathbb{E}$ to the whole of \mathbb{D} .) Dually, the left adjoint is called the *left Kan extension* and is defined $\text{Lan}_I F = \Lambda X : \mathbb{D} . \exists A : \mathbb{C} . \sum \mathbb{D}(I A, X) . F A$. The universally quantified object in the definition of Ran_I is a so-called *end*, which corresponds to a polymorphic type in Haskell. Dually, the existentially quantified object is a *coend*, which corresponds to an existential type in Haskell.

Instantiating Theorem 3.1, the parametric types μF and μG are related by $\text{Pre}_I(\mu F) \cong \mu G$ if $\text{Pre}_I \cdot F \cong G \cdot \text{Pre}_I$. The natural isomorphism *swap* realises the space-saving transformation as illustrated below.

Example 5.2 Continuing Example 5.1 let us show that $\text{Pre}_{\text{Maybe}} \cdot \text{List} \cong \text{Seq} \cdot \text{Pre}_{\text{Maybe}}$.

$$\begin{aligned} & \text{List } X \cdot \text{Maybe} \\ \cong & \quad \{ \text{definition of List} \} \\ & \Lambda A . 1 + \text{Maybe } A \times X (\text{Maybe } A) \\ \cong & \quad \{ \text{definition of Maybe} \} \\ & \Lambda A . 1 + (1 + A) \times X (\text{Maybe } A) \\ \cong & \quad \{ \times \text{ distributes over } + \text{ and } 1 \times B \cong B \} \\ & \Lambda A . 1 + X (\text{Maybe } A) + A \times X (\text{Maybe } A) \\ \cong & \quad \{ \text{definition of Seq} \} \\ & \text{Seq } (X \cdot \text{Maybe}) \end{aligned}$$

The central step is the application of distributivity: the law $(A + B) \times C \cong A \times C + B \times C$ turns the nested type on the left into a ‘flat’ sum, which can be represented space-efficiently in Haskell. The witness of the isomorphism, *swap*, makes this explicit.

$$\begin{aligned} \text{swap} &: \forall x a . \text{List } x (\text{Maybe } a) && \rightarrow \text{Seq } (x \cdot \text{Maybe } a) \\ \text{swap} & \quad (\text{Nil}) && = \text{Done} \\ \text{swap} & \quad (\text{Cons } (\text{Nothing}, x)) && = \text{Skip } x \\ \text{swap} & \quad (\text{Cons } (\text{Just } a, x)) && = \text{Yield } (a, x) \end{aligned}$$

The function *swap* is a natural transformation, whose components are again natural transformations, hence the nesting of universal quantifiers. \square

6. Application: tabulation

In this section we look at an intriguing application of type fusion: tabulation. It is well-known that functions from the naturals can be memoised using streams: $X^{\text{Nat}} \cong \text{Stream } X$, where $\text{Nat} = \mu\text{Nat}$ and $\text{Stream} = \nu\text{Stream}$ with base functors

$$\begin{aligned} \text{data Nat } \text{nat} &= \text{Zero} \mid \text{Succ } \text{nat} \\ \text{data Stream } \text{stream } a &= \text{Next } (a, \text{stream } a). \end{aligned}$$

The isomorphism holds for every return type X , so it can be generalised to an isomorphism between functors:

$$(-)^{\text{Nat}} \cong \text{Stream}. \quad (17)$$

Tabulations abound. We routinely use tabulation to represent or to visualise functions from small finite domains. Probably every textbook on computer architecture includes truth tables for the logical connectives.

$$(\wedge) : \text{Bool}^{\text{Bool} \times \text{Bool}} \quad \begin{array}{|c|c|} \hline \text{False} & \text{False} \\ \hline \text{False} & \text{True} \\ \hline \end{array}$$

A function from a pair of Booleans can be represented by a two-by-two table. Again, the construction is parametric.

$$(-)^{\text{Bool} \times \text{Bool}} \cong (\text{Id} \dot{\times} \text{Id}) \dot{\times} (\text{Id} \dot{\times} \text{Id})$$

Here, Id is the identity functor and $\dot{\times}$ is the lifted product defined $(F \dot{\times} G) X = F X \times G X$.

For finite argument types such as $\text{Bool} \times \text{Bool}$ tabulation rests on the well-known *laws of exponentials*:

$$X^0 \cong 1, X^1 \cong X, X^{A+B} \cong X^A \times X^B, X^{A \times B} \cong (X^B)^A.$$

Things become interesting when the types involved are recursive as in the introductory example, and this is where type fusion enters the scene. To be able to apply the framework, we first have to identify the left adjoint functor. Quite intriguingly, the underlying functor is a curried version of exponentiation: $\text{Exp} : \mathbb{C} \rightarrow (\mathbb{C}^{\mathbb{C}})^{\text{op}}$ with

$\text{Exp } K = \Lambda V . V^K$ and $\text{Exp } f = \Lambda V . (id_V)^f$. Using the functor *Exp*, Equation (17) can be rephrased as $\text{Exp } \text{Nat} \cong \text{Stream}$.

This is the first example where the left adjoint is a contravariant functor and this will have consequences when it comes to specialising *swap* and τ . Before we spell out the details, let us first determine the right adjoint of *Exp*, which exists if the underlying category has ends.

$$\begin{aligned} & (\mathbb{C}^{\mathbb{C}})^{\text{op}}(\text{Exp } A, B) \\ \cong & \quad \{ \text{definition of } (-)^{\text{op}} \} \\ & \mathbb{C}^{\mathbb{C}}(B, \text{Exp } A) \\ \cong & \quad \{ \text{natural transformation as an end} \} \\ & \forall X \in \mathbb{C} . \mathbb{C}(B X, \text{Exp } A X) \\ \cong & \quad \{ \text{definition of } \text{Exp} \} \\ & \forall X \in \mathbb{C} . \mathbb{C}(B X, X^A) \\ \cong & \quad \{ - \times Y \dashv (-)^Y \text{ and } Y \times Z \cong Z \times Y \} \\ & \forall X \in \mathbb{C} . \mathbb{C}(A, X^{B X}) \\ \cong & \quad \{ \text{the functor } \mathbb{C}(A, -) \text{ preserves ends} \} \\ & \mathbb{C}(A, \forall X \in \mathbb{C} . X^{B X}) \\ \cong & \quad \{ \text{define } \text{Sel } B := \forall X \in \mathbb{C} . X^{B X} \} \\ & \mathbb{C}(A, \text{Sel } B) \end{aligned}$$

The right adjoint is a higher-order functor that maps a functor B , a type of tables, to the type of *selectors* $\text{Sel } B$, polymorphic functions that select some entry from a given table.

Since *Exp* is a contravariant functor, *swap* and τ live in an opposite category. Moreover, μG in $(\mathbb{C}^{\mathbb{C}})^{\text{op}}$ is a final coalgebra in $\mathbb{C}^{\mathbb{C}}$. Formulated in terms of arrows in $\mathbb{C}^{\mathbb{C}}$, type fusion takes the following form

$$\tau : \nu G \cong \text{Exp } (\mu F) \quad \longleftarrow \quad \text{swap} : G \cdot \text{Exp} \cong \text{Exp} \cdot F,$$

and the isomorphisms τ and τ° are defined

$$\begin{aligned} \text{Exp } in \cdot \tau &= \text{swap} \cdot G \tau \cdot out \\ out \cdot \tau^\circ &= G \tau^\circ \cdot \text{swap}^\circ \cdot \text{Exp } in. \end{aligned}$$

Both arrows are natural in the return type of the exponential. The arrow $\tau : \nu G \dot{\rightarrow} \text{Exp } (\mu F)$ is a curried *look-up* function that maps a memo table to an exponential, which in turn maps an index, an element of μF , to the corresponding entry in the table. Its inverse, $\tau^\circ : \text{Exp } (\mu F) \dot{\rightarrow} \nu G$ *tabulates* a given exponential. Tabulation is a standard unfold, whereas look-up is an adjoint fold, whose transposed fold maps an index to a selector function. Before we look at a Haskell example, let us specialise the defining equations of τ and τ° to the category **Set**, so that we can see the correspondence to the Haskell code more clearly.

$$\text{lookup } (out^\circ t) (in i) = \text{swap } (G \text{lookup } t) i \quad (18)$$

$$\text{tabulate } f = out^\circ (G \text{tabulate } (\text{swap}^\circ (f \cdot in))) \quad (19)$$

Example 6.1 Let us instantiate tabulation to natural numbers and streams. The natural isomorphism $swap$ is defined

$$\begin{aligned} swap &: \forall x v . \mathcal{S}tream (Exp x) v \rightarrow (\mathcal{N}at x \rightarrow v) \\ swap & \quad (\mathcal{N}ext (v, t)) \quad (\mathcal{Z}ero) = v \\ swap & \quad (\mathcal{N}ext (v, t)) \quad (\mathcal{S}ucc n) = t n. \end{aligned}$$

It implements $V \times V^X \cong V^{1+X}$. Inlining $swap$ into Equation (18) yields the look-up function

$$\begin{aligned} lookup &: \forall v . \nu \mathcal{S}tream v \rightarrow (\mu \mathcal{N}at \rightarrow v) \\ lookup (Out^\circ (\mathcal{N}ext (v, t))) (In \mathcal{Z}ero) &= v \\ lookup (Out^\circ (\mathcal{N}ext (v, t))) (In (\mathcal{S}ucc n)) &= lookup t n \end{aligned}$$

that accesses the n th element of a sequence. The inverse of $swap$ implements $V^{1+X} \cong V \times V^X$ and is defined

$$\begin{aligned} swap^\circ &: \forall x . \forall v . (\mathcal{N}at x \rightarrow v) \rightarrow \mathcal{S}tream (Exp x) v \\ swap^\circ & \quad f = \mathcal{N}ext (f \mathcal{Z}ero, f \cdot \mathcal{S}ucc). \end{aligned}$$

If we inline $swap^\circ$ into Equation (19), we obtain

$$\begin{aligned} tabulate &: \forall v . (\mu \mathcal{N}at \rightarrow v) \rightarrow \nu \mathcal{S}tream v \\ tabulate f & \\ &= Out^\circ (\mathcal{N}ext (f (In \mathcal{Z}ero), tabulate (f \cdot In \cdot \mathcal{S}ucc))) \end{aligned}$$

that memoises a function from the naturals. By construction, $lookup$ and $tabulate$ are inverses. \square

The definitions of look-up and tabulate are *generic*: the same code works for any suitable combination of F and G . The natural transformation $swap$ on the other hand depends on the particulars of F and G . The best we can hope for is a *polymorphic* definition that covers a large class of functors. The laws of exponentials provide the basis for the simple class of so-called *polynomial functors*.

$$Exp 0 \cong K 1 \quad (20)$$

$$Exp 1 \cong Id \quad (21)$$

$$Exp (A + B) \cong Exp A \dot{\times} Exp B \quad (22)$$

$$Exp (A \times B) \cong Exp A \cdot Exp B \quad (23)$$

Throughout the paper we have used λ -notation to denote functors. We can extend tabulation to a much larger class of objects if we make this precise. The central idea is to interpret λ -notation using the cartesian closed structure on \mathbf{Alg} , the category of ω -cocomplete categories and ω -cocontinuous functors. The resulting calculus is dubbed Λ -calculus. The type constructors 0 , 1 , $+$, \times and μ are given as *constants* in this language. Naturally, the constants 0 and 1 are interpreted as initial and final objects; the constructors $+$ and \times are mapped to (curried versions of) the coproduct and the product functor. The interpretation of μ , however, is less obvious.

It turns out that the fixed point operator, which maps an endofunctor to its initial algebra, defines a higher-order

functor of type $\mu : (\mathbb{C}^{\mathbb{C}}) \rightarrow \mathbb{C}$. Its action on arrows is given by $\mu \omega = (in \cdot \omega)$. (The type dictates that μ maps a natural transformation to an ordinary arrow.) The reflection law implies that μ preserves identity: $\mu id = (in \cdot id) = id$. To show that μ preserves composition, we first state a simple fusion rule. Let $\beta : G \rightarrow F$ and $f : F A \rightarrow A$, then

$$(f) \cdot \mu \beta = (f \cdot \beta).$$

Preservation of composition is an immediate consequence of this law (set $f := in \cdot \omega$).

For reasons of space, we only sketch the syntax and semantics of the Λ -calculus, see [8] for a more leisurely exposition. Its raw syntax is given below.

$$\begin{aligned} \kappa &::= * \mid \kappa \rightarrow \kappa \\ T &::= C \mid X \mid T T \mid \Lambda X . T \\ C &::= 0 \mid 1 \mid + \mid \times \mid \mu \end{aligned}$$

The kinds of the constants are fixed as follows.

$$\begin{aligned} 0, 1 &: * \\ +, \times &: * \rightarrow (* \rightarrow *) \\ \mu &: (* \rightarrow *) \rightarrow * \end{aligned}$$

The interpretation of this calculus in a cartesian closed category is completely standard [6]. We provide, in fact, two interpretations, one for the types of keys and one for the types of memo tables, and then relate the two, showing that the latter interpretation is a tabulation of the former.

For keys, we specialise the standard interpretation to the category \mathbf{Alg} , fixing a category \mathbb{C} as the interpretation of $*$. For memo tables, the category of ω -complete categories and ω -continuous functors serves as the target. The semantics of $*$ is given by $(\mathbb{C}^{\mathbb{C}})^{op}$, which is ω -complete since \mathbb{C} is ω -cocomplete. In other words, $*$ is interpreted by the domain and the codomain of the adjoint functor Exp , respectively.

The semantics of types is determined by the interpretation of the constants. (We use the same names both for the syntactic and the semantic entities.)

$$\begin{array}{ll} \mathcal{H}[0] = 0 & \mathcal{T}[0] = K 1 \\ \mathcal{H}[1] = 1 & \mathcal{T}[1] = Id \\ \mathcal{H}[+] = + & \mathcal{T}[+] = \dot{\times} \\ \mathcal{H}[\times] = \times & \mathcal{T}[\times] = \cdot \\ \mathcal{H}[\mu] = \mu & \mathcal{T}[\mu] = \nu \end{array}$$

Finally, to relate the types of keys and memo tables, we set up a kind-indexed logical relation.

$$\begin{aligned} (A, F) \in \mathcal{R}_* &\iff Exp A \cong F \\ (A, F) \in \mathcal{R}_{\kappa_1 \rightarrow \kappa_2} &\iff \\ \forall X Y . (X, Y) \in \mathcal{R}_{\kappa_1} &\implies (A X, F Y) \in \mathcal{R}_{\kappa_2} \end{aligned}$$

The first clause expresses the intended relation between key types and memo-table functors. The second clause closes the logical relation under application and abstraction.

Theorem 6.1 (Tabulation) For closed type terms $T : \kappa$,

$$(\mathcal{K}[[T]], \mathcal{T}[[T]]) \in \mathcal{R}_\kappa.$$

Proof We show that \mathcal{R} relates the interpretation of constants. The statement then follows from the ‘basic lemma’ of logical relations. Equations (20)–(23) imply $(\mathcal{K}[[C]], \mathcal{T}[[C]]) \in \mathcal{R}_\kappa$ for $C = 0, 1, +$ and \times . By definition, $(\mu, \nu) \in \mathcal{R}_{(* \rightarrow *) \rightarrow *}$ iff $\forall X Y. (X, Y) \in \mathcal{R}_{* \rightarrow *} \implies (\mu X, \nu Y) \in \mathcal{R}_*$. Since the precondition is equivalent to $Exp \cdot X \cong Y \cdot Exp$, Theorem 3.1 is applicable and implies $Exp(\mu X) \cong \nu Y$, as desired. \square

Note that the functors $\mathcal{T}[[T]]$ contain only products, no coproducts, hence the terms memo table and tabulation.

7. Related work

The initial algebra approach to the semantics of datatypes originates in the work of Lambek on fixed points in categories [14]. Lambek suggests that lattice theory provides a fruitful source of inspiration for results in category theory. This viewpoint was taken up by Backhouse *et al.* [3], who generalise a number of lattice-theoretic fixed point rules to category theory, type fusion being one of them. (The paper contains no proofs; these are provided in an unpublished manuscript [2]. Type fusion is established by showing that $L \dashv R$ induces an adjunction between the categories of F- and G-algebras.) The rules are illustrated by deriving isomorphisms between list types (cons and snoc lists) — currying is the only adjunction considered. The author is not aware of any other published applications of type fusion.

Finite versions of memo tables are known as *tries* or *digital search trees*. Knuth [13] attributes the idea of a trie to Thue [17]. Connelly and Morris [5] formalised the concept of a trie in a categorical setting: they showed that a trie is a functor and that the corresponding look-up function is a natural transformation. The author gave a polytypic definition of tries and memo tables using type-indexed datatypes [9, 10], which Section 6 puts on a sound theoretical footing. The insight that a function from an inductive type is tabulated by a coinductive type is due to Altenkirch [1]. The paper also mentions fusion as a way of proving tabulation correct, but doesn’t spell out the details. (Altenkirch attributes the idea to Backhouse.)

Adjoint folds and unfolds were introduced in a recent paper by the author [11], which in turn was inspired by Bird and Paterson’s work on generalised folds [4]. The fact that μ is a higher-order functor is due to Gibbons and Paterson [8]. That paper also introduces the λ -calculus for types that we adopted for Theorem 6.1.

References

- [1] T. Altenkirch. Representations of first order function types as terminal coalgebras. In *Typed Lambda Calculi and Applications*, LNCS 2044, pages 62–78. Springer-Verlag, 2001.
- [2] R. Backhouse, M. Bijsterveld, R. van Geldrop, and J. van der Woude. Category theory as coherently constructive lattice theory, 1994. Available from <http://www.cs.nott.ac.uk/~rcb/MPC/CatTheory.ps.gz>.
- [3] R. Backhouse, M. Bijsterveld, R. van Geldrop, and J. van der Woude. Categorical fixed point calculus. In D. Pitt, D. E. Rydeheard, and P. Johnstone, editors, *Conference on Category Theory and Computer Science*, LNCS 953, pages 159–179. Springer-Verlag, August 1995.
- [4] R. Bird and R. Paterson. Generalised folds for nested datatypes. *Formal Aspects of Computing*, 11(2):200–222, 1999.
- [5] R. H. Connelly and F. L. Morris. A generalization of the trie data structure. *Mathematical Structures in Computer Science*, 5(3):381–418, September 1995.
- [6] R. L. Crole. *Categories for Types*. Cambridge University Press, 1994.
- [7] O. Danvy. An extensional characterization of lambda-lifting and lambda-dropping. In A. Middeldorp and T. Sato, editors, *Symposium on Functional and Logic Programming*, LNCS 1722, pages 241–250. Springer-Verlag, November 1999.
- [8] J. Gibbons and R. Paterson. Parametric datatype-genericity. In P. Jansson, editor, *ACM SIGPLAN workshop on Generic programming*, pages 85–93. ACM Press, August 2009.
- [9] R. Hinze. Generalizing generalized tries. *Journal of Functional Programming*, 10(4):327–351, July 2000.
- [10] R. Hinze. Memo functions, polytypically! In J. Jeuring, editor, *Workshop on Generic Programming*, pages 17–32, July 2000, technical report of Universiteit Utrecht, UU-CS-2000-19.
- [11] R. Hinze. Adjoint folds and unfolds, 2010. In submission.
- [12] J. Hughes. Type specialisation for the λ -calculus; or, A new paradigm for partial evaluation based on type inference. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation*, LNCS 1110, pages 183–215. Springer-Verlag, 1996.
- [13] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley Publishing Company, 2nd edition, 1998.
- [14] J. Lambek. A fixpoint theorem for complete categories. *Math. Zeitschr.*, 103:151–161, 1968.
- [15] S. MacLane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics. Springer-Verlag, Berlin, 2nd edition, 1998.
- [16] S. Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.
- [17] A. Thue. Über die gegenseitige Lage gleicher Teile gewisser Zeichenreihen. *Skrifter udgivne af Videnskaps-Selskabet i Christiania, Matematisk-Naturvidenskabelig Klasse*, 1:1–67, 1912. Reprinted in Thue’s “Selected Mathematical Papers” (Oslo: Universitetsforlaget, 1977), 413–477.
- [18] T. Uustalu and V. Vene. Coding recursion a la Mendler (extended abstract). In J. Jeuring, editor, *Workshop on Generic Programming*, pages 69–85, July 2000, technical report of Universiteit Utrecht, UU-CS-2000-19.