Causal Commutative Arrows

Hai (Paul) Liu, Paul Hudak

Yale University

Outline

- Arrows and FRP
  - Introduction
  - Arrow Laws and Properties
- Causal Commutative Arrows (CCA)
  - Syntax, types and Semantics
  - Implementation by Mealy Machine
- Optimization by Normalization
  - Causal Commutative Normal Form (CCNF)
  - Benchmarks

Contributions

- A minimal language that captures the essence of causal computation.
- Two additional laws that lead to normal forms.
- Substantial performance gain via optimization by normalization.

Exponential Example

A math definition of the exponential function:

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

Yampa program using the Arrow Syntax:

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

Functional Reactive Programming

Computations about time-varying quantities.

$$\texttt{Signal}\ \alpha \approx \texttt{Time} \rightarrow \alpha$$

Yampa (Hudak, et. al. 2002) is a version of FRP using the *Arrow* framework (Hughes, 2000). Arrows provide:

- Abstract computation over signals.

$$\texttt{SF}\ \alpha\ \beta \approx \texttt{Signal}\ \alpha \rightarrow \texttt{Signal}\ \beta$$

- A minimum set of *wiring* combinators.
- Mathematics root in category theory.

What is Arrow

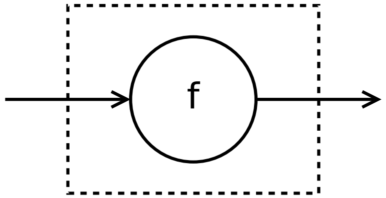
A generalization of Monads. In Haskell:

```
class Arrow a where
  arr   :: (b → c) → a b c
  (>>>) :: a b c → a c d → a b d
  first :: a b c → a (b,d) (c,d)
```

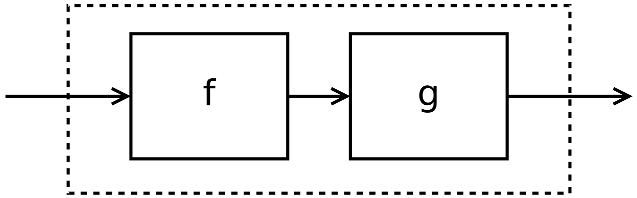
Support both sequential and parallel composition:

```
second :: (Arrow a) ⇒ a b c → a (d,b) (d,c)
second f = arr swap >>> first f >>> arr swap
  where swap (a, b) = (b, a)
(***) :: (Arrow a) ⇒ a b c → a b' c' → a (b, b') (c, c')
f *** g = first f >>> second g
```

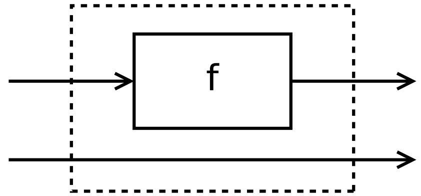
Arrows in Picture



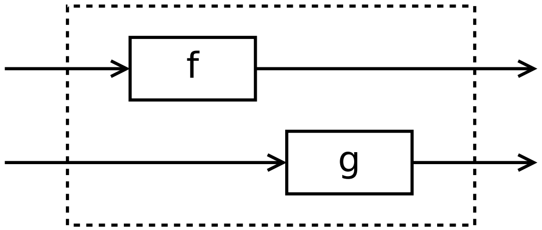
(a) `arr f`



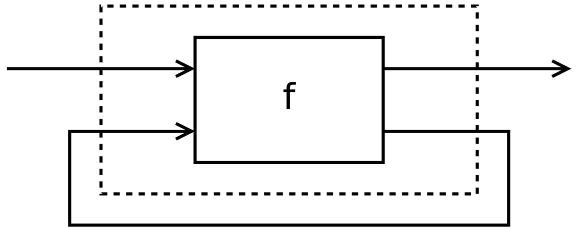
(b) `f >>> g`



(c) `first f`



(d) `f *** g`



(e) `loop f`

To model recursion, Paterson (2001) introduced ArrowLoop:

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

Arrows and FRP

Why do we need Arrows?

- Modular, both input and output are explicit.
- Eliminates a form of time and space leak (Liu and Hudak, 2007).
- Abstract, with properties enforced by arrow laws.

Why do we need abstraction?

- Think at the high level. Focus on the essence.
- Disciplines bring interesting properties and useful results.

Arrow Laws

left identity	$arr\ id \ggg f$	$=$	f
right identity	$f \ggg arr\ id$	$=$	f
associativity	$(f \ggg g) \ggg h$	$=$	$f \ggg (g \ggg h)$
composition	$arr\ (g \cdot f)$	$=$	$arr\ f \ggg arr\ g$
extension	$first\ (arr\ f)$	$=$	$arr (f \times id)$
functor	$first\ (f \ggg g)$	$=$	$first\ f \ggg first\ g$
exchange	$first\ f \ggg arr\ (id \times g)$	$=$	$arr\ (id \times g) \ggg first\ f$
unit	$first\ f \ggg arr\ fst$	$=$	$arr\ fst \ggg f$
association	$first\ (first f) \ggg arr\ assoc$	$=$	$arr\ assoc \ggg first f$
	where $assoc\ ((a, b), c)$	$=$	$(a, (b, c))$

Arrow Loop Laws

left tightening	$loop\ (first\ h \ggg f)$	$=$	$h \ggg loop\ f$
right tightening	$loop\ (f \ggg first\ h)$	$=$	$loop\ f \ggg h$
sliding	$loop\ (f \ggg arr\ (id \times k))$	$=$	$loop\ (arr\ (id \times k) \ggg f)$
vanishing	$loop\ (loop\ f)$	$=$	$loop\ (arr\ assoc^{-1} \ggg f \ggg arr\ assoc)$
superposing	$second\ (loop\ f)$	$=$	$loop\ (arr\ assoc \ggg second\ f \ggg arr\ assoc^{-1})$
extension	$loop\ (arr\ f)$	$=$	$arr(trace\ f)$
	where $trace\ f\ b$	$=$	$\texttt{let}\ (c, d) = f\ (b, d)\ \texttt{in}\ c$

Question

Are the arrow laws enough to capture the essence of FRP? Or more specifically, the notion of causal computation as in dataflow programming and stream processing?

(Causal: current output only depends on current and previous inputs.)

Question

Are the arrow laws enough to capture the essence of FRP? Or more specifically, the notion of causal computation as in dataflow programming and stream processing?

(Causal: current output only depends on current and previous inputs.)

No. They are too general, and we need a domain specific solution.

Causal Commutative Arrows

Introduce one new operator *init* (a.k.a. *delay*):

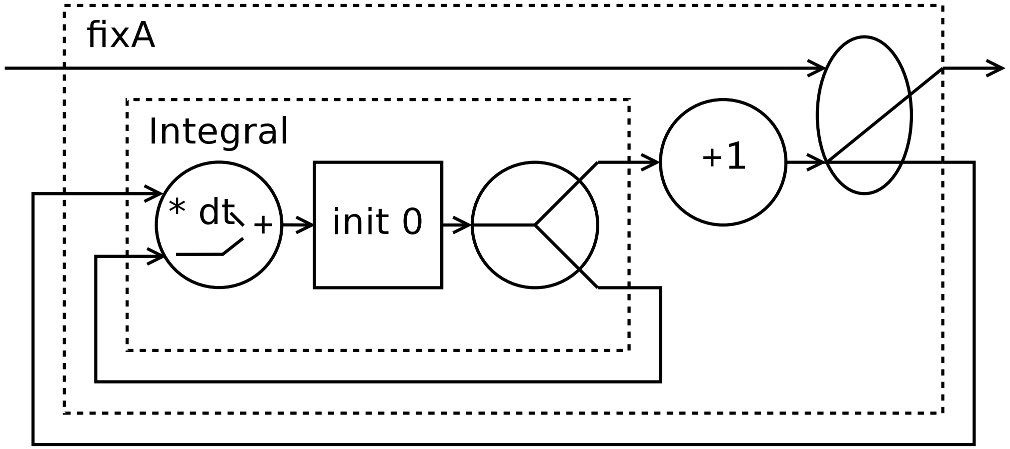
```
class ArrowLoop a ⇒ ArrowInit a where
  init :: b → a b b
```

two additional laws:

commutativity	$\mathit{first}\ f \ggg \mathit{second}\ g$	$=$	$\mathit{second}\ g \ggg \mathit{first}\ f$
product	$\mathit{init}\ i \star\!\star\!\star\ \mathit{init}\ j$	$=$	$\mathit{init}\ (i, j)$

and still remain *abstract*!

Exponential Example, Revisit




```
exp = fixA (integral >>> arr (+1))

fixA :: ArrowLoop a ⇒ a b b → a () b
fixA f = loop (second f >>> arr (λ((), y) → (y, y)))

integral :: ArrowInit a ⇒ a Double Double
integral = loop (arr (λ(v, i) → i + dt ∗ v) >>>
                 init 0 >>> arr (λi → (i, i)))
```

CCA, a Domain Specific Language

- Extend simply typed λ-calculus with tuples and arrows.
- Instead of type classes, use $\leadsto$ to represent the arrow type.

$$
\begin{array}{llll}
\text{Type} & t & ::= & \mathbb{R} \mid \alpha \mid t_1 \times t_2 \mid t_1 \rightarrow t_2 \mid t_1 \leadsto t_2 \\
\text{Exp} & e & ::= & \bot \mid n \mid x \mid (e_1, e_2) \mid \mathit{fst}\ e \mid \mathit{snd}\ e \mid \lambda x.e \mid e_1\ e_2 \mid \\
 & & & \mathit{arr} \mid \ggg \mid \mathit{first} \mid \mathit{loop} \mid \mathit{init} \\
\text{Env} & \Gamma & ::= & x_1 : \alpha_1, \ldots, x_n : \alpha_n
\end{array}
$$

CCA Types

$$\frac{(x : \alpha) \in \Gamma}{\Gamma \vdash x : \alpha} \qquad \frac{\Gamma, x : \alpha \vdash e : \beta}{\Gamma \vdash \lambda x.e : \alpha \rightarrow \beta} \qquad \frac{\Gamma \vdash e_1 : \alpha \rightarrow \beta \quad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash e_1\ e_2 : \beta}$$

$$\frac{\Gamma \vdash e_1 : \alpha \quad \Gamma \vdash e_2 : \beta}{\Gamma \vdash (e_1, e_2) : \alpha \times \beta} \qquad \frac{\Gamma \vdash e : \alpha \times \beta}{\Gamma \vdash \mathit{fst}\ e : \alpha} \qquad \frac{\Gamma \vdash e : \alpha \times \beta}{\Gamma \vdash \mathit{snd}\ e : \beta}$$

$$\begin{array}{rcl}
\mathit{arr} & : & (\alpha \rightarrow \beta) \rightarrow (\alpha \rightsquigarrow \beta) \\
(\ggg) & : & (\alpha \rightsquigarrow \beta) \rightarrow (\beta \rightsquigarrow \theta) \rightarrow (\alpha \rightsquigarrow \theta) \\
\mathit{first} & : & (\alpha \rightsquigarrow \beta) \rightarrow (\alpha \times \theta \rightsquigarrow \beta \times \theta)
\end{array}
\qquad
\begin{array}{rcl}
\mathit{loop} & : & (\alpha \times \theta \rightsquigarrow \beta \times \theta) \rightarrow (\alpha \rightsquigarrow \beta) \\
\mathit{init} & : & \alpha \rightarrow (\alpha \rightsquigarrow \alpha) \\
\perp & : & \alpha
\end{array}$$

CCA Utility Functions

$$
\begin{aligned}
\mathit{id} &: \alpha \to \alpha \\
\mathit{id} &= \lambda x.x \\
\mathit{assoc} &: (\alpha \times \beta) \times \theta \to \alpha \times (\beta \times \theta) \\
\mathit{assoc} &= \lambda z.(\mathit{fst}\ (\mathit{fst}\ z), (\mathit{snd}\ (\mathit{fst}\ z), \mathit{snd}\ z)) \\
\mathit{assoc}^{-1} &: \alpha \times (\beta \times \theta) \to (\alpha \times \beta) \times \theta \\
\mathit{assoc}^{-1} &= \lambda z.((\mathit{fst}\ z, \mathit{fst}\ (\mathit{snd}\ z)), \mathit{snd}\ (\mathit{snd}\ z)) \\
\mathit{juggle} &: (\alpha \times \beta) \times \theta \to (\alpha \times \theta) \times \beta \\
\mathit{juggle} &= \mathit{assoc}^{-1} \cdot (\mathit{id} \times \mathit{swap}) \cdot \mathit{assoc} \\
\mathit{transpose} &: (\alpha \times \beta) \times (\theta \times \eta) \to (\alpha \times \theta) \times (\beta \times \eta) \\
\mathit{transpose} &= \mathit{assoc} \cdot (\mathit{juggle} \times \mathit{id}) \cdot \mathit{assoc}^{-1} \\
\mathit{shuffle} &: \alpha \times ((\beta \times \delta) \times (\theta \times \eta)) \to (\alpha \times (\beta \times \theta)) \times (\delta \times \eta) \\
\mathit{shuffle} &= \mathit{assoc}^{-1} \cdot (\mathit{id} \times \mathit{transpose}) \\
\mathit{shuffle}^{-1} &: (\alpha \times (\beta \times \theta)) \times (\delta \times \eta) \to \alpha \times ((\beta \times \delta) \times (\theta \times \eta)) \\
\mathit{shuffle}^{-1} &= (\mathit{id} \times \mathit{transpose}) \cdot \mathit{assoc}
\end{aligned}
$$

$$
\begin{aligned}
(\cdot) &: (\beta \to \theta) \to (\alpha \to \beta) \to (\alpha \to \theta) \\
(\cdot) &= \lambda f.\lambda g.\lambda x.f(g\ x) \\
(\times) &: (\alpha \to \beta) \to (\theta \to \gamma) \to (\alpha \times \theta \to \beta \times \gamma) \\
(\times) &: \lambda f.\lambda g.\lambda z.(f\ (\mathit{fst}\ z), g\ (\mathit{snd}\ z)) \\
\mathit{dup} &: \alpha \to \alpha \times \alpha \\
\mathit{dup} &= \lambda x.(x, x) \\
\mathit{swap} &: \alpha \times \beta \to \beta \times \alpha \\
\mathit{swap} &= \lambda z.(\mathit{snd}\ z, \mathit{fst}\ z) \\
\mathit{second} &: (\alpha \leadsto \beta) \to (\theta \times \alpha \leadsto \theta \times \beta) \\
\mathit{second} &= \lambda f.\mathit{arr}\ \mathit{swap} \ggg \mathit{first}\ f \ggg \mathit{arr}\ \mathit{swap} \\
(\star\star\star) &: (\alpha \leadsto \beta) \to (\theta \leadsto \gamma) \to (\alpha \times \theta \leadsto \beta \times \gamma) \\
(\star\star\star) &= \lambda f.\lambda g.\mathit{first}\ f \ggg \mathit{second}\ g
\end{aligned}
$$

CCA Semantics

Interpretation of the arrow type:

$$\alpha \rightsquigarrow \beta \ \underset{\psi}{\overset{\phi}{\rightleftarrows}}\ \alpha \longrightarrow (\beta \times (\alpha \rightsquigarrow \beta))$$

Denotational Semantics

$$[\![-]\!] : Exp \longrightarrow \alpha \rightsquigarrow \beta$$

$$\begin{aligned}
[\![arr\ f]\!] &= \psi(h\ [\![f]\!]) & h\ f\ x &= \texttt{let}\ y = f\ x\ \texttt{in}\ (y, \psi(h\ f)) \\
[\![first\ f]\!] &= \psi(h\ [\![f]\!]) & h\ f\ (x,z) &= \texttt{let}\ (y, f') = \phi(f)\ x\ \texttt{in}\ ((y,z), \psi(h\ f')) \\
[\![f \ggg g]\!] &= \psi(h\ [\![f]\!]\ [\![g]\!]) & h\ f\ g\ x &= \texttt{let}\ \{(y, f') = \phi(f)\ x;\ (z, g') = \phi(g)\ y\}\ \texttt{in}\ (z, \psi(h\ f'\ g')) \\
[\![loop\ f]\!] &= \psi(h\ [\![f]\!]) & h\ f\ x &= \texttt{let}\ ((y,z), f') = \phi(f)\ (x,z)\ \texttt{in}\ (y, \psi(h\ f')) \\
[\![init\ i]\!] &= \psi(h\ [\![i]\!]) & h\ i\ x &= (i, \psi(h\ x))
\end{aligned}$$

($[\![-]\!]$ for λ expressions is omitted)

CCA and Mealy Machines

Mealy Machine (Mealy, 1955): (A, B, S, ϕ, s_0)

Inputs A, Outputs B, States S, and $\phi : S \longrightarrow (B \times S)^A$

A *CCA* term $s_0 : \alpha \rightsquigarrow \beta$ is a Mealy machine that maps input stream $<a_0, a_1, \cdots, a_k, \cdots>$ to output stream $<b_0, b_1, \cdots, b_k, \cdots>$

$$s_0 \xrightarrow{a_0 | b_0} s_1 \xrightarrow{a_1 | b_1} \ldots \xrightarrow{a_k | b_k} s_k \xrightarrow{a_{k+1} | b_{k+1}} \ldots$$

single-step transition:

$$s_i \xrightarrow{a_i | b_i} s_{i+1} \quad == \quad (b_i, s_{i+1}) = \phi(s_i)\ a_i$$

CCA and Mealy Machines

Functions as Mealy machine states:

$$\alpha \rightsquigarrow \beta \ \underset{\psi}{\overset{\phi}{\rightleftarrows}} \ \alpha \longrightarrow (\beta \times (\alpha \rightsquigarrow \beta))$$

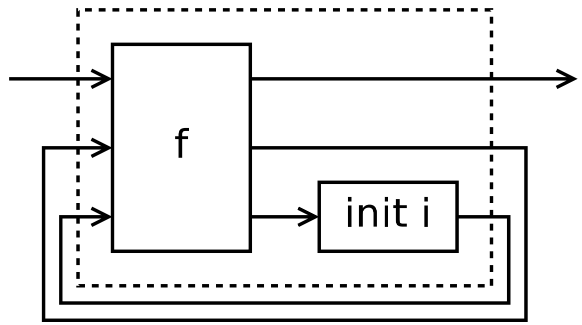
In Haskell, we borrow list type to represent streams:

$$\begin{array}{l} run :: (\alpha \rightsquigarrow \beta) \longrightarrow [\alpha] \longrightarrow [\beta] \\ run\ f\ (x : xs) = \texttt{let}\ (y, f') = \phi(f)\ x\ \texttt{in}\ y : run\ f'\ xs \end{array}$$

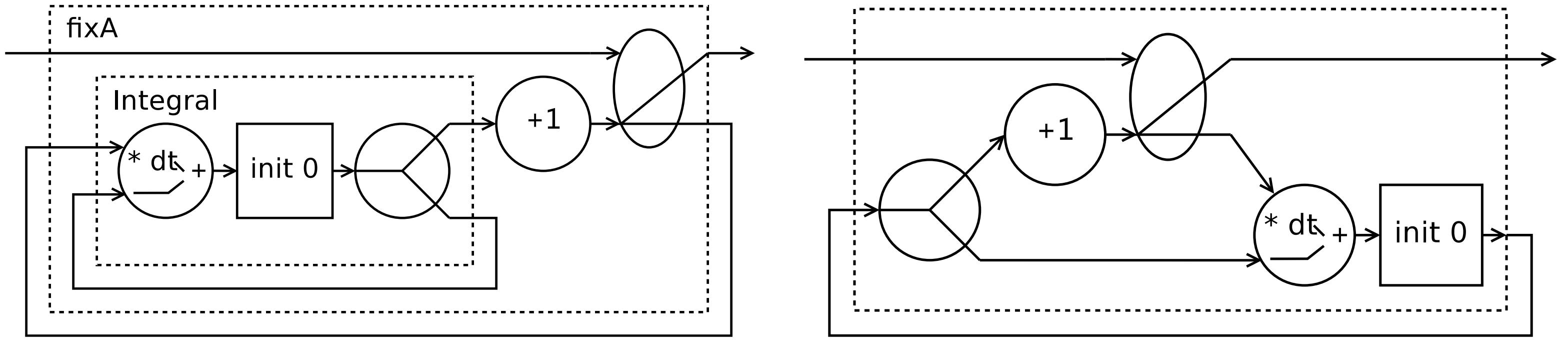
Causal Commutative Normal Form (CCNF)

For all $\vdash e : \alpha \rightsquigarrow \beta$, there exists a normal form e_{norm}, which is either a pure arrow $arr\ f$, or $loopB\ i\ (arr\ g)$, such that $\vdash e_{norm} : \alpha \rightsquigarrow \beta$ and $[\![e]\!] = [\![e_{norm}]\!]$.

$$loopB\ i\ f = loop\ (f \ggg second\ (second\ (init\ i)))$$



Exponential Example Normalized



(f) Original (g) Normalized

CCNF is a single loop containing one pure arrow and one initiated state.

Benchmarks (Speed Ratio, Greater is Better)

	1. GHC	2. arrowp	3. CCNF
exp	1.0	2.2	10.1
sine	1.0	2.9	12.6
oscSine	1.0	1.6	2.7
50's sci-fi	1.0	1.12	5.1
robotSim	1.0	1.4	3.8

- Same CCA source programs written in Arrow syntax.
- Same Haskell implementation of the CCA semantics.
- Only difference:
  1. Translated to arrow combinators by GHC's built-in arrow compiler.
  2. Translated to arrow combinators by Paterson's arrowp preprocessor.
  3. normalized combinator program.

One-step Reduction $\mapsto$

Intuition: extend Arrow Loop laws to $loopB$.

loop	$loop\ f$	$\mapsto$	$loopB\ \bot\ (arr\ assoc^{-1} \ggg first\ f \ggg arr\ assoc)$
init	$init\ i$	$\mapsto$	$loopB\ i\ (arr\ (swap \cdot juggle \cdot swap))$
composition	$arr\ f \ggg arr\ g$	$\mapsto$	$arr\ (g \cdot f)$
extension	$first\ (arr\ f)$	$\mapsto$	$arr\ (f \times id)$
left tightening	$h \ggg loopB\ i\ f$	$\mapsto$	$loopB\ i\ (first\ h \ggg f)$
right tightening	$loopB\ i\ f \ggg arr\ g$	$\mapsto$	$loopB\ i\ (f \ggg first\ (arr\ g))$
vanishing	$loopB\ i\ (loopB\ j\ f)$	$\mapsto$	$loopB\ (i, j)\ (arr\ shuffle \ggg f \ggg arr\ shuffle^{-1})$
superposing	$first\ (loopB\ i\ f)$	$\mapsto$	$loopB\ i\ (arr\ juggle \ggg first\ f \ggg arr\ juggle)$

Normalization Procedure $\mapsto_n$

$$\frac{}{e \mapsto_n e} \quad \exists i, f \text{ s.t. } e = arr\ f \text{ or } e = loopB\ i\ (arr\ f)$$

$$\frac{e_1 \mapsto_n e_1' \quad e_2 \mapsto_n e_2' \quad e_1' \ggg e_2' \mapsto e \quad e \mapsto_n e'}{e_1 \ggg e_2 \mapsto_n e'}$$

$$\frac{f \mapsto_n f' \quad first\ f' \mapsto e \quad e \mapsto_n e'}{first\ f \mapsto_n e'} \qquad \frac{f \mapsto_n f' \quad loopB\ i\ f' \mapsto e \quad e \mapsto_n e'}{loopB\ i\ f \mapsto_n e'}$$

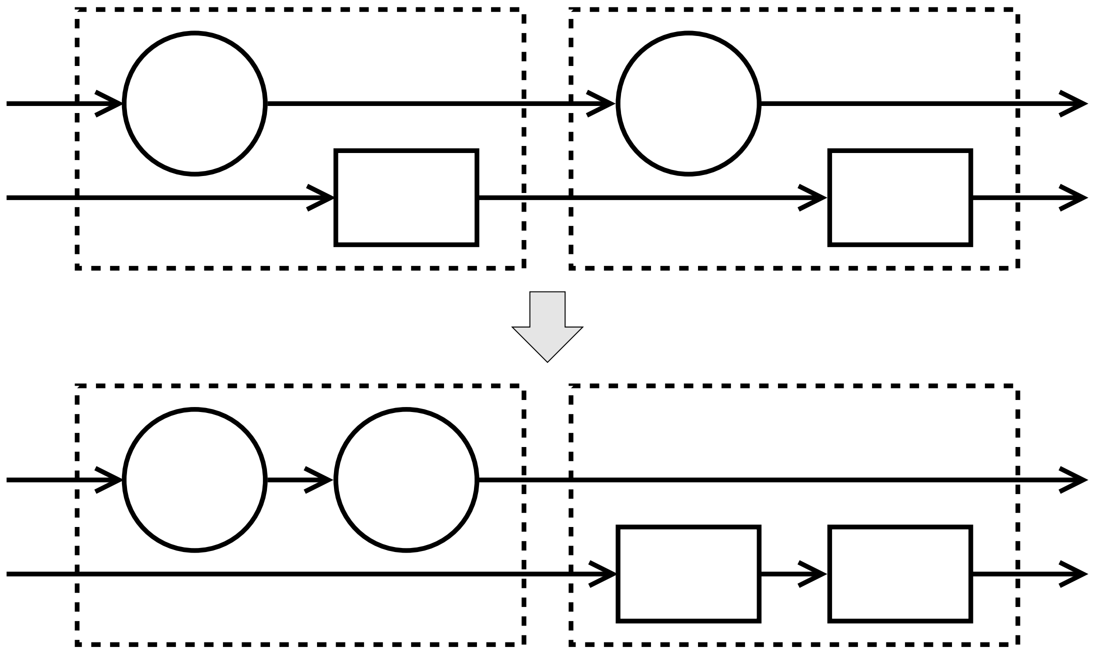
$$\frac{init\ i \mapsto e \quad e \mapsto_n e'}{init\ i \mapsto_n e'} \qquad \frac{loop\ f \mapsto e \quad e \mapsto_n e'}{loop\ f \mapsto_n e'}$$

- Always terminating.
- Preserving type and semantics due to arrow laws.
- Determinstic.

Normalization Explained

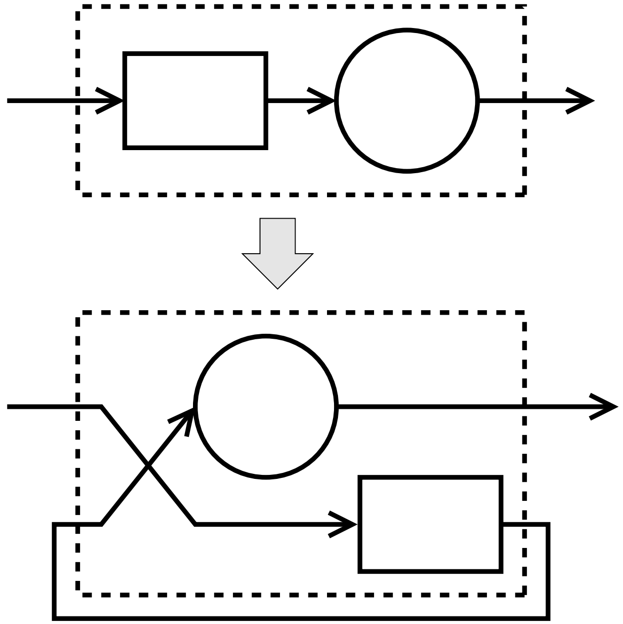
- Based on arrow laws, but directed.
- The two new laws, commutativity and product, are essential.
- Best illustrated by pictures...

Re-order Parallel pure and stateful arrows



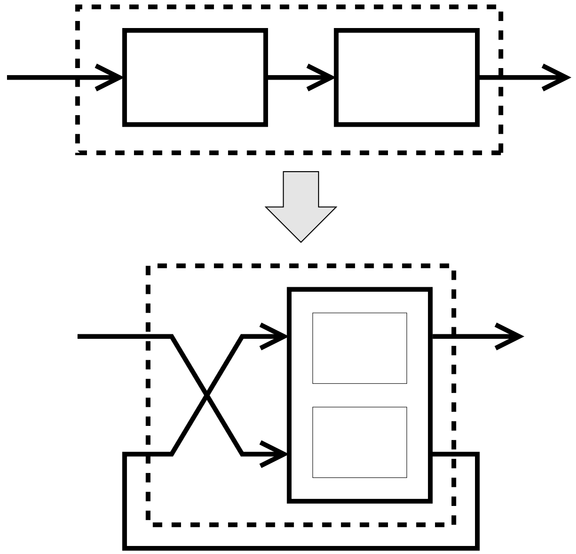
Related law: exchange (a special case for commutativity).

Re-order sequential pure and stateful arrows



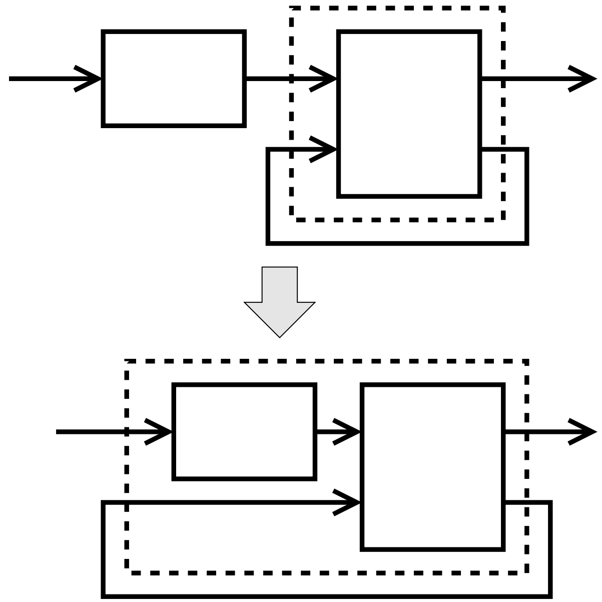
Related laws: tightening, sliding, and definition of second.

Change sequential to parallel



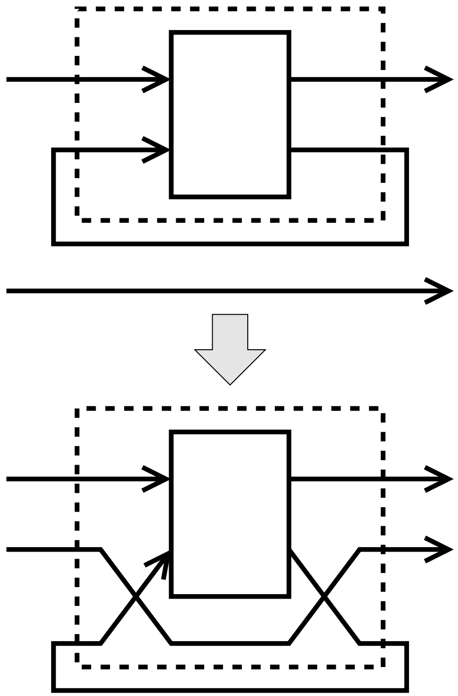
Related laws: product, tightening, sliding, and definition of second.

Move sequential into loop



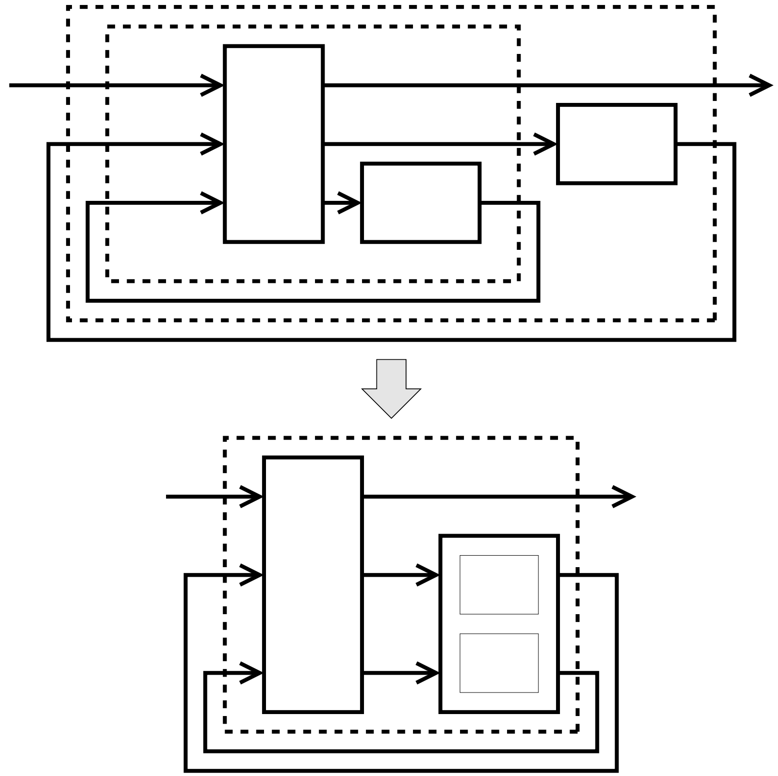
Related law: tightening.

Move parallel into loop



Related laws: superposing, and definition of second.

Fuse nested loops



Related laws: commutativity, product, tightening, and vanishing.

Conclusion

- CCA is a minimal language for FRP and dataflow languages.
- Arrow laws for CCA lead to the discovery of a normal form.
- CCNF is an effective optimization for CCA programs.

Conclusion

- CCA is a minimal language for FRP and dataflow languages.
- Arrow laws for CCA lead to the discovery of a normal form.
- CCNF is an effective optimization for CCA programs.

Abstraction, Absraction, and Abstraction!

Thank You!