JFP **20** (1): 51–69, 2010.
doi:10.1017/S095679680999027X

The arrow calculus

SAM LINDLEY, PHILIP WADLER and JEREMY YALLOP
University of Edinburgh
(*e-mail:* Sam.Lindley@ed.ac.uk, Philip.Wadler@ed.ac.uk, Jeremy.Yallop@ed.ac.uk)

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 (which fit two well-known patterns). 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.

1 Introduction

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

Arrows (Hughes 2000) generalize the *monads* of Moggi (1991) and the *idioms* of McBride and Paterson (2008). They are closely related to *Freyd categories*, discovered independently from Hughes by Power, Robinson, and Thielecke (Power & Robinson 1997; Power & Thielecke 1999). Arrows enjoy a wide range of applications, including parsers and printers (Jansson & Jeuring 1999), web interaction (Hughes 2000), circuits (Paterson 2001), graphic user interfaces (Courtney & 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. (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 (Paterson 2001). 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 fitting two familiar 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 (Lindley *et al.* 2008b). 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 that it 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?

First, Paterson translates his notation into classic arrows, and shows the five laws follow from the nine (soundness). Conversely, we give an 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.

Second, 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.

Third, 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 (1993). 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 *et al.* (2008b) 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. Interestingly, the proof of equational correspondence depends only on the translations, and is independent of the type rules.

Fourth, we 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.

Fifth, following Hughes and Paterson, we extend the arrow calculus to higher-order arrows, corresponding to arrows with apply, which in turn correspond to monads (Hughes 2000). We again show soundness and completeness, and uncover a second redundancy, that the three laws proposed by Hughes can be reduced to two.

The arrow calculus has already proven useful. It enabled us to clarify the relationship between idioms, arrows, and monads (Lindley *et al.* 2008b), and it provided the inspiration for a categorical semantics of arrows (Atkey 2008).

In private correspondence, Eugenio Moggi and Robert Atkey have suggested that there may be interesting generalizations of arrows, inspired by the formulation proposed in this paper, possibly related to Indexed Freyd Models (Atkey 2008) or to a generalization of Benton's Adjoint Calculus (Benton 1995).

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 higher-order arrows.

2 Classic arrows

We refer to the traditional 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. We 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 side effects. The three constants are: *arr*, which promotes a function to a pure arrow with no side effects; $(\ggg)$, which composes two arrows; 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 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 arrows rewrites to a pure arrow ($\rightsquigarrow_5$), *first* is a homomorphism for composition ($\rightsquigarrow_6$), *first* commutes with a pure arrow that is the identity on the first component of a pair ($\rightsquigarrow_7$), and *first* pushes through promotions of *fst* and *assoc* ($\rightsquigarrow_8$,$\rightsquigarrow_9$).

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

Syntax

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

Types

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

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

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

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.

f &&& g, which applies arrows f and g to the same argument and pairs the results. We also define the selector *snd*.

Every arrow of interest comes with additional operators, which perform side effects or combine arrows in other ways. 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 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 $\mathsf{let}\ x \Leftarrow P\ \mathsf{in}\ Q$, which resembles bind in a monad.

In addition to the term typing judgment

$$\Gamma \vdash M : A$$

Syntax

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

Types

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

Laws

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

Definitions

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

Fig. 2. Classic arrows.

Syntax

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

Types

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

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

Laws

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

Fig. 3. Arrow calculus.

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, $\lambda^{\bullet} x.\, Q$, and arrow bind, $\mathsf{let}\ x \Leftarrow P\ \mathsf{in}\ Q$.

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 $[\![P]\!]_{\Delta}$ satisfying the judgment $\Gamma \vdash [\![P]\!]_{\Delta} : \Delta \rightsquigarrow A$; so a command P denotes an arrow, and the second environment Δ corresponds to the argument type of the arrow. We explain the type rules of the language in this section, and the translation in the next.

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 builds a command from two terms.

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

Arrow application closely resembles function application. The argument term may contain variables from Δ, but the term denoting the arrow to be applied may not; this is because there is no way to apply an arrow that is itself yielded by another arrow. It is for this reason that we distinguish two environments, Γ and Δ. (Section 5 describes an arrow with an apply operator, which relinquishes this restriction and is equivalent to a monad.)

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).

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

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

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

Admissibility rules for substitution and weakening follow from these. Substitution of a term in a term is straightforward.

$$\frac{\Gamma \vdash M : A \qquad \Gamma, x : A \vdash N : B}{\Gamma \vdash N[x := M] : B}$$

Here the double line means that if the judgment on the top is derivable then the judgment on the bottom is also derivable.

Substitution of a term in a command may involve either environment. Substitution for a variable in the first environment is again straightforward.

$$\frac{\Gamma \vdash M : A \qquad \Gamma, x : A; \Delta \vdash Q \mathbin{!} B}{\Gamma; \Delta \vdash Q[x := M] \mathbin{!} B}$$

Substitution for a variable in the second environment is more interesting.

$$\frac{\Gamma, \Delta \vdash M : A \qquad \Gamma; \Delta, x : A \vdash Q \mathbin{!} B}{\Gamma; \Delta \vdash Q[x := M] \mathbin{!} B}$$

In this case the term may refer to variables in either environment.

The weakening rule for terms is straightforward. Write $\Gamma \subseteq \Gamma'$ to indicate that every type binding $x : A$ in Γ also occurs in Γ'. If $\Gamma \subseteq \Gamma'$, then

$$\frac{\Gamma \vdash M : A}{\Gamma' \vdash M : A}$$

The weakening rule for commands is slightly unusual because of the two typing contexts. If $\Gamma \subseteq \Gamma'$ and $\Gamma, \Delta \subseteq \Gamma', \Delta'$, then

$$\frac{\Gamma ; \Delta \vdash P \ ! \ A}{\Gamma' ; \Delta' \vdash P \ ! \ A}$$

Weakening permits both the Γ and Δ type enviroments to grow, and also permits variables to move from Δ into Γ (but not conversely).

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 (1991). 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.

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

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 $arrowA \prec M$ 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.

4 Translations

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

The translation takes an arrow calculus term M into a classic arrow $[\![M]\!]$:

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

Similarly, the translation takes an arrow calculus command P into a classic arrow $[\![P]\!]_{\Delta}$, parameterized by a sequence of variables:

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

The denotation of a command is an arrow, with argument corresponding to the command's (second) environment Δ and result corresponding to the command's type A.

Note that in $[\![P]\!]_{\Delta}$, we take Δ to stand for a tuple of the variables in the environment, and in $\Delta \rightsquigarrow A$ we take Δ to stand for the product of the types in the environment. We encode these as binary products associating to the left. For example, if Δ is $x_1 : A_1, x_2 : A_2, x_3 : A_3$ then Δ as a tuple of variables stands for $((x_1, x_2), x_3)$ and Δ as a type stands for $(A_1 \times A_2) \times A_3$. We also write $\lambda\Delta.\, M$

$$
\begin{array}{lcl}
[\![x]\!] & = & x \\
[\![(M,N)]\!] & = & ([\![M]\!],[\![N]\!]) \\
[\![\mathsf{fst}\ L]\!] & = & \mathsf{fst}\ [\![L]\!] \\
[\![\mathsf{snd}\ L]\!] & = & \mathsf{snd}\ [\![L]\!] \\
[\![\lambda x.N]\!] & = & \lambda x.[\![N]\!] \\
[\![L\ M]\!] & = & [\![L]\!]\ [\![M]\!] \\
[\![\lambda^{\bullet} x.Q]\!] & = & [\![Q]\!]_x \\
[\![L \bullet M]\!]_\Delta & = & arr\ (\lambda\Delta.[\![M]\!]) >\!\!>\!\!> [\![L]\!] \\
[\![\,[M]\,]\!]_\Delta & = & arr\ (\lambda\Delta.[\![M]\!]) \\
[\![\mathsf{let}\ x \Leftarrow P\ \mathsf{in}\ Q]\!]_\Delta & = & (arr\ id\ \&\!\&\!\&\ [\![P]\!]_\Delta) >\!\!>\!\!> [\![Q]\!]_{\Delta,x}
\end{array}
$$

Fig. 4. Translation from arrow calculus into classic arrows.

with the obvious meaning. For example, if Δ is as above then $\lambda\Delta.[\![M]\!]$ stands for $\lambda z.[\![M]\!][x_1 := \mathsf{fst}\ (\mathsf{fst}\ z), x_2 := \mathsf{snd}\ (\mathsf{fst}\ z), x_3 := \mathsf{snd}\ z]$.

The translation from arrow calculus to classic arrows is given in Figure 4. The translation of the constructs of the core lambda calculus are straightforward homomorphisms. We consider in turn the translation of the four new constructs.

The translation of an arrow abstraction is the translation of its body with respect to its bound variable:

$$[\![\lambda^{\bullet} x.Q]\!] \quad = \quad [\![Q]\!]_x$$

Or, in the context of its type rules:

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

The introduction rule for arrow abstraction in the arrow calculus becomes a no-op when translated to classic arrows.

It may be worth pausing to explain the above in detail. The translation of rules is to be read pointwise. The translation of commands is $[\![\Gamma;\Delta \vdash P\ !\ A]\!] = \Gamma \vdash [\![P]\!]_\Delta : \Delta \rightsquigarrow A$, so taking $\Delta := x : A, P := Q, B := A$ gives us the translation of the hypothesis. Similarly, the translation of terms is $[\![\Gamma \vdash M : A]\!] = \Gamma \vdash [\![M]\!] : A$, so taking $M := \lambda^{\bullet} x.Q, A := A \rightsquigarrow B$ and applying the definition $[\![\lambda^{\bullet} x.Q]\!] = [\![Q]\!]_x$ gives us the translation of the conclusion.

The translation of an arrow application is the composition of the promotion of the translation of the argument with the translation of the function:

$$[\![L \bullet M]\!]_\Delta \quad = \quad arr\ (\lambda\Delta.[\![M]\!]) >\!\!>\!\!> [\![L]\!]$$

Or, in the context of its type rules:

$$\left[\!\!\left[\frac{\Gamma \vdash L : A \rightsquigarrow B \qquad \Gamma,\Delta \vdash M : A}{\Gamma;\Delta \vdash L \bullet M\ !\ B}\right]\!\!\right] \quad = \quad \frac{\Gamma \vdash [\![L]\!] : A \rightsquigarrow B \qquad \Gamma,\Delta \vdash [\![M]\!] : A}{\Gamma \vdash arr\ (\lambda\Delta.[\![M]\!]) >\!\!>\!\!> [\![L]\!] : \Delta \rightsquigarrow B}$$

Here $\lambda\Delta.[\![M]\!]$ is a function of type $\Delta \to A$; applying *arr* to this yields an arrow $\Delta \rightsquigarrow A$ which is composed with the arrow $[\![L]\!]$ of type $A \rightsquigarrow B$ to yield an arrow $\Delta \rightsquigarrow B$ as required.

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

Fig. 5. Translation from classic arrows into arrow calculus.

The translation of the promotion of a term to a command is the promotion of the corresponding function:

$$[\![\,[M]\,]\!]_{\Delta} \quad = \quad arr\ (\lambda\Delta.\,[\![M]\!])$$

Or, in the context of its type rules:

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

This is exactly the same as the part of the previous transformation corresponding to the argument of the function.

Finally, the translation of a binding extends the environment by the translation of the definiens and composes this with the body:

$$[\![\mathsf{let}\ x \Leftarrow P\ \mathsf{in}\ Q]\!]_{\Delta} \quad = \quad (arr\ id \mathbin{\&\!\&\!\&} [\![P]\!]_{\Delta}) \ggg [\![Q]\!]_{(\Delta,x)}$$

Or, in the context of its type rules:

$$\left[\!\!\left[\frac{\Gamma;\,\Delta \vdash P\ !\ A \qquad \Gamma;\,\Delta,\,x : A \vdash Q\ !\ B}{\Gamma;\,\Delta \vdash \mathsf{let}\ x \Leftarrow P\ \mathsf{in}\ Q\ !\ B}\right]\!\!\right] \quad = \quad \frac{\Gamma \vdash [\![P]\!]_{\Delta} : \Delta \rightsquigarrow A \qquad \Gamma \vdash [\![Q]\!]_{\Delta,x} : \Delta \times A \rightsquigarrow B}{\Gamma \vdash (arr\ id \mathbin{\&\!\&\!\&} [\![P]\!]_{\Delta}) \ggg [\![Q]\!]_{\Delta,x} : \Delta \rightsquigarrow B}$$

This translation uses &&&, which is defined in terms of *first* (and *second*) in Figure 2. Here $arr\ id \mathbin{\&\!\&\!\&} [\![P]\!]_{\Delta}$ of type $\Delta \rightsquigarrow \Delta \times A$ extends the environment, and composition with $[\![Q]\!]_{\Delta,x}$ of type $\Delta \times A \rightsquigarrow B$ yields an arrow $\Delta \rightsquigarrow B$ as required.

The inverse translation, from classic arrows to the arrow calculus, is given in Figure 5. Again, the translations 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. Promotion accepts a function, and returns the corresponding arrow which applies the function:

$$[\![arr]\!]^{-1} \quad = \quad \lambda f.\lambda^{\bullet} x.\,[f\ x]$$

Composition of arrows looks just like ordinary function composition, but using arrow apply instead of function application:

$$[\![(\ggg)]\!]^{-1} \quad = \quad \lambda f.\lambda g.\lambda^{\bullet} x.\,\mathsf{let}\ y \Leftarrow f \bullet x\ \mathsf{in}\ g \bullet y$$

And the constant *first* accepts an arrow, and returns an arrow which takes a pair, applies the arrow to the first component of the pair and returns the second

component unchanged:

$$[\![\mathit{first}]\!]^{-1} \quad = \quad \lambda f.\, \lambda^{\bullet} z.\, \mathsf{let}\ x \Leftarrow f \bullet \mathsf{fst}\ z\ \mathsf{in}\ [(x, \mathsf{snd}\ z)]$$

Our previous admissibility rules give rise to two lemmas about translation. The substitution lemma relates the translation of a substitution to the translation of its components:

$$[\![Q[x := M]]\!]_{\Delta} \quad = \quad \mathit{arr}\ (\lambda \Delta.\, (\Delta, [\![M]\!])) \ggg [\![Q]\!]_{\Delta, x}$$

Or, in the context of its type rules:

$$\left[\!\!\left[\dfrac{\Gamma, \Delta \vdash M : A \qquad \Gamma;\, \Delta, x : A \vdash Q \mathbin{!} B}{\Gamma;\, \Delta \vdash Q[x := M] \mathbin{!} B} \right]\!\!\right] \quad = \quad \dfrac{\Gamma, \Delta \vdash [\![M]\!] : A \qquad \dfrac{\Gamma \vdash [\![Q]\!]_{\Delta, x} : \Delta \times A \rightsquigarrow B}{\Gamma \vdash \mathit{arr}\ (\lambda \Delta.\, (\Delta, [\![M]\!])) \ggg [\![Q]\!]_{\Delta, x} : \Delta \rightsquigarrow B}}{}$$

The proof is by induction on the structure of Q.

Similarly, the Weakening Lemma relates the translation of a weakened judgment to the translation of the original judgment:

$$[\![P]\!]_{\Delta'} \quad = \quad \mathit{arr}\ (\lambda \Delta'.\, \Delta) \ggg [\![P]\!]_{\Delta}$$

Or, in the context of its type rules, if $\Gamma \subseteq \Gamma'$ and $\Gamma, \Delta \subseteq \Gamma', \Delta'$, then

$$\left[\!\!\left[\dfrac{\Gamma;\, \Delta \vdash P \mathbin{!} A}{\Gamma';\, \Delta' \vdash P \mathbin{!} A} \right]\!\!\right] \quad = \quad \dfrac{\Gamma \vdash [\![P]\!]_{\Delta} : \Delta \rightsquigarrow A}{\Gamma' \vdash \mathit{arr}\ (\lambda \Delta'.\, \Delta) \ggg [\![P]\!]_{\Delta} : \Delta' \rightsquigarrow A}$$

The proof is by induction over P.

We can now show four properties.

(i) The five laws of the arrow calculus follow from the nine laws of classic arrows:

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

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.

(ii) The nine laws of classic arrows follow from the five laws of the arrow calculus:

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

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.

(iii) Translating from the arrow calculus into classic arrows and back again is the identity on terms:

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

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

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

Fig. 6. Proof of right from classic arrows.

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

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

translation yields a term rather than a command, but it does yield the term that is the arrow abstraction of the original command:

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

for all arrow calculus commands P. The proof requires four calculations, one for each construct of the arrow calculus.

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

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

(iv) Translating from classic arrows into the arrow calculus and back again is the identity:

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

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*.

Properties (i)–(iv) together constitute an *equational correspondence* between classic arrows and the arrow calculus (Sabry & Felleisen 1993). The full details of the proof appear in a companion technical report (Lindley *et al.* 2008a).

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

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

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

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.

5 Higher-order arrows

Arrows offer only a weak notion of structure, and it is common to impose additional structure by adding extra constants and extra laws. Such variants include higher-order arrows (Hughes 2000), arrows with choice (Hughes 2000), arrows with loops (Paterson 2001), and static arrows (Lindley *et al.* 2008b). Here we show how to treat higher-order computation in the arrow calculus; we believe the other extensions can be treated similarly.

Hughes (2000) described the additional structure required for classic arrows to be higher order, and showed that an arrow with this structure is equivalent to a monad. In practice, when such structure is present one would tend to use a monad rather than an arrow (since the monad calculus is simpler than the arrow calculus). But the structure is useful for understanding the relation between monads and arrows (Hughes 2000; Lindley *et al.* 2008b).

Hughes introduced a new constant, an arrow analogue of application,

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

satisfying three laws, as shown in Figure 10.

Syntax

$$\text{Terms} \quad L, M, N \quad ::= \quad \cdots \mid app$$

Types

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

Laws

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

Fig. 10. Higher-order classic arrows.

Syntax

$$\text{Commands} \quad P, Q, R \quad ::= \quad \cdots \mid L \star M$$

Types

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

Laws

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

Fig. 11. Higher-order arrow calculus.

Recall our previous rule for arrow application,

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

which imposes the restriction that the term denoting the arrow to apply cannot contain free variables in Δ. To extend the arrow calculus to higher-order arrows, we introduce a second apply operation (written $\star$ rather than $\bullet$) which lifts this restriction,

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

and satisfies a beta law,

$$(\beta^{app}) \quad (\lambda^{\bullet} x.\, Q) \star M \quad = \quad Q[x := M]$$

The changes are summarized in Figure 11.

$$
\begin{array}{ll}
 & [\![\mathit{first}\ (\mathit{arr}\ (g \ggg))) \ggg \mathit{app}]\!]^{-1} \\
= & \qquad \text{def } [\![-]\!]^{-1} \\
 & \lambda^\bullet a.\, \mathsf{let}\ b \Leftarrow (\lambda^\bullet z.\, \mathsf{let}\ x \Leftarrow (\lambda^\bullet f.\, [[\![g \ggg f]\!]^{-1}]) \bullet \mathsf{fst}\ z\ \mathsf{in}\ [\langle x, \mathsf{snd}\ z\rangle]) \bullet a\ \mathsf{in} \\
 & \quad (\lambda^\bullet w.\, \mathsf{fst}\ w \star \mathsf{snd}\ w) \bullet b \\
= & \qquad (\beta^{\rightsquigarrow}) \\
 & \lambda^\bullet a.\, \mathsf{let}\ b \Leftarrow (\mathsf{let}\ x \Leftarrow [[\![g \ggg \mathsf{fst}\ a]\!]^{-1}]\ \mathsf{in}\ [\langle x, \mathsf{snd}\ a\rangle])\ \mathsf{in}\ \mathsf{fst}\ b \star \mathsf{snd}\ b \\
= & \qquad (\text{assoc}) \\
 & \lambda^\bullet a.\, \mathsf{let}\ x \Leftarrow [[\![g \ggg \mathsf{fst}\ a]\!]^{-1}]\ \mathsf{in}\ \mathsf{let}\ b \Leftarrow [\langle x, \mathsf{snd}\ a\rangle]\ \mathsf{in}\ \mathsf{fst}\ b \star \mathsf{snd}\ b \\
= & \qquad (\text{left}) \\
 & \lambda^\bullet a.\, \mathsf{let}\ x \Leftarrow [[\![g \ggg \mathsf{fst}\ a]\!]^{-1}]\ \mathsf{in}\ x \star \mathsf{snd}\ a \\
= & \qquad \text{def } [\![-]\!]^{-1} \\
 & \lambda^\bullet a.\, \mathsf{let}\ x \Leftarrow [\lambda^\bullet d.\, \mathsf{let}\ y \Leftarrow g \bullet d\ \mathsf{in}\ \mathsf{fst}\ a \bullet y]\ \mathsf{in}\ x \star \mathsf{snd}\ a \\
= & \qquad (\eta^{app}, \beta^{\rightsquigarrow}) \\
 & \lambda^\bullet a.\, \mathsf{let}\ x \Leftarrow [\lambda^\bullet d.\, \mathsf{let}\ y \Leftarrow g \bullet d\ \mathsf{in}\ \mathsf{fst}\ a \star y]\ \mathsf{in}\ x \star \mathsf{snd}\ a \\
= & \qquad (\text{left}, \beta^{app}) \\
 & \lambda^\bullet a.\, \mathsf{let}\ y \Leftarrow g \bullet \mathsf{snd}\ a\ \mathsf{in}\ \mathsf{fst}\ a \star y \\
= & \qquad (\text{left}) \\
 & \lambda^\bullet a.\, \mathsf{let}\ y \Leftarrow g \bullet \mathsf{snd}\ a\ \mathsf{in}\ \mathsf{let}\ b \Leftarrow [\langle \mathsf{fst}\ a, y\rangle]\ \mathsf{in}\ \mathsf{fst}\ b \star \mathsf{snd}\ b \\
= & \qquad (\text{assoc}) \\
 & \lambda^\bullet a.\, \mathsf{let}\ b \Leftarrow (\mathsf{let}\ y \Leftarrow g \bullet \mathsf{snd}\ a\ \mathsf{in}\ [\langle \mathsf{fst}\ a, y\rangle])\ \mathsf{in}\ \mathsf{fst}\ b \star \mathsf{snd}\ b \\
= & \qquad (\beta^{\rightsquigarrow}) \\
 & \lambda^\bullet a.\, \mathsf{let}\ b \Leftarrow (\lambda^\bullet z.\, \mathsf{let}\ y \Leftarrow g \bullet \mathsf{snd}\ z\ \mathsf{in}\ [\langle \mathsf{fst}\ z, y\rangle]) \bullet a\ \mathsf{in} \\
 & \quad (\lambda^\bullet w.\, \mathsf{fst}\ w \star \mathsf{snd}\ w) \bullet b \\
= & \qquad \text{def } [\![-]\!]^{-1} \\
 & [\![\mathit{second}\ g \ggg \mathit{app}]\!]^{-1}
\end{array}
$$

Fig. 12. Proof of $\rightsquigarrow_{H2}$ in higher-order arrow calculus.

It is instructive to see the beta law rewritten with types:

$$\frac{\Gamma;\, x : A \vdash Q \mathbin{!} B \qquad \Gamma \vdash M : A}{\Gamma;\, \emptyset \vdash (\lambda^\bullet x.\, Q) \star M = Q[x := M] \mathbin{!} B}$$

where we write $\Gamma;\, \Delta \vdash P = Q \mathbin{!} A$ to abbreviate $\Gamma;\, \Delta \vdash P \mathbin{!} A$ and $\Gamma;\, \Delta \vdash Q \mathbin{!} A$ and $P = Q$. One might be tempted to adopt more general types,

$$\frac{\Gamma, \Delta;\, x : A \vdash Q \mathbin{!} B \qquad \Gamma, \Delta \vdash M : A}{\Gamma;\, \Delta \vdash (\lambda^\bullet x.\, Q) \star M = Q[x := M] \mathbin{!} B}$$

but this cannot be correct, because the more general typing for the right-hand side is not necessarily admissible under the more general typing for Q.

The (η^{app}) law follows from (β^{app}), since

$$\lambda^\bullet x.\, L \star x = \lambda^\bullet x.\, (\lambda^\bullet y.\, L \bullet y) \star x = \lambda^\bullet x.\, L \bullet x = L$$

where the equalities follow by $(\eta^{\rightsquigarrow})$, (β^{app}), and $(\eta^{\rightsquigarrow})$, respectively. Further, every first-order application is equivalent to a higher-order application. If we assume $\Gamma \vdash L : A \rightsquigarrow B$ and $\Gamma, \Delta \vdash M : A$ and $x \notin \Gamma, \Delta$, then

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

where the equalities follow by (β^{app}) and $(\eta^{\rightsquigarrow})$, respectively.

$$
\begin{array}{ll}
 & [\![(\lambda^{\bullet} x.\, Q) \star M]\!]_{\Delta} \\
= & \qquad \text{def } [\![-]\!] \\
 & arr\ (\lambda\Delta.\, ([\![Q]\!]_x, [\![M]\!])) >\!\!>\!\!> app \\
= & \qquad \text{def } (\times), dup \\
 & arr\ (((\lambda\Delta.\, [\![Q]\!]_x) \times (\lambda\Delta.\, [\![M]\!])) \cdot dup) >\!\!>\!\!> app \\
= & \qquad g \times f = swap \cdot (f \times id) \cdot swap \cdot (g \times id) \\
 & arr\ (swap \cdot ((\lambda\Delta.\, [\![M]\!]) \times id) \cdot swap \cdot ((\lambda\Delta.\, [\![Q]\!]_x) \times id) \cdot dup) >\!\!>\!\!> app \\
= & \qquad (\rightsquigarrow_4) \\
 & arr\ dup >\!\!>\!\!> arr\ ((\lambda\Delta.\, [\![Q]\!]_x) \times id) >\!\!>\!\!> arr\ swap >\!\!>\!\!> arr\ ((\lambda\Delta.\, [\![M]\!]) \times id) \\
 & \qquad >\!\!>\!\!> arr\ swap >\!\!>\!\!> app \\
= & \qquad (\rightsquigarrow_5) \\
 & arr\ dup >\!\!>\!\!> first\ (arr\ (\lambda\Delta.\, [\![Q]\!]_x)) >\!\!>\!\!> arr\ swap >\!\!>\!\!> first\ (arr\ (\lambda\Delta.\, [\![M]\!])) \\
 & \qquad >\!\!>\!\!> arr\ swap >\!\!>\!\!> app \\
= & \qquad \text{def } second, \&\!\&\!\& \\
 & (arr\ (\lambda\Delta.\, [\![Q]\!]_x) \mathbin{\&\!\&\!\&} arr\ (\lambda\Delta.\, [\![M]\!])) >\!\!>\!\!> app \\
= & \qquad \text{Weakening Lemma} \\
 & (arr\ (\lambda\Delta.\, arr\ (\lambda x.\, (\Delta, x)) >\!\!>\!\!> [\![Q]\!]_{\Delta,x}) \mathbin{\&\!\&\!\&} arr\ (\lambda\Delta.\, [\![M]\!])) >\!\!>\!\!> app \\
= & \qquad \text{let } i = \lambda\Delta.\, arr\ (\lambda x.\, (\Delta, x)), p = arr\ (\lambda\Delta.\, [\![M]\!]), q = [\![Q]\!]_{\langle\Delta,x\rangle} \\
 & (arr\ ((q >\!\!>\!\!>) \cdot i) \mathbin{\&\!\&\!\&} p) >\!\!>\!\!> app \\
= & \qquad \text{def } \&\!\&\!\& \\
 & arr\ dup >\!\!>\!\!> first\ (arr\ ((q >\!\!>\!\!>) \cdot i)) >\!\!>\!\!> second\ p >\!\!>\!\!> app \\
= & \qquad (\rightsquigarrow_5) \\
 & arr\ dup >\!\!>\!\!> arr\ (((q >\!\!>\!\!>) \cdot i) \times id) >\!\!>\!\!> second\ p >\!\!>\!\!> app \\
= & \qquad arr\ (f \times id) >\!\!>\!\!> second\ g = second\ g >\!\!>\!\!> arr\ (f \times id) \\
 & arr\ dup >\!\!>\!\!> second\ p >\!\!>\!\!> arr\ (((q >\!\!>\!\!>) \cdot i) \times id) >\!\!>\!\!> app \\
= & \qquad (\rightsquigarrow_5) \\
 & arr\ dup >\!\!>\!\!> second\ p >\!\!>\!\!> first\ (arr\ ((q >\!\!>\!\!>) \cdot i)) >\!\!>\!\!> app \\
= & \qquad (\rightsquigarrow_4) \\
 & arr\ dup >\!\!>\!\!> second\ p >\!\!>\!\!> first\ (arr\ i >\!\!>\!\!> arr\ (q >\!\!>\!\!>)) >\!\!>\!\!> app \\
= & \qquad (\rightsquigarrow_6) \\
 & arr\ dup >\!\!>\!\!> second\ p >\!\!>\!\!> first\ (arr\ i) >\!\!>\!\!> first\ (arr\ (q >\!\!>\!\!>)) >\!\!>\!\!> app \\
= & \qquad (\rightsquigarrow_{H3}) \\
 & arr\ dup >\!\!>\!\!> second\ p >\!\!>\!\!> first\ (arr\ i) >\!\!>\!\!> app >\!\!>\!\!> q \\
= & \qquad (\rightsquigarrow_{H1}) \\
 & arr\ dup >\!\!>\!\!> second\ p >\!\!>\!\!> arr\ id >\!\!>\!\!> q \\
= & \qquad (\rightsquigarrow_2) \\
 & arr\ dup >\!\!>\!\!> second\ p >\!\!>\!\!> q \\
= & \qquad (\rightsquigarrow_1) \\
 & arr\ dup >\!\!>\!\!> arr\ id >\!\!>\!\!> second\ p >\!\!>\!\!> q \\
= & \qquad id = id \times id \\
 & arr\ dup >\!\!>\!\!> arr\ (id \times id) >\!\!>\!\!> second\ p >\!\!>\!\!> q \\
= & \qquad (\rightsquigarrow_5) \\
 & arr\ dup >\!\!>\!\!> first\ (arr\ id) >\!\!>\!\!> second\ p >\!\!>\!\!> q \\
= & \qquad \text{def } \&\!\&\!\& \\
 & (arr\ id \mathbin{\&\!\&\!\&} p) >\!\!>\!\!> q \\
= & \qquad \text{def } p, q \\
 & (arr\ id \mathbin{\&\!\&\!\&} arr\ (\lambda\Delta.\, [\![M]\!])) >\!\!>\!\!> [\![Q]\!]_{\Delta,x} \\
= & \qquad \text{Substitution Lemma} \\
 & [\![Q[x := M]]\!]_{\Delta}
\end{array}
$$

Fig. 13. Proof of β^{app} from higher order classic arrows.

Replacing every occurrence of $L \bullet M$ by $L \star M$ renders the difference in the two environments, Γ and Δ, no longer relevant, since the only significant difference between Γ and Δ is that L in $L \bullet M$ may not use variables in Δ, and it is precisely this restriction that is lifted in $L \star M$.

The substitution and weakening lemmas extend straightforwardly. The translation from arrow calculus to classic arrows is extended by

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

and the inverse translation is extended by

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

As before, we must show that the translations preserve equations. There are three calculations for each of the three higher-order arrow laws. Figure 12 shows one of these, the calculation to derive $(\rightsquigarrow_{H2})$ from the laws of the arrow calculus and (β^{app}). Figure 13 shows the corresponding calculation for the inverse translation, a proof of (β^{app}) from the laws of classic arrows with apply.

Once again, the proof 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.

Acknowledgments

Our thanks to Robert Atkey, Samuel Bronson, John Hughes, Eugenio Moggi, Ross Paterson, and our referees.

References

Atkey, R. (2008) What is a categorical model of arrows? In *Mathematical Structures in Functional Programming*, Capreta, V. & McBride, C. (eds), Electronic Notes in Theoretical Computer Science, Reykjavic, Iceland.

Benton, N. (1995) A mixed linear and non-linear logic: Proofs, terms and models. In *Computer Science Logics*, Pacholski, L. & Tiuryn, J. (eds), Lecture Notes in Computer Science, vol. 933. Springer–Verlag, Kazimierz, Poland.

Courtney, A. & Elliott, C. (2001) *Genuinely Functional User Interfaces.* Haskell workshop, 41–69.

Hudak, P., Courtney, A., Nilsson, H. & Peterson, J. (2003) Arrows, robots, and functional reactive programming. In *Advanced Functional Programming, 4th International School*, Jeuring, J. & Jones, S. P. (eds), LNCS, vol. 2638. Springer-Verlag, Oxford, UK.

Hughes, J. (2000) Generalising monads to arrows, *Sci Comput Program.*, **37**: 67–111.

Jansson, P. & Jeuring, J. (1999) Polytypic compact printing and parsing. *Pages 273–287 of: European Symposium on Programming*, LNCS, vol. 1576. Springer-Verlag, Amsterdam, The Netherlands.

Lindley, S., Wadler, P. & Yallop, J. (2008a) *The Arrow Calculus.* Tech. rept. EDI-INF-RR-1258. School of Informatics, University of Edinburgh.

Lindley, S., Wadler, P. & Yallop, J. (2008b) Idioms are oblivious, arrows are meticulous, monads are promiscuous. In *Mathematical Structures in Functional Programming*, Capreta, V. & McBride, C. (eds), Electronic Notes in Theoretical Computer Science, Reykjavic, Iceland.

McBride, C. & Paterson, R. (2008) Applicative programming with effects, *J. Funct. Program.*, **18** (1): 1–13.

Moggi, E. (1991) Notions of computation and monads, *Inf. Comput.*, **93** (1): 55–92.

Paterson, R. (2001) A new notation for arrows. *Pages 229–240 of: International Conference on Functional Programming*, ACM Press, Florence, Italy.

Power, J. & Robinson, E. (1997) Premonoidal categories and notions of computation, *Math. Struct. Comput. Sci.*, **7** (5): 453–468.

Power, J. & Thielecke, H. (1999) Closed Freyd- and kappa-categories. In *International colloquium on automata, languages, and programming*, LNCS, vol. 1644. Springer, Prague, Czech Republic.

Sabry, A. & Felleisen, M. (1993) Reasoning about programs in continuation-passing style, *Lisp Symbol. Comput.*, **6** (3/4): 289–360.