Causal commutative arrows revisited

Jeremy Yallop    Hai Liu

WadlerFest
Edinburgh, April 2016
Links and web forms

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

Monads
- `return`
- `>>=`

Arrows
- `arr`
- `>>>`
- `first`

Applicatives
- `pure`
- `⊗`
Evaluators and the arrow calculus

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

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

  pure f ⊗ c₁ ⊗ c₂ ⊗ ... ⊗ cₙ
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

\[
\begin{align*}
\text{pure } f & \otimes (\text{pure } g \otimes \text{promote } h) \\
\Downarrow & \\
\text{Pure } f & \otimes (\text{Pure } g \otimes (\text{Pure } \text{id} :\otimes h)) \\
\Downarrow & \\
\text{Pure } f & \otimes ((\text{Pure } (.) \otimes \text{Pure } g \otimes \text{Pure } \text{id}) :\otimes h) \\
\Downarrow & \\
\text{Pure } f & \otimes ((\text{Pure } ((.) g) \otimes \text{Pure } \text{id}) :\otimes h) \\
\Downarrow & \\
\text{Pure } f & \otimes (\text{Pure } (g \cdot \text{id}) :\otimes h) \\
\Downarrow & \\
(\text{Pure } (.) \otimes \text{Pure } f \otimes \text{Pure } (g \cdot \text{id})) & :\otimes h \\
\Downarrow & \\
\text{Pure } (f \cdot g \cdot \text{id}) & :\otimes h
\end{align*}
\]
**Arrows**

```haskell
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

f

first

f

first
Arrow normal form

\[
\begin{align*}
((\text{arr } f_1 \rightarrow c_1) \&\& \text{arr } id) \rightarrow \\
((\text{arr } f_2 \rightarrow c_2) \&\& \text{arr } id) \rightarrow \\
\ldots \\
\rightarrow \\
((\text{arr } f_n \rightarrow c_n) \&\& \text{arr } id) \rightarrow \\
\text{arr } g
\end{align*}
\]

where

\[
f \&\& g = \text{arr } \text{dup } \rightarrow \text{first } f \rightarrow \text{second } g \\
\text{dup } a = (a, a)
\]
Arrows: normalizing implementation

```haskell
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)

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

Diagram:

```
loop
   f
```

Laws:

- `loop (arr f) ≡ arr (trace f)`
- `loop (first h >>> f) ≡ h >>> loop f`
- `loop (f >>> first h) ≡ loop f >>> h`
- `loop (f >>> arr (id ** k)) ≡ loop (arr (id ** k) >>> f)`
- `loop (loop f) ≡ loop (arr assoc⁻¹ . f . arr assoc)`
- `second (loop f) ≡ loop (arr assoc . second f . arr assoc⁻¹)`
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

\[
\text{loop } (\text{arr } f \ggg \text{second } (\text{init } i))
\]

\[
\text{exp = loop } (\text{arr } (\lambda(x, y) \rightarrow \text{let } i = y + 1 \text{ in } (i, y + dt \ast i)) \ggg \text{second } (\text{init } 0))
\]
**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
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

\[
\text{observe} :: \text{ArrowInit} (\sim) \Rightarrow \text{CCNF} \ a \ b \rightarrow (a \sim b)
\]

\[
\text{observe} (\text{ArrD} \ f) = \text{arr} \ f
\]

\[
\text{observe} (\text{LoopD} \ i \ f) = \text{loop} (\text{arr} \ f \ggg \text{second} (\text{init} \ i))
\]

Optimization opportunities

specialize to an instance          fuse the arrow operators
Specializing observe

```haskell
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

\[
\text{observeSF} \ (\text{LoopD } i \ f) = \text{loop} \ (\text{arr } f \gg\gg \text{second} \ (\text{init } i))
\]

\[
\text{observeSF} \ (\text{LoopD } i \ f) = \text{loop}_{\text{comp2}} \ i \ f
\]

\[
\text{where}
\]

\[
\begin{align*}
\text{arr}_{\text{swap}} &= \text{arr} \ \text{swap} \\
\text{arr}_{\text{swapf}} \ f &= \text{arr} \ (\text{swap} . \ f) \\
\text{first}_{\text{init}} \ i &= \text{first} \ (\text{init } i) \\
\text{comp}_{1} \ i \ f &= \text{arr}_{\text{swapf}} \ f \gg\gg \text{first}_{\text{init}} \ i \\
\text{comp}_{2} \ i \ f &= \text{comp}_{1} \ i \ f \gg\gg \text{arr}_{\text{swap}} \\
\text{loop}_{\text{comp2}} \ i \ f &= \text{loop} \ (\text{comp}_{2} \ i \ f)
\end{align*}
\]
Optimising the specialized observe

\[
\text{observeSF} \ (\text{LoopD} \ i \ f) = \text{loop}_{\text{comp2}} \ i \ f \\
\text{where} \\
\quad \text{arr}_{\text{swap}} = \text{arr} \ \text{swap} \\
\quad \text{arr}_{\text{swapf}} \ f = \text{arr} \ (\text{swap} \ . \ f) \\
\ldots
\]

rewrites to

\[
\text{observeSF} \ (\text{LoopD} \ i \ f) = \text{loop}_{\text{comp2}} \ i \ f \\
\text{where} \\
\quad \text{arr}_{\text{swap}} = \text{SF} \ \text{hswap} \\
\quad \text{hswap} \ (x,y) = ((y,x), \ \text{SF} \ \text{hswap}) \\
\quad \text{arr}_{\text{swapf}} \ f = \text{arr} \ (\text{swap} \ . \ f) \\
\ldots
\]
Optimising the specialized `observe`

...  

rewrites to  

...

rewrites to  

...

rewrites to

```haskell
observeSF (LoopD i f) = loopD f i  
where  
  loopD f i = SF (\x -> let (a,b) = f (x,i) in (a, loopD f b))
```
combining observe and runSF (to give runCCNF)

\[
\text{observeSF}\ (\text{LoopD}\ i\ f) = \text{loopD}\ f\ i \\
\text{where}
\text{loopD}\ f\ i = \text{SF}\ (\lambda x \to \text{let}\ (a,b) = f\ (x,i)\ \text{in}\ (a, \text{loopD}\ f\ b))}
\]

combined with

\[
\text{runSF} :: \text{SF}\ a\ b \to [a] \to [b] \\
\text{runSF}\ (\text{SF}\ f)\ (x:\text{xs}) = \text{let}\ (y,\ g) = f\ x \\text{in}\ y : \text{runSF}\ g\ \text{xs}
\]

gives

\[
\text{runCCNF} :: e \to ((b,e) \to (c,e)) \to [b] \to [c] \\
\text{runCCNF}\ i\ f = g\ i \\
\text{where}\ g\ i\ (x:\text{xs}) = \text{let}\ (y,\ i’) = f\ (x,\ i) \\text{in}\ y : g\ i’\ \text{xs}
\]
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

![Graph showing performance improvements for nth elem exp, nth n (observe exp), and nthCCNF n exp. The x-axis represents n (from 500 to 2500), and the y-axis represents Time (µs) from 0 to 2500. The graph shows a linear relationship between n and Time for each function.]
Conclusion