Type Fusion

Ralf Hinze

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

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.

Keywords: initial algebra, fold, fusion, adjunction, 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:

$$\ell\,(\mu f) = \mu g \quad \Longleftarrow \quad \ell \cdot f = g \cdot \ell \ , \tag{1}$$

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

$$\mathsf{L}\,(\mu\mathsf{F}) \cong \mu\mathsf{G} \quad \Longleftarrow \quad \mathsf{L} \cdot \mathsf{F} \cong \mathsf{G} \cdot \mathsf{L} \ ,$$

where F, G and L are functors between suitable categories and $\mu\mathsf{F}$ 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 [1], but we believe that it deserves to be better known. We provide a novel proof of type fusion based on adjoint folds [2], 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 adjoint folds. (The material is partly taken from [2], which introduces adjoint folds.) 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 $\mathsf{F} : \mathbb{C} \to \mathbb{C}$ be an endofunctor. An F-*algebra* is a pair $\langle A, f \rangle$ consisting of an object $A \in \mathbb{C}$ and an arrow $f \in \mathbb{C}(\mathsf{F}\, A, A)$. 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 $h \cdot f = g \cdot \mathsf{F}\, h$. 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* $\langle \mu\mathsf{F}, \mathit{in} \rangle$. The import of initiality is that there is a unique arrow from $\langle \mu\mathsf{F}, \mathit{in} \rangle$ to any other F-algebra $\langle A, f \rangle$. This unique arrow is written $(\!| f |\!)$ and is called a *fold*. Expressed in terms of the base category, it satisfies the following *universal property*.

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

The universal property has several important consequences [3, 4]. Setting $x := \mathit{id}$ and $f := \mathit{in}$, we obtain the *reflection law*: $(\!| \mathit{in} |\!) = \mathit{id}$. Substituting the left-hand side into the right-hand side gives the *computation law*: $(\!| f |\!) \cdot \mathit{in} = f \cdot \mathsf{F}\, (\!| f |\!)$. Finally and most importantly, it implies the *fusion law* for fusing an arrow with a fold to form another fold.

$$h \cdot (\!| f |\!) = (\!| g |\!) \iff h \cdot f = g \cdot \mathsf{F}\, h \tag{3}$$

Initial algebras provide semantics for inductive or recursive datatypes as found, for instance, in Haskell [5]. The following example illustrates the approach. In fact, Haskell is expressive enough to replay the development within the language.

Example 1 Consider the datatype *Stack* that models stacks of naturals.

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

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

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

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

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

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 that ties the recursive knot:

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

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

$$\begin{aligned}
&(\!|-|\!)\ :\ (\mathit{Functor}\, f) \Rightarrow (f\, b \to b) \to (\mu f \to b) \\
&(\!|f|\!) = f \cdot \mathit{fmap}\, (\!|f|\!) \cdot \mathit{in}^{\circ}
\end{aligned}$$

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:

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

For emphasis, base functors, algebras and, later, base functions are typeset in this $\mathfrak{font}$. □

To understand concepts in category theory it is helpful to look at a 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 special as there is at most one arrow between two objects: $a \to b$ is inhabited if and only if $a \leqslant b$. A functor between two preorders is a *monotone function*, which is a mapping on objects that respects the underlying ordering: $a \leqslant b \Longrightarrow f\ a \leqslant f\ b$. A natural transformation between two monotone functions corresponds to a point-wise ordering: $f \mathrel{\dot{\leqslant}} g \Longleftrightarrow \forall x\ .\ f\ x \leqslant g\ x$. Throughout the section we shall specialise the development to preorders. For type fusion, in 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 $f : P \to P$ be a monotone function. An f-algebra is an element a with $f\ a \leqslant a$, 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 simplifies considerably: all that matters is the type of an arrow. The type of *in* corresponds to the *fixed-point inclusion law*:

$$f\ (\mu f) \leqslant \mu f\ \ , \tag{4}$$

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

$$\mu f \leqslant b \quad \Longleftarrow \quad f\ b \leqslant b\ \ . \tag{5}$$

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. (Generalised to categories, this fact is known as *Lambek's Lemma* [6].) The inclusion law states $f\ (\mu f) \leqslant \mu f$, for the other direction we reason (left column)

$$\begin{array}{llcll}
 & \mu f \leqslant f\ (\mu f) & \qquad & & \llparenthesis \mathsf{F}\,\mathit{in} \rrparenthesis \in \mathbb{C}(\mu\mathsf{F}, \mathsf{F}\,(\mu\mathsf{F})) \\
\Longleftarrow & \{\ \text{induction (5)}\ \} & & \Longleftarrow & \{\ \text{type of fold}\ \} \\
 & f\ (f\ (\mu f)) \leqslant f\ (\mu f) & & & \mathsf{F}\,\mathit{in} \in \mathbb{C}(\mathsf{F}\,(\mathsf{F}\,(\mu\mathsf{F})), \mathsf{F}\,(\mu\mathsf{F})) \\
\Longleftarrow & \{\ f \text{ monotone}\ \} & & \Longleftarrow & \{\ \mathsf{F} \text{ functor}\ \} \\
 & f\ (\mu f) \leqslant \mu f & & & \mathit{in} \in \mathbb{C}(\mathsf{F}\,(\mu\mathsf{F}), \mu\mathsf{F}) \\
\Longleftrightarrow & \{\ \text{inclusion (4)}\ \} & & \Longleftrightarrow & \{\ \text{type of } \mathit{in}\ \} \\
 & \Box & & & \Box
\end{array}$$

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 $\llparenthesis \mathsf{F}\,\mathit{in} \rrparenthesis$, the inverse of *in*. This is made explicit above on the right. The proof on the left is given as a top-down backward implication, with the initial goal at top. While this style is natural for order-theoretic arguments, it is less so for typing derivations, as the witness, here $\llparenthesis \mathsf{F}\,\mathit{in} \rrparenthesis$, appears out of thin air. To follow the term construction, it is advisable to read typing derivations from bottom to top.

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* [7], and to generalise an order-theoretic result we additionally have to establish the required coherence conditions. Continuing the example above, to prove $\mathsf{F}\,(\mu\mathsf{F}) \cong \mu\mathsf{F}$ we must show that $\mathit{in} \cdot \llparenthesis \mathsf{F}\,\mathit{in} \rrparenthesis = \mathit{id}$,

$$\mathit{in} \cdot \llparenthesis \mathsf{F}\,\mathit{in} \rrparenthesis = \mathit{id}$$

$$\begin{array}{cl}
\Longleftrightarrow & \{\text{ reflection }\} \\
& in \cdot (\!|\mathsf{F}\, in|\!) = (\!|in|\!) \\
\Longleftarrow & \{\text{ fusion (3) }\} \\
& in \cdot \mathsf{F}\, in = in \cdot \mathsf{F}\, in \quad ,
\end{array}$$

and $(\!|\mathsf{F}\, in|\!) \cdot in = id$,

$$\begin{array}{cl}
& (\!|\mathsf{F}\, in|\!) \cdot in \\
= & \{\text{ computation }\} \\
& \mathsf{F}\, in \cdot \mathsf{F}\,(\!|\mathsf{F}\, in|\!) \\
= & \{\ \mathsf{F}\text{ functor }\} \\
& \mathsf{F}\,(in \cdot (\!|\mathsf{F}\, in|\!)) \\
= & \{\text{ see above }\} \\
& \mathsf{F}\, id \\
= & \{\ \mathsf{F}\text{ functor }\} \\
& id \quad .
\end{array}$$

Finally, let us remark that the development nicely dualises to F-*coalgebras* and *unfolds*, which give a semantics to coinductive types. The final F-coalgebra is denoted $\langle \nu\mathsf{F},\ out\rangle$.

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

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

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

Practical considerations dictate the introduction of a more general (co-) recursion scheme, christened *adjoint folds and unfolds* [2] 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, where the context is modelled by a functor (L and R below). The functor is subject to a certain condition, which we discuss shortly. The *adjoint fold* $(\!|\Psi|\!)_{\mathsf{L}} \in \mathbb{C}(\mathsf{L}\,(\mu\mathsf{F}), B)$ is then the unique solution of the equation

$$x \cdot \mathsf{L}\, in = \Psi\, x \quad , \tag{6}$$

where the *base function* Ψ has type $\Psi : \forall X \in \mathbb{D}\ .\ \mathbb{C}(\mathsf{L}\,X, B) \to \mathbb{C}(\mathsf{L}\,(\mathsf{F}\,X), B)$. It is important that Ψ is natural in X. Informally speaking, naturality ensures *termination* of $(\!|\Psi|\!)_{\mathsf{L}}$: the first argument of Ψ, the recursive call of x, can only be applied to proper sub-terms of x's argument — each embedded in the context L.

Dually, the *adjoint unfold* $[\!(\Psi)\!]_{\mathsf{R}} \in \mathbb{C}(A, \mathsf{R}\,(\nu\mathsf{F}))$ is the unique solution of the equation

$$\mathsf{R}\,\mathit{out} \cdot x = \Psi\,x\ \ , \qquad (7)$$

where the base function Ψ has type $\Psi : \forall X \in \mathbb{C}\ .\ \mathbb{D}(A, \mathsf{R}\,X) \to \mathbb{D}(A, \mathsf{R}\,(\mathsf{F}\,X))$. Again, the base function has to be natural in X, which now ensures *productivity*.

We have already indicated that the functors L and R cannot be arbitrary. For instance, for $\mathsf{L} := \mathsf{K}\,A$, where $\mathsf{K} : \mathbb{C} \to \mathbb{C}^{\mathbb{D}}$ is the constant functor and $\Psi := \mathit{id} : \mathbb{C}(A, B) \to \mathbb{C}(A, B)$, Equation (6) simplifies to the trivial $x = x$. One approach for ensuring uniqueness is to express x in terms of a standard fold. This is where adjunctions enter the scene: *we require* L *and* R *to be adjoint.* Briefly, let $\mathbb{C}$ and $\mathbb{D}$ be categories. The functors L and R are *adjoint*, $\mathsf{L} \dashv \mathsf{R}$,

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

if and only if there is a bijection

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

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 enables us to reduce an adjoint fold to a standard fold, for the proof see [2]. The standard fold $\phi\,(\!|\Psi|\!)_{\mathsf{L}} \in \mathbb{C}(\mu\mathsf{F}, \mathsf{R}\,B)$ is called the *transpose* of $(\!|\Psi|\!)_{\mathsf{L}}$.

Example 3 In the case of *shunt*, the adjoint functor is pairing defined $\mathsf{L}\,X = X \times \mathit{Stack}$ and $\mathsf{L}f = f \times \mathit{id}_{\mathit{Stack}}$. Its right adjoint is exponentiation defined $\mathsf{R}\,Y = Y^{\mathit{Stack}}$ and $\mathsf{R}f = f^{\mathit{id}_{\mathit{Stack}}}$. 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 $\mathfrak{shunt}$ that abstracts away from the recursive call and an adjoint equation that ties the recursive knot.

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

The last equation is the point-wise variant of $\mathit{shunt} \cdot \mathsf{L}\,\mathit{in} = \mathfrak{shunt}\,\mathit{shunt}$. The transposed fold is simply the curried variant of *shunt*. □

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

$$\ell\, a \leqslant b \iff a \leqslant r\, b\ . \tag{9}$$

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

$$\ell\,(\mu f) \leqslant b \Longleftarrow (\forall x \in Q\,.\, \ell\, x \leqslant b \Longrightarrow \ell\,(f\, x) \leqslant b) \tag{10}$$

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.

$$\mathsf{L}\,(\mu\mathsf{F}) \cong \mu\mathsf{G} \Longleftarrow \mathsf{L}\cdot\mathsf{F} \cong \mathsf{G}\cdot\mathsf{L}$$

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

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

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 $\mu g \leqslant \ell\,(\mu f) \Longleftarrow g\cdot\ell \mathrel{\dot{\leqslant}} \ell\cdot f$ and second that $\ell\,(\mu f) \leqslant \mu g \Longleftarrow \ell\cdot f \mathrel{\dot{\leqslant}} g\cdot\ell$.

$$\begin{aligned}
& \mu g \leqslant \ell\,(\mu f) \\
\Longleftarrow\quad & \{\text{ induction (5) }\} \\
& g\,(\ell\,(\mu f)) \leqslant \ell\,(\mu f) \\
\Longleftarrow\quad & \{\text{ transitivity }\} \\
& g\,(\ell\,(\mu f)) \leqslant \ell\,(f\,(\mu f)) \ \text{ and } \ \ell\,(f\,(\mu f)) \leqslant \ell\,(\mu f) \\
\Longleftarrow\quad & \{\text{ assumption } g\cdot\ell \mathrel{\dot{\leqslant}} \ell\cdot f \ \} \\
& \ell\,(f\,(\mu f)) \leqslant \ell\,(\mu f) \\
\Longleftarrow\quad & \{\ \ell \text{ monotone }\} \\
& f\,(\mu f) \leqslant \mu f \\
\Longleftrightarrow\quad & \{\text{ inclusion (4) }\} \\
& \Box
\end{aligned}$$

For part two, we apply adjoint induction (10), which leaves us with the obligation $\forall x \in Q \,.\, \ell\, x \leqslant \mu g \Longrightarrow \ell\,(f\,x) \leqslant \mu g$.

$$\begin{aligned}
& \ell\,(f\,x) \leqslant \mu g \\
\Longleftarrow\quad & \{\ \text{transitivity}\ \} \\
& \ell\,(f\,x) \leqslant g\,(\ell\,x) \ \text{ and }\ g\,(\ell\,x) \leqslant \mu g \\
\Longleftarrow\quad & \{\ \text{assumption } \ell \cdot f \mathrel{\dot{\leqslant}} g \cdot \ell\ \} \\
& g\,(\ell\,x) \leqslant \mu g \\
\Longleftarrow\quad & \{\ \text{transitivity}\ \} \\
& g\,(\ell\,x) \leqslant g\,(\mu g) \ \text{ and }\ g\,(\mu g) \leqslant \mu g \\
\Longleftarrow\quad & \{\ g \text{ monotone and inclusion (4)}\ \} \\
& \ell\,x \leqslant \mu g
\end{aligned}$$

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 $\mathsf{L} \cdot \mathsf{F} \cong \mathsf{G} \cdot \mathsf{L}$ is called *swap*.)

$$\begin{aligned}
& (\!|\mathsf{L}\,\mathit{in} \cdot \mathit{swap}^{\circ}|\!) \in \mathbb{C}(\mu\mathsf{G}, \mathsf{L}\,(\mu\mathsf{F})) \\
\Longleftarrow\quad & \{\ \text{type of fold}\ \} \\
& \mathsf{L}\,\mathit{in} \cdot \mathit{swap}^{\circ} \in \mathbb{C}(\mathsf{G}\,(\mathsf{L}\,(\mu\mathsf{F})), \mathsf{L}\,(\mu\mathsf{F})) \\
\Longleftarrow\quad & \{\ \text{composition}\ \} \\
& \mathit{swap}^{\circ} \in \mathbb{C}(\mathsf{G}\,(\mathsf{L}\,(\mu\mathsf{F})), \mathsf{L}\,(\mathsf{F}\,(\mu\mathsf{F}))) \ \text{ and }\ \mathsf{L}\,\mathit{in} \in \mathbb{C}(\mathsf{L}\,(\mathsf{F}\,(\mu\mathsf{F})), \mathsf{L}\,(\mu\mathsf{F})) \\
\Longleftarrow\quad & \{\ \text{assumption } \mathit{swap}^{\circ} : \mathsf{G} \cdot \mathsf{L} \mathrel{\dot{\to}} \mathsf{L} \cdot \mathsf{F}\ \} \\
& \mathsf{L}\,\mathit{in} \in \mathbb{C}(\mathsf{L}\,(\mathsf{F}\,(\mu\mathsf{F})), \mathsf{L}\,(\mu\mathsf{F})) \\
\Longleftarrow\quad & \{\ \mathsf{L} \text{ functor}\ \} \\
& \mathit{in} \in \mathbb{D}(\mathsf{F}\,(\mu\mathsf{F}), \mu\mathsf{F}) \\
\Longleftrightarrow\quad & \{\ \text{type of } \mathit{in}\ \} \\
& \square
\end{aligned}$$

Conversely, the arrow τ is the adjoint fold $(\!|\Psi|\!)_{\mathsf{L}}$ whose base function Ψ is given by the second proof.

$$\begin{aligned}
& \mathit{in} \cdot \mathsf{G}\,x \cdot \mathit{swap} \in \mathbb{C}(\mathsf{L}\,(\mathsf{F}\,X), \mu\mathsf{G}) \\
\Longleftarrow\quad & \{\ \text{composition}\ \} \\
& \mathit{swap} \in \mathbb{C}(\mathsf{L}\,(\mathsf{F}\,X), \mathsf{G}\,(\mathsf{L}\,X)) \ \text{ and }\ \mathit{in} \cdot \mathsf{G}\,x \in \mathbb{C}(\mathsf{G}\,(\mathsf{L}\,X), \mu\mathsf{G}) \\
\Longleftarrow\quad & \{\ \text{assumption } \mathit{swap} : \mathsf{L} \cdot \mathsf{F} \mathrel{\dot{\to}} \mathsf{G} \cdot \mathsf{L}\ \} \\
& \mathit{in} \cdot \mathsf{G}\,x \in \mathbb{C}(\mathsf{G}\,(\mathsf{L}\,X), \mu\mathsf{G}) \\
\Longleftarrow\quad & \{\ \text{composition}\ \} \\
& \mathsf{G}\,x \in \mathbb{C}(\mathsf{G}\,(\mathsf{L}\,X), \mathsf{G}\,(\mu\mathsf{G})) \ \text{ and }\ \mathit{in} \in \mathbb{C}(\mathsf{G}\,(\mu\mathsf{G}), \mu\mathsf{G}) \\
\Longleftarrow\quad & \{\ \mathsf{G} \text{ functor and type of } \mathit{in}\ \} \\
& x \in \mathbb{C}(\mathsf{L}\,X, \mu\mathsf{G})
\end{aligned}$$

We may conclude that $\tau = (\!|\lambda\, x \,.\, in \cdot \mathsf{G}\, x \cdot swap|\!)_{\mathsf{L}}$ and $\tau^{\circ} = (\!|\mathsf{L}\, in \cdot swap^{\circ}|\!)$ are the desired arrows. The diagram below visualises the type information.

$$\begin{tikzcd}
 & \mathsf{G}\,(\mathsf{L}\,(\mu\mathsf{F})) \arrow[dl, bend left=10, "swap^{\circ}"] \arrow[dr, bend left=10, "\mathsf{G}\,\tau^{\circ}"] & \\
\mathsf{L}\,(\mathsf{F}\,(\mu\mathsf{F})) \arrow[ur, bend left=10, "swap"] \arrow[d, "\mathsf{L}\,in"', "\mathsf{L}\,in"] & & \mathsf{G}\,(\mu\mathsf{G}) \arrow[ul, bend left=10, "\mathsf{G}\,\tau"] \arrow[d, "in"', "in"] \\
\mathsf{L}\,(\mu\mathsf{F}) \arrow[rr, bend right=10, "\tau"'] & & \mu\mathsf{G} \arrow[ll, bend right=10, "\tau^{\circ}"']
\end{tikzcd}$$

All that remains is to establish that they are inverses.

Theorem 1 (Type fusion). *Let* $\mathbb{C}$ *and* $\mathbb{D}$ *be categories, let* $\mathsf{L} \dashv \mathsf{R}$ *be an adjoint pair of functors* $\mathsf{L} : \mathbb{D} \to \mathbb{C}$ *and* $\mathsf{R} : \mathbb{C} \to \mathbb{D}$, *and let* $\mathsf{F} : \mathbb{D} \to \mathbb{D}$ *and* $\mathsf{G} : \mathbb{C} \to \mathbb{C}$ *be two endofunctors. Then*

$$\mathsf{L}\,(\mu\mathsf{F}) \cong \mu\mathsf{G} \iff \mathsf{L} \cdot \mathsf{F} \cong \mathsf{G} \cdot \mathsf{L} \ ; \tag{11}$$

$$\nu\mathsf{F} \cong \mathsf{R}\,(\nu\mathsf{G}) \iff \mathsf{F} \cdot \mathsf{R} \cong \mathsf{R} \cdot \mathsf{G} \ . \tag{12}$$

Proof. We show type fusion for initial algebras (11), the corresponding statement for final coalgebras (12) follows by duality. The isomorphisms τ and $\tau^{\circ}$are given as solutions of adjoint fixed point equations:

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

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

$$\begin{aligned}
& (\tau \cdot \tau^{\circ}) \cdot in \\
= \quad & \{ \text{ definition of } \tau^{\circ} \text{ and } \tau \ \} \\
& in \cdot \mathsf{G}\,\tau \cdot swap \cdot swap^{\circ} \cdot \mathsf{G}\,\tau^{\circ} \\
= \quad & \{ \text{ inverses } \} \\
& in \cdot \mathsf{G}\,\tau \cdot \mathsf{G}\,\tau^{\circ} \\
= \quad & \{ \ \mathsf{G} \text{ functor } \} \\
& in \cdot \mathsf{G}\,(\tau \cdot \tau^{\circ}) \ .
\end{aligned}$$

The equation $x \cdot in = in \cdot \mathsf{G}\, x$ has a unique solution — the base function $\Psi\, x = in \cdot \mathsf{G}\, x$ is a natural transformation of type $\forall X \in \mathbb{C} \,.\, \mathbb{C}(X, \mu\mathsf{G}) \to \mathbb{C}(\mathsf{G}\, X, \mu\mathsf{G})$. Since id is also a solution, the result follows.

Proof of $\tau^{\circ} \cdot \tau = id_{\mathsf{L}\,(\mu\mathsf{F})}$:

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

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

Note that in order to apply type fusion for initial algebras (11) it is sufficient to know that the functor L is part of an adjoint situation — there is no need to make R explicit. Likewise, for fusing final coalgebras (12) it is sufficient to know that R has a left adjoint.

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 Recall the type of stacks of natural numbers.

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

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

$$\mathbf{data}\, \mathsf{List}\; a = Nil \mid Cons\,(a \times \mathsf{List}\; a)\ \ .$$

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

The inverse of abstraction is application or instantiation. We regain the original concept by instantiating the parameter to the constant. Continuing Example 4, we expect that

$$\mathsf{List}\, Nat \cong Stack\ \ . \qquad (13)$$

The isomorphism is non-trivial, as both types are recursively defined. The transformation of $\mathsf{List}\, Nat$ into $Stack$ can be seen as an instance of *firstification* [8] or λ-*dropping* [9] 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 an object $T \in \mathbb{D}$ define $\mathsf{App}_T : \mathbb{C}^{\mathbb{D}} \to \mathbb{C}$ by $\mathsf{App}_T\, \mathsf{F} = \mathsf{F}\, T$ and $\mathsf{App}_T\, \alpha = \alpha\, T$. Using type application we can rephrase Equation (13) as $\mathsf{App}_{Nat}\,(\mu\mathfrak{List}) \cong \mu\mathfrak{Stack}$, where $\mathfrak{List}$ is the higher-order base functor of List defined

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

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 [2]. Consequently, we can firstify both inductive and coinductive parametric types. Generalising the original problem, Equation (13), the second-order type $\mu\mathsf{F}$ and the first-order type $\mu\mathsf{G}$ are related by $\mathsf{App}_T\,(\mu\mathsf{F}) \cong \mu\mathsf{G}$ if $\mathsf{App}_T \cdot \mathsf{F} \cong \mathsf{G} \cdot \mathsf{App}_T$. Despite the somewhat complicated type, the natural isomorphism $swap$ is usually straightforward to define: it simply renames the constructors, as illustrated below.

Example 5 Let us show that $\mathsf{List}\, Nat \cong Stack$. We have to discharge the obligation $\mathsf{App}_{Nat} \cdot \mathfrak{List} \cong \mathfrak{Stack} \cdot \mathsf{App}_{Nat}$.

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

The proof above is entirely straightforward. The isomorphisms are

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

The transformations rename $\mathfrak{Nil}$ to $\mathfrak{Empty}$ and $\mathfrak{Cons}$ to $\mathfrak{Push}$, and vice versa.

Finally, τ and τ° implement Λ-lifting and Λ-dropping.

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

Since type application is invisible in Haskell, the adjoint fold Λ-*drop* deceptively resembles a standard fold. □

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 [10] defined

$$\mathbf{data}\ \mathsf{Perfect}\ a = Zero\ a \mid Succ\, (\mathsf{Perfect}\, (a \times a))$$

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

Definition 1. *Let* $\mathsf{F} : \mathbb{C}^{\mathbb{D}} \to \mathbb{C}^{\mathbb{D}}$ *be a higher-order functor. The parametric datatype* $\mu\mathsf{F} : \mathbb{D} \to \mathbb{C}$ *is* regular *if and only if there exists a functor* $\mathsf{G} : \mathbb{C} \to \mathbb{C}$ *such that* $\mathsf{App}_T \cdot \mathsf{F} \cong \mathsf{G} \cdot \mathsf{App}_T$ *for all objects* $T : \mathbb{D}$. □

The regularity condition can be simplified to $\mathsf{F}\,\mathsf{H}\,T \cong \mathsf{G}\,(\mathsf{H}\,T)$, 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 6 Lists of optional values, $\mathsf{List} \cdot \mathsf{Maybe}$, where Maybe is given by

$$\mathbf{data}\,\mathsf{Maybe}\;a = \mathit{Nothing} \mid \mathit{Just}\;a \quad ,$$

can be represented more compactly using

$$\mathbf{data}\,\mathsf{Seq}\;a = \mathit{Done} \mid \mathit{Skip}\,(\mathsf{Seq}\;a) \mid \mathit{Yield}\,(a \times \mathsf{Seq}\;a) \quad .$$

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

The goal of this section is to prove that

$$\mathsf{List} \cdot \mathsf{Maybe} \cong \mathsf{Seq} \quad , \tag{14}$$

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

To fit Equation (14) under the umbrella of type fusion, we have to view pre-composition as a functor. Given a functor $\mathsf{J} : \mathbb{C} \to \mathbb{D}$, define the higher-order functor $\mathsf{Pre}_{\mathsf{J}} : \mathbb{E}^{\mathbb{D}} \to \mathbb{E}^{\mathbb{C}}$ by $\mathsf{Pre}_{\mathsf{J}}\,\mathsf{F} = \mathsf{F} \cdot \mathsf{J}$ and $\mathsf{Pre}_{\mathsf{J}}\,\alpha = \alpha \cdot \mathsf{J}$. Using the functor we can rephrase Equation (14) as $\mathsf{Pre}_{\mathsf{Maybe}}\,(\mu\mathfrak{List}) \cong \mu\mathfrak{Seq}$.

Of course, we first have to verify that $\mathsf{Pre}_{\mathsf{J}}$ has a adjoint. It turns out that this is a well-studied problem in category theory [11, X.3]. Similar to the situation of the previous section, $\mathsf{Pre}_{\mathsf{J}}$ has both a left and a right adjoint, the so-called left and right Kan extension. Instantiating Theorem 1, the parametric types $\mu\mathsf{F}$ and $\mu\mathsf{G}$ are related by $\mathsf{Pre}_{\mathsf{J}}\,(\mu\mathsf{F}) \cong \mu\mathsf{G}$ if $\mathsf{Pre}_{\mathsf{J}} \cdot \mathsf{F} \cong \mathsf{G} \cdot \mathsf{Pre}_{\mathsf{J}}$. The natural isomorphism *swap* realises the space-saving transformation as illustrated below.

Example 7 Continuing Example 6 we demonstrate that $\mathsf{Pre}_{\mathsf{Maybe}} \cdot \mathfrak{List} \cong \mathfrak{Seq} \cdot \mathsf{Pre}_{\mathsf{Maybe}}$.

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

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 — *swap*'s definition makes this explicit.

$$\begin{array}{lll} \mathit{swap} : \forall x \,.\, \forall a \,.\, \mathfrak{List}\, x\, (\mathsf{Maybe}\, a) & & \to \mathfrak{Seq}\, (x \cdot \mathsf{Maybe})\, a \\ \mathit{swap} & (\mathfrak{Nil}) & = \mathfrak{Done} \\ \mathit{swap} & (\mathfrak{Cons}\, (\mathit{Nothing}, x)) & = \mathfrak{Skip}\, x \\ \mathit{swap} & (\mathfrak{Cons}\, (\mathit{Just}\, a, \quad x)) & = \mathfrak{Yield}\, (a, x) \end{array}$$

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

6 Application: Tabulation

In this section we look at an intriguing application of type fusion: tabulation. Tabulation means to put something into tabular form. Here, the "something" is a function. For example, it is well-known that functions from the naturals can be memoised or tabulated using streams: $X^{Nat} \cong \mathsf{Stream}\, X$, where $Nat = \mu\mathfrak{Nat}$ and $\mathsf{Stream} = \nu\mathfrak{Stream}$ with

$$\begin{array}{l} \mathbf{data}\, \mathfrak{Nat}\, \mathit{nat} = \mathfrak{Zero} \mid \mathfrak{Succ}\, \mathit{nat} \\ \mathbf{data}\, \mathfrak{Stream}\, \mathit{stream}\ a = \mathfrak{Next}\, (a, \mathit{stream}\ a) \\ \mathbf{newtype}\, \nu f\ a = \mathit{Out}^{\circ}\, \{\, \mathit{out} : f\, (\nu f)\, a \,\} \ . \end{array}$$

The last definition introduces second-order final coalgebras, which model parametric coinductive datatypes.

The isomorphism $X^{Nat} \cong \mathsf{Stream}\, X$ holds for every return type X, so it can be generalised to an isomorphism between functors:

$$(-)^{Nat} \cong \mathsf{Stream} \ . \qquad (15)$$

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.

$(\wedge) : Bool^{Bool \times Bool}$

False	*False*
False	*True*

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

$$(-)^{Bool \times Bool} \cong (\mathsf{Id} \mathbin{\dot\times} \mathsf{Id}) \mathbin{\dot\times} (\mathsf{Id} \mathbin{\dot\times} \mathsf{Id}) \ ,$$

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

For finite argument types such as $Bool \times Bool$, where $Bool = 1+1$, tabulation rests on the well-known *laws of exponentials*:

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

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: $\mathsf{Exp} : \mathbb{C} \to (\mathbb{C}^{\mathbb{C}})^{\mathsf{op}}$ with $\mathsf{Exp}\,\mathsf{K} = \Lambda\, V \,.\, V^{\mathsf{K}}$ and $\mathsf{Exp}\,f = \Lambda\, V \,.\, V^{f}$. Using the functor Exp, Equation (15) can be rephrased as $\mathsf{Exp}\,\mathit{Nat} \cong \mathsf{Stream}$.

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 so-called ends.

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

The universally quantified object introduced in the second step is an *end*, which corresponds to a polymorphic type in Haskell. We refer the interested reader to Mac Lane's textbook [11, IX.5] for further information.

The derivation shows that the right adjoint of Exp is a higher-order functor that maps a functor B, a type of tables, to the type of *selectors* $\mathsf{Sel}\,\mathsf{B}$, 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, $\mu\mathsf{G}$ in $(\mathbb{C}^{\mathbb{C}})^{\mathsf{op}}$ 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

$$\tau : \nu\mathsf{G} \cong \mathsf{Exp}\,(\mu\mathsf{F}) \quad \Longleftarrow \quad \mathit{swap} : \mathsf{G} \cdot \mathsf{Exp} \cong \mathsf{Exp} \cdot \mathsf{F} \ ,$$

and the isomorphisms τ and τ° are defined

$$\mathsf{Exp}\,\mathit{in} \cdot \tau = \mathit{swap} \cdot \mathsf{G}\,\tau \cdot \mathit{out} \ ; \tag{16}$$

$$\mathit{out} \cdot \tau^{\circ} \quad = \mathsf{G}\,\tau^{\circ} \cdot \mathit{swap}^{\circ} \cdot \mathsf{Exp}\,\mathit{in} \ . \tag{17}$$

Both arrows are natural in the return type of the exponential. The arrow $\tau : \nu\mathsf{G} \dot{\to} \mathsf{Exp}\,(\mu\mathsf{F})$ is a curried *look-up* function that maps a memo table to an exponential, which in turn maps an index, an element of $\mu\mathsf{F}$, to the corresponding entry in the table. Its inverse, $\tau^\circ : \mathsf{Exp}\,(\mu\mathsf{F}) \dot{\to} \nu\mathsf{G}$ *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.

$$\mathit{lookup}\,(\mathit{out}^\circ\,t)\,(\mathit{in}\,i) = \mathit{swap}\,(\mathsf{G}\,\mathit{lookup}\,t)\,i \tag{18}$$

$$\mathit{tabulate}\,f = \mathit{out}^\circ\,(\mathsf{G}\,\mathit{tabulate}\,(\mathit{swap}^\circ\,(f \cdot \mathit{in}))) \tag{19}$$

Example 8 Let us instantiate tabulation to natural numbers and streams. The natural isomorphism *swap* is defined

$$\begin{array}{lllll}
\mathit{swap} : \forall x\,.\,\forall v\,.\,\mathfrak{Stream}\,(\mathsf{Exp}\,x)\,v \to & (\mathfrak{Nat}\,x & \to v) \\
\mathit{swap} \qquad\quad (\mathfrak{Next}\,(v,t)) & (\mathfrak{Zero}) & = v \\
\mathit{swap} \qquad\quad (\mathfrak{Next}\,(v,t)) & (\mathfrak{Succ}\,n) & = t\,n \quad .
\end{array}$$

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

$$\begin{array}{llll}
\mathit{lookup} : \forall v\,.\,\nu\mathfrak{Stream}\,v \to & (\mu\mathfrak{Nat} & \to v) \\
\mathit{lookup} \quad (\mathit{Out}^\circ\,(\mathfrak{Next}\,(v,t))) & (\mathit{In}\,\mathfrak{Zero}) & = v \\
\mathit{lookup} \quad (\mathit{Out}^\circ\,(\mathfrak{Next}\,(v,t))) & (\mathit{In}\,(\mathfrak{Succ}\,n)) & = \mathit{lookup}\,t\,n
\end{array}$$

that accesses the nth element of a sequence. Its transpose

$$\begin{array}{llll}
\mathit{lookup}' : \mu\mathfrak{Nat} \to \forall v\,.\,\nu\mathfrak{Stream}\,v & & \to v \\
\mathit{lookup}' \quad (\mathit{In}\,\mathfrak{Zero}) & (\mathit{Out}^\circ\,(\mathfrak{Next}\,(v,t))) & = v \\
\mathit{lookup}' \quad (\mathit{In}\,(\mathfrak{Succ}\,n)) & (\mathit{Out}^\circ\,(\mathfrak{Next}\,(v,t))) & = \mathit{lookup}'\,n\,t
\end{array}$$

simply swaps the two value arguments: given a natural number n, it returns a tailor-made, polymorphic access function that extracts the nth element.

The inverse of *swap* implements $V^{1+X} \cong V \times V^X$ and is defined

$$\begin{array}{lll}
\mathit{swap}^\circ : \forall x\,.\,\forall v\,.\,(\mathfrak{Nat}\,x \to v) \to \mathfrak{Stream}\,(\mathsf{Exp}\,x)\,v \\
\mathit{swap}^\circ \qquad\qquad \mathfrak{f} \qquad\qquad = \mathfrak{Next}\,(\mathfrak{f}\,\mathfrak{Zero}, \mathfrak{f} \cdot \mathfrak{Succ}) \quad .
\end{array}$$

If we inline swap° into Equation (19), we obtain

$$\begin{array}{l}
\mathit{tabulate} : \forall v\,.\,(\mu\mathfrak{Nat} \to v) \to \nu\mathfrak{Stream}\,v \\
\mathit{tabulate} \qquad f = \mathit{Out}^\circ\,(\mathfrak{Next}\,(f\,(\mathit{In}\,\mathfrak{Zero}), \mathit{tabulate}\,(f \cdot \mathit{In} \cdot \mathfrak{Succ})))
\end{array}$$

that memoises a function from the naturals. By construction, *lookup* and *tabulate* are inverses. □

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 *polytypic* definition that covers a large class of functors. The laws of exponentials provide the basis for the simple class of so-called *polynomial functors.*

$$\mathsf{Exp}\,0 \cong \mathsf{K}\,1 \tag{20}$$
$$\mathsf{Exp}\,1 \cong \mathsf{Id} \tag{21}$$
$$\mathsf{Exp}\,(A+B) \cong \mathsf{Exp}\,A \mathbin{\dot{\times}} \mathsf{Exp}\,B \tag{22}$$
$$\mathsf{Exp}\,(A\times B) \cong \mathsf{Exp}\,A \cdot \mathsf{Exp}\,B \tag{23}$$

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 **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 $\mu : \mathbb{C}^{\mathbb{C}} \to \mathbb{C}$, whose action on arrows is given by $\mu\alpha = (\!| in \cdot \alpha |\!)$ [12].

For reasons of space, we only sketch the syntax and semantics of the Λ-calculus, see [12] for a more leisurely exposition. Its raw syntax is given below.

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

The so-called kind $*$ represents types that contain values. The kind $\kappa_1 \to \kappa_2$ represents type constructors that map type constructors of kind κ_1 to those of kind κ_2. The kinds of the constants in C are fixed as follows.

$$0, 1 : * \qquad +, \times : * \to (* \to *) \qquad \mu : (* \to *) \to *$$

The interpretation of this calculus in a cartesian closed category is completely standard [13]. 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 **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 $(\mathbb{C}^{\mathbb{C}})^{\mathsf{op}}$, 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.)

$$\begin{array}{lllll}
\mathscr{K}[\![0]\!] = 0 & \mathscr{K}[\![1]\!] = 1 & \mathscr{K}[\![+]\!] = + & \mathscr{K}[\![\times]\!] = \times & \mathscr{K}[\![\mu]\!] = \mu\\
\mathscr{T}[\![0]\!] = \mathsf{K}\,1 & \mathscr{T}[\![1]\!] = \mathsf{Id} & \mathscr{T}[\![+]\!] = \dot{\times} & \mathscr{T}[\![\times]\!] = \cdot & \mathscr{T}[\![\mu]\!] = \nu
\end{array}$$

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

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

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

Theorem 2 (Tabulation). *For closed type terms $T : \kappa$,*

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

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

Note that the functors $\mathscr{T}[\![T]\!]$ 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 [6]. 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.* [1], 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 [7]. Type fusion is established by showing that $\mathsf{L} \dashv \mathsf{R}$ 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.

Finite versions of memo tables are known as *tries* or *digital search trees*. Knuth [14] attributes the idea of a trie to Thue [15]. Connelly and Morris [16] formalised the concept of a trie in a categorical setting: they showed that a trie is a functor and that the corresponding look-up function is a natural transformation. The author gave a polytypic definition of memo tables using type-indexed datatypes [17, 18], 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 [19]. He also mentions fusion as a way of proving tabulation correct, but does not spell out the details. (Altenkirch attributes the idea to Backhouse.)

Adjoint folds and unfolds were introduced in a recent paper by the author [2], which in turn was inspired by Bird and Paterson's work on generalised folds [20]. The fact that μ is a higher-order functor seems to be folklore, see [12] for a recent reference. That paper also introduces the λ-calculus for types that we adopted for Theorem 2.

Acknowledgement

Thanks are due to the anonymous referees of AMAST 2010 for helpful suggestions and for pointing out some presentational problems.

References

1. Backhouse, R., Bijsterveld, M., van Geldrop, R., van der Woude, J.: Categorical fixed point calculus. In Pitt, D., Rydeheard, D.E., Johnstone, P., eds.: Proceedings of the 6th International Conference on Category Theory and Computer Science (CTCS '95), Cambridge, UK. Volume 953 of Lecture Notes in Computer Science., Springer-Verlag (August 1995) 159–179
2. Hinze, R.: Adjoint folds and unfolds. In Bolduc, C., Desharnais, J., Ktari, B., eds.: 10th International Conference on Mathematics of Program Construction (MPC '10). Volume 6120 of Lecture Notes in Computer Science., Springer-Verlag (July 2010) 195–228
3. Bird, R., de Moor, O.: Algebra of Programming. Prentice Hall Europe, London (1997)
4. Backhouse, R., Jansson, P., Jeuring, J., Meertens, L.: Generic Programming: An Introduction. In Swierstra, S.D., Henriques, P.R., Oliveira, J.N., eds.: 3rd International Summer School on Advanced Functional Programming, Braga, Portugal. Volume 1608 of Lecture Notes in Computer Science. Springer-Verlag, Berlin (1999) 28–115
5. Peyton Jones, S.: Haskell 98 Language and Libraries. Cambridge University Press (2003)
6. Lambek, J.: A fixpoint theorem for complete categories. Math. Zeitschr. **103** (1968) 151–161
7. Backhouse, R., Bijsterveld, M., van Geldrop, R., van der Woude, J.: Category theory as coherently constructive lattice theory (1994) Available from `http://www.cs.nott.ac.uk/~rcb/MPC/CatTheory.ps.gz`.
8. Hughes, J.: Type specialisation for the λ-calculus; or, A new paradigm for partial evaluation based on type inference. In Danvy, O., Glück, R., Thiemann, P., eds.: Partial Evaluation. Dagstuhl Castle, Germany, February 1996. Volume 1110 of Lecture Notes in Computer Science., Springer-Verlag (1996) 183–215
9. Danvy, O.: An extensional characterization of lambda-lifting and lambda-dropping. In Middeldorp, A., Sato, T., eds.: 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99), Tsukuba, Japan. Volume 1722 of Lecture Notes in Computer Science., Springer-Verlag (November 1999) 241–250
10. Hinze, R.: Functional Pearl: Perfect trees and bit-reversal permutations. Journal of Functional Programming **10**(3) (May 2000) 305–317
11. Mac Lane, S.: Categories for the Working Mathematician. 2nd edn. Graduate Texts in Mathematics. Springer-Verlag, Berlin (1998)
12. Gibbons, J., Paterson, R.: Parametric datatype-genericity. In Jansson, P., ed.: Proceedings of the 2009 ACM SIGPLAN workshop on Generic programming, ACM Press (August 2009) 85–93
13. Crole, R.L.: Categories for Types. Cambridge University Press (1994)
14. Knuth, D.E.: The Art of Computer Programming, Volume 3: Sorting and Searching. 2nd edn. Addison-Wesley Publishing Company (1998)

15. Thue, A.: Über die gegenseitige Lage gleicher Teile gewisser Zeichenreihen. Skrifter udgivne af Videnskaps-Selskabet i Christiania, Mathematisk-Naturvidenskabelig Klasse **1** (1912) 1–67 Reprinted in Thue's "Selected Mathematical Papers" (Oslo: Universitetsforlaget, 1977), 413–477.
16. Connelly, R.H., Morris, F.L.: A generalization of the trie data structure. Mathematical Structures in Computer Science **5**(3) (September 1995) 381–418
17. Hinze, R.: Generalizing generalized tries. Journal of Functional Programming **10**(4) (July 2000) 327–351
18. Hinze, R.: Memo functions, polytypically! In Jeuring, J., ed.: Proceedings of the 2nd Workshop on Generic Programming, Ponte de Lima, Portugal. (July 2000) 17–32 The proceedings appeared as a technical report of Universiteit Utrecht, UU-CS-2000-19.
19. Altenkirch, T.: Representations of first order function types as terminal coalgebras. In: Typed Lambda Calculi and Applications, TLCA 2001. Volume 2044 of Lecture Notes in Computer Science., Springer-Verlag (2001) 62–78
20. Bird, R., Paterson, R.: Generalised folds for nested datatypes. Formal Aspects of Computing **11**(2) (1999) 200–222