MSFP 2008

Idioms are oblivious, arrows are meticulous, monads are promiscuous

Sam Lindley, Philip Wadler and Jeremy Yallop

Laboratory for Foundations of Computer Science
The University of Edinburgh

Abstract

We revisit the connection between three notions of computation: Moggi's *monads*, Hughes's *arrows* and McBride and Paterson's *idioms* (also called *applicative functors*). We show that idioms are equivalent to arrows that satisfy the type isomorphism $A \rightsquigarrow B \simeq 1 \rightsquigarrow (A \to B)$ and that monads are equivalent to arrows that satisfy the type isomorphism $A \rightsquigarrow B \simeq A \to (1 \rightsquigarrow B)$. Further, idioms embed into arrows and arrows embed into monads.

Keywords: applicative functors, idioms, arrows, monads

$$\begin{tikzcd}
\text{idioms} \arrow[d, bend left] & & \text{monads} \arrow[d, bend left] \\
\begin{array}{c}\text{static arrows}\\ (A \rightsquigarrow B \simeq 1 \rightsquigarrow (A \to B))\end{array} \arrow[u, bend left] \arrow[r, hook] & \text{arrows} \arrow[r, hook] & \begin{array}{c}\text{higher-order arrows}\\ (A \rightsquigarrow B \simeq A \to (1 \rightsquigarrow B))\end{array} \arrow[u, bend left]
\end{tikzcd}$$

Fig. 1. Idioms, arrows and monads

1 Introduction

Assumptions and guarantees

The Internet Robustness Principle states [10]

Be conservative in what you do; be liberal in what you accept from others.

In other words, robust systems make the weakest possible assumptions about input and give the strongest possible guarantees about output. Programs that accept only integers are less flexible than programs that accept all kinds of number. Contrariwise, programs that may output any kind of number are less flexible than programs that are guaranteed to output only integers.

To follow the principle we need to know which sets of values generalise which other sets. While there are certainly more numbers than integers, the ordering is not so obvious at higher-order types, such as function and computation types. Can a program that manipulates *arrow* computations be made more flexible by specifying that the input must be an *idiom* rather than an *arrow*? Can a library that exposes an *idiom* instance be made more flexible by exposing an *arrow* instance instead? In his original work on arrows [1], Hughes shows how each monad gives rise to an arrow, and gives an extended arrow interface, *ArrowApp*, that is equivalent to the monad interface. In their later work introducing idioms (also called applicative functors) [5], McBride and Paterson show how to obtain an idiom from either a monad or an arrow, and how to combine an idiom and an arrow to yield another arrow. However, the precise relationship between the three notions of computation has remained obscure. In particular, McBride and Paterson informally describe idioms as

> an abstract notion of effectful computation lying between Arrow and Monad in strength

which we show in the following pages to be mistaken: idioms are, in fact, weaker than both arrows and monads. The diagram in Figure 1 gives a high-level view of the situation: idioms correspond to a language which may be extended to obtain arrows; a further extension yields a language corresponding to monads.

The main contributions of this paper are:

(i) A presentation of idioms, arrows and monads as variations on a single calculus.

(ii) A precise characterisation of the relationship between the three notions of computation which shows that, in contrast to the folklore ordering, idioms are less powerful than arrows.

The remainder of this paper is organised as follows. Section 2 introduces the notion of *equational equivalence*, a generalisation of *equational correspondence* [11]. Section 3 defines standard equational theories corresponding to idioms, arrows and monads, and a more convenient equational theory for arrows which highlights how arrows *meticulously* maintain the distinction between terms and commands. Section 4 gives an informal comparison of the expressive power of the three notions of computation by means of an example. Section 5 presents idioms as a variant of arrows, characterised by either a type isomorphism or an additional equation, in which commands are *oblivious* to input. Section 6 presents monads as a variant of arrows, characterised by either a type isomorphism or an additional operator and accompanying equations, which allay the distinction between terms and commands, resulting in a *promiscuous* admixture. Section 7 concludes.

2 Preliminaries

Definition 2.1 A typed *equational theory* T consists of the following

- variables x, y, z
- types A, B, C

- terms L, M, N
- type environments $\Gamma ::= \cdot \mid x : A, \Gamma$
- typing judgements $\Gamma \vdash_T M : A$
- equational judgements $\Gamma \vdash_T M = N : A$

Equational judgements must be well-formed: if $\Gamma \vdash_T M = N : A$ then $\Gamma \vdash_T M : A$ and $\Gamma \vdash_T N : A$. We present the equational judgements via laws relating terms, writing $M = N$ as shorthand for $\Gamma \vdash_T M = N : A$ for all Γ, A in T such that $\Gamma \vdash_T M : A$ and $\Gamma \vdash_T N : A$. The equational theory is defined as the contextual and equivalence closure of the laws.

Definition 2.2 Let T be an equational theory with typing judgements $x : A \vdash_T f : B$ and $x : B \vdash_T f^{-1} : A$. (These typing judgements can be viewed as translations on terms: f from A to B and f^{-1} from B to A. For convenience, we write $f(M)$ for $f[x := M]$ and $f^{-1}(N)$ for $f^{-1}[x := N]$.) We say that A is isomorphic to B and f, f^{-1} witness the isomorphism ($f : A \simeq B$) if

- Translating from A to B and back is the identity,

  $$\Gamma \vdash_T f^{-1}(f(M)) = M : A$$

  for all $\Gamma \vdash_T M : A$ in T.
- Translating from B to A and back is the identity,

  $$\Gamma \vdash_T f(f^{-1}(N)) = N : B$$

  for all $\Gamma \vdash_T N : B$ in T.

As all of the theories we consider include function types and lambda abstractions, we choose to express the isomorphisms more concisely as pairs of closed terms $f : A \to B$ and $f^{-1} : B \to A$ rather than typing judgements $x : A \vdash_T f : B$ and $x : B \vdash_T f^{-1} : A$.

Definition 2.3 Let S, T be equational theories, with a compositional translation on terms and types $[\![-]\!]$ from S to T that preserves typing,

$$\Gamma \vdash_S M : A \ \text{ implies } \ [\![\Gamma]\!] \vdash_T [\![M]\!] : [\![A]\!]$$

for all Γ, M, A in S, and with a compositional inverse translation $\langle\!|-|\!\rangle$ from T to S that also preserves typing,

$$\Gamma \vdash_T M : A \ \text{ implies } \ \langle\!|\Gamma|\!\rangle \vdash_S \langle\!|M|\!\rangle : \langle\!|A|\!\rangle$$

for all Γ, M, A in T. Further, translating a type from S to T and back yields a type isomorphic to the original type,

$$f_A : A \simeq \langle\!|[\![A]\!]|\!\rangle$$

for all A in S. Similarly, translating a type from T to S and back yields a type isomorphic to the original type,

$$g_A : A \simeq [\![\langle\!| A |\!\rangle]\!]$$

for all A in T. We say these translations form an *equational equivalence* ($S \sim T$) if

- The translation from S to T preserves equations,

$$\Gamma \vdash_S M = N : A \;\; \text{implies} \;\; [\![\Gamma]\!] \vdash_T [\![M]\!] = [\![N]\!] : [\![A]\!]$$

  for all Γ, M, N, A in S.
- The translation from T to S preserves equations,

$$\Gamma \vdash_T M = N : A \;\; \text{implies} \;\; \langle\!| \Gamma |\!\rangle \vdash_S \langle\!| M |\!\rangle = \langle\!| N |\!\rangle : \langle\!| A |\!\rangle$$

  for all Γ, M, N, A in T.
- Translating from S to T and back yields a term isomorphic to the original term,

$$\Gamma \vdash_S M : A \;\; \text{implies} \;\; \Gamma \vdash_S \langle\!| [\![M]\!] |\!\rangle[\Gamma := f(\Gamma)] = f_A(M) : \langle\!| [\![A]\!] |\!\rangle$$

  for all Γ, M, A in S (writing $N[\Gamma := f(\Gamma)]$ for $N[x_1 := f_{A_1}(x_1), \ldots, x_n := f_{A_n}(x_n)]$, given $\Gamma = x_1 : A_1, \ldots, x_n : A_n$).
- Translating from T to S and back yields a term isomorphic to the original term,

$$\Gamma \vdash_T M : A \;\; \text{implies} \;\; \Gamma \vdash_T [\![\langle\!| M |\!\rangle]\!][\Gamma := g(\Gamma)] = g_A(M) : [\![\langle\!| A |\!\rangle]\!]$$

  for all Γ, M, A in T.

(This definition amounts to saying that we have an equivalence of categories [3], where $[\![-]\!]$ is left adjoint to $\langle\!| - |\!\rangle$ with unit f_A and counit g_A^{-1}.)

The special case of an equational equivalence where both isomorphisms are the identity is an *equational correspondence*.

Definition 2.4 An *equational correspondence* between theories S and T ($S \cong T$) is an equational equivalence with translations $[\![-]\!] : S \to T, \langle\!| - |\!\rangle : T \to S$ where both f_A and g_A are the identity at each type A. (This amounts to saying that we have an isomorphism of categories [3] given by the translations $[\![-]\!] : S \to T$ and $\langle\!| - |\!\rangle : T \to S$.)

We also introduce the notion of *equational embedding*, a map from a weaker into a stronger theory. An equational embedding of a theory S into a theory T may be defined as an equational equivalence between S and a *subtheory* of T. We instead use the following more direct definition, which is more convenient in practice.

Definition 2.5 Let S, T be equational theories with a compositional translation on terms and types $[\![-]\!]$ from S to T that preserves typing,

$$\Gamma \vdash_S M : A \;\; \text{implies} \;\; [\![\Gamma]\!] \vdash_T [\![M]\!] : [\![A]\!]$$

for all Γ, M, A in S, and with a compositional inverse translation $\langle\!\langle - \rangle\!\rangle$ from $[\![S]\!]$ to S that also preserves typing,

$$[\![\Gamma]\!] \vdash_T [\![M]\!] : [\![A]\!] \ \text{ implies } \ \langle\!\langle [\![\Gamma]\!] \rangle\!\rangle \vdash_S \langle\!\langle [\![M]\!] \rangle\!\rangle : \langle\!\langle [\![A]\!] \rangle\!\rangle$$

for all Γ, M, A in S. Further, translating a type from S to T and back yields a type isomorphic to the original type,

$$f_A : A \simeq \langle\!\langle [\![A]\!] \rangle\!\rangle$$

for all A in S. We say these translations form an *equational embedding* of S into T ($S \hookrightarrow T$) if

- The translation from S to T preserves equations,

  $$\Gamma \vdash_S M = N : A \ \text{ implies } \ [\![\Gamma]\!] \vdash_T [\![M]\!] = [\![N]\!] : [\![A]\!]$$

  for all Γ, M, N, A in S.
- The translation from $[\![S]\!]$ to S preserves equations,

  $$[\![\Gamma]\!] \vdash_T [\![M]\!] = [\![N]\!] : [\![A]\!] \ \text{ implies } \ \langle\!\langle [\![\Gamma]\!] \rangle\!\rangle \vdash_S \langle\!\langle [\![M]\!] \rangle\!\rangle = \langle\!\langle [\![N]\!] \rangle\!\rangle : \langle\!\langle [\![A]\!] \rangle\!\rangle$$

  for all Γ, M, N, A in S.
- Translating from S to $[\![S]\!]$ and back yields a term isomorphic to the original term,

  $$\Gamma \vdash_S M : A \ \text{ implies } \ \Gamma \vdash_S \langle\!\langle [\![M]\!] \rangle\!\rangle[\Gamma := f(\Gamma)] = f_A(M) : \langle\!\langle [\![A]\!] \rangle\!\rangle$$

  for all Γ, M, A in S.

3 Theories

This section outlines theories for simply-typed lambda calculus extended with pairs and unit: $\lambda^{\to\times 1}$, idioms: $\mathcal{I}$, monads: $\mathcal{M}$ and two different theories for arrows: $\mathcal{C}$ and $\mathcal{A}$.

Figure 2 (page 7) gives a standard definition of the theory of typed lambda calculus extended with pairs and unit, $\lambda^{\to\times 1}$. We use this definition as a starting point for each of the theories which follow. For convenience we define a number of functions, such as *id*.

Figure 3 (page 8) defines the theory of idioms, $\mathcal{I}$ [5]. Idioms extend $\lambda^{\to\times 1}$ with a unary type constructor I for computations of type $I\,A$ which return a value of type A. There are two constants: *pure*, which takes a value and constructs a computation which returns the value, and $(\circledast)$, which combines two computations, applying the value returned by the first to the value returned by the second. There are four laws which, together with the laws of the lambda calculus, define the equivalence relation of the theory. For idioms to serve as a useful programming language we would additionally need constants for constructing basic computations. These play no significant role in the theory, so we omit them here.

Syntax

$$\begin{array}{lll}
\text{Types} & A, B, C & ::= \mathcal{B} \mid 1 \mid A{\times}B \mid A \to B \\
\text{Terms} & L, M, N & ::= x \mid \langle\rangle \mid \langle M, N\rangle \mid \mathsf{fst}\ L \mid \mathsf{snd}\ L \mid \lambda x.\, N \mid L\ M \\
\text{Environments} & \Gamma & ::= x_1 : A_1, \ldots, x_n : A_n
\end{array}$$

Types

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \Gamma \vdash \langle\rangle : 1$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N\rangle : A{\times}B} \qquad \frac{\Gamma \vdash L : A{\times}B}{\Gamma \vdash \mathsf{fst}\ L : A} \qquad \frac{\Gamma \vdash L : A{\times}B}{\Gamma \vdash \mathsf{snd}\ L : B}$$

$$\frac{\Gamma, x : A \vdash N : B}{\Gamma \vdash \lambda x.\, N : A \to B} \qquad \frac{\Gamma \vdash L : A \to B \qquad \Gamma \vdash M : A}{\Gamma \vdash L\ M : B}$$

Definitions

$$\begin{array}{rcl@{\qquad}rcl}
id & : & A \to A & (\times) & : & (A \to C) \to (B \to D) \to (A{\times}B \to C{\times}D) \\
id & = & \lambda x.\, x & (\times) & = & \lambda f.\, \lambda g.\, \lambda z.\, \langle f\ (\mathsf{fst}\ z), g\ (\mathsf{snd}\ z)\rangle \\[1ex]
dup & : & A \to A{\times}A & (\cdot) & : & (B \to C) \to (A \to B) \to (A \to C) \\
dup & = & \lambda x.\, \langle x, x\rangle & (\cdot) & = & \lambda f.\, \lambda g.\, \lambda x.\, f\ (g\ x) \\[1ex]
swap & : & A{\times}B \to B{\times}A & (;) & : & (A \to B) \to (B \to C) \to (A \to C) \\
swap & = & \lambda z.\, \langle \mathsf{snd}\ z, \mathsf{fst}\ z\rangle & (;) & = & \lambda f.\, \lambda g.\, \lambda x.\, g\ (f\ x) \\[1ex]
fst & : & A{\times}B \to A & assoc & : & (A{\times}B){\times}C \to A{\times}(B{\times}C) \\
fst & = & \lambda z.\, \mathsf{fst}\ z & assoc & = & \lambda z.\, \langle \mathsf{fst}\ (\mathsf{fst}\ z), \langle \mathsf{snd}\ (\mathsf{fst}\ z), \mathsf{snd}\ z\rangle\rangle \\[1ex]
snd & : & A{\times}B \to B & apply & : & (A \to B){\times}A \to B \\
snd & = & \lambda z.\, \mathsf{snd}\ z & apply & = & \lambda z.\, (\mathsf{fst}\ z\ (\mathsf{snd}\ z))
\end{array}$$

Laws

$$\begin{array}{lrcl}
(\beta_1^\times) & \mathsf{fst}\ \langle M, N\rangle & = & M \\
(\beta_2^\times) & \mathsf{snd}\ \langle M, N\rangle & = & N \\
(\eta^\times) & \langle \mathsf{fst}\ L, \mathsf{snd}\ L\rangle & = & L \\
(\beta^\to) & (\lambda x.\, N)\ M & = & N[x := M] \\
(\eta^\to) & \lambda x.\, (L\ x) & = & L \\
(\eta^1) & \langle\rangle & = & M
\end{array}$$

Fig. 2. Lambda calculus, $\lambda^{\to\times 1}$

Syntax

$$\text{Types} \quad A, B, C ::= \cdots \mid I\,A$$

Constants

$$\begin{aligned} \mathit{pure} &: A \to I\,A \\ (\otimes) &: I\,(A \to B) \to I\,A \to I\,B \end{aligned}$$

Laws

$$\begin{array}{lrcl} (I_1) & u & = & \mathit{pure}\ \mathit{id} \otimes u \\ (I_2) & \mathit{pure}\ f \otimes \mathit{pure}\ p & = & \mathit{pure}\ (f\ p) \\ (I_3) & u \otimes (v \otimes w) & = & \mathit{pure}\ (\cdot) \otimes u \otimes v \otimes w \\ (I_4) & u \otimes \mathit{pure}\ x & = & \mathit{pure}\ (\lambda f.\, f\ x) \otimes u \end{array}$$

Fig. 3. Idioms, $\mathcal{I}$

Figure 4 (page 8) defines the theory of arrows [1,8], $\mathcal{C}$, which (following [4]) we refer to as *classic arrows*. Classic arrows extend $\lambda^{\to\times 1}$ with a binary type constructor $\rightsquigarrow$ for computations with input and output and three constants for creating and composing computations. The first constant, *arr*, constructs a computation from a function. The second, $(\ggg)$, combines two computations, passing the output of the first as input to the second. The third, *first*, transforms a computation to pass through additional data untouched. These primitives make it possible to construct a wide range of combinators for computations. Finally, there are nine laws which, together with the laws of the lambda calculus, define the equivalence relation of the theory.

Figure 5 (page 9) defines the arrow calculus $\mathcal{A}$ [4]. Arrow calculus extends $\lambda^{\to\times 1}$ with four constructs satisfying five laws. As with $\mathcal{C}$, the type $A \rightsquigarrow B$ denotes a computation that accepts a value of type A and returns a value of type B, possibly performing some side effects. There are now two syntactic categories: terms, ranged over by L, M, N, and commands, ranged over by P, Q, R. In addition to the terms of $\lambda^{\to\times 1}$, there is one new term form: arrow abstraction $\lambda^\bullet x.\, Q$. There are three command forms: arrow application $L \bullet M$, arrow unit $[M]$ (analogous to *arr*), and arrow bind $\mathsf{let}\ x = P\ \mathsf{in}\ Q$.

In addition to the term typing judgement $\Gamma \vdash M : A$ we now also have a command typing judgement

$$\Gamma; \Delta \vdash P \,!\, A.$$

Similarly, in addition to the equational judgement on terms $\Gamma \vdash M = N : A$ we now also have an equational judgement on commands

$$\Gamma; \Delta \vdash P = Q \,!\, A.$$

(In specifying the laws we write $P = Q$ as shorthand for $\Gamma; \Delta \vdash P = Q \,!\, A$ for all Γ, Δ, A such that $\Gamma; \Delta \vdash P \,!\, A$ and $\Gamma; \Delta \vdash Q \,!\, A$.) An important feature of the arrow calculus is that these judgements have two environments, Γ and Δ, where variables in Γ come from ordinary lambda abstractions $\lambda x.\, N$, while variables in Δ come from arrow abstractions $\lambda^\bullet x.\, Q$. The *meticulousness* of the title refers to the

Syntax

$$\text{Types} \quad A, B, C ::= \cdots \mid A \leadsto B$$

Constants

$$\begin{aligned} \mathit{arr} &: (A \to B) \to (A \leadsto B) \\ (\ggg) &: (A \leadsto B) \to (B \leadsto C) \to (A \leadsto C) \\ \mathit{first} &: (A \leadsto B) \to (A \times C \leadsto B \times C) \end{aligned}$$

Definitions

$$\begin{aligned} &\mathit{second} \;:\; (A \leadsto B) \to (C \times A \leadsto C \times B) \\ &\mathit{second} = \lambda f.\, \mathit{arr}\ \mathit{swap} \ggg \mathit{first}\ f \ggg \mathit{arr}\ \mathit{swap} \\ &(\&\&\&) \quad : (C \leadsto A) \to (C \leadsto B) \to (C \leadsto A \times B) \\ &(\&\&\&) \quad = \lambda f.\, \lambda g.\, \mathit{arr}\ \mathit{dup} \ggg \mathit{first}\ f \ggg \mathit{second}\ g \end{aligned}$$

Laws

$$\begin{array}{lrcl} (\leadsto_1) & \mathit{arr}\ \mathit{id} \ggg f &=& f \\ (\leadsto_2) & f \ggg \mathit{arr}\ \mathit{id} &=& f \\ (\leadsto_3) & (f \ggg g) \ggg h &=& f \ggg (g \ggg h) \\ (\leadsto_4) & \mathit{arr}\ (g \cdot f) &=& \mathit{arr}\ f \ggg \mathit{arr}\ g \\ (\leadsto_5) & \mathit{first}\ (\mathit{arr}\ f) &=& \mathit{arr}\ (f \times \mathit{id}) \\ (\leadsto_6) & \mathit{first}\ (f \ggg g) &=& \mathit{first}\ f \ggg \mathit{first}\ g \\ (\leadsto_7) & \mathit{first}\ f \ggg \mathit{arr}\ (\mathit{id} \times g) &=& \mathit{arr}\ (\mathit{id} \times g) \ggg \mathit{first}\ f \\ (\leadsto_8) & \mathit{first}\ f \ggg \mathit{arr}\ \mathit{fst} &=& \mathit{arr}\ \mathit{fst} \ggg f \\ (\leadsto_9) & \mathit{first}\ (\mathit{first}\ f) \ggg \mathit{arr}\ \mathit{assoc} &=& \mathit{arr}\ \mathit{assoc} \ggg \mathit{first}\ f \end{array}$$

Fig. 4. Arrows, $\mathcal{C}$

careful maintenance of this distinction in the typing rules.

Figure 6 (page 9) defines the theory of monads, $\mathcal{M}$ [7]. Like idioms, monads extend $\lambda^{\to\times 1}$ with a unary type constructor M for computations of type $M\ A$ which return a value of type A. The constant *return* is analogous to the idiomatic *pure*, while $\gg\!=$ constructs a computation from a computation and a computation-constructing function, supplying the value returned by the former as argument to the latter. (We might just as well have used Moggi's computational metalanguage [7] instead of adding constants to $\mathcal{C}$, but we chose to define constants for consistency with our treatment of idioms and classic arrows.)

We find it convenient to use the arrow calculus rather than classic arrows as a basis for comparison. The following result allows us to move freely between the two theories.

Proposition 3.1 *The theories of arrow calculus and classic arrows are in equational correspondence:* $\mathcal{A} \cong \mathcal{C}$.

(Here and throughout we elide the definition of the homomorphic translations on terms of the lambda calculus.)

Syntax

$$\begin{array}{lrcl} \text{Types} & A,B,C & ::= & \cdots \mid A \leadsto B \\ \text{Terms} & L,M,N & ::= & \cdots \mid \lambda^{\bullet}x.\,Q \\ \text{Commands} & P,Q,R & ::= & L \bullet M \mid [M] \mid \mathsf{let}\ x = P\ \mathsf{in}\ Q \end{array}$$

Types

$$\frac{\Gamma;\, x : A \vdash Q \,!\, B}{\Gamma \vdash \lambda^{\bullet}x.\,Q : A \leadsto B} \qquad \frac{\Gamma \vdash L : A \leadsto B \qquad \Gamma, \Delta \vdash M : A}{\Gamma;\, \Delta \vdash L \bullet M \,!\, B}$$

$$\frac{\Gamma,\, \Delta \vdash M : A}{\Gamma;\, \Delta \vdash [M] \,!\, A}$$

$$\frac{\Gamma;\, \Delta \vdash P \,!\, A \qquad \Gamma;\, \Delta,\, x : A \vdash Q \,!\, B}{\Gamma;\, \Delta \vdash \mathsf{let}\ x = P\ \mathsf{in}\ Q \,!\, B}$$

Laws

$$\begin{array}{lrcl} (\beta^{\leadsto}) & (\lambda^{\bullet}x.\,Q) \bullet M & = & Q[x := M] \\ (\eta^{\leadsto}) & \lambda^{\bullet}x.\,(L \bullet x) & = & L \\ (\text{left}) & \mathsf{let}\ x = [M]\ \mathsf{in}\ Q & = & Q[x := M] \\ (\text{right}) & \mathsf{let}\ x = P\ \mathsf{in}\ [x] & = & P \\ (\text{assoc}) & \mathsf{let}\ y = (\mathsf{let}\ x = P\ \mathsf{in}\ Q)\ \mathsf{in}\ R & = & \mathsf{let}\ x = P\ \mathsf{in}\ (\mathsf{let}\ y = Q\ \mathsf{in}\ R) \end{array}$$

Fig. 5. The arrow calculus, $\mathcal{A}$

Syntax

$$\text{Types} \quad A,B,C ::= \cdots \mid M\,A$$

Constants

$$\begin{array}{l} \mathit{return} : A \to M\,A \\ (\gg\!=) : M\,A \to (A \to M\,B) \to M\,B \end{array}$$

Laws

$$\begin{array}{lrcl} (M_1) & \mathit{return}\ a \gg\!= f & = & f\ a \\ (M_2) & m \gg\!= \mathit{return} & = & m \\ (M_3) & (m \gg\!= k) \gg\!= h & = & m \gg\!= (\lambda x.k\ x \gg\!= h) \end{array}$$

Fig. 6. Monads, $\mathcal{M}$

Arrow calculus to classic arrows:

$$\begin{aligned} [\![A]\!] &= A \\ [\![\lambda^\bullet x.\, Q]\!] &= [\![Q]\!]_x \end{aligned}$$

where

$$\begin{aligned} [\![\Gamma; \Delta \vdash P \,!\, A]\!] &= \Gamma \vdash [\![P]\!]_\Delta : \Delta \rightsquigarrow A \\ [\![L \bullet M]\!]_\Delta &= \mathit{arr}\ (\lambda\Delta.\, [\![M]\!]) \ggg [\![L]\!] \\ [\![[M]]\!]_\Delta &= \mathit{arr}\ (\lambda\Delta.\, [\![M]\!]) \\ [\![\mathsf{let}\ x = P\ \mathsf{in}\ Q]\!]_\Delta &= (\mathit{arr}\ \mathit{id} \mathbin{\&\&\&} [\![P]\!]_\Delta) \ggg [\![Q]\!]_{\Delta,x} \end{aligned}$$

Classic arrows to arrow calculus:

$$\begin{aligned} \langle\!|A|\!\rangle &= A \\ \langle\!|\mathit{arr}|\!\rangle &= \lambda f.\, \lambda^\bullet x.\, [f\ x] \\ \langle\!|(\ggg)|\!\rangle &= \lambda f.\, \lambda g.\, \lambda^\bullet x.\, \mathsf{let}\ y = f \bullet x\ \mathsf{in}\ g \bullet y \\ \langle\!|\mathit{first}|\!\rangle &= \lambda f.\, \lambda^\bullet z.\, \mathsf{let}\ x = f \bullet (\mathsf{fst}\ z)\ \mathsf{in}\ [\langle x, \mathsf{snd}\ z\rangle] \end{aligned}$$

Further details may be found in [4]. Note that despite the special form of typing judgement for commands, we do not need to generalise our definition of equational correspondence as we only care about equational correspondence between terms. However, the proof [4] does rely on showing a correspondence property involving commands: $\Gamma \vdash \langle\!|[\![P]\!]_\Delta|\!\rangle = \lambda^\bullet\Delta.\, P : \Delta \rightsquigarrow A$.

Note that an arrow calculus term judgement maps into a classic arrow judgement

$$\Gamma \vdash M : A \ \text{ maps to } \ \Gamma \vdash [\![M]\!] : A$$

while an arrow calculus command judgement maps into a classic arrow judgement

$$\Gamma; \Delta \vdash P \,!\, A \ \text{ maps to } \ \Gamma \vdash [\![P]\!]_\Delta : \Delta \rightsquigarrow A.$$

In $[\![P]\!]_\Delta$, we take Δ to stand for the sequence of variables in the environment, and in $\Delta \rightsquigarrow A$ we take Δ to stand for the left-nested product of the types in the environment. The denotation of a command of type A is an arrow whose arguments correspond to the environment Δ and whose result has type A.

The translation uses the notation $\lambda\Delta.\, N$, which is given the obvious meaning: $\lambda x.\, N$ stands for itself, $\lambda x_1, x_2.\, N$ stands for $\lambda z.\, N[x_1 := \mathsf{fst}\ z, x_2 := \mathsf{snd}\ z]$, $\lambda x_1, x_2, x_3.\, N$ stands for $\lambda z.\, N[x_1 := \mathsf{fst}\ (\mathsf{fst}\ z), x_2 := \mathsf{snd}\ (\mathsf{fst}\ z), x_3 := \mathsf{snd}\ z]$, and so on.

4 Example

We now turn to an informal comparison of idioms, arrows and monads, in order to illustrate the relative expressive power of each before returning to a more formal investigation in Sections 5 and 6.

In Section 3 we presented the theories of idioms, arrows and monads. However, in actual programs we do not simply use monads, arrows or idioms in the abstract, but particular *instances* of these interfaces in which the type expressions $A \leadsto B$, $M\,A$ or $I\,A$ denote concrete types. For example, we can use the well-known state monad with integers as the encapsulated state by instantiating $M\,A$ to the type $\mathit{Int} \to \mathit{Int}{\times}A$ and providing additional constants for reading and writing the state:

$$\begin{aligned} \mathit{get} &: 1 \to M\,\mathit{Int} \\ \mathit{put} &: \mathit{Int} \to M\,1 \end{aligned}$$

(We give a slightly non-standard type for *get* in order to simplify the translation to arrows and idioms in what follows. The type of our *get* is isomorphic to the more common $M\,\mathit{Int}$ and the behaviour of our $\mathit{get}\,\langle\rangle$ identical to that of the standard *get*.) We can then use these constants along with the standard monad operators *return* and $(\gg\!=)$, to write programs such as the following, which uses the encapsulated state to generate fresh names (given a type of names *Name* and a name-construction function $\mathit{makeName} : \mathit{Int} \to \mathit{Name}$):

$$\begin{aligned} \mathit{freshName} &: M\,\mathit{Name} \\ \mathit{freshName} &= \mathit{get}\,\langle\rangle \gg\!= \lambda s.\,\mathit{put}\,(s+1) \gg\!= \lambda u.\,\mathit{return}\,(\mathit{makeName}\;s) \end{aligned}$$

or the following, which branches on the current state in order to choose which of two computations to execute:

$$\begin{aligned} \mathit{ifZero} &: (M\,A{\times}M\,A) \to M\,A \\ \mathit{ifZero} &= \lambda k.\,\mathit{get}\,\langle\rangle \gg\!= \lambda s.\,\mathsf{if}\;s=0\;\;\mathsf{then}\;\;\mathsf{fst}\;k\;\;\mathsf{else}\;\;\mathsf{snd}\;k \end{aligned}$$

or the following, which reads, transforms and returns the current state:

$$\begin{aligned} \mathit{getTransformed} &: (\mathit{Int} \to A) \to M\,A \\ \mathit{getTransformed} &= \lambda f.\,\mathit{get}\,\langle\rangle \gg\!= \lambda s.\,\mathit{return}\,(f\;s) \end{aligned}$$

We can obtain an arrow from the state monad using the standard Kleisli construction [1], setting $A \leadsto B$ to $A \to M\,B$:

$$\begin{aligned} \mathit{arr} &: (A \to B) \to (A \leadsto B) \\ \mathit{arr} &= \lambda f.\,\lambda a.\,\mathit{return}\,(f\;\;a) \\ (\ggg) &: (A \leadsto B) \to (B \leadsto C) \to (A \leadsto C) \\ (\ggg) &= \lambda f.\,\lambda g.\,\lambda a.\,f\;\;a \gg\!= g \\ \mathit{first} &: (A \leadsto B) \to (A{\times}C \leadsto B{\times}C) \\ \mathit{first} &= \lambda f.\,\lambda a.\,f\;\;(\mathsf{fst}\;a) \gg\!= \lambda b.\,\mathit{return}\,\langle b, c\rangle \end{aligned}$$

The constants *get* and *put* now have the following types

$$\begin{aligned} get_{\leadsto} &: 1 \leadsto Int \\ put_{\leadsto} &: Int \leadsto 1 \end{aligned}$$

and we can use them to write an arrow equivalent of *freshName*:

$$\begin{aligned} freshName_{\leadsto} &: 1 \leadsto Int \\ freshName_{\leadsto} &= get_{\leadsto} \ggg arr\,(\lambda x.\,\langle x+1, x\rangle) \ggg first\ put_{\leadsto} \ggg arr\ snd \end{aligned}$$

or an arrow equivalent of *getTransformed*:

$$\begin{aligned} getTransformed_{\leadsto} &: (Int \to A) \to 1 \leadsto A \\ getTransformed_{\leadsto} &= \lambda f.\,(get_{\leadsto} \ggg arr\ f) \end{aligned}$$

However, there is no way to write an arrow equivalent of *ifZero*. This is due to the “first-orderness” of arrows: the arrow interface does not provide a method for running a computation received as input. Similarly, we can obtain a state idiom from the state monad using standard techniques [5], setting $I\,A$ to $M\,A$:

$$\begin{aligned} pure &: A \to I\,A \\ pure &= return \\ (\otimes) &: I\,(A \to B) \to (I\,A \to I\,B) \\ (\otimes) &= \lambda f.\,\lambda p.\,f \ggg\!= \lambda g.\,p \ggg\!= \lambda q.\,return\ (g\ q) \end{aligned}$$

The constants *get* and *put* now have the following types

$$\begin{aligned} get_I &: 1 \to I\,Int \\ put_I &: Int \to I\,1 \end{aligned}$$

and we can use them to write *getTransformed* idiomatically:

$$\begin{aligned} getTransformed_I &: (Int \to A) \to I\,A \\ getTransformed_I &= \lambda f.\,pure\ f \otimes get_I\,\langle\rangle \end{aligned}$$

However, we cannot write either *freshName* or *ifZero* using the idiom operations since, as we show in the next section, the idiom interface does not provide a means for one computation to depend on the value returned by another. The put_I function is therefore much less useful than its monad and arrow counterparts, since its argument cannot be a value arising from the computation of which it forms part.

Since monads are the most powerful of the three notions, the question naturally arises whether it might not be better to use monads in every case. In fact, it is precisely because monads are more expressive that they are not suitable for every situation: they offer more to users, but demand more of implementers. There are consequently many interesting instances of the idiom and arrow interfaces which do not satisfy the more stringent requirements for monad instances [1,2,5,8,9].

5 Relating idioms and arrows

In order to compare idioms and arrows we formalise static arrow computations. First, we describe a variant of classic arrows that supports static computation $\mathcal{C}_{\mathrm{S}}$ by adding an extra constant and two laws. Then, we introduce a variant of the arrow calculus, *static arrows* $\mathcal{S}$, by adding one command and three laws. We show that $\mathcal{C}_{\mathrm{S}}$ and $\mathcal{S}$ are in equational correspondence. We then give an equational embedding of static arrows into arrow calculus and show that idioms are equationally equivalent to static arrows.

In the arrow calculus, $\mathcal{A}$, computations accept input via the command application operation $L \bullet M$. The semantics of a computation in $\mathcal{S}$ is independent of input. We capture this property with an additional command that allows an arrow computation to be *run* before supplying it with an input, and an additional equation which treats computations with input as equivalent to computations without input. Thus static arrow computations are *oblivious* to their inputs.

Definition 5.1 The theory $\mathcal{C}_{\mathrm{S}}$ of *classic arrows with delay* is the extension of theory $\mathcal{C}$ with the constant

$$\mathit{delay} : (A \rightsquigarrow B) \to (1 \rightsquigarrow (A \to B))$$

and the additional laws:

$$\begin{array}{ll} (\rightsquigarrow_{S1}) & \mathit{force}\ (\mathit{delay}\ (a)) = a \\ (\rightsquigarrow_{S2}) & \mathit{delay}\ (\mathit{force}\ (a)) = a \end{array}$$

where

$$\begin{array}{l} \mathit{force} : (1 \rightsquigarrow (A \to B)) \to (A \rightsquigarrow B) \\ \mathit{force} = \lambda f.\ \mathit{arr}\ (\lambda x.\ \langle\langle\rangle, x\rangle) \ggg \mathit{first}\ f \ggg \mathit{arr}\ (\mathit{apply}) \end{array}$$

Definition 5.2 The theory $\mathcal{S}$ of *static arrows* is the extension of the theory $\mathcal{A}$ with an additional syntactic construct given by the typing rule:

$$\frac{\Gamma \vdash L : A \rightsquigarrow B}{\Gamma;\, \Delta \vdash \mathsf{run}\ L\ !\ A \to B}$$

and the additional laws:

$$\begin{array}{lrcl} (ob_1) & L \bullet M & = & \mathsf{let}\ f = \mathsf{run}\ L\ \mathsf{in}\ [f\ \ M] \\ (ob_2) & \mathsf{run}\ (\lambda^{\bullet} x.\, [M]) & = & [\lambda x.\, M] \\ (ob_3) & \mathsf{run}\ (\lambda^{\bullet} x.\, \mathsf{let}\ y = P\ \mathsf{in}\ Q) & = & \mathsf{let}\ y = P\ \mathsf{in} \\ & & & \quad \mathsf{let}\ f = \mathsf{run}\ (\lambda^{\bullet}\langle x, y\rangle.\, Q)\ \mathsf{in}\ [\lambda x.\, f\ \langle x, y\rangle] \end{array}$$

Proposition 5.3 *The theories of static arrows and classic arrows with delay are in equational correspondence:* $\mathcal{S} \cong \mathcal{C}_S$.

The translations are each extended with an extra clause.
Static arrows to classic arrows with delay:

$$[\![\mathsf{run}\ L]\!]_\Delta = arr\ (\lambda\Delta.\ \langle\rangle) \ggg delay\ [\![L]\!]$$

Classic arrows with delay to static arrows:

$$(\!|delay|\!) = \lambda x.\ \lambda^\bullet u.\ \mathsf{run}\ x$$

(By convention we use the variable u to bind variables of type 1.)

Proposition 5.4 *The theories of idioms and static arrows are equationally equivalent: $\mathcal{I} \sim \mathcal{S}$.*

(Here and throughout we elide the definition of the homomorphic translations on types and the corresponding type isomorphisms.)

Idioms to static arrows:

$$[\![I\,A]\!] = 1 \rightsquigarrow [\![A]\!]$$

$$\begin{aligned} [\![pure]\!] &= \lambda x.\ \lambda^\bullet u.\ [x] \\ [\![(\otimes)]\!] &= \lambda h.\ \lambda a.\ \lambda^\bullet u.\ \mathsf{let}\ k = h \bullet \langle\rangle\ \mathsf{in}\ \mathsf{let}\ x = a \bullet \langle\rangle\ \mathsf{in}\ [k\ x] \end{aligned}$$

Static arrows to idioms:

$$(\!|A \rightsquigarrow B|\!) = I\,((\!|A|\!) \to (\!|B|\!))$$

$$(\!|\lambda^\bullet x.\,P|\!) = (\!|P|\!)_x$$

where

$$\begin{aligned} (\!|\Gamma;\ \Delta \vdash P\ !\ A|\!) &= (\!|\Gamma|\!) \vdash (\!|P|\!)_\Delta : I\,((\!|\Delta|\!) \to (\!|A|\!)) \\ (\!|L \bullet M|\!)_\Delta &= pure\ (\lambda l.\ \lambda\Delta.\ l\ (\!|M|\!)) \otimes (\!|L|\!) \\ (\!|\mathsf{run}\ L|\!)_\Delta &= pure\ (\lambda l.\ \lambda\Delta.\ l) \otimes (\!|L|\!) \\ (\!|[M]|\!)_\Delta &= pure\ (\lambda\Delta.\ (\!|M|\!)) \\ (\!|\mathsf{let}\ x = P\ \mathsf{in}\ Q|\!)_\Delta &= pure\ (\lambda p.\ \lambda q.\ \lambda\Delta.\ q\ \langle\Delta, p\ \Delta\rangle) \otimes (\!|P|\!)_\Delta \otimes (\!|Q|\!)_{\Delta,x} \end{aligned}$$

Type isomorphism on idioms:

$$\begin{aligned} &f_{I(A)} : I\,A \simeq I\,(1 \to A) \\ &f_{I(A)} = \lambda a.\ pure\ (\lambda x.\ \lambda u.\ x) \otimes a \\ &f^{-1}_{I(A)} = \lambda a.\ pure\ (\lambda x.\ x\ \langle\rangle) \otimes a \end{aligned}$$

Type isomorphism on static arrows:

$$\begin{aligned} &g_{A \rightsquigarrow B} : A \rightsquigarrow B \simeq 1 \rightsquigarrow (A \to B) \\ &g_{A \rightsquigarrow B} = \lambda a.\ \lambda^\bullet u.\ \mathsf{run}\ a \\ &g^{-1}_{A \rightsquigarrow B} = \lambda a.\ \lambda^\bullet x.\ \mathsf{let}\ h = a \bullet \langle\rangle\ \mathsf{in}\ [h\ x] \end{aligned}$$

Remark 5.5 The type isomorphism $g_{A \rightsquigarrow B} : A \rightsquigarrow B \simeq 1 \rightsquigarrow (A \to B)$ gives an alternative characterisation of static arrows. The type isomorphism and $\mathsf{run}\ L$ are inter-definable. The definition of $g_{A \rightsquigarrow B}$ in terms of $\mathsf{run}\ L$ is given above. The definition of $\mathsf{run}\ L$ in terms of $g_{A \rightsquigarrow B}$ follows.

$$\mathsf{run}\ L \equiv (g_{A \rightsquigarrow B}\ L) \bullet \langle\rangle$$

Proposition 5.6 *There is an equational embedding of static arrows into arrow calculus: $\mathcal{S} \hookrightarrow \mathcal{A}$*

Static arrows to arrows:

$$\begin{aligned}
\llbracket A \rightsquigarrow B \rrbracket &= 1 \rightsquigarrow (\llbracket A \rrbracket \to \llbracket B \rrbracket) \\
\llbracket \lambda^\bullet x.\, P \rrbracket &= \lambda^\bullet u.\, \llbracket P \rrbracket_x
\end{aligned}$$

where

$$\begin{aligned}
\llbracket \Gamma;\, \Delta \vdash P \mathbin{!} A \rrbracket &= \llbracket \Gamma \rrbracket; \cdot \vdash \llbracket P \rrbracket_\Delta \mathbin{!} \llbracket \Delta \rrbracket \to \llbracket A \rrbracket \\
\llbracket L \bullet M \rrbracket_\Delta &= \mathsf{let}\ l = \llbracket L \rrbracket \bullet \langle\rangle\ \mathsf{in}\ [\lambda\Delta.\, l\ \llbracket M \rrbracket] \\
\llbracket \mathsf{run}\ L \rrbracket_\Delta &= \mathsf{let}\ h = \llbracket L \rrbracket \bullet \langle\rangle\ \mathsf{in}\ [\lambda\Delta.\, h] \\
\llbracket [M] \rrbracket_\Delta &= [\lambda\Delta.\, \llbracket M \rrbracket] \\
\llbracket \mathsf{let}\ x = P\ \mathsf{in}\ Q \rrbracket_\Delta &= \ \mathsf{let}\ p = \llbracket P \rrbracket_\Delta\ \mathsf{in} \\
&\qquad \mathsf{let}\ q = \llbracket Q \rrbracket_{\Delta,x}\ \mathsf{in}\ [\lambda\Delta.\, q\ \langle \Delta, p\ \Delta \rangle]
\end{aligned}$$

$\llbracket \mathcal{S} \rrbracket$ *to static arrows:*

$$\begin{aligned}
\llparenthesis 1 \rightsquigarrow (A \to B) \rrparenthesis &= \llparenthesis A \rrparenthesis \rightsquigarrow \llparenthesis B \rrparenthesis \\
\llparenthesis \lambda^\bullet u.\, P \rrparenthesis &= \lambda^\bullet x.\, \mathsf{let}\ h = \llparenthesis P \rrparenthesis\ \mathsf{in}\ [h\ x]
\end{aligned}$$

where

$$\begin{aligned}
\llparenthesis \Gamma;\, \Delta \vdash P \mathbin{!} A \rrparenthesis &= \llparenthesis \Gamma \rrparenthesis; \llparenthesis \Delta \rrparenthesis \vdash \llparenthesis P \rrparenthesis \mathbin{!} \llparenthesis A \rrparenthesis \\
\llparenthesis L \bullet M \rrparenthesis &= \mathsf{run}\ \llparenthesis L \rrparenthesis \\
\llparenthesis [M] \rrparenthesis &= [\llparenthesis M \rrparenthesis] \\
\llparenthesis \mathsf{let}\ x = P\ \mathsf{in}\ Q \rrparenthesis &= \mathsf{let}\ x = \llparenthesis P \rrparenthesis\ \mathsf{in}\ \llparenthesis Q \rrparenthesis
\end{aligned}$$

Type isomorphism on static arrows: $f : A \simeq A$ is the identity isomorphism.

$$\begin{tikzcd}
\text{idioms } \mathcal{I} \arrow[d, bend left] \arrow[d, phantom, "\sim"] & \\
\text{static arrows } \mathcal{S} \arrow[u, bend left] \arrow[r, hook] \arrow[d, bend left] \arrow[d, phantom, "\cong"] & \text{arrow calculus } \mathcal{A} \arrow[d, bend left] \arrow[d, phantom, "\cong"] \\
\text{classic arrows with delay } \mathcal{C}_{\mathrm{S}} \arrow[u, bend left] & \text{classic arrows } \mathcal{C} \arrow[u, bend left]
\end{tikzcd}$$

In summary: idioms are equationally equivalent to static arrows, which embed into arrow calculus.

6 Relating arrows and monads

In order to compare monads and arrows we consider arrows extended with application. First, we describe Hughes's theory of classic arrows with apply $\mathcal{C}_{\text{app}}$ by adding an extra constant and three laws. We then introduce an extension of the arrow calculus, *higher-order arrows* $\mathcal{H}$, that is in equational correspondence with $\mathcal{C}_{\text{app}}$, by adding one command and two laws.

An arrow with apply permits us to apply an arrow that is itself yielded by another arrow. As explained by Hughes [1] an arrow with apply is equivalent to a monad. It is equipped with an additional constant

$$\mathit{app} : (A \rightsquigarrow B) \times A \rightsquigarrow B$$

which is an arrow analogue of function application.

For the arrow calculus, equivalent structure is provided by a second version of arrow application, where the arrow to apply may itself be computed by an arrow.

$$\frac{\Gamma, \Delta \vdash L : A \rightsquigarrow B \qquad \Gamma, \Delta \vdash M : A}{\Gamma; \Delta \vdash L \star M \; ! \; B}$$

This lifts the central restriction on arrow application. Now the arrow to apply may contain free variables in both Γ and Δ. We therefore dub arrows with apply *promiscuous* (in the broader sense of *undiscriminating*), to highlight the departure from the careful maintenance of the distinction between the two environments in the standard arrow calculus. Indeed, Moggi's metalanguage for monads is exactly like the arrow calculus but with no distinction between Γ and Δ, $\lambda x. N$ and $\lambda^\bullet x.$, or $L\ M$ and $L \bullet M$, and with $A \rightsquigarrow B$ replaced by $A \rightarrow M\,B$.

In this section we re-derive Hughes's result in the arrow calculus to give an equational equivalence between monads and arrows with apply.

Definition 6.1 The theory $\mathcal{C}_{\text{app}}$ of *classic arrows with apply* is the extension of theory $\mathcal{C}$ with the constant

$$\mathit{app} : (A \rightsquigarrow B) \times A \rightsquigarrow B$$

and the additional laws:

$$\begin{array}{lrcl}
(\rightsquigarrow_{H1}) & \mathit{first}\ (\mathit{arr}\ (\lambda x.\ \mathit{arr}\ (\lambda y.\ \langle x, y \rangle))) \ggg \mathit{app} & = & \mathit{arr}\ \mathit{id} \\
(\rightsquigarrow_{H2}) & \mathit{first}\ (\mathit{arr}\ (g \ggg)) \ggg \mathit{app} & = & \mathit{second}\ g \ggg \mathit{app} \\
(\rightsquigarrow_{H3}) & \mathit{first}\ (\mathit{arr}\ (\ggg h)) \ggg \mathit{app} & = & \mathit{app} \ggg h
\end{array}$$

Definition 6.2 The theory $\mathcal{H}$ of higher-order arrows is the extension of the theory $\mathcal{A}$ with an additional syntactic construct given by the typing rule:

$$\frac{\Gamma, \Delta \vdash L : A \leadsto B \qquad \Gamma, \Delta \vdash M : A}{\Gamma; \Delta \vdash L \star M \, ! \, B}$$

with the additional laws:

$$\begin{array}{lrcl} (\beta^{app}) & (\lambda^\bullet x.\, Q) \star M & = & Q[x := M] \\ (\eta^{app}) & \lambda^\bullet x.\, (L \star x) & = & L \end{array}$$

Proposition 6.3 *The theories of higher-order arrows and classic arrows with apply are in equational correspondence: $\mathcal{H} \cong \mathcal{C}_{app}$.*

The translations are each extended with an extra clause.

Higher-order arrows to classic arrows with apply:

$$[\![L \star M]\!]_\Delta = arr\ (\lambda\Delta.\, [\![L]\!] \ \&\&\&\ [\![M]\!]) \ggg app$$

Classic arrows with apply to higher-order arrows:

$$(\!|app|\!) = \lambda^\bullet p.\, (\mathsf{fst}\ p) \star (\mathsf{snd}\ p)$$

Proposition 6.4 *The theories of monads and higher-order arrows are equationally equivalent: $\mathcal{M} \sim \mathcal{H}$.*

Monads to higher-order arrows:

$$\begin{aligned} [\![M\,A]\!] &= 1 \leadsto [\![A]\!] \\ [\![return]\!] &= \lambda x.\, \lambda^\bullet u.\, [x] \\ [\![(\gg\!=)]\!] &= \lambda a.\, \lambda h.\, \lambda^\bullet u.\, \mathsf{let}\ x = a \star \langle\rangle\ \mathsf{in}\ (h\ x) \star \langle\rangle \end{aligned}$$

Higher-order arrows to monads:

$$\begin{aligned} (\!|A \leadsto B|\!) &= (\!|A|\!) \to M\,(\!|B|\!) \\ (\!|\lambda^\bullet x.\, P|\!) &= \lambda x.\, (\!|P|\!) \end{aligned}$$

where

$$\begin{aligned} (\!|\Gamma; \Delta \vdash P \,!\, A|\!) &= (\!|\Gamma, \Delta|\!) \vdash (\!|P|\!) : M\,(\!|A|\!) \\ (\!|L \bullet M|\!) &= (\!|L|\!)\ (\!|M|\!) \\ (\!|L \star M|\!) &= (\!|L|\!)\ (\!|M|\!) \\ (\!|[M]|\!) &= return\ (\!|M|\!) \\ (\!|\mathsf{let}\ x = P\ \mathsf{in}\ Q|\!) &= (\!|P|\!) \gg\!= \lambda x.\, (\!|Q|\!) \end{aligned}$$

Type isomorphism on monads:

$$\begin{aligned} & f_{M(A)} : M\,A \simeq 1 \to M\,A \\ & f_{M(A)} = \lambda a.\, \lambda u.\, a \\ & f^{-1}_{M(A)} = \lambda h.\, h\ \langle\rangle \end{aligned}$$

Type isomorphism on higher-order arrows:

$$\begin{aligned} g_{A \rightsquigarrow B} &: A \rightsquigarrow B \simeq A \to (1 \rightsquigarrow B) \\ g_{A \rightsquigarrow B} &= \lambda a.\, \lambda x.\, \lambda^{\bullet} u.\, a \bullet x \\ g^{-1}_{A \rightsquigarrow B} &= \lambda h.\, \lambda^{\bullet} x.\, (h\ x) \star \langle\rangle \end{aligned}$$

Remark 6.5 The type isomorphism $g_{A \rightsquigarrow B} : A \rightsquigarrow B \simeq A \to (1 \rightsquigarrow B)$ gives an alternative characterisation of higher-order arrows. The type isomorphism and $L \star M$ are inter-definable. The definition of $g^{-1}_{A \rightsquigarrow B}$ in terms of $L \star M$ is given above. The definition of $L \star M$ in terms of $g^{-1}_{A \rightsquigarrow B}$ follows.

$$L \star M \equiv g^{-1}_{(A \rightsquigarrow B) \times A \rightsquigarrow B} (\lambda p.\, \lambda^{\bullet} u.\, (\mathsf{fst}\ p)\ (\mathsf{snd}\ p)) \bullet \langle L, M \rangle$$

Proposition 6.6 *There is an equational embedding of arrow calculus into higher-order arrows: $\mathcal{A} \hookrightarrow \mathcal{H}$.*

The translation $[\![-]\!]$ is the inclusion map from $\mathcal{A}$ to $\mathcal{H}$, the translation $\langle\!|-|\!\rangle$ is the identity on $\mathcal{A}$, and f is the identity isomorphism.

Proof. Clearly, $[\![-]\!]$ preserves equality, as every equation of $\mathcal{A}$ is an equation of $\mathcal{H}$. It remains to show that $\langle\!|-|\!\rangle$ preserves equality.

Both the rewriting theories of the arrow calculus and of higher-order arrows are strongly normalising and confluent [4]. We establish that $\langle\!|-|\!\rangle$ preserves equality by examining normal forms. The normal form for arrow calculus commands is:

$$\mathsf{let}\ x_1 = L_1 \bullet M_1\ \mathsf{in}\ \ldots \mathsf{let}\ x_n = L_n \bullet M_n\ \mathsf{in}\ [N]$$

where $L_1, \ldots{}_n$, $M_1, \ldots, M_n$, N are all in normal form. The normal form for higher-order arrow calculus commands is:

$$\mathsf{let}\ x_1 = L_1 \star M_1\ \mathsf{in}\ \ldots \mathsf{let}\ x_n = L_n \star M_n\ \mathsf{in}\ [N]$$

where $L_1, \ldots{}_n$, $M_1, \ldots, M_n$, N are all in normal form. Since $L \bullet M = L \star M$ it is the case that both $[\![-]\!]$ and $\langle\!|-|\!\rangle$ map distinct normal forms to distinct normal forms, hence $\langle\!|-|\!\rangle$ preserves equality. □

$$\begin{tikzcd}
& \text{monads } \mathcal{M} \arrow[d, bend left] \arrow[d, phantom, "\sim"] \\
\text{arrow calculus } \mathcal{A} \arrow[r, hook] \arrow[d, bend left] \arrow[d, phantom, "\cong"] & \text{higher-order arrows } \mathcal{H} \arrow[u, bend left] \arrow[d, bend left] \arrow[d, phantom, "\cong"] \\
\text{classic arrows } \mathcal{C} \arrow[u, bend left] & \text{classic arrows with application } \mathcal{C}_{\text{app}} \arrow[u, bend left]
\end{tikzcd}$$

In summary: monads are equationally equivalent to higher-order arrows, and there is an equational embedding of arrow calculus into higher-order arrows.

7 Conclusions and future work

We have characterised idioms, monads and arrows as variations on a single calculus, establishing the relative order of strength as *idiom, arrow, monad* in contrast to the putative order of *arrow, idiom, monad.* The variations that bring the arrow calculus into correspondence with idioms and with monads may be characterised either by type isomorphisms or by extensions to the equational theory.

The arrow calculus is the analogue for arrows of Moggi's computational metalanguage [7]. For the future, we plan to investigate analogues for arrows and idioms of the computational lambda calculus [6].

Acknowledgement

We thank Robert Atkey for illuminating discussions, Ezra Cooper, Ben Moseley and Brent Yorgey for comments on a draft of this paper, and Conor McBride and anonymous reviewers for helpful feedback. This work was supported by EPSRC grant number EP/D046769/1.

References

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

[2] Hughes, J., *Programming with arrows*, in: *5th International Summer School in Advanced Functional Programming*, LNCS **3622**, Springer-Verlag, 2005 pp. 73–129.

[3] Lane, S. M., "Categories for the working mathematician," Springer, 1998.

[4] Lindley, S., P. Wadler and J. Yallop, *The arrow calculus*, Technical Report EDI-INF-RR-1258, School of Informatics, University of Edinburgh (2008).

[5] Mcbride, C. and R. Paterson, *Applicative programming with effects*, Journal of Functional Programming **18** (2008), pp. 1–13.

[6] Moggi, E., *Computational lambda-calculus and monads*, in: *Proceedings of the Fourth Annual Symposium on Logic in computer science* (1989), pp. 14–23.

[7] Moggi, E., *Notions of computation and monads*, Information and Computation **93** (1991), pp. 55–92.

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

[9] Paterson, R., *Arrows and computation*, in: J. Gibbons and O. de Moor, editors, *The Fun of Programming*, Palgrave, 2003 pp. 201–222.

[10] *RFC 793* (1981),
http://tools.ietf.org/html/rfc793.

[11] Sabry, A. and M. Felleisen, *Reasoning about programs in continuation-passing style*, LISP and Symbolic Computation **6** (1993), pp. 287–358.