Arrows, like Monads, are Monoids

Chris Heunen and Bart Jacobs [1]

Institute for Computing and Information Sciences
Radboud University, Nijmegen, the Netherlands,
Email: {c.heunen,b.jacobs}@cs.ru.nl

Abstract

Monads are by now well-established as programming construct in functional languages. Recently, the notion of "Arrow" was introduced by Hughes as an extension, not with one, but with two type parameters. At first, these Arrows may look somewhat arbitrary. Here we show that they are categorically fairly civilised, by showing that they correspond to monoids in suitable subcategories of bifunctors $\mathbb{C}^{\mathrm{op}} \times \mathbb{C} \to \mathbb{C}$. This shows that, at a suitable level of abstraction, arrows are like monads — which are monoids in categories of functors $\mathbb{C} \to \mathbb{C}$.

Freyd categories have been introduced by Power and Robinson to model computational effects, well before Hughes' Arrows appeared. It is often claimed (informally) that Arrows are simply Freyd categories. We shall make this claim precise by showing how monoids in categories of bifunctors exactly correspond to Freyd categories.

Key words: Arrow, Bifunctor, Monad, Monoid, Freyd category.

1 Introduction

The main result of this article can be expressed in one sentence: *Hughes' Arrows are monoids in categories of bifunctors* $\mathbb{C}^{op} \times \mathbb{C} \to \mathbb{C}$. In fact, this entire paper is just an explanation of this one-line summary.

There are several reasons why this result might appeal.

- Arrows have been introduced by Hughes [11,19] in the context of functional programming, to the rather applied end of fitting certain parsers into a general interface (see [10] for more applications). The notion comes without much motivation, and looks somewhat arbitrary. It is then reassuring — and maybe even surprising — that this notion appears as very natural in a completely different (categorical) setting.

[1] Also part-time at Technical University Eindhoven, the Netherlands.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

- Arrows are meant as extensions of Monads — which are well-established in functional programming [18,30]. Indeed, for a monad M, the mapping $(X, Y) \mapsto M(Y)^X$ is a well-known instance of an Arrow. But the intended analogy between Monads and Arrows has not yet been further substantiated.

  It is a standard category theory textbook result [17, VII.3] that a monad is an example of a "monoid in a category", namely in a category $\mathbb{C} \to \mathbb{C}$ of endofunctors on a category $\mathbb{C}$ (see below for more details). Our main observation is that Arrows are also monoids, also in a category of functors, not of the form $\mathbb{C} \to \mathbb{C}$, but $\mathbb{C}^{\mathrm{op}} \times \mathbb{C} \to \mathbb{C}$.
- When we make the minor change of considering monoids in categories of bifunctors $\mathbb{C}^{\mathrm{op}} \times \mathbb{C} \to \mathbf{Sets}$ — with $\mathbf{Sets}$ instead of $\mathbb{C}$ as codomain — we can establish a one-to-one correspondence with Freyd categories from [23]. Hence the monoid description gives a different view on the subject, namely one in which the emphasis lies on the fact that an Arrow is a mapping from both input and output types to a type of computations, with suitable composition operations. The perspective of Freyd categories puts more emphasis on the (slightly tricky) premonoidal aspects involved.

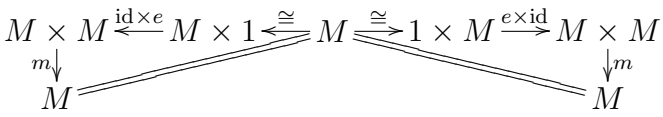
After introducing Arrows in Section 2, their structure is analysed categorically in Section 3. It turns out that the elaboration of the main result does require some work. The most technical part is the construction of suitable monoidal (tensor) infrastructure in categories of functors $\mathbb{C}^{\mathrm{op}} \times \mathbb{C} \to \mathbb{C}$, which is described in Subsection 3.2. This finally gives rise to our model of Arrows as monoids in Section 4.

We start with a gradual introduction of the various notions involved. In doing so we assume a basic level of familiarity with categorical notions and techniques.

Let us first recall that the Monad construct in functional programming languages corresponds directly with a strong monad in category theory [30]. To fix notation, we shortly recall that a monad on a category $\mathbb{C}$ consists of an endofunctor $M : \mathbb{C} \to \mathbb{C}$, a unit natural transformation $\eta\colon 1 \xrightarrow{\bullet} M$, and a multiplication natural transformation $\mu\colon M^2 \xrightarrow{\bullet} M$ satisfying familiar equations [17].

Usually, a *monoid* is described as a set M together with an associative binary operation $m : M \times M \to M$ with a unit $e \in M$. Category theory provides an abstract framework to work in different "universes". A basic illustration of this abstraction is that the notion of monoid can also be formulated in an arbitrary category with suitable structure: for instance a monoid in a category $\mathbb{C}$ with Cartesian products $(\times, 1)$ is an object M of $\mathbb{C}$, together with morphisms $m\colon M \times M \to M$ and $e\colon 1 \to M$ that make certain diagrams commute, corresponding to the monoid equations. The unit equations $m(x, e) = x = m(e, x)$,

for instance, are:



This formulation leads to monoids in universes of for instance topological spaces or dcpo's. The carrier M then has suitable structure, that is preserved by the operations.

Even stronger, we do not even need Cartesian products $(\times, 1)$ for this formulation: monoidal structure (tensors) $(\otimes, I)$ suffices, because we don't need productions or diagonals. Section VII.3 of [17] lists several examples of familiar notions (such as groups and rings) that appear in this way as monoid in a category.

One example is a monad (M, η, μ) on a category $\mathbb{C}$. The category $\mathbb{C} \to \mathbb{C}$ of endofunctors (functors from $\mathbb{C}$ to itself, with natural transformations between them) carries a rather trivial monoidal structure given by functor composition: $F \otimes G = F \circ G$, with identity functor I as obvious unit. Indeed, a monad can be described as a pair of maps $(M \otimes M) \xrightarrow{\mu} M \xleftarrow{\eta} I$ satisfying (precisely) the monoid equations. A forteriori, a strong monad on $\mathbb{C}$ is (precisely) a monoid in the category of strong functors $\mathbb{C} \to \mathbb{C}$, with natural transformations that commute with strength.

Likewise, we will argue that monoids in suitable subcategories of bifunctors $\mathbb{C}^{\mathrm{op}} \times \mathbb{C} \to \mathbb{C}$ model Arrows. The most technical part is the construction of a tensor in this category that has exponentiation as unit. In doing so we shall give a simplification of one of the Arrow operations (namely `first`) introduced by Hughes, see Proposition 3.4.

Arrows have been studied categorically before: [26] use Freyd categories (from [23]) as models of Arrows. We shall elaborate on the relation with our work in Section 5. The occurrence of the name Freyd is also very appropriate in our setting, because we encounter several issues, like dinaturality and small completeness, on which he has worked [8,2,9]. Our view of Arrows as monoids also provides another approach to Freyd categories. Section 5 concludes by proving that both models, monoids and Freyd categories, coincide in the relevant case.

2 Arrows in functional languages

This section introduces Arrows and their use in functional programming languages. We briefly consider monads first, since this construction from category theory historically paved the way for Arrows.

2.1 Monads

A major reason for the initial reluctance in the adoption of functional programming languages is the need to pass state data around explicitly, even through functions that do even not use it. Monadic programming [30] provides an answer to this inconvenience. Through the use of a monad one can encapsulate the changes to the state data, the "side-effects", without explicitly carrying states around. Monads can efficiently structure functional programs while improving genericity. This mechanism is even deemed important enough to be incorporated into Haskell syntax [21]. A monad in Haskell is defined as a type class:

$$\begin{array}{l}\texttt{class Monad M where}\\ \quad \texttt{return} :: X \to \texttt{M}\ X\\ \quad (\texttt{>>=}) :: \texttt{M}\ X \to (X \to \texttt{M}\ Y) \to \texttt{M}\ Y\end{array}$$

To ensure the desired behaviour, the programmer herself should prove certain *monad laws* about the operations `return` and `>>=`. These boil down to the axioms that `M` be a strong monad, in the categorical sense.

In effect, monads are functional combinators. They enable the combination of functions very generally, without many assumptions on the precise functions to combine. However, these restrictions are strict enough to exclude certain classes of libraries from implementation with a monadic interface, most notably efficient parser combinators [27,14].

2.2 Arrows

Arrows [11,20] are even more general functional combinators, and can be seen as a generalisation of monads, as we will see in example 2.2. An Arrow in Haskell is a type constructor class of the form:

$$\begin{array}{l}\texttt{class Arrow A where}\\ \quad \texttt{arr} :: (X \to Y) \to \texttt{A}\ X\ Y\\ \quad (\ggg) :: \texttt{A}\ X\ Y \to \texttt{A}\ Y\ Z \to \texttt{A}\ X\ Z\\ \quad \texttt{first} :: \texttt{A}\ X\ Y \to \texttt{A}\ (X,Z)\ (Y,Z)\end{array}$$

Analogous to monads, an Arrow must furthermore satisfy the following *arrow laws*, the proof of which is up to the programmer:

$$(a \ggg b) \ggg c = a \ggg (b \ggg c), \tag{1}$$
$$\texttt{arr}\ (g \circ f) = \texttt{arr}\ f \ggg \texttt{arr}\ g, \tag{2}$$
$$\texttt{arr}\ \mathrm{id} \ggg a = a = a \ggg \texttt{arr}\ \mathrm{id}, \tag{3}$$
$$\texttt{first}\ a \ggg \texttt{arr}\ \pi_1 = \texttt{arr}\ \pi_1 \ggg a, \tag{4}$$
$$\texttt{first}\ a \ggg \texttt{arr}\ (\mathrm{id} \times f) = \texttt{arr}\ (\mathrm{id} \times f) \ggg \texttt{first}\ a, \tag{5}$$
$$\texttt{first}\ a \ggg \texttt{arr}\ \alpha = \texttt{arr}\ \alpha \ggg \texttt{first}\ (\texttt{first}\ a), \tag{6}$$
$$\texttt{first}\ (\texttt{arr}\ f) = \texttt{arr}\ (f \times \mathrm{id}), \tag{7}$$
$$\texttt{first}\ (a \ggg b) = \texttt{first}\ a \ggg \texttt{first}\ b, \tag{8}$$

In these equations we use:

$$X_1 \xleftarrow{\pi_1} X_1 \times X_2 \xrightarrow{\pi_2} X_2, \qquad X \times (Y \times Z) \xrightarrow{\alpha} (X \times Y) \times Z,$$

for the familiar product maps `fst`, `snd` and `assoc`. Also, `arr` (id $\times$ f) is sometimes written as `second` (`arr` f), where

$$\texttt{second}\ a = \texttt{arr}\ \gamma \ggg \texttt{first}\ a \ggg \texttt{arr}\ \gamma,$$

and $\gamma : X \times Y \xrightarrow{\cong} Y \times X$ is the well-known `swap`-map.

Throughout this article we use a, b, c for arrows, f, g for functions, and X, Y, Z for types, wherever possible.

The arrow laws might appear arbitrary, and [11] does not derive them systematically. However, we shall show they are quite natural from a suitable categorical perspective.

It turns out that the Arrow interface is general enough to allow most known libraries that are incompatible with a monadic interface. Applications can be found on the website [10], for example in the aforementioned parser combinators, in reactive programming [15] and in user interfaces [6].

Example 2.1 (Pure functions) The first example of Arrows that comes to mind is, naturally, ordinary functions. In Haskell this is written as follows.

```
instance Arrow (→) where
  arr f = f
  f ⋙ g = g ∘ f
  first f = f × id
```

One sees at a glance that this satisfies the arrow laws (1)–(8).

To distinguish between normal functions and functions as Arrows, we also call the former *pure functions*.

Example 2.2 (Kleisli Arrows from monads) In category theory, given a monad one can construct the Kleisli category of free algebras. Likewise, given a monad in a functional programming language, we can cast it as an Arrow. This standard example of an Arrow is already present in [11].

```
newtype Kleisli M X Y = K (X → M Y)

instance Monad M ⇒ Arrow (Kleisli M) where
  arr f = K (return ∘ f)
  K f ⋙ K g = K (λX. f X >>= g)
  first (K f) = K (λ(X, Z). f X >>= λY. return (Y, Z))
```

The arrow laws (1)–(8) (for `K`) follow readily from the monad laws (for `M`). Intuitively we can think of a Kleisli Arrow as a computation that allows for monadic behaviour in its codomain (*i.e.* in its output).

Further examples of Arrows will appear in Section 4, in categorical language.

3 Analysing Arrows

In this section we shall formulate several results to make the underlying categorical structure of Arrows in Haskell explicit. We also provide an alternative for the 'first' operations in terms of what we call 'internal strength' (in Proposition 3.4). The main outcome of this section is a reformulation of an Arrow as a monoid.

We fix a category $\mathbb{T}$ with types as objects and terms as morphisms, such as for example the category **Hask** of Haskell types and functions. We assume that $\mathbb{T}$ is Cartesian closed, and carries a binary operation $A(-,-)$ on objects/types that satisfies the arrow laws (1)–(8) for given collections of maps arr, $\ggg$ and first. We shall identify the categorical structure involved in a series of results.

Lemma 3.1 *The operation $A(-,-)$ extends to a functor $\mathbb{T}^{op} \times \mathbb{T} \to \mathbb{T}$, with action on maps $f\colon X' \to X$ and $g\colon Y \to Y'$ given by:*

$$A(f,g) \stackrel{defn}{=} \lambda h.\, arr(f) \ggg h \ggg arr(g) : A(X,Y) \to A(X',Y').$$

This functor is strong in its second argument, and costrong in its first, via

$$\begin{aligned} st_2 &= \lambda(z,h).\, arr(\lambda x.\, \langle z, x\rangle) \ggg second(h) \\ &\qquad : Z \times A(X,Y) \to A(X, Z \times Y), \\ cost_2 &= \lambda(z,h).\, arr(\lambda f.\, f(z)) \ggg h \\ &\qquad : Z \times A(X,Y) \to A(X^Z, Y). \end{aligned}$$

Proof. It is easy to check that A preserves identities and composition, and also that the strength and costrength maps are natural. These maps furthermore satisfy the following equations, which will come in useful:

$$\begin{aligned} A(\mathrm{id}, \pi_2) \circ \mathrm{st}_2 &= \pi_2, \\ A(\mathrm{id}, \alpha) \circ \mathrm{st}_2 \circ (\mathrm{id} \times \mathrm{st}_2) &= \mathrm{st}_2 \circ \alpha, \\ \mathrm{cost}_2 \circ \lambda h.\, \langle z, h\rangle &= A(\lambda f.\, f(z), \mathrm{id}), \\ A(\beta, \mathrm{id}) \circ \mathrm{cost}_2 \circ (\mathrm{id} \times \mathrm{cost}_2) &= \mathrm{cost}_2 \circ \alpha, \end{aligned}$$

where $\beta\colon X^{(Y \times Z)} \xrightarrow{\cong} (X^Y)^Z$ is the familiar canonical isomorphism. □

Notice that we could just as well have used $\mathrm{st}_1 = A(\mathrm{id}, \gamma) \circ \mathrm{st}_2 \circ \gamma : A(X,Y) \times Z \to A(X, Y \times Z)$, which then satisfies similar equations, like $\pi_1 = \mathrm{st}_1 \circ A(\mathrm{id}, \pi_1)$.

Lemma 3.2 *The maps arr: $Y^X \to A(X,Y)$ form a natural transformation $(+)^{(-)} \xrightarrow{\bullet} A(-,+)$ from exponents to arrows.*

Similarly, the maps first: $A(X,Y) \to A(X\times Z, Y\times Z)$ are natural in X,Y. This may be formulated as: first yields a natural transformation $\langle first\rangle$ from A to the functor $A\times$ given by $(X,Y)\mapsto \prod_Z A(X\times Z, Y\times Z)$.

Proof. This follows from easy calculations: for maps $f\colon X'\to X$, $g\colon Y\to Y'$ in $\mathbb{T}$ and $h\colon Y^X$ we have:

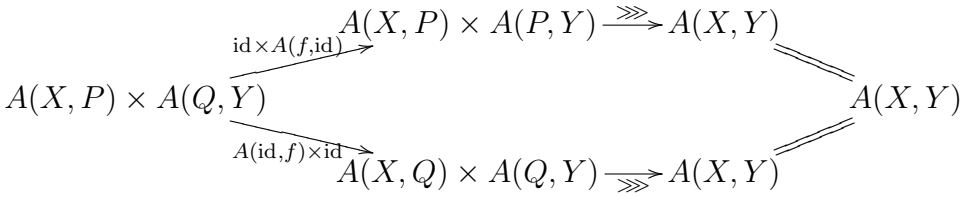
$$\begin{aligned}\big(A(f,g)\circ \mathrm{arr}\big)(h) &= \mathrm{arr}(f) \ggg \mathrm{arr}(h) \ggg \mathrm{arr}(g)\\ &\overset{(2)}{=} \mathrm{arr}(g\circ h\circ f)\\ &= \mathrm{arr}(g^f(h))\\ &= \big(\mathrm{arr}\circ g^f\big)(h),\end{aligned}$$

and

$$\begin{aligned}\big(A\times(f,g)\circ\langle\mathrm{first}\rangle\big)(h) &= \langle A(f\times\mathrm{id}, g\times\mathrm{id})\circ\pi_Z\rangle_Z(\langle\mathrm{first}(h)\rangle)\\ &= \langle A(f\times\mathrm{id}, g\times\mathrm{id})(\mathrm{first}(h))\rangle\\ &= \langle \mathrm{arr}(f\times\mathrm{id}) \ggg \mathrm{first}(h) \ggg \mathrm{arr}(g\times\mathrm{id})\rangle\\ &\overset{(7)}{=} \langle \mathrm{first}(\mathrm{arr}(f)) \ggg \mathrm{first}(h) \ggg \mathrm{first}(\mathrm{arr}(g))\rangle\\ &\overset{(8)}{=} \langle \mathrm{first}(\mathrm{arr}(f) \ggg h \ggg \mathrm{arr}(g))\rangle\\ &= \langle \mathrm{first}(A(f,g)(h))\rangle\\ &= \big(\langle\mathrm{first}\rangle\circ A(f,g)\big)(h).\end{aligned}$$

□

Lemma 3.3 *The maps $\ggg\colon A(X,P)\times A(P,Y)\to A(X,Y)$ are natural in X,Y, and dinatural in P. The latter means (see [17, IX.4] or [2]) that for each map $f\colon P\to Q$ the following diagram commutes.*



Proof. We shall only do dinaturality: for $a : A(X,P)$ and $b : A(Q,Y)$,

$$\begin{aligned}\big(\ggg\circ\ \mathrm{id}\times A(f,\mathrm{id})\big)(a,b) &= a \ggg A(f,\mathrm{id})(b)\\ &= a \ggg \mathrm{arr}(f) \ggg b\\ &= A(\mathrm{id},f)(a) \ggg b\\ &= \big(\ggg\circ\ A(\mathrm{id},f)\times\mathrm{id}\big)(a,b).\end{aligned}$$

□

Intuitively, dinaturality in P signifies that this middle parameter 'is not really important'; it could just have well been another one, as long as it is the

same across the second argument of the first factor, and the first argument of the second. This suggests that $\ggg$ should really be seen as an operation from a tensor product. This will be elaborated in Section 3.2.

3.1 The operation first

We now prove that the maps 'first' satisfying (4)-(8) can be simplified into maps called 'ist', for *internal strength*.

Proposition 3.4 *The maps first*: $A(X,Y) \to A(X \times Z, Y \times Z)$ *satisfying equations* (4)–(8) *correspond to "internal strength" maps ist*: $A(X,Y) \to A(X, Y \times X)$ *which are natural in* Y *and dinatural in* X*, and satisfy*

$$ist(arr(f)) = arr(\langle f, \mathrm{id}\rangle), \tag{9}$$
$$ist(a) \ggg arr(\pi_1) = a, \tag{10}$$
$$ist(a \ggg b) = ist(a) \ggg ist(arr(\pi_1) \ggg b) \ggg arr(\mathrm{id} \times \pi_2), \tag{11}$$
$$ist(ist(a)) = ist(a) \ggg arr(\langle \mathrm{id}, \pi_2\rangle). \tag{12}$$

The alternative formulation in terms of internal strength 'ist' in this result is convenient because it has only two parameters – instead of three for 'first' – and its (di)naturality is clearly described. The proof of the equivalence of 'first' and 'ist' involves many basic calculations, of which we only present a few exemplaric cases.

Proof. Given the maps 'first' satisfying (4)–(8), we define internal strength on $a : A(X,Y)$ as:

$$\mathrm{ist}(a) = \mathrm{arr}(\Delta) \ggg \mathrm{first}(a),$$

where $\Delta = \langle \mathrm{id}, \mathrm{id}\rangle$. One then checks naturality in Y, dinaturality in X, and (9)–(12). The (di)naturality equations can be formulated as:

$$\mathrm{ist}(a) \ggg \mathrm{arr}(g \times \mathrm{id}) = \mathrm{ist}(a \ggg \mathrm{arr}(g)) \tag{13}$$
$$\mathrm{arr}(f) \ggg \mathrm{ist}(a) = \mathrm{ist}(\mathrm{arr}(f) \ggg a) \ggg \mathrm{arr}(\mathrm{id} \times f). \tag{14}$$

As illustration we check equation (10):

$$\begin{aligned}
\mathrm{ist}(a) \ggg \mathrm{arr}(\pi_1) &= \mathrm{arr}(\Delta) \ggg \mathrm{first}(a) \ggg \mathrm{arr}(\pi_1) \\
&\overset{(4)}{=} \mathrm{arr}(\Delta) \ggg \mathrm{arr}(\pi_1) \ggg a \\
&\overset{(2)}{=} \mathrm{arr}(\pi_1 \circ \Delta) \ggg a \\
&= \mathrm{arr}(\mathrm{id}) \ggg a \\
&\overset{(3)}{=} a.
\end{aligned}$$

Conversely, given internal strength 'ist' satisfying (9)–(12), we define:

$$\mathrm{first}(a) = \mathrm{ist}(\mathrm{arr}(\pi_1) \ggg a) \ggg \mathrm{arr}(\mathrm{id} \times \pi_2),$$

where $\pi_1 \colon X \times Z \to X$ and $\mathrm{id} \times \pi_2 \colon Y \times (X \times Z) \to Y \times Z$. This yields a natural operation, in the sense that:

$$\mathrm{arr}(f \times \mathrm{id}) \ggg \mathrm{first}(a) \ggg \mathrm{arr}(g \times \mathrm{id}) = \mathrm{first}(f \ggg a \ggg g).$$

We shall prove equation (8) in detail, and leave the rest to the interested reader.

$$\begin{aligned}
&\mathrm{first}\ a \ggg \mathrm{first}\ b \\
&\quad = \mathrm{ist}(\mathrm{arr}(\pi_1) \ggg a) \ggg \mathrm{arr}(\mathrm{id} \times \pi_2) \ggg \mathrm{ist}(\mathrm{arr}(\pi_1) \ggg b) \\
&\qquad\quad \ggg \mathrm{arr}(\mathrm{id} \times \pi_2) \\
&\overset{\text{(dinat)}}{=} \mathrm{ist}(\mathrm{arr}(\pi_1) \ggg a) \ggg \mathrm{ist}(\mathrm{arr}(\mathrm{id} \times \pi_2) \ggg \mathrm{arr}(\pi_1) \ggg b) \\
&\qquad\quad \ggg \mathrm{arr}(\mathrm{id} \times (\mathrm{id} \times \pi_2)) \ggg \mathrm{arr}(\mathrm{id} \times \pi_2) \\
&\overset{(2)}{=} \mathrm{ist}(\mathrm{arr}(\pi_1) \ggg a) \ggg \mathrm{ist}(\mathrm{arr}(\pi_1) \ggg b) \\
&\qquad\quad \ggg \mathrm{arr}(\mathrm{id} \times \pi_2) \ggg \mathrm{arr}(\mathrm{id} \times \pi_2) \\
&\overset{(11)}{=} \mathrm{ist}(\mathrm{arr}(\pi_1) \ggg a \ggg b) \ggg \mathrm{arr}(\mathrm{id} \times \pi_2) \\
&\quad = \mathrm{first}\ (a \ggg b). \qquad \square
\end{aligned}$$

3.2 Monoidal structure

Recall from the introduction that describing an Arrow $(A, \mathrm{arr}, \ggg)$ as a monoid in a category of bifunctors $\mathbb{C}^{\mathrm{op}} \times \mathbb{C} \to \mathbb{C}$ requires a monoidal structure $(\otimes, I)$ on such bifunctors, so that the Arrow A appears as monoid of the form:

$$A \otimes A \xrightarrow{\ggg} A \xleftarrow{\mathrm{arr}} I$$

The naturality $\mathrm{arr} \colon (+)^{(-)} \xrightarrow{\bullet} A$ observed in lemma 3.2 suggests to take exponentiation as unit $I = (+)^{(-)}$ of the intended monoidal structure. The big question then is: what is $\otimes$?

We now sketch the main idea, still in the category $\mathbb{T}$ of types and terms. Composition $\ggg$ is given by a collection of maps:

$$A(X, P) \times A(P, Y) \xrightarrow{\ggg} A(X, Y)$$

which can be combined by a coproduct on the left-hand-side:

$$\left(\coprod_{P \in \mathbb{T}} A(X, P) \times A(P, Y) \right) \xrightarrow{\ggg} A(X, Y)$$

This is a coproduct over all objects/types in $\mathbb{T}$, and thus size aspects become relevant. But such coproducts (or sums) can be taken in polymorphic type theories, so for the time being we simply proceed and postpone issues of size until the next section.

Once we have this (big) coproduct, it becomes "natural" to take the dinaturality observed in lemma 3.3 into account. This is done by using a suitable coequaliser c to form the required tensor $\otimes$ as in:

$$\left(\coprod_{P_1,P_2\in\mathbb{T}} A(X,P_1)\times P_2^{P_1}\times A(P_2,Y)\right) \underset{d_2}{\overset{d_1}{\rightrightarrows}} \left(\coprod_{P\in\mathbb{T}} A(X,P)\times A(P,Y)\right) \longrightarrow A(X,Y)$$
$$\left(\coprod_{P\in\mathbb{T}} A(X,P)\times A(P,Y)\right) \xrightarrow{c} (A\otimes A)(X,Y) \overset{\ggg}{\longrightarrow} A(X,Y)$$

The composition map $\ggg$ then appears as a map $(A\otimes A)\to A$ by construction. The two maps d_1, d_2 capture dinaturality via the composites:

$$\begin{aligned} A(X,P_1)\times P_2^{P_1} &\xrightarrow{\mathrm{st}_1} A(X,P_1\times P_2^{P_1}) \xrightarrow{A(\mathrm{id},\mathrm{ev}\circ\gamma)} A(X,P_2),\\ P_2^{P_1}\times A(P_2,Y) &\xrightarrow{\mathrm{cost}_2} A(P_2^{(P_2^{P_1})},Y) \xrightarrow{A(\Lambda(\mathrm{ev}\circ\gamma),\mathrm{id})} A(P_1,Y). \end{aligned}$$

in which strength and costrength play a crucial role.

Let us make sure that exponentiation $I=(-)^{(-)}$ is indeed a unit for this tensor. We concentrate on the required isomorphism $\rho\colon A\xrightarrow{\cong} A\otimes I$. It is obtained for each pair X,Y of objects as a composite

$$A(X,Y)\xrightarrow{\lambda a.\,\kappa_Y(a,\mathrm{id}_Y)} \left(\coprod_{P\in\mathbb{T}} A(X,P)\times Y^P\right)\xrightarrow{c}(A\otimes I)(X,Y),$$

where κ_Y is the Y-injection into the coproduct. The inverse of ρ is obtained in:

$$\bullet \underset{d_2}{\overset{d_1}{\rightrightarrows}} \left(\coprod_{P\in\mathbb{T}} A(X,P)\times Y^P\right) \xrightarrow{c} (A\otimes I)(X,Y)$$
$$\left(\coprod_{P\in\mathbb{T}} A(X,P)\times Y^P\right) \xrightarrow{e} A(X,Y) \overset{\rho^{-1}}{\longrightarrow} (A\otimes I)(X,Y)$$

where e is the cotuple of maps $A(X,P)\times Y^P\to A(X,P\times Y^P)\to A(X,Y)$ given as $A(\mathrm{id},\mathrm{ev}\circ\gamma)\circ\mathrm{st}_1$. One can then check that ρ and ρ^{-1} are indeed each other's inverses.

Finally, we have to check that the monoid equations hold for the span $(A\otimes A)\xrightarrow{\ggg} A\xleftarrow{\mathrm{arr}} I$. We shall do one of the equations, namely

$$A\xrightarrow{\rho} A\otimes I\xrightarrow{\mathrm{id}\otimes\mathrm{arr}} A\otimes A \overset{\ggg}{\longrightarrow} A,$$

which for $a\colon A(X,Y)$ becomes

$$a \longmapsto (a, \mathrm{id}) \longmapsto (a, \mathrm{arr}(\mathrm{id})) \downarrow a \ggg \mathrm{arr}(\mathrm{id}).$$

Hence commutation of this diagram amounts to arrow law (3), which states $a \ggg \mathrm{arr}(\mathrm{id}) = a$.

Likewise, the associativity isomorphism $\alpha : (A \otimes B) \otimes C \to A \otimes (B \otimes C)$ is given by associativity $(A \times B) \times C \to A \times (B \times C)$ via the coproduct. The required coherence properties, like $\alpha \circ \alpha = (\mathrm{id} \otimes \alpha) \circ \alpha \circ (\alpha \otimes \mathrm{id})$, are proven by using the arrow laws, notably (6). Strength, costrength en internal strength on $A \otimes A$ are inherited from A, again via the coproduct and the equaliser.

Summarising, in a type theoretic setting we have sketched that an Arrow is the same thing as a monoid in the category of bifunctors $\mathbb{T}^{\mathrm{op}} \times \mathbb{T} \to \mathbb{T}$ with strength, costrength, and internal strength.

4 Arrows, categorically

In the previous section we have reformulated an Arrow A in Haskell as a monoid $A \otimes A \to A \leftarrow I$ in a category of bifunctors acting on types and terms. The most complicated ingredient was the tensor product $\otimes$. It was constructed like the tensor product of profunctors (or distributors) generalising relation composition [16], see also [4].

One other complication involves size. The big coproduct in the previous section uses all objects of the category $\mathbb{T}$ as indices. This requires $\mathbb{T}$ to be both small and (co)complete. At this stage it is important to recall the basic result of Freyd [8, Chapter 3, Exercise D] that there are no categories that are both small and complete, except preorders (see also [13, 8.3.2]). However, small complete *internal* categories do exist [12], and can indeed be used as models for polymorphic type theory. Working in such a universe is very similar to working in a polymorphic type theory as we have done in the previous section.

Thus it becomes natural to consider the more general setting of bifunctors $\mathbb{C}^{\mathrm{op}} \times \mathbb{C} \to \mathbb{B}$, where $\mathbb{C}$ is $\mathbb{B}$-enriched, so that size issues are separated. So far we have considered $\mathbb{B} = \mathbb{C}$, for $\mathbb{C}$ Cartesian closed (and thus enriched over itself). Another obvious case is to take $\mathbb{B} = \mathbf{Sets}$ and $\mathbb{C}$ small. The situation then reduces to profunctors, the monoidal structure of which is well-known, see [4, Section 7.8] or [22, Section 6.5]. In the remainder of this paper we shall focus on this (profunctor) situation, and take the following as categorical formalisation of the notion of Arrow, as originally formulated [11] in the language of Haskell.

Definition 4.1 Let $\mathbb{C}$ be a small category. An Arrow over $\mathbb{C}$ is a monoid in the category of profunctors $\mathbb{C}^{\mathrm{op}} \times \mathbb{C} \to \mathbf{Sets}$, with its usual monoidal structure, that carry an internal strength.

The restriction to *small* categories is maybe a bit too strong. It is needed to construct the usual tensor $\otimes$, via a big coproduct over objects of $\mathbb{C}$ in the category **Sets**, like in the previous section. We can also formulate the composition operation $\ggg$ of an Arrow A via collections of maps $A(X,P) \times A(P,Y) \to A(X,Y)$ — which are natural in X, Y and dinatural in P — and satisfy the equations (1)–(8). In this manner we can relax the restriction to *locally small* categories.

In the remainder of this section we shall sketch how to understand the earlier examples 2.1 and 2.2 in this setting, and extend them a bit. We therefor fix a locally small category $\mathbb{C}$.

The Hom-bifunctor $\mathbb{C}^{\mathrm{op}} \times \mathbb{C} \to \mathbf{Sets}$ given by $(X,Y) \mapsto \mathrm{Hom}_{\mathbb{C}}(X,Y)$ is perhaps the most obvious example of an Arrow — like the pure functions of example 2.1.

If $M \colon \mathbb{C} \to \mathbb{C}$ is a strong monad, then the mapping $(X,Y) \mapsto \mathrm{Hom}_{\mathbb{C}}(X, MY)$ is also an Arrow — namely the Kleisli Arrow as described in example 2.2.

That Kleisli Arrows have monadic behaviour in their codomain leads us to consider Arrows behaving monadically in their domain. In functional language: Kleisli Arrows take care of bookkeeping in the output, so why not build Arrows that take care of bookkeeping in the input [29,28]? Since an Arrow $\mathbb{C}^{\mathrm{op}} \times \mathbb{C} \to \mathbf{Sets}$ is contravariant in its 'domain', we must start from a monad on $\mathbb{C}^{\mathrm{op}}$. That is, if N is a comonad, then we obtain an Arrow via $(X,Y) \mapsto \mathrm{Hom}_{\mathbb{C}}(NX, Y)$.

Even more generally, as in [29], [24] or even [5], we can try to build Arrows behaving monadically in both their domain and codomain at once. Given a strong monad M, a comonad N, and a distributive law $\lambda \colon NM \overset{\bullet}{\to} MN$ between them we obtain an Arrow A via $A(X,Y) = \mathrm{Hom}_{\mathbb{C}}(NX, MY)$. The unit of the monoid A is readily defined using the (co)units of M and N, and the internal strength $\mathrm{ist} : \mathrm{Hom}_{\mathbb{C}}(NX, MY) \to \mathrm{Hom}_{\mathbb{C}}(NX, M(Y \times X))$ is easily given by $\mathrm{ist}(f) = \mathrm{st}_2^M \circ \langle \varepsilon, f \rangle$, where ε is the counit of N. The distributive law is needed to give the multiplication/composition $a \ggg b$ of $a \in \mathrm{Hom}_{\mathbb{C}}(NX, MY)$ and $b \in \mathrm{Hom}_{\mathbb{C}}(NY, MZ)$:

$$NX \xrightarrow{\delta} N^2X \xrightarrow{Na} NMY \xrightarrow{\lambda} MNY \xrightarrow{Mb} M^2Z \xrightarrow{\mu} MZ.$$

5 Arrows as Freyd categories

One could question our approach to Arrows in the previous sections because of the oft-heard statement "Arrows are Freyd categories" [19]: Freyd categories are claimed to already provide categorical semantics for Arrows. However, this claim has always remained very informal. We think our approach is closer to the original functional intuitions underlying Arrows in Haskell, because:

- the description of Arrows as bifunctors emphasises that they are binary operations on types;

- the precise formulation of the premonoidal structure of Freyd categories is rather delicate, and distracts from the essence of the structure of Arrows: all this structure corresponds only to 'first'.

But more interestingly, our alternative formalisation of Arrows as monoids makes it possible to give precise mathematical meaning to the statement "Arrows are Freyd categories". That is the aim of the current section. We shall compare this result with the situation for monads [2].

Let us first recall what a Freyd category is. Therefor we need the notion of a premonoidal category, which we intuitively think of as a monoidal category in which the tensor need not be a bifunctor, though it is functorial in each variable separately.

Definition 5.1 A *binoidal category* is a category $\mathbb{D}$, with for every object X two functors $(-) \ltimes X : \mathbb{D} \to \mathbb{D}$ and $X \rtimes (-) : \mathbb{D} \to \mathbb{D}$ such that $X \ltimes Y = X \rtimes Y$. Hence we write $X \boxtimes Y = X \ltimes Y = X \rtimes Y$. A morphism f is called *central* if for each g, both:

- $(f \ltimes \mathrm{id}) \circ (\mathrm{id} \rtimes g) = (\mathrm{id} \rtimes g) \circ (f \ltimes \mathrm{id})$, and
- $(\mathrm{id} \rtimes f) \circ (g \ltimes \mathrm{id}) = (g \ltimes \mathrm{id}) \circ (\mathrm{id} \rtimes f)$.

For such a central f it makes sense to write $f \boxtimes g$ or $g \boxtimes f$ for these composites.

Definition 5.2 A *symmetric premonoidal category* is a binoidal category $\mathbb{D}$ together with an object $I \in \mathbb{D}$ and natural isomorphisms with central components $\alpha : (X \boxtimes Y) \boxtimes Z \to X \boxtimes (Y \boxtimes Z)$, $\lambda : I \boxtimes X \to X$, $\rho : X \boxtimes I \to X$ and $\gamma : X \boxtimes Y \to Y \boxtimes X$ that obey the familiar coherence properties for monoidal categories.

The non-bifunctoriality reflects the order of side-effects when we think of $\mathbb{D}$ as a category of 'computations'. When we include a category $\mathbb{C}$ of 'values', we arrive at the notion of a Freyd category [23,25,26].

Definition 5.3 A *Freyd category* consists of a symmetric premonoidal category $\mathbb{D}$ together with a category $\mathbb{C}$ with finite products, and an identity-on-objects functor $J : \mathbb{C} \to \mathbb{D}$ that preserves all structure: $J(X \times Y) = X \boxtimes Y$, $J(\alpha) = \alpha$, $J(\lambda) = \lambda$, etc. A Freyd category $\mathbb{C} \to \mathbb{D}$ is called (locally) small if the category $\mathbb{D}$ is (locally) small.

The comparison between monoids and Freyd categories begins with the well-known (and easily seen) fact that the following are in one-to-one correspondence:

- Monoids in the category of functors $\mathbb{C} \to \mathbb{C}$,
- Monads M on $\mathbb{C}$, and
- Identity-on-objects functors $J : \mathbb{C} \to \mathbb{D}$ that have a right adjoint,

[2] As suggested by John Power.

The functor J arises from M by the Kleisli construction, while J gives a monad M induced by the adjunction.

This can be generalised to the following equivalence.

Theorem 5.4 *For a locally small category $\mathbb{C}$ with finite products, there is a one-to-one correspondence between*

(i) *Arrows A over $\mathbb{C}$, that is (cf. definition 4.1), monoids A in the category of profunctors $\mathbb{C}^{op} \times \mathbb{C} \to$ **Set** that are internally strong.*

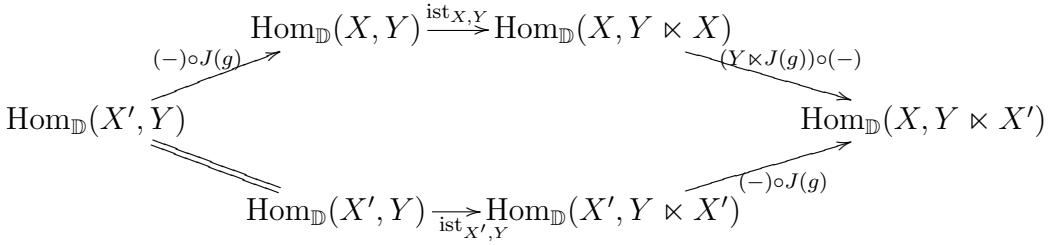
(ii) *Locally small Freyd categories $\mathbb{C} \to \mathbb{D}$.*

Proof. Suppose we are given a monoid $A : \mathbb{C}^{\mathrm{op}} \times \mathbb{C} \to \mathbf{Set}$ with $\ggg$, arr, and ist, satisfying the by now familiar properties of an Arrow. Put $\mathbb{D} = \mathbb{C}_A$, the "Kleisli category" of A,[3] with objects $X \in \mathbb{C}$ and $a \in A(X, Y)$ as morphisms $a : X \to Y$. Then $\mathbb{D}$ is symmetric premonoidal by defining $I = 1 \in \mathbb{D}$ and $X \boxtimes Y = X \times Y$. The premonoidal tensor $\boxtimes$ extends to a functor (on morphisms) by virtue of the provided ist (or equivalently first, see proposition 3.4, and second), since every morphism $a \in A(X, Y)$ yields $a \boxtimes Z = \mathrm{first}_Z(a) : X \boxtimes Z \to Y \boxtimes Z$ and $Z \boxtimes a = \mathrm{second}_Z(a) : Z \boxtimes X \to Z \boxtimes Y$. The implication (i) $\Rightarrow$ (ii) is completed by defining $J : \mathbb{C} \to \mathbb{D}$ to act as the identity on objects, and as arr on morphisms.

Conversely, suppose given a Freyd category $J : \mathbb{C} \to \mathbb{D}$. We then define $A : \mathbb{C}^{\mathrm{op}} \times \mathbb{C} \to \mathbf{Set}$ by $A(X, Y) = \mathrm{Hom}_{\mathbb{D}}(X, Y)$. This A is made into a monoid in the category of profunctors $\mathbb{C}^{\mathrm{op}} \times \mathbb{C} \to \mathbf{Set}$ by taking the unit $\mathrm{arr} = J : \mathrm{Hom}_{\mathbb{C}}(X, Y) \to \mathrm{Hom}_{\mathbb{D}}(X, Y)$, and taking as multiplication $\ggg$ the composition $A(X, P) \times A(P, Y) \to A(X, Y)$ in $\mathbb{D}$. Furthermore we can define

$$\mathrm{ist}_{X,Y} : A(X, Y) \to A(X, Y \times X) \quad \text{by} \quad \mathrm{ist}_{X,Y}(f) = (f \ltimes X) \circ J(\langle \mathrm{id}, \mathrm{id} \rangle).$$

Naturality of ist in Y is obvious, dinaturality in X boils down to the fact that the diagram



commutes for every morphism $g : X \to X'$ in $\mathbb{C}$. The crux here is that it need only commute for morphisms g of $\mathbb{C}$, *i.e.* morphisms of $\mathbb{D}$ of the form $J(g)$ (cf. equations (13) and (14)), which are central. Since one also readily checks (9)–(12), we see that (ii) implies (i). □

Another, similar, equivalence that comes to mind is that of

[3] The reason for calling $\mathbb{C}_A$ the Kleisli category will be presented elsewhere.

- Monoids in the category of strong functors $\mathbb{C} \to \mathbb{C}$ (and natural transformations commuting with strength),
- Strong monads on $\mathbb{C}$, and
- Freyd categories $\mathbb{C} \to \mathbb{D}$ with a right adjoint.

However, there seems to be no meaningful 'Arrow-analogue' of this correspondence, because Arrows are automatically (co)strong (cf. lemma 3.1).

Conclusion and future work

This paper contains two reformulations of the notion of Arrow, introduced in the context of functional programming by Hughes: one minor and one major. The minor reformulation concerns an easier alternative to the 'first' operation. The major reformulation is the description of an Arrow as a monoid in a category of bifunctors.

We have used the latter reformulation to justify the informal claim that Arrows are Freyd categories. On **Sets** the two semantics, Freyd categories and internally strong monoids, coincide. However, the monoid approach generalises in a different direction than the Freyd category approach.

A (minor) topic left open is how the monoidal structure on the category of bifunctors $\mathbb{C}^{\mathrm{op}} \times \mathbb{C} \to \mathbb{C}$ relates to the monoidal structure on the category of functors $\mathbb{C} \to \mathbb{C}$.

In the end the question comes up: does this help functional programmers in any way? At this stage we have no such claim. But one does notice several variations and extensions of Arrows appearing, such as Biarrows [1], or a need for recursion schemes, and thus the need for a foundation, of monads/Arrows [7,3,19]. Our categorical reformulation as monoids might give guidance for the proper formulation of such variations.

Acknowledgement

We are indebted to the anonymous reviewers for their comments and suggestions, and to John Power for pushing us to make the connection between monoids and Freyd categories explicit.

References

[1] Alimarine, A., S. Smetsers, A. van Weelden, M. van Eekelen and R. Plasmeijer, *There and back again: arrows for invertible programming*, in: *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell* (2005), pp. 86–97, (ISBN 1-59593-071-X).

[2] Bainbridge, E., P. Freyd, A. Scedrov and P. Scott, *Functorial polymorphism*, Theoretical Computer Science **70(1)** (1990), pp. 35–64, corrigendum in *Theor.*

Comp. Sci. 71(3):431, 1990.

[3] Benton, N. and M. Hyland, *Traced premonoidal categories*, Theoretical Informatics and Applications **37** (2003), pp. 273–299.

[4] Borceux, F., "Handbook of Categorical Algebra," Encyclopedia of Mathematics and its applications **1**, Cambridge University Press, 1994.

[5] Cockett, J., J. Koslowski, R. Seely and R. Wood, *Modules*, Theory and Applications of Categories **11** (2003), pp. 375–396.

[6] Courtney, A., "Modeling User Interfaces in a Functional Language," Ph.D. thesis, Yale University (2004).

[7] Erkök, L. and J. Launchbury, *A recursive do for haskell*, in: *Haskell Workshop*, 2002, pp. 29–37.

[8] Freyd, P., "Abelian Categories: An Introduction to the Theory of Functor," Harper and Row, New York, 1964.

[9] Freyd, P., *Structural polymorphism*, Theoretical Computer Science **115** (1993), pp. 107–129.

[10] Haskell.org, http://www.haskell.org/arrows.

[11] Hughes, J., *Generalising monads to arrows*, Science of Computer Programming **37** (2000), pp. 67–111.

[12] Hyland, J., *A small complete category*, Annals of Pure & Applied Logic **40** (1988), pp. 135–165.

[13] Jacobs, B., "Categorical Logic and Type Theory," North Holland, Amsterdam, 1999.

[14] Jansson, P. and J. Jeuring, *Polytypic data conversion programs*, Science of Computer Programming **43** (2002), pp. 35–75.

[15] Jeuring, J. and S. P. Jones, editors, "Arrows, Robots and Functional Reactive Programming," Lect. Notes Comp. Sci. **2638**, Springer, 2003.

[16] Lawvere, F., *Metric spaces, generalized logic, and closed categories*, Seminario Matematico e Fisico. Rendiconti di Milano **43** (1973), pp. 135–166, reprint available via `www.tac.mta.ca/tac/reprints`.

[17] Mac Lane, S., "Categories for the Working Mathematician," Springer, 1971.

[18] Moggi, E., *Computational lambda-calculus and monads*, Logic in Computer Science (1989).

[19] Paterson, R., *A new notation for arrows*, in: *International Conference on Functional Programming* (2001), pp. 229–240.

[20] Paterson, R., *Arrows and computation*, in: J. Gibbons and O. de Moor, editors, *The Fun of Programming*, Palgrave, 2003 pp. 201–222, http://www.soi.city.ac.uk/~ross/papers/fop.html.

[21] Peyton Jones, S. and J. Hughes, *Report on the programming language haskell 98, a non-strict purely functional programming language*, Technical report, `www.haskell.org/onlinereport` (1999).

[22] Poigne, A., *Basic category theory*, in: S. Abramsky, D. M. Gabbay and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science: Background - Mathematical Structures (Volume 1)*, Clarendon Press, Oxford, 1992 pp. 413–640.

[23] Power, J. and E. Robinson, *Premonoidal categories and notions of computation*, Mathematical Structures in Computer Science **7** (1997).

[24] Power, J. and E. Robinson, *Modularity and dyads*, Mathematical Foundations of Programming Semantics XV **20**, 1999, pp. 467–480.

[25] Power, J. and H. Thielecke, *Environments, continuation semantics and indexed categories*, in: M. Abadi and T. Ito, editors, *Theoretical Aspects of Computer Software*, number 1281 in Lect. Notes Comp. Sci. (1997), pp. 391–414.

[26] Power, J. and H. Thielecke, *Closed Freyd- and κ-categories*, in: J. Wiedermann, P. van Emde Boas and M. Nielsen, editors, *International Colloquium on Automata, Languages and Programming*, number 1644 in Lect. Notes Comp. Sci. (1999), pp. 625–634.

[27] Swierstra, S. and L. Duponcheel, *Deterministic, error-correcting combinator parsers*, in: J. Launchbury, E. Meijer and T. Sheard, editors, *Advanced Functional Programming*, Lect. Notes Comp. Sci. **1129** (1996), pp. 184–207.

[28] Uustalu, T. and V. Vene, *Signals and comonads*, Journ. of Universal Comp. Sci. **11(7)** (2005), pp. 1310–1326.

[29] Uustalu, T. and V. Vene, *The essence of dataflow programming*, Revised Lectures from Central European Functional Programming School (Budapest, 2005), to appear, http://www.cs.ioc.ee/~tarmo/papers/.

[30] Wadler, P., *Monads for functional programming*, in: *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School* (1993).