The Arrow Calculus

Sam Lindley, Philip Wadler, and Jeremy Yallop

University of Edinburgh

Abstract. We introduce the arrow calculus, a metalanguage for manipulating Hughes's arrows with close relations both to Moggi's metalanguage for monads and to Paterson's arrow notation.

Arrows are classically defined by extending lambda calculus with three constructs satisfying nine (somewhat idiosyncratic) laws. In contrast, the arrow calculus adds four constructs satisfying five laws. Two of the constructs are arrow abstraction and application (satisfying beta and eta laws) and two correspond to unit and bind for monads (satisfying left unit, right unit, and associativity laws). The five laws were previously known to be sound; we show that they are also complete, and hence that the five laws may replace the nine.

We give a translation from classic arrows into the arrow calculus to complement Paterson's desugaring and show that the two translations form an *equational correspondence* in the sense of Sabry and Felleisen.

We are also the first to publish formal type rules (which are unusual in that they require two contexts), which greatly aided our understanding of arrows.

The first fruit of our new calculus is to reveal some redundancies in the classic formulation: the nine classic arrow laws can be reduced to eight, and the three additional classic arrow laws for arrows with apply can be reduced to two. The calculus has also been used to clarify the relationship between idioms, arrows and monads and as the inspiration for a categorical semantics of arrows.

1 Introduction

Arrows belong in the quiver of every functional programmer, ready to pierce hard problems through their heart.

Arrows [4] generalise the *monads* of Moggi [9] and the *idioms* of McBride and Paterson [8]. They are closely related to *Freyd categories*, discovered independently from Hughes by Power, Robinson and Thielecke [11, 12]. Arrows enjoy a wide range of applications, including parsers and printers [5], web interaction [4], circuits [10], graphic user interfaces [2], and robotics [3].

Formally, arrows are defined by extending simply-typed lambda calculus with three constants satisfying nine laws. And here is where the problems start. While some of the laws are easy to remember, others are not. Further, arrow expressions written with these constants use a 'pointless' style of expression that can be hard to read and to write. (Not to mention that 'pointless' is the last thing arrows should be.)

Fortunately, Paterson introduced a notation for arrows that is easier to read and to write, and in which some arrow laws may be directly expressed [10]. But for all its benefits, Paterson's notation is only a partial solution. It simply abbreviates terms built from the three constants, and there is no claim that its laws are adequate for all reasoning with arrows. Syntactic sugar is an apt metaphor: it sugars the pill, but the pill still must be swallowed.

Here we define the *arrow calculus*, which closely resembles both Paterson's notation for arrows and Moggi's metalanguage for monads. Instead of augmenting simply typed lambda calculus with three constants and nine laws, we augment it with four constructs satisfying five laws. Two of these constructs resemble function abstraction and application, and satisfy beta and eta laws. The remaining two constructs resemble the unit and bind of a monad, and satisfy left unit, right unit, and associativity laws. So instead of nine (somewhat idiosyncratic) laws, we have five laws that fit two well-known patterns.

The arrow calculus is equivalent to the classic formulation. We give a translation of the four constructs into the three constants, and show the five laws follow from the nine. Conversely, we also give a translation of the three constants into the four constructs, and show the nine laws follow from the five. Hence, the arrow calculus is no mere syntactic sugar. Instead of understanding it by translation into the three constants, we can understand the three constants by translating them to it!

Elsewhere, we have already applied the arrow calculus to elucidate the connections between idioms, arrows, and monads [7]. Arrow calculus was not the main focus of that paper, where it was a tool to an end, and that paper has perhaps too terse a description of the calculus. This paper was in fact written before the other, and we hope provides a readable introduction to the arrow calculus.

Our notation is a minor syntactic variation of Paterson's notation, and Paterson's paper contains essentially the same laws we give here. So what is new?

- Paterson translates his notation into classic arrows, and shows the five laws follow from the nine (Soundness). We are the first to give the inverse translation, and show that the nine laws follow from the five (Completeness).

  Completeness is not just a nicety: Paterson regards his notation as syntactic sugar for the classic arrows; completeness lets us claim our calculus can supplant classic arrows.
- We are also the first to publish concise formal type rules. The type rules are unusual in that they involve two contexts, one for variables bound by ordinary lambda abstraction and one for variables bound by arrow abstraction. Discovering the rules greatly improved our understanding of arrows.

  Or rather, we should say *rediscovering*. It turns out that the type rules were known to Paterson, and he used them to implement the arrow notation extension to the Glasgow Haskell Compiler. But Paterson never published the type rules; he explained to us that "Over the years I spent trying to get the arrow notation published, I replaced formal rules with informal descriptions because referees didn't like them." We are glad to help the formal rules finally into print.

- We show the two translations from classic arrows to arrow calculus and back are exactly inverse, providing an *equational correspondence* in the sense of Sabry and Felleisen [13].
  The reader's reaction may be to say, 'Of course the translations are inverses, how could they be otherwise?' But in fact the more common situation is for forward and backward translations to compose to give an isomorphism (category theorists call this an *equivalence* of categories), rather than compose to give the identity on the nose (an *isomorphism* of categories). Lindley, Wadler and Yallop [7] give forward and backward translations between variants of idioms, arrows, and monads, and only some turn out to be equational correspondences; we had to invent a more general notion of *equational equivalence* to characterize the others.
- The first fruit of our new calculus is to reveal a redundancy: the nine classic arrow laws can be reduced to eight. Notation alone was not adequate to lead to this discovery; it flowed from our attempts to show the translations between classic arrows and arrow calculus preserve the laws. We also discovered another redundancy this way: the three additional classic arrow laws for arrows with apply can be reduced to two.
- The arrow calculus has already proven useful in practice. It enabled us to clarify the relationship between idioms, arrows and monads [7]. Further, it provided the inspiration for the categorical semantics of arrows [1].

This paper is organized as follows. Section 2 reviews the classic formulation of arrows. Section 3 introduces the arrow calculus. Section 4 translates the arrow calculus into classic arrows, and vice versa, showing that the laws of each can be derived from the other. Section 5 extends the arrow calculus to the higher-order arrow calculus.

2 Classic arrows

We refer to the classic presentation of arrows as classic arrows, and to our new metalanguage as the arrow calculus.

The core of both is an entirely pedestrian simply-typed lambda calculus with products and functions, as shown in Figure 1. Let A, B, C range over types, L, M, N range over terms, and Γ, Δ range over environments. A type judgment $\Gamma \vdash M : A$ indicates that in environment Γ term M has type A. We use a Curry formulation, eliding types from terms. Products and functions satisfy beta and eta laws. The $(\eta^{\rightarrow})$ law has the usual side-condition, that x is not free in L.

Classic arrows extends lambda calculus with one type and three constants satisfying nine laws, as shown in Figure 2. 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. The three constants are: *arr*, which promotes a function to a pure arrow with no side effects; $\ggg$, which composes two arrows;

Syntax

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

Types

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}$$

$$\frac{\begin{array}{l}\Gamma \vdash M : A \\ \Gamma \vdash N : B\end{array}}{\Gamma \vdash (M, N) : A \times B} \qquad \frac{\Gamma \vdash L : A \times B}{\Gamma \vdash \mathsf{fst}\ L : A} \qquad \frac{\Gamma \vdash L : A \times B}{\Gamma \vdash \mathsf{snd}\ L : B}$$

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

Laws

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

Fig. 1. Lambda calculus

and *first*, which extends an arrow to act on the first component of a pair leaving the second component unchanged. We allow infix notation as usual, writing $M \ggg N$ in place of $(\ggg)\ M\ N$.

The figure defines ten auxiliary functions, all of which are standard. The identity function *id*, selector *fst*, associativity *assoc*, function composition $f \cdot g$, and product bifunctor $f \times g$ are required for the nine laws. Functions *dup* and *swap* are used to define *second*, which is like *first* but acts on the second component of a pair, and $f \mathbin{\&\&\&} g$, which applies arrows f and g to the same argument and pairs the results. We also define the selector *snd*.

The nine laws state that arrow composition has a left and right unit ($\rightsquigarrow_1$,$\rightsquigarrow_2$), arrow composition is associative ($\rightsquigarrow_3$), composition of functions promotes to composition of arrows ($\rightsquigarrow_4$), *first* on pure functions rewrites to a pure function ($\rightsquigarrow_5$), *first* is a homomorphism for composition ($\rightsquigarrow_6$), *first* commutes with a pure function that is the identity on the first component of a pair ($\rightsquigarrow_7$), and *first* pushes through promotions of *fst* and *assoc* ($\rightsquigarrow_8$,$\rightsquigarrow_9$).

Every arrow of interest comes with additional operators, which perform side effects or combine arrows in other ways (such as choice or parallel composition). The story for these additional operators is essentially the same for classic arrows and the arrow calculus, so we say little about them.

Syntax

$$\begin{array}{lll} \text{Types} & A, B, C & ::= \cdots \mid A \rightsquigarrow B \\ \text{Terms} & L, M, N & ::= \cdots \mid \mathit{arr} \mid (\ggg) \mid \mathit{first} \end{array}$$

Types

$$\begin{array}{r} \mathit{arr} : (A \to B) \to (A \rightsquigarrow B) \\ (\ggg) : (A \rightsquigarrow B) \to (B \rightsquigarrow C) \to (A \rightsquigarrow C) \\ \mathit{first} : (A \rightsquigarrow B) \to (A{\times}C \rightsquigarrow B{\times}C) \end{array}$$

Definitions

$$\begin{array}{rlrl} \mathit{id} & : A \to A & (\times) & : (A \to C) \to (B \to D) \to (A{\times}B \to C{\times}D) \\ \mathit{id} & = \lambda x.\, x & (\times) & = \lambda f.\, \lambda g.\, \lambda z.\, (f\ (\mathsf{fst}\ z), g\ (\mathsf{snd}\ z)) \\ \mathit{fst} & : A{\times}B \to A & \mathit{assoc} & : (A{\times}B){\times}C \to A{\times}(B{\times}C) \\ \mathit{fst} & = \lambda z.\, \mathsf{fst}\ z & \mathit{assoc} & = \lambda z.\, (\mathsf{fst}\ (\mathsf{fst}\ z), (\mathsf{snd}\ (\mathsf{fst}\ z), \mathsf{snd}\ z)) \\ \mathit{snd} & : A{\times}B \to B & (\cdot) & : (B \to C) \to (A \to B) \to (A \to C) \\ \mathit{snd} & = \lambda z.\, \mathsf{snd}\ z & (\cdot) & = \lambda f.\, \lambda g.\, \lambda x.\, f\ (g\ x) \\ \mathit{dup} & : A \to A{\times}A & \mathit{second} & : (A \rightsquigarrow B) \to (C{\times}A \rightsquigarrow C{\times}B) \\ \mathit{dup} & = \lambda x.\, (x, x) & \mathit{second} & = \lambda f.\, \mathit{arr}\ \mathit{swap} \ggg \mathit{first}\ f \ggg \mathit{arr}\ \mathit{swap} \\ \mathit{swap} & : A{\times}B \to B{\times}A & (\&\&\&) & : (C \rightsquigarrow A) \to (C \rightsquigarrow B) \to (C \rightsquigarrow A{\times}B) \\ \mathit{swap} & = \lambda z.\, (\mathsf{snd}\ z, \mathsf{fst}\ z) & (\&\&\&) & = \lambda f.\, \lambda g.\, \mathit{arr}\ \mathit{dup} \ggg \mathit{first}\ f \ggg \mathit{second}\ g \end{array}$$

Laws

$$\begin{array}{lc} (\rightsquigarrow_1) & \mathit{arr}\ \mathit{id} \ggg f = f \\ (\rightsquigarrow_2) & f \ggg \mathit{arr}\ \mathit{id} = f \\ (\rightsquigarrow_3) & (f \ggg g) \ggg h = f \ggg (g \ggg h) \\ (\rightsquigarrow_4) & \mathit{arr}\ (g \cdot f) = \mathit{arr}\ f \ggg \mathit{arr}\ g \\ (\rightsquigarrow_5) & \mathit{first}\ (\mathit{arr}\ f) = \mathit{arr}\ (f \times \mathit{id}) \\ (\rightsquigarrow_6) & \mathit{first}\ (f \ggg g) = \mathit{first}\ f \ggg \mathit{first}\ g \\ (\rightsquigarrow_7) & \mathit{first}\ f \ggg \mathit{arr}\ (\mathit{id} \times g) = \mathit{arr}\ (\mathit{id} \times g) \ggg \mathit{first}\ f \\ (\rightsquigarrow_8) & \mathit{first}\ f \ggg \mathit{arr}\ \mathit{fst} = \mathit{arr}\ \mathit{fst} \ggg f \\ (\rightsquigarrow_9) & \mathit{first}\ (\mathit{first}\ f) \ggg \mathit{arr}\ \mathit{assoc} = \mathit{arr}\ \mathit{assoc} \ggg \mathit{first}\ f \end{array}$$

Fig. 2. Classic arrows

3 The arrow calculus

Arrow calculus extends the core lambda calculus with four constructs satisfying five laws, as shown in Figure 3. As before, 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.

We now have two syntactic categories. Terms, as before, are ranged over by L, M, N, and commands are ranged over by P, Q, R. In addition to the terms of the core lambda calculus, there is one new term form: arrow abstraction $\lambda^{\bullet} x.\, Q$. There are three command forms: arrow application $L \bullet M$, arrow unit $[M]$ (which resembles unit in a monad), and arrow bind $\mathsf{let}\ x = P\ \mathsf{in}\ Q$ (which resembles bind in a monad).

Syntax

Types	A, B, C	$::= \cdots \mid A \rightsquigarrow B$
Terms	L, M, N	$::= \cdots \mid \lambda^{\bullet} x.\, Q$
Commands	P, Q, R	$::= L \bullet M \mid [M] \mid \mathsf{let}\ x = P\ \mathsf{in}\ Q$

Types

$$\frac{\Gamma;\ x : A \vdash Q \mathrel{!} B}{\Gamma \vdash \lambda^{\bullet} x.\, Q : A \rightsquigarrow B} \qquad \frac{\begin{array}{l}\Gamma \vdash L : A \rightsquigarrow B \\ \Gamma,\ \Delta \vdash M : A\end{array}}{\Gamma;\ \Delta \vdash L \bullet M \mathrel{!} B}$$

$$\frac{\Gamma,\ \Delta \vdash M : A}{\Gamma;\ \Delta \vdash [M] \mathrel{!} A} \qquad \frac{\begin{array}{l}\Gamma;\ \Delta \vdash P \mathrel{!} A \\ \Gamma;\ \Delta,\ x : A \vdash Q \mathrel{!} B\end{array}}{\Gamma;\ \Delta \vdash \mathsf{let}\ x = P\ \mathsf{in}\ Q \mathrel{!} B}$$

Laws

$(\beta^{\rightsquigarrow})$	$(\lambda^{\bullet} x.\, Q) \bullet M = Q[x := M]$
$(\eta^{\rightsquigarrow})$	$\lambda^{\bullet} x.\, (L \bullet x) = L$
(left)	$\mathsf{let}\ x = [M]\ \mathsf{in}\ Q = Q[x := M]$
(right)	$\mathsf{let}\ x = P\ \mathsf{in}\ [x] = P$
(assoc)	$\mathsf{let}\ y = (\mathsf{let}\ x = P\ \mathsf{in}\ Q)\ \mathsf{in}\ R = \mathsf{let}\ x = P\ \mathsf{in}\ (\mathsf{let}\ y = Q\ \mathsf{in}\ R)$

Fig. 3. Arrow calculus

In addition to the term typing judgment

$$\Gamma \vdash M : A.$$

we now also have a command typing judgment

$$\Gamma;\ \Delta \vdash P \mathrel{!} A.$$

An important feature of the arrow calculus is that the command type judgment has 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 $\lambda^{\bullet} x.\, Q$.

We will give a translation of commands to classic arrows, such that a command P satisfying the judgment

$$\Gamma;\ \Delta \vdash P \mathrel{!} A$$

translates to a term $[\![P]\!]_{\Delta}$ satisfying the judgment

$$\Gamma \vdash [\![P]\!]_{\Delta} : \Delta \rightsquigarrow A.$$

That is, the command P denotes an arrow, taking an argument of type Δ and returning a result of type A. We explain this translation further in Section 4.

Here are the type rules for the four constructs. Arrow abstraction converts a command into a term.

$$\frac{\Gamma;\ x : A \vdash Q \mathrel{!} B}{\Gamma \vdash \lambda^{\bullet} x.\, Q : A \rightsquigarrow B}$$

Arrow abstraction closely resembles function abstraction, save that the body Q is a command (rather than a term) and the bound variable x goes into the second environment (separated from the first by a semicolon).

Conversely, arrow application embeds a term into a command.

$$\frac{\begin{array}{l}\Gamma \vdash L : A \rightsquigarrow B \\ \Gamma, \Delta \vdash M : A\end{array}}{\Gamma; \Delta \vdash L \bullet M \mathbin{!} B}$$

Arrow application closely resembles function application. The arrow to be applied is denoted by a term, not a command; this is because there is no way to apply an arrow that is itself yielded by another arrow. This is why we distinguish two environments, Γ and Δ: variables in Γ may be used to compute arrows that are applied to arguments, but variables in Δ may not.

Arrow unit promotes a term to a command.

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

Note that in the hypothesis we have a term judgment with one environment (there is a comma between Γ and Δ), while in the conclusion we have a command judgment with two environments (there is a semicolon between Γ and Δ).

Lastly, the value returned by a command may be bound.

$$\frac{\begin{array}{l}\Gamma; \Delta \vdash P \mathbin{!} A \\ \Gamma; \Delta, x : A \vdash Q \mathbin{!} B\end{array}}{\Gamma; \Delta \vdash \mathsf{let}\ x = P\ \mathsf{in}\ Q \mathbin{!} B}$$

This resembles a traditional let term, save that the bound variable goes into the second environment, not the first.

Arrow abstraction and application satisfy beta and eta laws, ($\beta^{\rightsquigarrow}$) and ($\eta^{\rightsquigarrow}$), while arrow unit and bind satisfy left unit, right unit, and associativity laws, (left), (right), and (assoc). Similar laws appear in the computational lambda calculus of Moggi [9]. The (assoc) law has the usual side condition, that x is not free in R. We do not require a side condition for ($\eta^{\rightsquigarrow}$), because the type rules guarantee that x does not appear free in L.

Paterson's notation is closely related to ours. Here is a translation table, with our notation on the left and his on the right.

$\lambda^{\bullet} x.\, Q$	$\mathsf{proc}\ x \to Q$
$L \bullet M$	$L \;\prec\; M$
$[M]$	$\mathit{arrowA} \;\prec\; M$
$\mathsf{let}\ x = P\ \mathsf{in}\ Q$	$\mathsf{do}\ \{x \leftarrow P; Q\}$

In essence, each is a minor syntactic variant of the other. The only difference of note is that we introduce arrow unit as an explicit construct, $[M]$, while Paterson uses the equivalent form $\mathit{arrowA} \prec M$ where arrowA is *arr id*. Our introduction of a separate construct for arrow unit is slightly neater, and avoids the need to introduce *arrowA* as a constant in the arrow calculus.

Translation

$$\begin{aligned}
[\![x]\!] &= x\\
[\![(M,N)]\!] &= ([\![M]\!],[\![N]\!])\\
[\![\mathsf{fst}\ L]\!] &= \mathsf{fst}\ [\![L]\!]\\
[\![\mathsf{snd}\ L]\!] &= \mathsf{snd}\ [\![L]\!]\\
[\![\lambda x.\,N]\!] &= \lambda x.\,[\![N]\!]\\
[\![L\ M]\!] &= [\![L]\!]\ [\![M]\!]\\
[\![\lambda^{\bullet} x.\,Q]\!] &= [\![Q]\!]_x
\end{aligned}$$

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

Translation preserves types

$$\left[\!\!\left[\frac{\Gamma;\, x : A \vdash Q\ !\ B}{\Gamma \vdash \lambda^{\bullet} x.\,Q : A \rightsquigarrow B}\right]\!\!\right] = \frac{\Gamma \vdash [\![Q]\!]_x : A \rightsquigarrow B}{\Gamma \vdash [\![Q]\!]_x : A \rightsquigarrow B}$$

$$\left[\!\!\left[\frac{\begin{array}{l}\Gamma \vdash L : A \rightsquigarrow B\\ \Gamma,\, \Delta \vdash M : A\end{array}}{\Gamma;\, \Delta \vdash L \bullet M\ !\ B}\right]\!\!\right] = \frac{\begin{array}{l}\Gamma \vdash [\![L]\!] : A \rightsquigarrow B\\ \Gamma,\, \Delta \vdash [\![M]\!] : A\end{array}}{\Gamma \vdash \mathit{arr}\ (\lambda\Delta.\,[\![M]\!]) \ggg [\![L]\!] : \Delta \rightsquigarrow B}$$

$$\left[\!\!\left[\frac{\Gamma,\, \Delta \vdash M : A}{\Gamma;\, \Delta \vdash [M]\ !\ A}\right]\!\!\right] = \frac{\Gamma,\, \Delta \vdash [\![M]\!] : A}{\Gamma \vdash \mathit{arr}\ (\lambda\Delta.\,[\![M]\!]) : \Delta \rightsquigarrow A}$$

$$\left[\!\!\left[\frac{\begin{array}{l}\Gamma;\, \Delta \vdash P\ !\ A\\ \Gamma;\, \Delta,\, x : A \vdash Q\ !\ B\end{array}}{\Gamma;\, \Delta \vdash \mathsf{let}\ x = P\ \mathsf{in}\ Q\ !\ B}\right]\!\!\right] = \frac{\begin{array}{l}\Gamma \vdash [\![P]\!]_{\Delta} : \Delta \rightsquigarrow A\\ \Gamma \vdash [\![Q]\!]_{\Delta,x} : \Delta{\times}A \rightsquigarrow B\end{array}}{\Gamma \vdash (\mathit{arr}\ \mathit{id} \,\&\&\&\, [\![P]\!]_{\Delta}) \ggg [\![Q]\!]_{\Delta,x} : \Delta \rightsquigarrow B}$$

Fig. 4. Translating Arrow Calculus into Classic Arrows

Translation

$$\begin{aligned}
[\![x]\!]^{-1} &= x\\
[\![(M,N)]\!]^{-1} &= ([\![M]\!]^{-1},[\![N]\!]^{-1})\\
[\![\mathsf{fst}\ L]\!]^{-1} &= \mathsf{fst}\ [\![L]\!]^{-1}\\
[\![\mathsf{snd}\ L]\!]^{-1} &= \mathsf{snd}\ [\![L]\!]^{-1}\\
[\![\lambda x.\,N]\!]^{-1} &= \lambda x.\,[\![N]\!]^{-1}\\
[\![L\ M]\!]^{-1} &= [\![L]\!]^{-1}\ [\![M]\!]^{-1}\\
[\![\mathit{arr}]\!]^{-1} &= \lambda f.\,\lambda^{\bullet} x.\,[f\ x]\\
[\![(\ggg)]\!]^{-1} &= \lambda f.\,\lambda g.\,\lambda^{\bullet} x.\,\mathsf{let}\ y = f \bullet x\ \mathsf{in}\ g \bullet y\\
[\![\mathit{first}]\!]^{-1} &= \lambda f.\,\lambda^{\bullet} z.\,\mathsf{let}\ x = f \bullet \mathsf{fst}\ z\ \mathsf{in}\ [(x,\mathsf{snd}\ z)]
\end{aligned}$$

Fig. 5. Translating Classic Arrows into Arrow Calculus

4 Translations

We now consider translations between our two formulations, and show they are equivalent.

The translation from the arrow calculus into classic arrows is shown in Figure 4. An arrow calculus term judgment

$$\Gamma \vdash M : A$$

maps into a classic arrow judgment

$$\Gamma \vdash [\![M]\!] : A$$

while an arrow calculus command judgment

$$\Gamma;\, \Delta \vdash P \mathbin{!} A$$

maps into a classic arrow judgment

$$\Gamma \vdash [\![P]\!]_\Delta : \Delta \leadsto A.$$

In $[\![P]\!]_\Delta$, we take Δ to stand for the sequence of variables in the environment, and in $\Delta \leadsto A$ we take Δ to stand for the product of the types in the environment. Hence, the denotation of a command is an arrow, with arguments corresponding to the environment Δ and result of type A.

The translation of the constructs of the core lambda calculus are straightforward homomorphisms. The translations of the remaining four constructs are shown twice, in the top half of the figure as equations on syntax, and in the bottom half in the context of type derivations; the latter are longer, but may be easier to understand. We comment briefly on each of the four:

- $\lambda^\bullet x.\, Q$ translates straightforwardly; it is a no-op. Notice that the correpronding introduction rule is redundant once translated into the classical setting.
- $L \bullet M$ translates to $\ggg$.
- $[M]$ translates to *arr*.
- $\mathsf{let}\ x = P\ \mathsf{in}\ Q$ translates to pairing &&& (to extend the environment with P) and composition $\ggg$ (to then apply Q). The pairing operator &&& is defined in Figure 2.

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

The inverse translation, from classic arrows to the arrow calculus, is given in Figure 5. Again, the translation of the constructs of the core lambda calculus are straightforward homomorphisms. Each of the three constants translates to an appropriate term in the arrow calculus.

We can now show the following four properties.

- The five laws of the arrow calculus follow from the nine laws of classic arrows. That is,

$$M = N \;\; \text{implies} \;\; [\![M]\!] = [\![N]\!]$$
$$\text{and}$$
$$P = Q \;\; \text{implies} \;\; [\![P]\!]_{\Delta} = [\![Q]\!]_{\Delta}$$

  for all arrow calculus terms M, N and commands P, Q. The proof requires five calculations, one for each law of the arrow calculus. Figure 6 shows one of these, the calculation to derive (right) from the classic arrow laws.
- The nine laws of classic arrows follow from the five laws of the arrow calculus. That is,

$$M = N \;\; \text{implies} \;\; [\![M]\!]^{-1} = [\![N]\!]^{-1}$$

  for all classic arrow terms M, N. The proof requires nine calculations, one for each classic arrow law. Figure 7 shows one of these, the calculation to derive ($\rightsquigarrow_2$) from the laws of the arrow calculus.
- Translating from the arrow calculus into classic arrows and back again is the identity on terms. That is,

$$[\![\, [\![M]\!] \,]\!]^{-1} = M$$

  for all arrow calculus terms M. Translating a command of the arrow calculus into classic arrows and back again cannot be the identity, because the back translation yields a term rather than a command, but it does yield the term that is the arrow abstraction of the original command. That is,

$$[\![\, [\![P]\!]_{\Delta} \,]\!]^{-1} = \lambda^{\bullet}\Delta.\, P$$

  for all arrow calculus commands P. The proof requires four calculations, one for each construct of the arrow calculus.
- Translating from classic arrows into the arrow calculus and back again is the identity. That is,

$$[\![\, [\![M]\!]^{-1} \,]\!] = M$$

  for all classic arrow terms M. The proof requires three calculations, one for each classic arrow constant. Figure 8 shows one of these, the calculation for *first*.

These four properties together constitute an *equational correspondence* between classic arrows and the arrow calculus [13]. The full details of the proof appear in a companion technical report [6].

A look at Figure 6 reveals a mild surprise: ($\rightsquigarrow_2$), the right unit law of classic arrows, is not required to prove (right), the right unit law of the arrow calculus. Further, it turns out that ($\rightsquigarrow_2$) is also not required to prove the other four laws. But this is a big surprise! From the classic arrow laws—excluding ($\rightsquigarrow_2$)—we can prove the laws of the arrow calculus, and from these we can in turn prove the classic arrow laws—including ($\rightsquigarrow_2$). It follows that ($\rightsquigarrow_2$) must be redundant.

Once the arrow calculus provided the insight, it was not hard to find a direct proof of redundancy, as presented in Figure 9. We believe we are the first to observe that the nine classic arrow laws can be reduced to eight.

$$
\begin{array}{ll}
& [\![\mathsf{let}\ x = M\ \mathsf{in}\ [x]]\!]_\Delta \\
= & \qquad \text{def } [\![-]\!] \\
& (arr\ id \mathbin{\&\&\&} [\![M]\!]_\Delta) \ggg arr\ snd \\
= & \qquad \text{def } \&\&\& \\
& arr\ dup \ggg first\ (arr\ id) \ggg second\ [\![M]\!]_\Delta \ggg arr\ snd \\
= & \qquad (\rightsquigarrow_5) \\
& arr\ dup \ggg arr\ (id \times id) \ggg second\ [\![M]\!]_\Delta \ggg arr\ snd \\
= & \qquad id \times id = id \\
& arr\ dup \ggg arr\ id \ggg second\ [\![M]\!]_\Delta \ggg arr\ snd \\
= & \qquad (\rightsquigarrow_1) \\
& arr\ dup \ggg second\ [\![M]\!]_\Delta \ggg arr\ snd \\
= & \qquad \text{def } second \\
& arr\ dup \ggg arr\ swap \ggg first\ [\![M]\!]_\Delta \ggg arr\ swap \ggg arr\ snd \\
= & \qquad (\rightsquigarrow_4) \\
& arr\ (swap \cdot dup) \ggg first\ [\![M]\!]_\Delta \ggg arr\ (snd \cdot swap) \\
= & \qquad swap \cdot dup = dup,\ snd \cdot swap = fst \\
& arr\ dup \ggg first\ [\![M]\!]_\Delta \ggg arr\ fst \\
= & \qquad (\rightsquigarrow_8) \\
& arr\ dup \ggg arr\ fst \ggg [\![M]\!]_\Delta \\
= & \qquad (\rightsquigarrow_4) \\
& arr\ (fst \cdot dup) \ggg [\![M]\!]_\Delta \\
= & \qquad fst \cdot dup = id \\
& arr\ id \ggg [\![M]\!]_\Delta \\
= & \qquad (\rightsquigarrow_1) \\
& [\![M]\!]_\Delta
\end{array}
$$

Fig. 6. Proof of (right) from classic arrows

$$
\begin{array}{ll}
& [\![f \ggg arr\ id]\!]^{-1} \\
= & \qquad \text{def } [\![-]\!]^{-1} \\
& \lambda^\bullet x.\ \mathsf{let}\ y = f \bullet x\ \mathsf{in}\ (\lambda^\bullet z.\ [id\ z]) \bullet y \\
= & \qquad (\beta^{\rightarrow}) \\
& \lambda^\bullet x.\ \mathsf{let}\ y = f \bullet x\ \mathsf{in}\ (\lambda^\bullet z.\ [z]) \bullet y \\
= & \qquad (\beta^{\rightsquigarrow}) \\
& \lambda^\bullet x.\ \mathsf{let}\ y = f \bullet x\ \mathsf{in}\ [y] \\
= & \qquad \text{(right)} \\
& \lambda^\bullet x.\ f \bullet x \\
= & \qquad (\eta^{\rightsquigarrow}) \\
& f \\
= & \qquad \text{def } [\![-]\!]^{-1} \\
& [\![f]\!]^{-1}
\end{array}
$$

Fig. 7. Proof of ($\rightsquigarrow_2$) in arrow calculus

$$
\begin{array}{ll}
 & \llbracket\, \llbracket \mathit{first}\ f \rrbracket^{-1} \,\rrbracket \\
= & \qquad \text{def } \llbracket - \rrbracket^{-1}, (\beta^{\rightarrow}) \\
 & \llbracket\, \lambda^{\bullet} z.\ \mathsf{let}\ x = f \bullet (\mathsf{fst}\ z)\ \mathsf{in}\ [(x, \mathsf{snd}\ z)] \,\rrbracket \\
= & \qquad \text{def } \llbracket - \rrbracket \\
 & (\mathit{arr}\ \mathit{id} \mathbin{\&\&\&} (\mathit{arr}\ (\lambda u.\ \mathsf{fst}\ u) \ggg f)) \ggg \mathit{arr}\ (\lambda v.\ (\mathsf{snd}\ v, \mathsf{snd}\ (\mathsf{fst}\ v))) \\
= & \qquad \text{def } \mathit{fst} \\
 & (\mathit{arr}\ \mathit{id} \mathbin{\&\&\&} (\mathit{arr}\ \mathit{fst} \ggg f)) \ggg \mathit{arr}\ (\lambda v.\ (\mathsf{snd}\ v, \mathsf{snd}\ (\mathsf{fst}\ v))) \\
= & \qquad \text{def } \&\&\& \\
 & \mathit{arr}\ \mathit{dup} \ggg \mathit{first}\ (\mathit{arr}\ \mathit{id}) \ggg \mathit{second}\ (\mathit{arr}\ \mathit{fst} \ggg f) \ggg \\
 & \qquad\qquad\qquad\qquad\qquad\qquad \mathit{arr}\ (\lambda v.\ (\mathsf{snd}\ v, \mathsf{snd}\ (\mathsf{fst}\ v))) \\
= & \qquad (\rightsquigarrow_5) \\
 & \mathit{arr}\ \mathit{dup} \ggg \mathit{arr}\ (\mathit{id} \times \mathit{id}) \ggg \mathit{second}\ (\mathit{arr}\ \mathit{fst} \ggg f) \ggg \\
 & \qquad\qquad\qquad\qquad\qquad\qquad \mathit{arr}\ (\lambda v.\ (\mathsf{snd}\ v, \mathsf{snd}\ (\mathsf{fst}\ v))) \\
= & \qquad \mathit{id} \times \mathit{id} = \mathit{id} \\
 & \mathit{arr}\ \mathit{dup} \ggg (\mathit{arr}\ \mathit{id}) \ggg \mathit{second}\ (\mathit{arr}\ \mathit{fst} \ggg f) \ggg \mathit{arr}\ (\lambda v.\ (\mathsf{snd}\ v, \mathsf{snd}\ (\mathsf{fst}\ v))) \\
= & \qquad (\rightsquigarrow_1) \\
 & \mathit{arr}\ \mathit{dup} \ggg \mathit{second}\ (\mathit{arr}\ \mathit{fst} \ggg f) \ggg \mathit{arr}\ (\lambda v.\ (\mathsf{snd}\ v, \mathsf{snd}\ (\mathsf{fst}\ v))) \\
= & \qquad \text{def } \mathit{second} \\
 & \mathit{arr}\ \mathit{dup} \ggg \mathit{arr}\ \mathit{swap} \ggg \mathit{first}\ (\mathit{arr}\ \mathit{fst} \ggg f) \ggg \mathit{arr}\ \mathit{swap} \ggg \\
 & \qquad\qquad\qquad\qquad\qquad\qquad \mathit{arr}\ (\lambda v.\ (\mathsf{snd}\ v, \mathsf{snd}\ (\mathsf{fst}\ v))) \\
= & \qquad (\rightsquigarrow_4) \\
 & \mathit{arr}\ (\mathit{swap} \cdot \mathit{dup}) \ggg \mathit{first}\ (\mathit{arr}\ \mathit{fst} \ggg f) \ggg \mathit{arr}\ \mathit{swap} \ggg \\
 & \qquad\qquad\qquad\qquad\qquad\qquad \mathit{arr}\ (\lambda v.\ (\mathsf{snd}\ v, \mathsf{snd}\ (\mathsf{fst}\ v))) \\
= & \qquad \mathit{swap} \cdot \mathit{dup} = \mathit{dup} \\
 & \mathit{arr}\ \mathit{dup} \ggg \mathit{first}\ (\mathit{arr}\ \mathit{fst} \ggg f) \ggg \mathit{arr}\ \mathit{swap} \ggg \mathit{arr}\ (\lambda v.\ (\mathsf{snd}\ v, \mathsf{snd}\ (\mathsf{fst}\ v))) \\
= & \qquad (\rightsquigarrow_4) \\
 & \mathit{arr}\ \mathit{dup} \ggg \mathit{first}\ (\mathit{arr}\ \mathit{fst} \ggg f) \ggg \mathit{arr}\ ((\lambda v.\ (\mathsf{snd}\ v, \mathsf{snd}\ (\mathsf{fst}\ v))) \cdot \mathit{swap}) \\
= & \qquad (\lambda v.\ (\mathsf{snd}\ v, \mathsf{snd}\ (\mathsf{fst}\ v))) \cdot \mathit{swap} = \mathit{id} \times \mathit{snd} \\
 & \mathit{arr}\ \mathit{dup} \ggg \mathit{first}\ (\mathit{arr}\ \mathit{fst} \ggg f) \ggg \mathit{arr}\ (\mathit{id} \times \mathit{snd}) \\
= & \qquad (\rightsquigarrow_6) \\
 & \mathit{arr}\ \mathit{dup} \ggg \mathit{first}\ (\mathit{arr}\ \mathit{fst}) \ggg \mathit{first}\ f \ggg \mathit{arr}\ (\mathit{id} \times \mathit{snd}) \\
= & \qquad (\rightsquigarrow_5) \\
 & \mathit{arr}\ \mathit{dup} \ggg \mathit{arr}\ (\mathit{fst} \times \mathit{id}) \ggg \mathit{first}\ f \ggg \mathit{arr}\ (\mathit{id} \times \mathit{snd}) \\
= & \qquad (\rightsquigarrow_7) \\
 & \mathit{arr}\ \mathit{dup} \ggg \mathit{arr}\ (\mathit{fst} \times \mathit{id}) \ggg \mathit{arr}\ (\mathit{id} \times \mathit{snd}) \ggg \mathit{first}\ f \\
= & \qquad (\rightsquigarrow_4) \\
 & \mathit{arr}\ ((\mathit{id} \times \mathit{snd}) \cdot (\mathit{fst} \times \mathit{id}) \cdot \mathit{dup}) \ggg \mathit{first}\ f \\
= & \qquad (\mathit{id} \times \mathit{snd}) \cdot (\mathit{fst} \times \mathit{id}) \cdot \mathit{dup} = \mathit{id} \\
 & \mathit{arr}\ \mathit{id} \ggg \mathit{first}\ f \\
= & \qquad (\rightsquigarrow_1) \\
 & \mathit{first}\ f
\end{array}
$$

Fig. 8. Translating *first* to arrow calculus and back is the identity

$$
\begin{array}{ll}
 & f \ggg arr\ id \\
= & \qquad (\rightsquigarrow_1) \\
 & arr\ id \ggg f \ggg arr\ id \\
= & \qquad fst \cdot dup = id \\
 & arr\ (fst \cdot dup) \ggg f \ggg arr\ id \\
= & \qquad (\rightsquigarrow_4) \\
 & arr\ dup \ggg arr\ fst \ggg f \ggg arr\ id \\
= & \qquad (\rightsquigarrow_8) \\
 & arr\ dup \ggg first\ f \ggg arr\ fst \ggg arr\ id \\
= & \qquad (\rightsquigarrow_4) \\
 & arr\ dup \ggg first\ f \ggg arr\ (id \cdot fst) \\
= & \qquad id \cdot fst = fst \\
 & arr\ dup \ggg first\ f \ggg arr\ fst \\
= & \qquad (\rightsquigarrow_8) \\
 & arr\ dup \ggg arr\ fst \ggg f \\
= & \qquad (\rightsquigarrow_4) \\
 & arr\ (fst \cdot dup) \ggg f \\
= & \qquad fst \cdot dup = id \\
 & arr\ id \ggg f \\
= & \qquad (\rightsquigarrow_1) \\
 & f
\end{array}
$$

Fig. 9. Proof that ($\rightsquigarrow_2$) is redundant

5 Higher-order arrows

Arrows generalise a whole range of different notions of computation. We can obtain specific kinds of arrows by adding extra syntax and extra laws to the arrow calculus. Here we describe how to capture *higher-order* computation. One might also capture other variants of arrows such as static arrows, arrows with choice and arrows with fixed points.

A *higher-order arrow* permits us to *apply* an arrow that is itself yielded by another arrow. As explained by Hughes [4], a classic arrow with apply is equivalent to a monad. It is equipped with an additional constant

$$app : (A \rightsquigarrow B) \times A \rightsquigarrow B$$

which is an arrow analogue of function application. For the higher-order 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.

$$\frac{\begin{array}{l}\Gamma, \Delta \vdash L : A \rightsquigarrow B \\ \Gamma, \Delta \vdash M : A\end{array}}{\Gamma, \Delta \vdash L \star M \ ! \ B}$$

This lifts the central restriction on arrow application. Now the arrow to apply may be the result of a command, and the command denoting the arrow may contain free variables in both Γ and Δ.

For classic arrows the additional laws for arrows with apply are:

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

For the higher-order arrow calculus the additional laws are simply the beta and eta laws for $\star$:

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

One small subtlety deserves mention: it appears that the β^{app} law does not preserve well-typing. Consider the following term.

$$\lambda^\bullet f.\, (\lambda^\bullet x.\, f \bullet x) \star M$$

This can be assigned a type, but under the equation β^{app} it is apparently equal to the following term, which cannot.

$$\lambda^\bullet f.\, f \bullet M$$

In fact, there is no problem: the equational judgment on commands is only defined when both sides can be assigned types.

Remark The η^{app} law is in fact redundant.

$$\begin{array}{ll}
 & \lambda^\bullet x.\, L \star x \\
= & \qquad (\eta^{\rightsquigarrow}) \\
 & \lambda^\bullet x.\, (\lambda^\bullet y.\, L \bullet y) \star x \\
= & \qquad (\beta^{app}) \\
 & \lambda^\bullet x.\, L \bullet x \\
= & \qquad (\eta^{\rightsquigarrow}) \\
 & L
\end{array}$$

These extensions to classic arrows and the arrow calculus maintain the equational correspondence. The translations are each extended with an extra clause. Higher-order arrow calculus to classic arrows with apply:

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

Classic arrows with apply to higher-order arrow calculus:

$$[\![\mathit{app}]\!]^{-1} = \lambda^\bullet p.\, (\mathsf{fst}\ p) \star (\mathsf{snd}\ p)$$

Once again, the proof (see the accompanying technical report [6] for details) reveals a surprise: $(\rightsquigarrow_{H2})$, is not required to prove (β^{app}). From the classic laws—with apply but excluding $(\rightsquigarrow_{H2})$—we can prove the laws of higher-order arrow calculus, and from these we can in turn prove the classic laws—including $(\rightsquigarrow_{H2})$. It follows that $(\rightsquigarrow_{H2})$ must be redundant.

We believe we are the first to observe that the three classic arrow laws for *app* can be reduced to two.

Acknowledgements

Our thanks to Robert Atkey, Samuel Bronson, John Hughes, and Ross Paterson.

References

1. R. Atkey. What is a categorical model of arrows? In *Mathematical Structures in Functional Programming*, ENTCS, 2008.
2. A. Courtney and C. Elliott. Genuinely functional user interfaces. In *Haskell Workshop*, pages 41–69, Sept. 2001.
3. P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. In J. Jeuring and S. P. Jones, editors, *Advanced Functional Programming, 4th International School*, volume 2638 of *LNCS*. Springer-Verlag, 2003.
4. J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.
5. P. Jansson and J. Jeuring. Polytypic compact printing and parsing. In *European Symposium on Programming*, volume 1576 of *LNCS*, pages 273–287. Springer-Verlag, 1999.
6. S. Lindley, P. Wadler, and J. Yallop. The arrow calculus. Technical Report EDI-INF-RR-1258, School of Informatics, University of Edinburgh, 2008.
7. S. Lindley, P. Wadler, and J. Yallop. Idioms are oblivious, arrows are meticulous, monads are promiscuous. In *Mathematical Structures in Functional Programming*, ENTCS, 2008.
8. C. McBride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.
9. E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
10. R. Paterson. A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240. ACM Press, Sept. 2001.
11. J. Power and E. Robinson. Premonoidal categories and notions of computation. *Mathematical Structures in Computer Science*, 7(5):453–468, Oct. 1997.
12. J. Power and H. Thielecke. Closed Freyd- and kappa-categories. In *ICALP*, volume 1644 of *LNCS*. Springer, 1999.
13. A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3/4):289–360, 1993.