Causal commutative arrows revisited

Jeremy Yallop Hai Liu

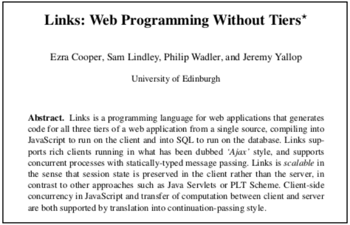
WadlerFest
Edinburgh, April 2016

Links and web forms

Month:

Day:

```
let date =
 formlet
   <div>
     Month: {input int ⇒ month}
     Day: {input int ⇒ day}
   </div>
 yields {month, day}
```

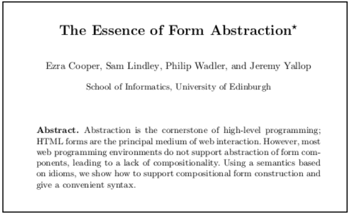


Links: Web Programming Without Tiers*

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop

University of Edinburgh

Abstract. Links is a programming language for web applications that generates code for all three tiers of a web application from a single source, compiling into JavaScript to run on the client and into SQL to run on the database. Links supports rich clients running in what has been dubbed *'Ajax'* style, and supports concurrent processes with statically-typed message passing. Links is *scalable* in the sense that session state is preserved in the client rather than the server, in contrast to other approaches such as Java Servlets or PLT Scheme. Client-side concurrency in JavaScript and transfer of computation between client and server are both supported by translation into continuation-passing style.



The Essence of Form Abstraction*

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop

School of Informatics, University of Edinburgh

Abstract. Abstraction is the cornerstone of high-level programming; HTML forms are the principal medium of web interaction. However, most web programming environments do not support abstraction of form components, leading to a lack of compositionality. Using a semantics based on idioms, we show how to support compositional form construction and give a convenient syntax.

Monads vs arrows vs applicatives

`return`

`>>=`

Monads

`arr`

$\ggg$

`first`

Arrows

`pure`

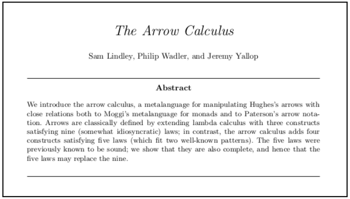
$\circledast$

Applicatives

Evaluators and the arrow calculus

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

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

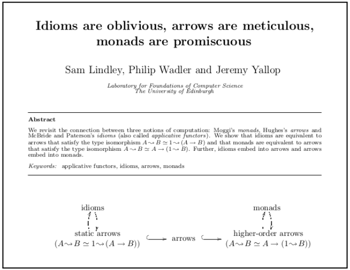


The Arrow Calculus

Sam Lindley, Philip Wadler, and Jeremy Yallop

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.



Idioms are oblivious, arrows are meticulous, monads are promiscuous

Sam Lindley, Philip Wadler and Jeremy Yallop

Laboratory for Foundations of Computer Science
The University of Edinburgh

Abstract

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

Keywords: applicative functors, idioms, arrows, monads

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

Normal forms in Haskell

```
class Applicative f where
   pure :: α → f α
   (⊛) :: f (α → β) → f α → f β
```

```
   pure (f v)    ≡   pure f ⊛ pure v
            u    ≡   pure id ⊛ u
  u ⊛ (v ⊛ w)    ≡   pure (.) ⊛ u ⊛ v ⊛ w
   v ⊛ pure x    ≡   pure (λf → f x) ⊛ v
```

$$\texttt{pure f} \circledast \texttt{c}_1 \circledast \texttt{c}_2 \circledast \ldots \circledast \texttt{c}_n$$

Normal forms in Haskell (continued)

```
data AppNF :: (* → *) → (* → *) where
  Pure :: α → AppNF i α
  (:⊛) :: AppNF i (α → β) → i α → AppNF i β


instance Applicative (AppNF i) where
  pure = Pure
  Pure f ⊛ Pure x = Pure (f x)
  u      ⊛ v :⊛ w = (Pure (.) ⊛ u ⊛ v) :⊛ w
  u      ⊛ Pure x = Pure (λf → f x) ⊛ u


promote :: Applicative i ⇒ i α → AppNF i α
promote i = Pure id :⊛ i


observe :: Applicative i ⇒ AppNF i α → i α
observe (Pure v) = pure v
observe (f :⊛ v) = observe f ⊛ v
```

Normal forms in Haskell: example

```
pure f ⊛ (pure g ⊛ promote h)
   ⇓
Pure f ⊛ (Pure g ⊛ (Pure id :⊛ h))
   ⇓
Pure f ⊛ ((Pure (.) ⊛ Pure g ⊛ Pure id) :⊛ h)
   ⇓
Pure f ⊛ ((Pure ((.) g) ⊛ Pure id) :⊛ h)
   ⇓
Pure f ⊛ (Pure (g . id) :⊛ h)
   ⇓
(Pure (.) ⊛ Pure f ⊛ Pure (g . id)) :⊛ h
   ⇓
Pure (f . g . id) :⊛ h
```

Arrows

```
class Arrow (⇝) where
  pure :: (α → β) → (α ⇝ β)
  (⋙) :: (α ⇝ β) → (β ⇝ γ) → (α ⇝ γ)
  first :: (α ⇝ β) → ((α, γ) ⇝ (β, γ))
```

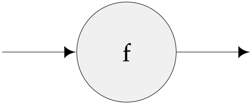
```
              arr id ⋙ f ≡ f
              f ⋙ arr id ≡ f
           (f ⋙ g) ⋙ h ≡ f ⋙ (g ⋙ h)
               arr (g . f) ≡ arr f ⋙ arr g
            arr (f ** id) ≡ first (arr f)
          first (f ⋙ g) ≡ first f ⋙ first g
second (arr g) ⋙ first f ≡ first f ⋙ second (arr g)
            arr fst ⋙ f ≡ first f ⋙ arr fst
    arr assoc ⋙ first f ≡ first (first f) ⋙ arr assoc
```

where

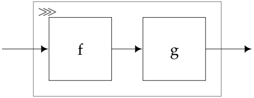
```
  second f = arr swap ⋙ first f ⋙ arr swap
  f ** g = λ(x,y) → (f x, g y)
  assoc ((a, b), c) = (a, (b, c))
  swap (a, b) = (b, a)
```

Arrow diagrams

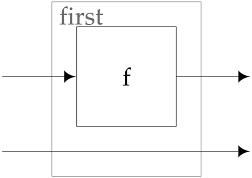
`arr`



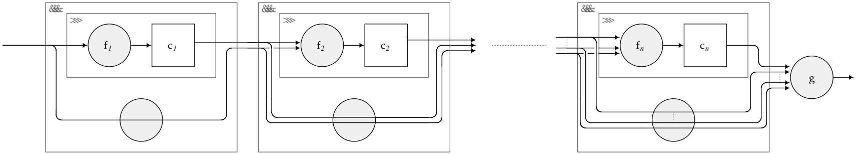
⋙



`first`



Arrow normal form



```
((arr f1 ⋙ c1) &&& arr id) ⋙
((arr f2 ⋙ c2) &&& arr id) ⋙
  ...
  ⋙
((arr fn ⋙ cn) &&& arr id) ⋙
 arr g
```

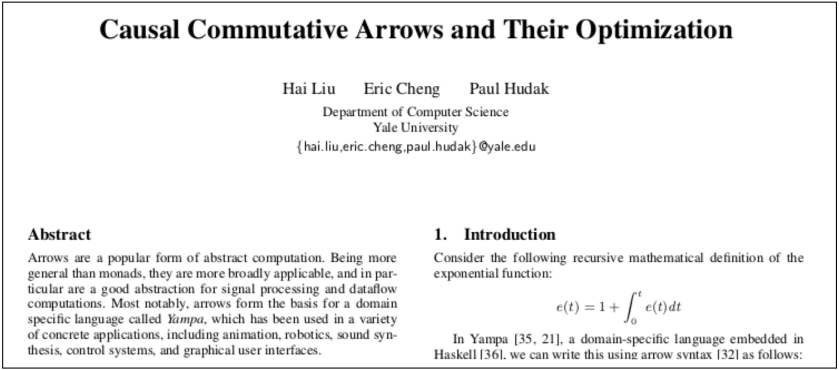
where

```
f &&& g = arr dup ⋙ first f ⋙ second g
dup a = (a, a)
```

Arrows: normalizing implementation

```
data ArrNF :: (* → * → *) → (* → * → *) where
 Arr :: (α → β) → ArrNF (⇝) α β
 Seq :: (α→δ) → (δ⇝γ) → ArrNF (⇝) (γ,α) β → ArrNF (⇝) α β
```

```
instance Arrow (ArrNF (⇝)) where
  arr                = Arr
  Arr f ⋙ Arr g      = Arr (g . f)
  Arr f ⋙ Seq g c h  = Seq (g . f) c
                          (Arr (id ** f) ⋙ h)
  Seq g c h ⋙ s      = Seq g c (h ⋙ s)
  first (Arr f)      = Arr (f ** id)
  first (Seq g c h)  = Seq (g . fst) c
                          (Arr assoc^{-1} ⋙ first h)
    where assoc^{-1} (x,(y,z)) = ((x,y),z)
```



Causal Commutative Arrows and Their Optimization

Hai Liu Eric Cheng Paul Hudak
Department of Computer Science
Yale University
{hai.liu,eric.cheng,paul.hudak}@yale.edu

Abstract

Arrows are a popular form of abstract computation. Being more general than monads, they are more broadly applicable, and in particular are a good abstraction for signal processing and dataflow computations. Most notably, arrows form the basis for a domain specific language called *Yampa*, which has been used in a variety of concrete applications, including animation, robotics, sound synthesis, control systems, and graphical user interfaces.

1. Introduction

Consider the following recursive mathematical definition of the exponential function:

$$e(t) = 1 + \int_0^t e(t)\,dt$$

In Yampa [35, 21], a domain-specific language embedded in Haskell [36], we can write this using arrow syntax [32] as follows:

Programming with CCA

```
exp = proc () → do
       rec let e = 1 + i
           i ←integral⤙ e
           returnA ⤙ e
```

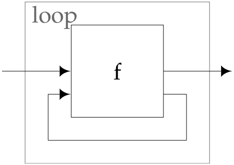
$$e(t) = 1 + \int_0^t e(t)dt$$

```
exp :: ArrowInit (⇝) ⇒ () ⇝ Double
exp = loop (second (integral ⋙ arr (+1)) ⋙
            arr snd ⋙ arr dup)

integral :: ArrowInit (⇝) ⇒ Double ⇝ Double
integral = loop (arr (λ(v, i) → i + dt * v) ⋙
                 init 0 ⋙ arr dup)
```

CCA: new operators, new laws (loop)

```
class Arrow (⇝) ⇒ ArrowLoop (⇝) where
  loop :: ((a, c) ⇝ (b, c)) → (a ⇝ b)
```



$$\begin{array}{rcl}
\texttt{loop (arr f)} & \equiv & \texttt{arr (trace f)} \\
\texttt{loop (first h} \ggg \texttt{f)} & \equiv & \texttt{h} \ggg \texttt{loop f} \\
\texttt{loop (f} \ggg \texttt{first h)} & \equiv & \texttt{loop f} \ggg \texttt{h} \\
\texttt{loop (f} \ggg \texttt{arr (id ** k))} & \equiv & \texttt{loop (arr (id ** k)} \ggg \texttt{f)} \\
\texttt{loop (loop f)} & \equiv & \texttt{loop (arr assoc}^{-1}\texttt{ . f . arr assoc)} \\
\texttt{second (loop f)} & \equiv & \texttt{loop (arr assoc . second f} \\
 & & \qquad\qquad \texttt{. arr assoc}^{-1}\texttt{)}
\end{array}$$

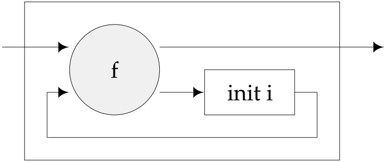
CCA: new operators, new laws (init)

```
class ArrowLoop (⇝) ⇒ ArrowInit (⇝) where
  init :: a → (a ⇝ a)
```

init i

```
first f ⋙ second g  ≡  second g ⋙ first f
  init i *** init j  ≡  init (i,j)
```

CCA normal form



```
loop (arr f ⋙ second (init i))
```

```
exp =
  loop (arr (λ(x, y) → let i = y + 1 in
                            (i, y + dt * i)) ⋙
        second (init 0))
```

CCA Normal form

```
data CCNF :: * → * → * where
   ArrD  :: (a → b) → CCNF a b
   LoopD :: e → ((b,e) → (c,e)) → CCNF b c

instance Arrow CCNF where
   arr = ArrD
   ArrD f ⋙ ArrD g    = ArrD (g . f)
   ArrD f ⋙ LoopD i g = LoopD i (g . first f)
   [...]

instance ArrowLoop CCNF where
   loop (ArrD f ) = ArrD (trace f)
   loop (LoopD i f) = LoopD i (trace (juggle' f))

instance ArrowInit CCNF where init i = LoopD i swap

observe :: ArrowInit (⇝) ⇒ CCNF a b → (a ⇝ b)
observe (ArrD f) = arr f
observe (LoopD i f) = loop (arr f ⋙ second (init i))
```

Performance improvements

Normalization: performance improvements

from: Paul Liu
to: Jeremy Yallop
cc: Paul Hudak, Eric Cheng
date: 18 June 2009

I wonder if there is any way to optimize GHC's output based on your code since the CCNF is actually running slower.

Optimizing observation

```
observe :: ArrowInit (⇝) ⇒ CCNF a b → (a ⇝ b)
observe (ArrD f) = arr f
observe (LoopD i f) = loop (arr f ⋙ second (init i))
```

Optimization opportunities

specialize to an instance

fuse the arrow operators

Specializing observe

```
newtype SF a b = SF (a → (b, SF a b))
```

```
instance Arrow SF where
  arr f = SF h
    where h x = (f x, SF h)

  f ⋙ g = SF (h f g)
    where h (SF f) (SF g) x = let (y, f’) = f x
                                  (z, g’) = g y
                              in (z, SF (h f’ g’))
  ...
```

```
observeSF :: CCNF a b → SF a b
observeSF (ArrD f) = arr f
observeSF (LoopD i f) = loop (arr f ⋙ second (init i))
```

Optimising the specialized observe

```
observeSF (LoopD i f) = loop (arr f ⋙ second (init i))
```

```
observeSF (LoopD i f) = loop_comp2 i f
  where
    arr_swap = arr swap
    arr_swapf f = arr (swap . f)
    first_init i = first (init i)
    comp_1 i f = arr_swapf f ⋙ (first_init i)
    comp_2 i f = comp_1 i f ⋙ arr_swap
    loop_comp2 i f = loop (comp_2 i f)
```

Optimising the specialized `observe`

```
observeSF (LoopD i f) = loop_comp2 i f
  where
    arr_swap = arr swap
    arr_swapf f = arr (swap . f)
    ...
```

rewrites to

```
observeSF (LoopD i f) = loop_comp2 i f
  where
    arr_swap = SF hswap
    hswap (x,y) = ((y,x), SF hswap)
    arr_swapf f = arr (swap . f)
    ...
```

Optimising the specialized `observe`

. . .

rewrites to

. . .

rewrites to

. . .

rewrites to

```
observeSF (LoopD i f) = loopD f i
   where
     loopD f i = SF (λx → let (a,b) = f (x,i) in
                               (a, loopD f b))
```

combining observe and runSF (to give runCCNF)

```
observeSF (LoopD i f) = loopD f i
   where
     loopD f i = SF (λx → let (a,b) = f (x,i) in
                              (a, loopD f b))
```

combined with

```
runSF :: SF a b → [a] → [b]
runSF (SF f) (x:xs) = let (y, g) = f x
                      in y : runSF g xs
```

gives

```
runCCNF :: e → ((b,e) → (c,e)) → [b] → [c]
runCCNF i f = g i
  where g i (x:xs) = let (y, i') = f (x, i)
                     in y : g i' xs
```

combining runCCNF and nth

```
runCCNF :: e → ((b,e) → (c,e)) → [b] → [c]
runCCNF i f = g i
  where g i (x:xs) = let (y, i’) = f (x, i)
                      in y : g i’ xs
```

combined with

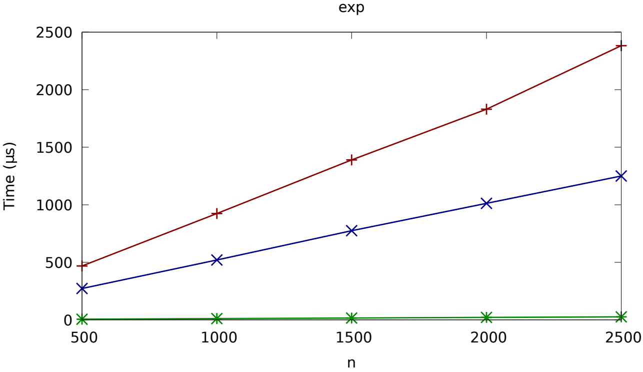
```
nth :: [a] → Int → a
(x:_)  ‘nth‘ 0         =  x
(_:xs) ‘nth‘ n         =  xs ‘nth‘ (n-1)
```

gives

```
nthCCNF :: Int → CCNF () a → a
nthCCNF n (ArrD f) = f ()
nthCCNF n (LoopD i f) = aux n i
  where
    aux n i = x ‘seq‘ if n == 0 then x else aux (n-1) j
     where (x, j) = f ((), i)
```

Performance improvements

`nth elem exp` `nth n (observe exp)` `nthCCNF n exp`



Conclusion