Theory SIG Reading Group
Arrows - Part 1(b)

Mike Stannett

Department of Computer Science, University of Sheffield
Regent Court, 211 Portobello Street, Sheffield S1 4DP, UK.
M.Stannett@dcs.shef.ac.uk

18 November 2005

Overview

(4.2) Arrows and Interpreters

A Tiny language

Monadic Version

Arrow Version

Comparison

A Slightly Bigger language

Interpreting Conditionals

Arrow-based Conditionals

ArrowChoice

Arrow implementation of evaluation

Interpreting λ-calculus

ArrowApply

ArrowApply for Kleisli arrows

ArrowApply as a Monad

Further Reading

Overview

This is Part 1.2 of the Theory SIG Reading Group discussion of the online 1998 version of John Hughes' paper *Generalising Monads to Arrows* [Hug00]. Parts 1.1 and 1.3 are in a separate document by Simon Foster. Part 2 will be dealt with another week, time and inclination permitting.

1.
   1.1 $\S\S 1 - 4.1$ Motivation and Definition
   1.2 $\S 4.2$ Arrows and Interpreters
   1.3 $\S 5$ S & D's Parsers as Arrows
2. $\S\S 6 - 10$ Another time

(4.2) Arrows and Interpreters

- How difficult is it to program with arrows instead of monads?
- How expressive are arrows? Are there things you can do with $\gg=$ and `return` but not with `arr`, $\ggg$ and `first`?

Hughes addresses these questions by considering a language interpreter based on arrows, and comparing with one based on monads.

A Tiny Language

$$\begin{array}{lcl} \textbf{data}\ \mathtt{Exp} & = & \mathtt{Var\ String} \mid \mathtt{Add\ Exp\ Exp} \\ \textbf{data}\ \mathtt{Val} & = & \mathtt{Number\ Int} \\ \textbf{type}\ \mathtt{Env} & = & [(\mathtt{String}, \mathtt{Val})] \end{array}$$

- An *expression* is either a variable, or else it's obtained by adding two existing expressions together.
- A *value* is an integer tagged with the label `Number` (Hughes uses `Num`, but this has a specific meaning these days, so I've changed it).
- An *environment* comprises a list of pairs, where each pair says what value is assigned to which variable.

Monadic Version

According to Hughes' interpretation, a monad `M` maps objects of type `a` to computations of type `M a`. He wants to map *expressions* to *value computations*, so he is interested in using a monad `M Val`. Given an expression, *evaluating* that expression in a given environment means identifying the computation that generates the appropriate result. Hughes defines

$$
\begin{array}{l}
\mathtt{eval} \quad :: \quad \mathtt{Exp} \rightarrow \mathtt{Env} \rightarrow \mathtt{M\ Val} \\
\mathtt{eval} \quad (\mathtt{Var\ s})\ \mathtt{env} \quad = \quad \mathtt{return}\ (\mathtt{lookup\ s\ env}) \\
\mathtt{eval} \quad (\mathtt{Add}\ \mathtt{e}_1\ \mathtt{e}_2)\ \mathtt{env} \quad = \quad \mathtt{liftM2\ add}\ (\mathtt{eval}\ \mathtt{e}_1\ \mathtt{env})(\mathtt{eval}\ \mathtt{e}_2\ \mathtt{env}) \\
\quad \mathbf{where} \\
\qquad \mathtt{add}\ (\mathtt{Number\ u})(\mathtt{Number\ v}) \quad = \quad \mathtt{Number}\ (\mathtt{u} + \mathtt{v})
\end{array}
$$

Arrow Version

According to Hughes' interpretation, an arrow also represents computations. What should the *input* to the arrow be? Since the result of the computation depends crucuially upon the current environment, Hughes' takes the input to *be* the environment. This time he defines

$$
\begin{array}{lcl}
\mathtt{eval} & :: & \mathtt{Exp} \rightarrow \mathtt{A\ Env\ Val} \\
\mathtt{eval} & (\mathtt{Var\ s}) & = \ \mathtt{arr}\ (\mathtt{lookup\ s}) \\
\mathtt{eval} & (\mathtt{Add}\ \mathtt{e}_1\ \mathtt{e}_2) & = \ \mathtt{liftA2\ add}\ (\mathtt{eval\ e}_1)(\mathtt{eval\ e}_2) \\
\quad \textbf{where} & & \\
\quad\quad \mathtt{add}\ (\mathtt{Number\ u})(\mathtt{Number\ v}) & = & \mathtt{Number}\ (\mathtt{u} + \mathtt{v})
\end{array}
$$

Comparison

There's not much difference – in this case, at least, arrows are just as easy to use as monads. In fact, arrows are easier to use in this situation, as we don't have to keep passing the `Env` variable to our functions.

$$
\begin{aligned}
&\mathtt{evalM} :: \mathtt{Exp} \to \mathtt{Env} \to \mathtt{M\ Val} \\
&\mathtt{evalM}\ (\mathtt{Var\ s})\ \mathtt{env} = \mathtt{return}\ (\mathtt{lookup\ s\ env}) \\
&\mathtt{evalM}\ (\mathtt{Add}\ \mathtt{e}_1\ \mathtt{e}_2)\ \mathtt{env} = \mathtt{liftM2\ add}\ (\mathtt{eval}\ \mathtt{e}_1\ \mathtt{env})(\mathtt{eval}\ \mathtt{e}_2\ \mathtt{env}) \\
&\quad \textbf{where} \\
&\qquad \mathtt{add}\ (\mathtt{Number\ u})(\mathtt{Number\ v}) = \mathtt{Number}\ (\mathtt{u} + \mathtt{v})
\end{aligned}
$$

$$
\begin{aligned}
&\mathtt{evalA} :: \mathtt{Exp} \to \mathtt{A\ Env\ Val} \\
&\mathtt{evalA}\ (\mathtt{Var\ s}) = \mathtt{arr}\ (\mathtt{lookup\ s}) \\
&\mathtt{evalA}\ (\mathtt{Add}\ \mathtt{e}_1\ \mathtt{e}_2) = \mathtt{liftA2\ add}\ (\mathtt{eval}\ \mathtt{e}_1)(\mathtt{eval}\ \mathtt{e}_2) \\
&\quad \textbf{where} \\
&\qquad \mathtt{add}\ (\mathtt{Number\ u})(\mathtt{Number\ v}) = \mathtt{Number}\ (\mathtt{u} + \mathtt{v})
\end{aligned}
$$

A Slightly Bigger Language

$$\begin{array}{lcl} \textbf{data } \texttt{Exp} & = & \texttt{Var String} \mid \texttt{Add Exp Exp} \mid \texttt{If Exp Exp Exp} \\ \textbf{data } \texttt{Val} & = & \texttt{Number Int} \mid \texttt{Boolean Bool} \\ \textbf{type } \texttt{Env} & = & [(\texttt{String}, \texttt{Val})] \end{array}$$

- An *expression* is either a variable, or else it's obtained by adding two existing expressions together, or it's a conditional statement.
- A *value* is an integer tagged with the label `Number` or a boolean tagged with the label `Boolean`.
- An *environment* comprises a list of pairs, where each pair says what value is assigned to which variable.

Interpreting Conditionals

Easy with monads:

$$
\begin{aligned}
&\texttt{evalM} \;::\; \texttt{Exp} \to \texttt{Env} \to \texttt{M Val} \\
&\quad \ldots \\
&\texttt{evalM} \;\; (\texttt{If } \texttt{e}_1 \; \texttt{e}_2 \; \texttt{e}_3) \;\; = \\
&\texttt{evalM } \texttt{e}_1 \texttt{ env} \;\; \gg\!= \\
&\quad \lambda \; \texttt{Boolean b} \;\; \to \\
&\qquad\quad \textbf{if } b \textbf{ then } \;\texttt{eval } \texttt{e}_2 \texttt{ env} \;\; \textbf{else} \;\; \texttt{eval } \texttt{e}_3 \texttt{ env}
\end{aligned}
$$

Not so easy with arrows! The 'obvious' answer would be

$$
\begin{aligned}
&\texttt{evalA} \;::\; \texttt{Exp} \to \texttt{Env} \to \texttt{M Val} \\
&\quad \ldots \\
&\texttt{evalA} \;\; (\texttt{If } \texttt{e}_1 \; \texttt{e}_2 \; \texttt{e}_3) \;\; = \\
&\quad \texttt{evalA } \texttt{e}_1 \;\&\&\&\; \texttt{evalA } \texttt{e}_2 \;\&\&\&\; \texttt{evalA } \texttt{e}_3 \;\; \ggg \\
&\qquad\quad \texttt{arr} \; (\lambda \; (\texttt{Boolean b}, (\texttt{v}_1, \texttt{v}_2)) \;\; \to \\
&\qquad\qquad\quad \textbf{if } b \textbf{ then } \; \texttt{v}_1 \;\; \textbf{else} \;\; \texttt{v}_2
\end{aligned}
$$

Problem: Evaluates both branches, not just the relevant one.

Arrow-based Conditionals

We need to choose between two arrows based on the input, so we use the `Maybe` type:

$$\mathbf{data}\ \texttt{Either a b} \quad = \quad \texttt{Left a} \mid \texttt{Right b}$$

Hughes aims to define a new class `ArrowChoice` and a new function $|||$ to choose between arrows:

$$||| \quad :: \quad \texttt{ArrowChoice a} \Rightarrow \texttt{a b d} \rightarrow \texttt{a c d} \rightarrow \texttt{a (Either b c) d}$$

The expression $(\texttt{f} \ ||| \ \texttt{g})$ passes `Left` inputs to `f` and `Right` inputs to `g`.

ArrowChoice

First, Hughes defines the `ArrowChoice` class:

$$
\begin{array}{l}
\textbf{class}\ \ \texttt{Arrow a} \Rightarrow \texttt{ArrowChoice a}\ \ \textbf{where} \\
\quad \texttt{left} \ \ :: \ \ \mathsf{a\ b\ c} \longrightarrow \mathsf{a}\ (\texttt{Either}\ \mathsf{b\ d})\ (\texttt{Either}\ \mathsf{c\ d})
\end{array}
$$

The expression `left f` invokes `f` only on `Left` inputs, leaving `Right` inputs unaffected. For example,

$$
\begin{array}{l}
\textbf{instance}\ \ \texttt{Monad m} \Rightarrow \texttt{ArrowChoice}\ (\texttt{Kleisli m})\ \ \textbf{where} \\
\quad \texttt{left}\ (\texttt{K f}) \ = \ \texttt{K}\ (\lambda \mathrm{x} \rightarrow \\
\qquad\quad \textbf{case}\ x\ \textbf{of} \\
\qquad\qquad \mathsf{Left\ b} \qquad\quad \rightarrow \ \mathsf{f\ b} \gg\!= \lambda\ \ \texttt{c} \ \rightarrow \ \ \texttt{return}\ (\texttt{Left}\ \mathsf{c}) \\
\qquad\qquad \mathsf{Right\ d} \qquad\quad \rightarrow \ \texttt{return}\ (\texttt{Right}\ \mathsf{d}))
\end{array}
$$

ArrowChoice - 2

Now Hughes defines

```
right f = arr mirror >>> left f >>> arr mirror
  where
    mirror (Left x) = Right x
    mirror (Right x) = Left x
```

```
f <+> g = left f >>> right g
```

```
f ||| g = (f <+> g) >>> arr untag
  where
    untag (Left x) = x
    untag (Right x) = x
```

ArrowChoice - 2

Now Hughes defines

```
right f  =  arr mirror >>> left f >>> arr mirror
  where
    mirror (Left x) = Right x
    mirror (Right x) = Left x
```

```
f <+> g  =  left f >>> right g
```

```
f ||| g  =  (f <+> g) >>> arr untag
  where
    untag (Left x) = x
    untag (Right x) = x
```

ArrowChoice - 2

Now Hughes defines

```
right f = arr mirror >>> left f >>> arr mirror
  where
    mirror (Left x) = Right x
    mirror (Right x) = Left x
```

```
f <+> g = left f >>> right g
```

```
f ||| g = (f <+> g) >>> arr untag
  where
    untag (Left x) = x
    untag (Right x) = x
```

Arrow implementation of evaluation

Hughes is now able to define arrow-based evaluation.

$$
\begin{array}{l}
\texttt{eval}\ (\texttt{If}\ \texttt{e}_1\ \texttt{e}_2\ \texttt{e}_3)\ = \\
\quad (\texttt{eval}\ \texttt{e}_1\ \&\&\&\ \texttt{arr}\ \texttt{id})\ \ggg \\
\qquad \texttt{arr}\ (\lambda(\texttt{Boolean}\ \texttt{b}, \texttt{env}) \rightarrow \\
\qquad\quad \textbf{if}\ \texttt{b}\ \textbf{then}\ \texttt{Left}\ \texttt{env}\ \textbf{else}\ \texttt{Right}\ \texttt{env})\ \ggg \\
\qquad\qquad (\texttt{eval}\ \texttt{e}_2\ \|\!|\!|\ \texttt{eval}\ \texttt{e}_3)
\end{array}
$$

Arrow implementation of evaluation

Hughes is now able to define arrow-based evaluation.

$$
\begin{array}{l}
\mathtt{eval}\ (\mathtt{If}\ \mathtt{e}_1\ \mathtt{e}_2\ \mathtt{e}_3)\ = \\
\quad (\mathtt{eval}\ \mathtt{e}_1\ \&\&\&\ \mathtt{arr}\ \mathtt{id})\ \ggg \\
\quad\quad \mathtt{arr}\ (\lambda(\mathtt{Boolean}\ \mathtt{b}, \mathtt{env}) \rightarrow \\
\quad\quad\quad \mathbf{if}\ \mathtt{b}\ \mathbf{then}\ \mathtt{Left}\ \mathtt{env}\ \mathbf{else}\ \mathtt{Right}\ \mathtt{env})\ \ggg \\
\quad\quad\quad\quad (\mathtt{eval}\ \mathtt{e}_2\ |||\ \mathtt{eval}\ \mathtt{e}_3)
\end{array}
$$

[mps: This is horrible!]

Arrow implementation of evaluation

Hughes is now able to define arrow-based evaluation.

$$
\begin{array}{l}
\texttt{eval}\ (\texttt{If}\ \texttt{e}_1\ \texttt{e}_2\ \texttt{e}_3)\ =\\
\quad (\texttt{eval}\ \texttt{e}_1\ \&\&\&\ \texttt{arr}\ \texttt{id})\ \ggg\\
\qquad \texttt{arr}\ (\lambda(\texttt{Boolean}\ \texttt{b}, \texttt{env}) \to\\
\qquad\quad \textbf{if}\ \texttt{b}\ \textbf{then}\ \texttt{Left}\ \texttt{env}\ \textbf{else}\ \texttt{Right}\ \texttt{env})\ \ggg\\
\qquad\qquad (\texttt{eval}\ \texttt{e}_2\ \|\!|\ \texttt{eval}\ \texttt{e}_3)
\end{array}
$$

Hughes suggests a simplification:

$$
\begin{array}{l}
\texttt{test}\ ::\ \texttt{Arrow}\ \texttt{a} \Rightarrow \texttt{a}\ \texttt{b}\ \texttt{Bool} \to \texttt{a}\ \texttt{b}\ (\texttt{Either}\ \texttt{b}\ \texttt{b})\\
\texttt{test}\ \texttt{f}\ =\ (\texttt{f}\ \&\&\&\ \texttt{arr}\ \texttt{id})\ \ggg\ \texttt{arr}\ (\lambda(\texttt{Boolean}\ \texttt{b}, \texttt{env}) \to\\
\quad \textbf{if}\ b\ \textbf{then}\ \texttt{Left}\ \texttt{env}\ \textbf{else}\ \texttt{Right}\ \texttt{env}\ )\\
\texttt{eval}\ (\texttt{If}\ \texttt{e}_1\ \texttt{e}_2\ \texttt{e}_3)\ =\\
\quad \texttt{test}\ (\texttt{eval}\ \texttt{e}_1 \ggg \texttt{arr}\ (\lambda(\texttt{Boolean}\ \texttt{b}) \to \texttt{b}))(\texttt{eval}\ \texttt{e}_2\ \|\!|\ \texttt{eval}\ \texttt{e}_3)
\end{array}
$$

Arrow implementation of evaluation

Hughes is now able to define arrow-based evaluation.

$$
\begin{array}{l}
\texttt{eval}\ (\texttt{If}\ \texttt{e}_1\ \texttt{e}_2\ \texttt{e}_3) \ = \\
\quad (\texttt{eval}\ \texttt{e}_1\ \&\&\&\ \texttt{arr id}) \ \ggg \\
\qquad \texttt{arr}\ (\lambda(\texttt{Boolean b}, \texttt{env}) \rightarrow \\
\qquad\quad \textbf{if}\ \texttt{b}\ \textbf{then}\ \texttt{Left env}\ \textbf{else}\ \texttt{Right env}) \ \ggg \\
\qquad\qquad (\texttt{eval}\ \texttt{e}_2\ \|\!|\ \texttt{eval}\ \texttt{e}_3)
\end{array}
$$

Hughes suggests a 'simplification'???:

$$
\begin{array}{l}
\texttt{test}\ ::\ \texttt{Arrow a} \Rightarrow \texttt{a b Bool} \rightarrow \texttt{a b (Either b b)} \\
\texttt{test f}\ = \ (\texttt{f}\ \&\&\&\ \texttt{arr id})\ \ggg\ \texttt{arr}\ (\lambda(\texttt{Boolean b}, \texttt{env}) \rightarrow \\
\quad \textbf{if}\ b\ \textbf{then}\ \texttt{Left env}\ \textbf{else}\ \texttt{Right env}\ ) \\
\texttt{eval}\ (\texttt{If}\ \texttt{e}_1\ \texttt{e}_2\ \texttt{e}_3) \ = \\
\quad \texttt{test}\ (\texttt{eval}\ \texttt{e}_1\ \ggg\ \texttt{arr}\ (\lambda(\texttt{Boolean b}) \rightarrow \texttt{b}))(\texttt{eval}\ \texttt{e}_2\ \|\!|\ \texttt{eval}\ \texttt{e}_3)
\end{array}
$$

[mps: This is still horrible!]

Interpreting λ-calculus

We can obviously extend interpretation to handle a complete first-order language.

But what about higher-order stuff?

Monadic version

We add *lambda expressions* and *applications*.

$$\textbf{data}\ \texttt{Exp} = \texttt{Var String} \mid \texttt{Add Exp Exp} \mid \texttt{If Exp Exp Exp} \mid \texttt{Lam String Exp} \mid \texttt{App Exp Exp}$$

```
data Exp =
  Var String |
  Add Exp Exp |
  If Exp Exp Exp |
  Lam String Exp |
  App Exp Exp
```

A *function* maps a value to another value, possibly with a side-effect.

```
data Val =
  Number Int |
  Boolean Bool |
  Fun (Val → M Val)
```

Monadic version - 2

Evaluation is extended accordingly:

$$
\begin{array}{l}
\mathtt{eval}\ (\mathtt{Lam}\ \mathtt{x}\ \mathtt{e})\ \mathtt{env}\ =\ \mathtt{return}\ (\mathtt{Fun}\ (\lambda \mathtt{v} \rightarrow \mathtt{eval}\ \mathtt{e}\ ((\mathtt{x}, \mathtt{v}) : \mathtt{env})))\\
\mathtt{eval}\ (\mathtt{App}\ \mathtt{e}_1\ \mathtt{e}_2)\ \mathtt{env}\ =\\
\qquad \mathtt{eval}\ \mathtt{e}_1\ \mathtt{env} \gg\!= \lambda \mathtt{f} \rightarrow\\
\qquad \mathtt{eval}\ \mathtt{e}_2\ \mathtt{env} \gg\!= \lambda \mathtt{v} \rightarrow\\
\qquad \mathtt{f}\ \mathtt{v}
\end{array}
$$

Arrow version

We again add *lambda expressions* and *applications*.

```
data Exp =
  Var String |
  Add Exp Exp |
  If Exp Exp Exp |
  Lam String Exp |
  App Exp Exp
```

A *function* maps a value to another value, this time via an arrow representation:

```
data Val =
  Number Int |
  Boolean Bool |
  Fun (A Val Val)
```

Arrow version -2

Can evaluate Lam easily enough (?), but App is trickier. Here's Lam:

```
eval (Lam x e) =
  arr (λenv →
      Fun (arr (λv → (x, v): env) >>> eval e))
```

Arrow version -2

Can evaluate `Lam` easily enough (?), but `App` is trickier. Here's `Lam`:

$$
\begin{array}{l}
\texttt{eval}\ (\texttt{Lam x e}) \ = \\
\quad \texttt{arr}\ (\lambda\texttt{env} \rightarrow \\
\qquad\quad \texttt{Fun}\ (\texttt{arr}\ (\lambda\texttt{v} \rightarrow (\texttt{x}, \texttt{v}) : \texttt{env}) \ggg \texttt{eval e}))
\end{array}
$$

Discussion: How does this work?

ArrowApply

As before, Hughes defines a new class to handle the new behaviour.

$$\begin{array}{l}\textbf{class}\ \ \texttt{Arrow a} \Rightarrow \texttt{ArrowApply a}\ \ \textbf{where}\\ \quad \texttt{app}\ \ ::\ \ \texttt{a}\ (\texttt{a b c}, \texttt{b})\ \texttt{c}\end{array}$$

and defines `eval` by

$$\begin{array}{l}\texttt{eval}\ (\texttt{App}\ \texttt{e}_1\ \texttt{e}_2)\ \ =\\ \quad ((\texttt{eval}\ \texttt{e}_1 \ggg (\lambda(\texttt{Fun f}) \rightarrow \texttt{f}))\ \&\&\&\ \texttt{eval}\ \texttt{e}_2) \ggg \texttt{app}\end{array}$$

ArrowApply

As before, Hughes defines a new class to handle the new behaviour.

$$\textbf{class}\ \ \mathtt{Arrow\ a} \Rightarrow \mathtt{ArrowApply\ a}\ \ \textbf{where}$$
$$\quad \mathtt{app} \ :: \ \mathtt{a}\ (\mathtt{a\ b\ c}, \mathtt{b})\ \mathtt{c}$$

and defines `eval` by

$$\mathtt{eval}\ (\mathtt{App}\ \mathtt{e}_1\ \mathtt{e}_2) \ =$$
$$\quad ((\mathtt{eval}\ \mathtt{e}_1 \ggg (\lambda(\mathtt{Fun\ f}) \rightarrow \mathtt{f}))\ \&\&\&\ \mathtt{eval}\ \mathtt{e}_2) \ggg \mathtt{app}$$

Discussion: What's going on here?

ArrowApply for Kleisli arrows

Hughes shows how to use his technology with Kleisli arrows.

$$
\begin{array}{l}
\textbf{instance}\ \ \texttt{Monad m} \Rightarrow \texttt{ArrowApply}\ (\texttt{Kleisli m})\ \ \textbf{where} \\
\quad app \ = \ K\ (\lambda(K\ f, x) \rightarrow f\ x)
\end{array}
$$

ArrowApply for Kleisli arrows

Hughes next seems to say that arrows are unnecessary – any arrow type that supports app can be represented as a Monad.

$$\begin{array}{l}\textbf{data}\ \ \texttt{Void}\ \ =\ \ \texttt{undefined} \\ \textbf{newtype}\ \ \texttt{ArrowApply a} \Rightarrow \texttt{ArrowMonad a b}\ \ =\ \ \texttt{M}\ (\texttt{a Void b})\end{array}$$

and

$$\begin{array}{l}\textbf{instance}\ \ \texttt{ArrowApply a} \Rightarrow \texttt{Monad}\ (\texttt{ArrowMonad a})\ \ \textbf{where} \\ \quad \texttt{return x}\ \ =\ \ \texttt{M}(\texttt{arr}\ (\lambda \texttt{z} \rightarrow \texttt{x})) \\ \quad \texttt{M m} \ggg\!= \texttt{f}\ \ = M(\\ \qquad\qquad\quad \texttt{m}\ \ggg \\ \qquad\qquad\quad \texttt{arr}(\lambda \texttt{x} \rightarrow\ \ \textbf{let}\ \ \texttt{M h} = \texttt{f x}\ \ \textbf{in}\ \ (\texttt{h}, \texttt{undefined}))\ \ \ggg \\ \qquad\qquad\quad \texttt{app})\end{array}$$

Discussion: What do you think so far?

Further Reading

J. Hughes.
Generalising Monads to Arrows.
Science of Computer Programming, 37(1–3):67–111, 2000.