

The Arrow Calculus (Functional Pearl)

Sam Lindley Philip Wadler 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.

1. Introduction

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

Arrows were discovered independently twice. Hughes (2000) coined the name *arrow*, while Power and Thielecke (1999) used the name *Freyd categories*. Arrows generalise the *monads* of Moggi (1991) and the *idioms* of McBride and Paterson (2008). Arrows enjoy a wide range of applications, including parsers and printers (Jansson and Jeuring 1999), web interaction (Hughes 2000), circuits (Paterson 2001), graphic user interfaces (Courtney and Elliott 2001), and robotics (Hudak et al. 2003).

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.

Fortunately, Paterson (2001) introduced a notation for arrows that is easier to read and to write, and in which some arrow laws may be directly expressed. 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!

We show the two translations are exactly inverse, providing an *equational correspondence* in the sense of Sabry and Felleisen (1993). The first fruit of the new calculus will be to uncover a previously unknown fact about the classic nine laws. We also sketch how to extend the arrow calculus to deal with additional structure on arrows, such as arrows with choice or arrows with apply.

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.

Classic arrows extends the core 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; \gg , which composes two arrows; and *fst*, 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 $M \gg N$ in place of $(\gg) M N$.

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 *fst* but acts on the second component of a pair, and $f \&\&g$, 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), *fst* on pure functions rewrites to composition of arrows (\rightsquigarrow_5), *fst* is a homomorphism for composition (\rightsquigarrow_6), *fst* commutes with a pure function that is the identity on the first component of a pair (\rightsquigarrow_7), and *fst* 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

[Copyright notice will appear here once ‘preprint’ option is removed.]

Syntax

Types $A, B, C ::= X \mid A \times B \mid A \rightarrow B$
 Terms $L, M, N ::= x \mid (M, N) \mid \mathbf{fst} L \mid \mathbf{snd} L \mid \lambda x. N \mid L M$
 Environments $\Gamma, \Delta ::= x_1 : A_1, \dots, x_n : A_n$

Types

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

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash (M, N) : A \times B} \quad \frac{\Gamma \vdash L : A \times B}{\Gamma \vdash \mathbf{fst} L : A} \quad \frac{\Gamma \vdash L : A \times B}{\Gamma \vdash \mathbf{snd} L : B}$$

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

Laws

$$\begin{aligned} (\beta_1^\times) \quad \mathbf{fst} (M, N) &= M \\ (\beta_2^\times) \quad \mathbf{snd} (M, N) &= N \\ (\eta^\times) \quad (\mathbf{fst} L, \mathbf{snd} L) &= L \\ (\beta^\rightarrow) \quad (\lambda x. N) M &= N[x := M] \\ (\eta^\rightarrow) \quad \lambda x. (L x) &= L \end{aligned}$$

Figure 1. Lambda calculus

Syntax

Types $A, B, C ::= \dots \mid A \rightsquigarrow B$
 Terms $L, M, N ::= \dots \mid \mathbf{arr} \mid (\ggg) \mid \mathbf{first}$

Types

$$\begin{aligned} \mathbf{arr} &: (A \rightarrow B) \rightarrow (A \rightsquigarrow B) \\ (\ggg) &: (A \rightsquigarrow B) \rightarrow (B \rightsquigarrow C) \rightarrow (A \rightsquigarrow C) \\ \mathbf{first} &: (A \rightsquigarrow B) \rightarrow (A \times C \rightsquigarrow B \times C) \end{aligned}$$

Definitions

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

Laws

$$\begin{aligned} (\sim_1) \quad \mathbf{arr} \mathbf{id} \ggg f &= f \\ (\sim_2) \quad f \ggg \mathbf{arr} \mathbf{id} &= f \\ (\sim_3) \quad (f \ggg g) \ggg h &= f \ggg (g \ggg h) \\ (\sim_4) \quad \mathbf{arr} (g \cdot f) &= \mathbf{arr} f \ggg \mathbf{arr} g \\ (\sim_5) \quad \mathbf{first} (\mathbf{arr} f) &= \mathbf{arr} (f \times \mathbf{id}) \\ (\sim_6) \quad \mathbf{first} (f \ggg g) &= \mathbf{first} f \ggg \mathbf{first} g \\ (\sim_7) \quad \mathbf{first} f \ggg \mathbf{arr} (\mathbf{id} \times g) &= \mathbf{arr} (\mathbf{id} \times g) \ggg \mathbf{first} f \\ (\sim_8) \quad \mathbf{first} f \ggg \mathbf{arr} \mathbf{fst} &= \mathbf{arr} \mathbf{fst} \ggg f \\ (\sim_9) \quad \mathbf{first} (\mathbf{first} f) \ggg \mathbf{arr} \mathbf{assoc} &= \mathbf{arr} \mathbf{assoc} \ggg \mathbf{first} f \end{aligned}$$

Figure 2. Classic arrows

Syntax

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

Types

$$\frac{\Gamma; x : A \vdash Q ! B}{\Gamma \vdash \lambda^\bullet x. Q : A \rightsquigarrow B} \quad \frac{\Gamma \vdash L : A \rightsquigarrow B \quad \Gamma; \Delta \vdash P ! A}{\Gamma; \Delta \vdash L \bullet P ! B}$$

$$\frac{\Gamma, \Delta \vdash M : A}{\Gamma; \Delta \vdash [M] ! A} \quad \frac{\Gamma; \Delta \vdash P ! A \quad \Gamma; \Delta, x : A \vdash Q ! B}{\Gamma; \Delta \vdash \text{let } x = P \text{ in } Q ! B}$$

Laws

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

Figure 3. Arrow calculus

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.

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 $L \bullet P$, arrow unit $[M]$ (which resembles unit in a monad), and arrow bind $\text{let } x = P \text{ in } Q$ (which resembles bind in a monad).

In addition to the term typing judgment

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

we now also have a command typing judgment

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

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

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

translates to a term $\llbracket P \rrbracket_\Delta$ satisfying the judgment

$$\Gamma \vdash \llbracket P \rrbracket_\Delta : \Delta \rightsquigarrow A.$$

That is, the command P denotes an arrow, taking 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.

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

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.

$$\frac{\Gamma \vdash L : A \rightsquigarrow B \quad \Gamma; \Delta \vdash P ! A}{\Gamma; \Delta \vdash L \bullet P ! B}$$

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 denote arrows that are applied to arguments, but variables in Δ may not. (As we shall see in Section 6, an arrow with an apply operator—which is equivalent to a monad—relinquishes this restriction.)

Arrow unit promotes a term to a command.

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

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.

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

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

Translation

$$\begin{aligned}
\llbracket x \rrbracket &= x \\
\llbracket (M, N) \rrbracket &= (\llbracket M \rrbracket, \llbracket N \rrbracket) \\
\llbracket \mathbf{fst} L \rrbracket &= \mathbf{fst} \llbracket L \rrbracket \\
\llbracket \mathbf{snd} L \rrbracket &= \mathbf{snd} \llbracket L \rrbracket \\
\llbracket \lambda x. N \rrbracket &= \lambda x. \llbracket N \rrbracket \\
\llbracket L M \rrbracket &= \llbracket L \rrbracket \llbracket M \rrbracket \\
\llbracket \lambda^\bullet x. Q \rrbracket &= \llbracket Q \rrbracket_x \\
\llbracket L \bullet P \rrbracket_\Delta &= \llbracket P \rrbracket_\Delta \ggg \llbracket L \rrbracket \\
\llbracket \llbracket M \rrbracket \rrbracket_\Delta &= \mathit{arr} (\lambda \Delta. \llbracket M \rrbracket) \\
\llbracket \mathbf{let} x = P \mathbf{in} Q \rrbracket_\Delta &= (\mathit{arr} \mathit{id} \&\& \llbracket P \rrbracket_\Delta) \ggg \llbracket Q \rrbracket_{\Delta, x}
\end{aligned}$$

Translation preserves types

$$\begin{aligned}
\left[\frac{\Gamma; x : A \vdash Q ! B}{\Gamma \vdash \lambda^\bullet x. Q : A \rightsquigarrow B} \right] &= \frac{\Gamma \vdash \llbracket Q \rrbracket_x : A \rightsquigarrow B}{\Gamma \vdash \llbracket Q \rrbracket_x : A \rightsquigarrow B} \\
\left[\frac{\frac{\Gamma \vdash L : A \rightsquigarrow B}{\Gamma; \Delta \vdash P ! A}}{\Gamma; \Delta \vdash L \bullet P ! B} \right] &= \frac{\frac{\Gamma \vdash \llbracket L \rrbracket : A \rightsquigarrow B}{\Gamma \vdash \llbracket P \rrbracket_\Delta : \Delta \rightsquigarrow A}}{\Gamma \vdash \llbracket P \rrbracket_\Delta \ggg \llbracket L \rrbracket : \Delta \rightsquigarrow B} \\
\left[\frac{\Gamma, \Delta \vdash M : A}{\Gamma; \Delta \vdash \llbracket M \rrbracket ! A} \right] &= \frac{\Gamma, \Delta \vdash \llbracket M \rrbracket : A}{\Gamma \vdash \mathit{arr} (\lambda \Delta. \llbracket M \rrbracket) : \Delta \rightsquigarrow A} \\
\left[\frac{\frac{\Gamma; \Delta \vdash P ! A}{\Gamma; \Delta, x : A \vdash Q ! B}}{\Gamma; \Delta \vdash \mathbf{let} x = P \mathbf{in} Q ! B} \right] &= \frac{\frac{\Gamma \vdash \llbracket P \rrbracket_\Delta : \Delta \rightsquigarrow A}{\Gamma \vdash \llbracket Q \rrbracket_{\Delta, x} : \Delta \times A \rightsquigarrow B}}{\Gamma \vdash (\mathit{arr} \mathit{id} \&\& \llbracket P \rrbracket_\Delta) \ggg \llbracket Q \rrbracket_{\Delta, x} : \Delta \rightsquigarrow B}
\end{aligned}$$

Figure 4. Translating Arrow Calculus into Classic Arrows

Translation

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

Figure 5. Translating Classic Arrows into Arrow Calculus

appear in the computational lambda calculus of Moggi (1991). The beta law equates the application of an abstraction to a bind; substitution is not part of beta, but instead appears in the left unit law. The (assoc) law has the usual side condition, that x is not free in R . We do not require a side condition for (η^\sim) , 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.

$$\begin{array}{ll}
\lambda^\bullet x. Q & \mathbf{proc} x \rightarrow Q \\
L \bullet P & \mathbf{do} x \leftarrow P; L \leftarrow x \\
\llbracket M \rrbracket & \mathit{arr} \mathit{id} \leftarrow M \\
\mathbf{let} x = P \mathbf{in} Q & \mathbf{do} x \leftarrow P; Q
\end{array}$$

For arrow abstraction and binding the differences are purely syntactic, but his form of arrow application is merged with arrow unit, he writes $L \leftarrow M$ where we write $L \bullet [M]$. Our treatment of arrow unit as separate from arrow application permits neater expression of the laws.

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

$$\Gamma \vdash \llbracket M \rrbracket : A$$

while an arrow calculus command judgment

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

maps into a classic arrow judgment

$$\Gamma \vdash \llbracket P \rrbracket_\Delta : \Delta \rightsquigarrow A.$$

In $\llbracket P \rrbracket_\Delta$, 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 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:

- $\lambda^{\bullet}x. N$ translates straightforwardly; it is a no-op.
- $L \bullet P$ translates to \gggg .
- $[M]$ translates to *arr*.
- $\text{let } x = P \text{ in } Q$ translates to pairing $\&\&$ (to extend the environment with P) and composition \gggg (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 $\lambda z. N[x_1 := \text{fst } z, x_2 := \text{snd } z]$, and $\lambda x_1, x_2, x_3. N$ stands for $\lambda z. N[x_1 := \text{fst } \text{fst } z, x_2 := \text{snd } \text{fst } z, x_3 := \text{snd } z]$, 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,

$$\begin{aligned} M = N \text{ implies } \llbracket M \rrbracket &= \llbracket N \rrbracket \\ &\text{and} \\ P = Q \text{ implies } \llbracket P \rrbracket_{\Delta} &= \llbracket Q \rrbracket_{\Delta} \end{aligned}$$

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,

$$M = N \text{ implies } \llbracket M \rrbracket^{-1} = \llbracket N \rrbracket^{-1}$$

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. That is,

$$\llbracket \llbracket M \rrbracket \rrbracket^{-1} = M \text{ and } \llbracket \llbracket P \rrbracket_{\Delta} \rrbracket^{-1} = P$$

for all arrow calculus terms M and 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,

$$\llbracket \llbracket M \rrbracket^{-1} \rrbracket = M$$

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 (Sabry and Felleisen 1993).

5. A surprise

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 laws—excluding (\rightsquigarrow_2) —we can prove the laws of the arrow calculus, and from these we can it turn prove the classic 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.

6. Additional structure

Arrows are often equipped with additional structure, such as arrows with choice or arrows with apply.

An arrow with choice permits us to use the result of one arrow to choose between other arrows. Assume the core calculus has sums as well as products. An arrow with choice is equipped with an additional constant

$$\text{left} : (A \rightsquigarrow B) \rightarrow (A + C \rightsquigarrow B + C)$$

analogous to *first* but acting on sums rather than products. For the arrow calculus, equivalent structure is provided by a case expression where the branches are commands:

$$\frac{\begin{array}{l} \Gamma; \Delta \vdash R ! A + B \\ \Gamma; \Delta, x : A \vdash P ! C \\ \Gamma; \Delta, y : B \vdash Q ! C \end{array}}{\Gamma; \Delta \vdash \text{case } R \text{ of } \text{inl } x \rightarrow P \mid \text{inr } y \rightarrow Q ! C}$$

An arrow with apply permits us to apply an arrow that is itself yielded by another arrow. As explained by Hughes (2000), an arrow with apply is equivalent to a monad. It is equipped with an additional constant

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

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.

$$\frac{\begin{array}{l} \Gamma; \Delta \vdash R ! A \rightsquigarrow B \\ \Gamma; \Delta \vdash P ! A \end{array}}{\Gamma; \Delta \vdash R \bullet P ! B}$$

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 both of these extensions, we could go on to recall the laws they satisfy for classic arrows, to formulate suitable laws for the arrow calculus, to provide translations, and to show the extended systems still satisfy an equational correspondence.

But let's keep it short. We'll leave that joy for another day.

Acknowledgements

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

References

- Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Haskell Workshop*, pages 41–69, September 2001.
- Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In Johan Jeuring and Simon Peyton Jones, editors, *Advanced Functional Programming, 4th International School*, volume 2638 of *LNCS*. Springer-Verlag, 2003.

$$\begin{aligned}
& \llbracket \text{let } x = M \text{ in } [x] \rrbracket_{\Delta} \\
= & \text{def'n } \llbracket - \rrbracket \\
& (\text{arr id} \ggg \llbracket M \rrbracket_{\Delta}) \ggg \text{arr snd} \\
= & \text{def'n } \ggg \\
& \text{arr dup} \ggg \text{first (arr id)} \ggg \text{second } \llbracket M \rrbracket_{\Delta} \ggg \text{arr snd} \\
= & (\rightsquigarrow_5) \\
& \text{arr dup} \ggg \text{arr (id} \times \text{id)} \ggg \text{second } \llbracket M \rrbracket_{\Delta} \ggg \text{arr snd} \\
= & \text{id} \times \text{id} = \text{id} \\
& \text{arr dup} \ggg \text{arr id} \ggg \text{second } \llbracket M \rrbracket_{\Delta} \ggg \text{arr snd} \\
= & (\rightsquigarrow_1) \\
& \text{arr dup} \ggg \text{second } \llbracket M \rrbracket_{\Delta} \ggg \text{arr snd} \\
= & \text{def'n second} \\
& \text{arr dup} \ggg \text{arr swap} \ggg \text{first } \llbracket M \rrbracket_{\Delta} \ggg \text{arr swap} \ggg \text{arr snd} \\
= & (\rightsquigarrow_4) \\
& \text{arr (swap} \cdot \text{dup)} \ggg \text{first } \llbracket M \rrbracket_{\Delta} \ggg \text{arr (snd} \cdot \text{swap)} \\
= & \text{dup} \cdot \text{swap} = \text{dup}, \text{snd} \cdot \text{swap} = \text{fst} \\
& \text{arr dup} \ggg \text{first } \llbracket M \rrbracket_{\Delta} \ggg \text{arr fst} \\
= & (\rightsquigarrow_8) \\
& \text{arr dup} \ggg \text{arr fst} \ggg \llbracket M \rrbracket_{\Delta} \\
= & (\rightsquigarrow_4) \\
& \text{arr (fst} \cdot \text{dup)} \ggg \llbracket M \rrbracket_{\Delta} \\
= & \text{fst} \cdot \text{dup} = \text{id} \\
& \text{arr id} \ggg \llbracket M \rrbracket_{\Delta} \\
= & (\rightsquigarrow_1) \\
& \llbracket M \rrbracket_{\Delta}
\end{aligned}$$

Figure 6. Proof of (right) from classic arrows

$$\begin{aligned}
& \llbracket f \ggg \text{arr id} \rrbracket^{-1} \\
= & \text{def'n } \llbracket - \rrbracket^{-1} \\
& \lambda^{\bullet} x. (\lambda^{\bullet} y. [\text{id } y]) \bullet (f \bullet [x]) \\
= & (\beta^{\rightarrow}) \\
& \lambda^{\bullet} x. (\lambda^{\bullet} y. [y]) \bullet (f \bullet [x]) \\
= & (\beta^{\rightarrow}) \\
& \lambda^{\bullet} x. \text{let } y = f \bullet [x] \text{ in } [y] \\
= & (\text{right}) \\
& \lambda^{\bullet} x. f \bullet [x] \\
= & (\eta^{\rightarrow}) \\
& f \\
= & \text{def'n } \llbracket - \rrbracket^{-1} \\
& \llbracket f \rrbracket^{-1}
\end{aligned}$$

Figure 7. Proof of (\rightsquigarrow_2) in arrow calculus

John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.

Patrik Jansson and Johan Jeuring. Polytropic compact printing and parsing. In *European Symposium on Programming*, volume 1576 of *LNCS*, pages 273–287. Springer-Verlag, 1999.

Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.

Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

Ross Paterson. A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240. ACM Press, September 2001.

John Power and Hayo Thielecke. Closed Freyd- and kappa-categories. In *ICALP*, volume 1644 of *LNCS*. Springer, 1999.

Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3/4):289–360, 1993.

$$\begin{aligned}
& \llbracket \llbracket \text{first } f \rrbracket^{-1} \rrbracket \\
= & \text{def'n } \llbracket [-]^{-1}, (\beta^{\rightarrow}) \\
& \llbracket \lambda \bullet z. \text{let } x = f \bullet \text{fst } z \text{ in } [(x, \text{snd } z)] \rrbracket \\
= & \text{def'n } \llbracket [-] \\
& (\text{arr id} \&\& (\text{arr } (\lambda u. \text{fst } u) \gg\gg f)) \gg\gg \text{arr } (\lambda v. (\text{snd } v, \text{snd } \text{fst } v)) \\
= & \text{def'n } \text{fst} \\
& (\text{arr id} \&\& (\text{arr } \text{fst} \gg\gg f)) \gg\gg \text{arr } (\lambda v. (\text{snd } v, \text{snd } \text{fst } v)) \\
= & \text{def'n } \&\& \\
& \text{arr dup} \gg\gg \text{first } (\text{arr id}) \gg\gg \text{second } (\text{arr } \text{fst} \gg\gg f) \gg\gg \text{arr } (\lambda v. (\text{snd } v, \text{snd } \text{fst } v)) \\
= & (\sim_5) \\
& \text{arr dup} \gg\gg (\text{arr id} \times \text{arr id}) \gg\gg \text{second } (\text{arr } \text{fst} \gg\gg f) \gg\gg \text{arr } (\lambda v. (\text{snd } v, \text{snd } \text{fst } v)) \\
= & \text{id} \times \text{id} = \text{id} \\
& \text{arr dup} \gg\gg (\text{arr id}) \gg\gg \text{second } (\text{arr } \text{fst} \gg\gg f) \gg\gg \text{arr } (\lambda v. (\text{snd } v, \text{snd } \text{fst } v)) \\
= & (\sim_1) \\
& \text{arr dup} \gg\gg \text{second } (\text{arr } \text{fst} \gg\gg f) \gg\gg \text{arr } (\lambda v. (\text{snd } v, \text{snd } \text{fst } v)) \\
= & \text{def'n } \text{second} \\
& \text{arr dup} \gg\gg \text{arr swap} \gg\gg \text{first } (\text{arr } \text{fst} \gg\gg f) \gg\gg \text{arr swap} \gg\gg \text{arr } (\lambda v. (\text{snd } v, \text{snd } \text{fst } v)) \\
= & (\sim_4) \\
& \text{arr } (\text{swap} \cdot \text{dup}) \gg\gg \text{first } (\text{arr } \text{fst} \gg\gg f) \gg\gg \text{arr swap} \gg\gg \text{arr } (\lambda v. (\text{snd } v, \text{snd } \text{fst } v)) \\
= & \text{swap} \cdot \text{dup} = \text{dup} \\
& \text{arr dup} \gg\gg \text{first } (\text{arr } \text{fst} \gg\gg f) \gg\gg \text{arr swap} \gg\gg \text{arr } (\lambda v. (\text{snd } v, \text{snd } \text{fst } v)) \\
= & (\sim_4) \\
& \text{arr dup} \gg\gg \text{first } (\text{arr } \text{fst} \gg\gg f) \gg\gg \text{arr } ((\lambda v. (\text{snd } v, \text{snd } \text{fst } v)) \cdot \text{swap}) \\
= & (\lambda v. (\text{snd } v, \text{snd } \text{fst } v)) \cdot \text{swap} = \text{id} \times \text{snd} \\
& \text{arr dup} \gg\gg \text{first } (\text{arr } \text{fst} \gg\gg f) \gg\gg \text{arr } (\text{id} \times \text{snd}) \\
= & (\sim_6) \\
& \text{arr dup} \gg\gg \text{first } (\text{arr } \text{fst}) \gg\gg \text{first } f \gg\gg \text{arr } (\text{id} \times \text{snd}) \\
= & (\sim_5) \\
& \text{arr dup} \gg\gg \text{arr } (\text{fst} \times \text{id}) \gg\gg \text{first } f \gg\gg \text{arr } (\text{id} \times \text{snd}) \\
= & (\sim_7) \\
& \text{arr dup} \gg\gg \text{arr } (\text{fst} \times \text{id}) \gg\gg \text{arr } (\text{id} \times \text{snd}) \gg\gg \text{first } f \\
= & (\sim_4) \\
& \text{arr } ((\text{id} \times \text{snd}) \cdot (\text{fst} \times \text{id}) \cdot \text{dup}) \gg\gg \text{first } f \\
= & (\text{id} \times \text{snd}) \cdot (\text{fst} \times \text{id}) \cdot \text{dup} = \text{id} \\
& \text{arr id} \gg\gg \text{first } f \\
= & (\sim_1) \\
& \text{first } f
\end{aligned}$$

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

$$\begin{aligned}
& f \gg\gg \text{arr id} \\
= & (\sim_1) \\
& \text{arr id} \gg\gg f \gg\gg \text{arr id} \\
= & \text{fst} \cdot \text{dup} = \text{id} \\
& \text{arr } (\text{fst} \cdot \text{dup}) \gg\gg f \gg\gg \text{arr id} \\
= & (\sim_4) \\
& \text{arr dup} \gg\gg \text{arr } \text{fst} \gg\gg f \gg\gg \text{arr id} \\
= & (\sim_8) \\
& \text{arr dup} \gg\gg \text{first } f \gg\gg \text{arr } \text{fst} \gg\gg \text{arr id} \\
= & (\sim_4) \\
& \text{arr dup} \gg\gg \text{first } f \gg\gg \text{arr } (\text{id} \cdot \text{fst}) \\
= & \text{id} \cdot \text{fst} = \text{fst} \\
& \text{arr dup} \gg\gg \text{first } f \gg\gg \text{arr } \text{fst} \\
= & (\sim_8) \\
& \text{arr dup} \gg\gg \text{arr } \text{fst} \gg\gg f \\
= & (\sim_4) \\
& \text{arr } (\text{fst} \cdot \text{dup}) \gg\gg f \\
= & \text{fst} \cdot \text{dup} = \text{id} \\
& \text{arr id} \gg\gg f \\
= & (\sim_1) \\
& f
\end{aligned}$$

Figure 9. Proof that (\sim_2) is redundant