Causal Commutative Arrows

Hai (Paul) Liu, Eric Cheng, and Paul Hudak

Computer Science Department
Yale University

The 14th ACM SIGPLAN ICFP 2009

Example

A mathematical definition of the exponential function:

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

FRP program using *arrow syntax* (Paterson, 2001):

$$
\begin{array}{l}
exp = \mathbf{proc}\ () \rightarrow \mathbf{do} \\
\quad \mathbf{rec}\ \mathbf{let}\ e = 1 + i \\
\qquad\quad i \leftarrow integral \prec e \\
\quad returnA \prec e
\end{array}
$$

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 small set of *wiring* combinators.
- Mathematical background in category theory.

What is Arrow

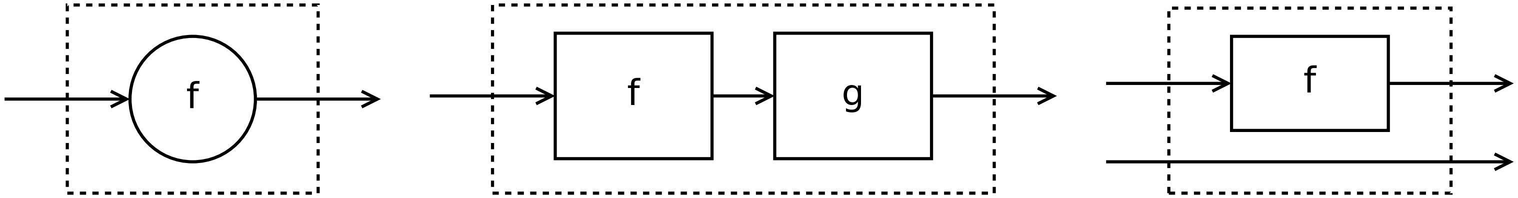
A generalization of monads. In Haskell:

$$
\begin{aligned}
&\textbf{class}\ Arrow\ a\ \textbf{where}\\
&\quad arr \quad :: (b \to c) \to a\ b\ c\\
&\quad (\ggg) :: a\ b\ c \to a\ c\ d \to a\ b\ d\\
&\quad first\ :: a\ b\ c \to a\ (b, d)\ (c, d)
\end{aligned}
$$

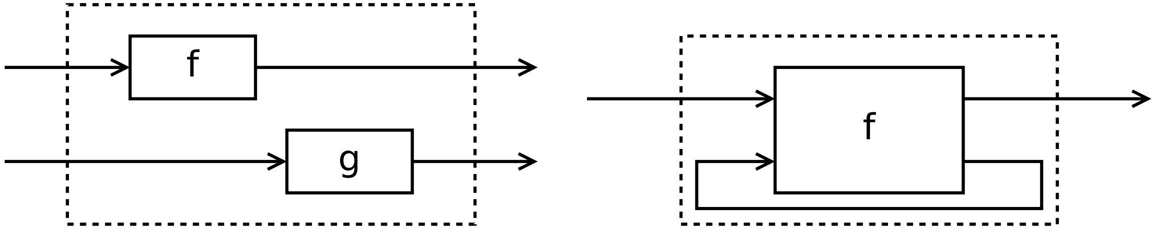
Support both sequential and parallel composition:

$$
\begin{aligned}
&second \quad :: (Arrow\ a) \Rightarrow a\ b\ c \to a\ (d, b)\ (d, c)\\
&second\ f = arr\ swap \ggg first\ f \ggg arr\ swap\\
&\quad \textbf{where}\ swap\ (a, b) = (b, a)\\
&(\star\star\star) \quad :: (Arrow\ a) \Rightarrow a\ b\ c \to a\ b'\ c' \to a\ (b, b')\ (c, c')\\
&f \star\star\star g \quad = first\ f \ggg second\ g
\end{aligned}
$$

Picturing an Arrow



(a) $arr\ f$ (b) $f \ggg g$ (c) $first\ f$



(d) $f \star\!\star\!\star\ g$ (e) $loop\ f$

To model recursion, Paterson (2001) introduces $ArrowLoop$:

$$\textbf{class}\ Arrow\ a \Rightarrow ArrowLoop\ a\ \textbf{where}$$
$$\quad loop :: a\ (b, d)\ (c, d) \rightarrow a\ b\ c$$

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 described by arrow laws.

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	$arr\ (g\ .\ f) = arr\ f \ggg arr\ g$
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	$first\ (first\ f) \ggg arr\ assoc = arr\ assoc \ggg first\ f$
	$\textbf{where}\ assoc\ ((a, b), c) = (a, (b, c))$

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	$loop\ (f \ggg arr\ (id * k)) = loop\ (arr\ (id \times k) \ggg f)$
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	$loop\ (arr\ f) = arr\ (trace\ f)$

$$\textbf{where}\ trace\ f\ b = \textbf{let}\ (c, d) = f\ (b, d)\ \textbf{in}\ c$$

Question

What makes a good abstraction for FRP?

Question

What makes a good abstraction for FRP?

Arrows?

Question

What makes a good abstraction for FRP?

Arrows? *Too general. They don't describe causality.*

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

Question

What makes a good abstraction for FRP?

Arrows? *Too general. They don't describe causality.*

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

Can we refine the arrow abstraction to capture causality?

Causal Commutative Arrows

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

$$\textbf{class } ArrowLoop\ a \Rightarrow ArrowInit\ a \textbf{ where}$$
$$\quad init :: b \rightarrow a\ b\ b$$

Causal Commutative Arrows

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

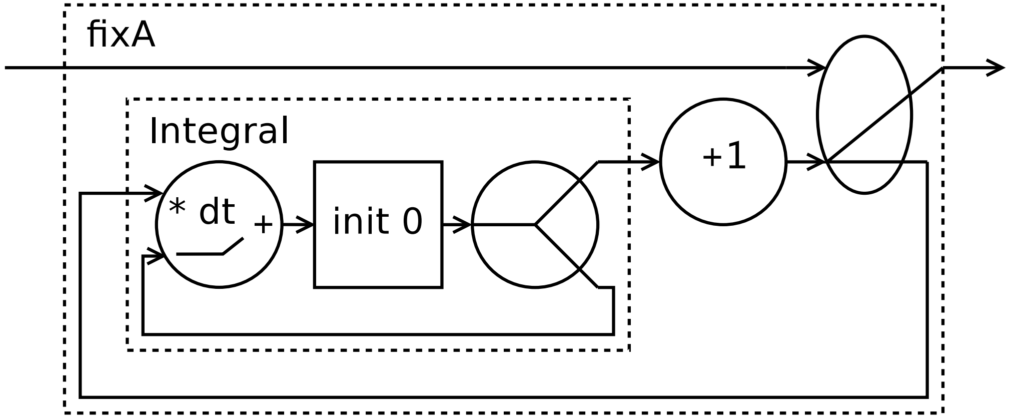
$$\textbf{class}\ ArrowLoop\ a \Rightarrow ArrowInit\ a\ \textbf{where}$$
$$init :: b \rightarrow a\ b\ b$$

and two additional laws:

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

and still remain *abstract*!

Exponential Example, Revisited



$$exp = \mathit{fixA}\ (integral \ggg arr\ (+1))$$

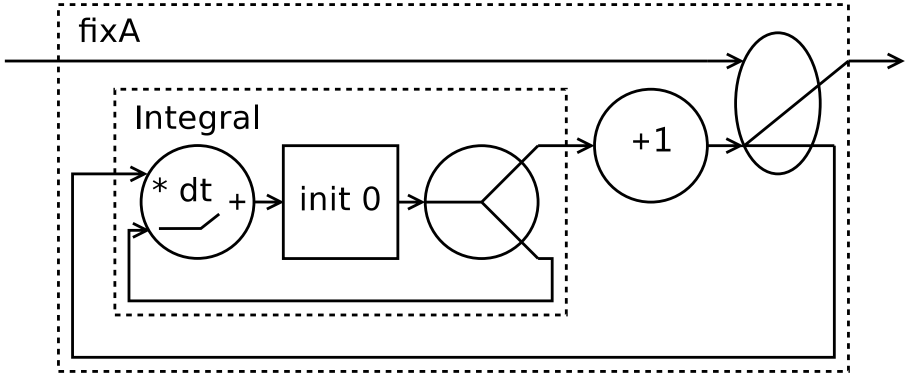
$$\mathit{fixA} :: ArrowLoop\ a \Rightarrow a\ b\ b \rightarrow a\ ()\ b$$

$$\mathit{fixA}\ f = loop\ (second\ f \ggg arr\ (\lambda((), y) \rightarrow (y, y)))$$

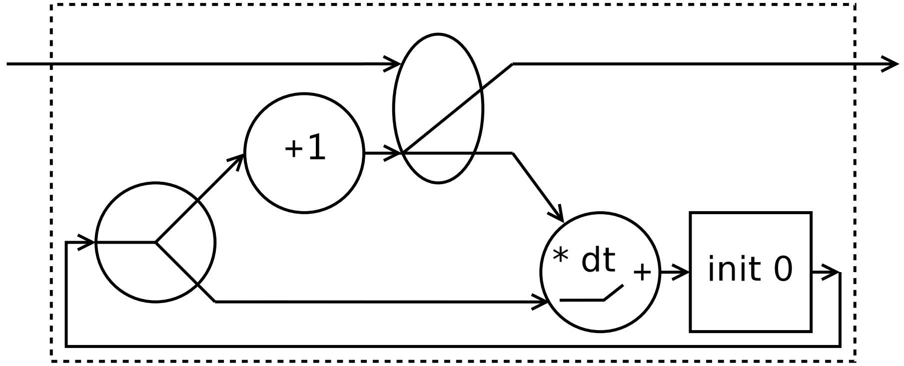
$$integral :: ArrowInit\ a \Rightarrow a\ Double\ Double$$

$$integral = loop\ (arr\ (\lambda(v, i) \rightarrow i + dt * v) \ggg init\ 0 \ggg arr\ (\lambda i \rightarrow (i, i)))$$

Exponential Example, Normalized

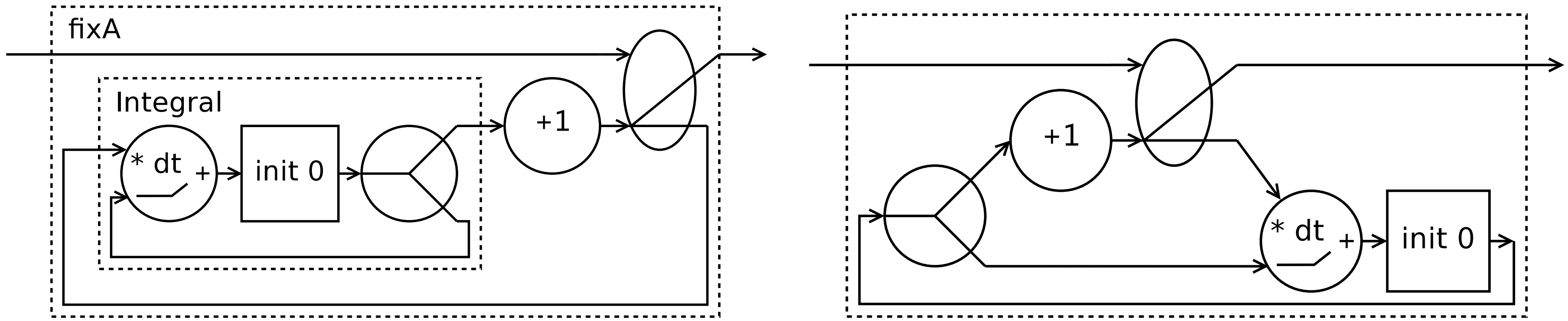


(f) Original



(g) Normalized

Exponential Example, Normalized



(f) Original (g) Normalized

Causal Commutative Normal Form (CCNF):

- A single loop containing one pure arrow and one initial state.
- Translation only based on abstract laws without committing to any particular implementation.

Benchmarks (Speed Ratio, Greater is Better)

Name (LOC)	1. GHC	2. arrowp	3. CCNF
exp (4)	1.0	2.4	13.9
sine (6)	1.0	2.66	12.0
oscSine (4)	1.0	1.75	4.1
50's sci-fi (5)	1.0	1.28	10.2
robotSim (8)	1.0	1.48	8.9

- Same arrow source programs written in arrow syntax.
- Same arrow implementation in Haskell.
- Only difference is syntactic:
  1. Translated to combinators by GHC's built-in arrow compiler.
  2. Translated to combinators by Paterson's arrowp preprocessor.
  3. Normalized combinator program through CCA.

CCA, a Domain Specific Language

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

Variables	V	$::=$	$x \mid y \mid z \mid \dots$
Primitive Types	t	$::=$	$1 \mid Int \mid Bool \mid \dots$
Types	α, β, θ	$::=$	$t \mid \alpha \times \beta \mid \alpha \to \beta \mid \alpha \rightsquigarrow \beta$
Expressions	E	$::=$	$V \mid (E_1, E_2) \mid \mathit{fst}\ E \mid \mathit{snd}\ E \mid$
			$\lambda x : \alpha.E \mid E_1\ E_2 \mid () \mid \dots$
Environment	Γ	$::=$	$x_1 : \alpha_1, \dots, x_n : \alpha_n$

CCA Types

$$(\text{UNIT}) \quad \Gamma \vdash () : 1 \qquad (\text{VAR}) \; \frac{(x : \alpha) \in \Gamma}{\Gamma \vdash x : \alpha}$$

$$(\text{ABS}) \; \frac{\Gamma, x : \alpha \vdash E : \beta}{\Gamma \vdash \lambda x : \alpha . E : \alpha \longrightarrow \beta} \qquad (\text{APP}) \; \frac{\begin{array}{c}\Gamma \vdash E_1 : \alpha \longrightarrow \beta \\ \Gamma \vdash E_2 : \alpha\end{array}}{\Gamma \vdash E_1 \; E_2 : \beta}$$

$$(\text{PAIR}) \; \frac{\begin{array}{c}\Gamma \vdash E_1 : \alpha \\ \Gamma \vdash E_2 : \beta\end{array}}{\Gamma \vdash (E_1, E_2) : \alpha \times \beta} \qquad (\text{FST}) \; \frac{\Gamma \vdash E : \alpha \times \beta}{\Gamma \vdash \mathit{fst} \; E : \alpha} \qquad (\text{SND}) \; \frac{\Gamma \vdash E : \alpha \times \beta}{\Gamma \vdash \mathit{snd} \; E : \beta}$$

CCA Constants

$$
\begin{aligned}
arr_{\alpha,\beta} &: (\alpha \rightarrow \beta) \rightarrow (\alpha \rightsquigarrow \beta) \\
\ggg_{\alpha,\beta,\theta} &: (\alpha \rightsquigarrow \beta) \rightarrow (\beta \rightsquigarrow \theta) \rightarrow (\alpha \rightsquigarrow \theta) \\
first_{\alpha,\beta,\theta} &: (\alpha \rightsquigarrow \beta) \rightarrow (\alpha \times \theta \rightsquigarrow \beta \times \theta) \\
loop_{\alpha,\beta,\theta} &: (\alpha \times \theta \rightsquigarrow \beta \times \theta) \rightarrow (\alpha \rightsquigarrow \beta) \\
init_{\alpha} &: \alpha \rightarrow (\alpha \rightsquigarrow \alpha)
\end{aligned}
$$

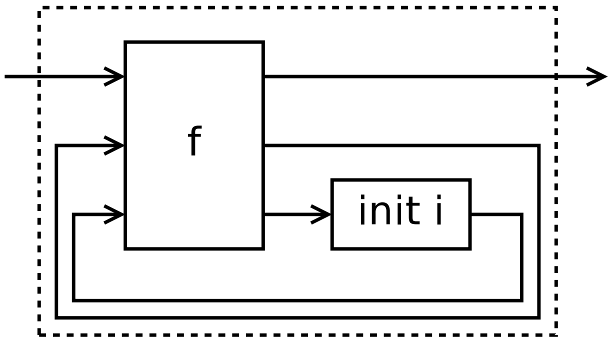
CCA Definitions

$$
\begin{array}{ll}
assoc & : (\alpha \times \beta) \times \theta \rightarrow \alpha \times (\beta \times \theta) \\
assoc & = \lambda z \, . \, (fst\ (fst\ z), (snd\ (fst\ z), snd\ z)) \\
assoc^{-1} & : \alpha \times (\beta \times \theta) \rightarrow (\alpha \times \beta) \times \theta \\
assoc^{-1} & = \lambda z \, . \, ((fst\ z, fst\ (snd\ z)), snd\ (snd\ z)) \\
juggle & : (\alpha \times \beta) \times \theta \rightarrow (\alpha \times \theta) \times \beta \\
juggle & = assoc^{-1} \, . \, (id \times swap) \, . \, assoc \\
transpose & : (\alpha \times \beta) \times (\theta \times \eta) \rightarrow (\alpha \times \theta) \times (\beta \times \eta) \\
transpose & = assoc \, . \, (juggle \times id) \, . \, assoc^{-1} \\
shuffle^{-1} & : \alpha \times ((\beta \times \delta) \times (\theta \times \eta)) \rightarrow (\alpha \times (\beta \times \theta)) \times (\delta \times \eta) \\
shuffle^{-1} & = assoc^{-1} \, . \, (id \times transpose) \\
shuffle' & : (\alpha \times (\beta \times \theta)) \times (\delta \times \eta) \rightarrow \alpha \times ((\beta \times \delta) \times (\theta \times \eta)) \\
shuffle' & = (id \times transpose) \, . \, assoc \\
id & : \alpha \rightarrow \alpha \\
id & = \lambda x \, . \, x \\
(\,.\,) & : (\beta \rightarrow \theta) \rightarrow (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \theta) \\
(\,.\,) & = \lambda f \, . \, \lambda g \, . \, \lambda x \, . \, f\ (g\ x) \\
(\times) & : (\alpha \rightarrow \beta) \rightarrow (\theta \rightarrow \gamma) \rightarrow (\alpha \times \theta \rightarrow \beta \times \gamma) \\
(\times) & : \lambda f \, . \, \lambda g \, . \, \lambda z \, . \, (f\ (fst\ z), g\ (snd\ z)) \\
dup & : \alpha \rightarrow \alpha \times \alpha \\
dup & = \lambda x \, . \, (x, x) \\
swap & : \alpha \times \beta \rightarrow \beta \times \alpha \\
swap & = \lambda z \, . \, (snd\ z, fst\ z) \\
second & : (\alpha \rightsquigarrow \beta) \rightarrow (\theta \times \alpha \rightsquigarrow \theta \times \beta) \\
second & = \lambda f \, . \, arr\ swap \ggg first\ f \ggg arr\ swap
\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 $[\![e]\!] = [\![e_{norm}]\!]$.

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



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 $\Downarrow$

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

$$(\text{SEQ}) \quad \frac{e_1 \Downarrow e_1' \quad e_2 \Downarrow e_2' \quad e_1' \ggg e_2' \mapsto e \quad e \Downarrow e'}{e_1 \ggg e_2 \Downarrow e'}$$

$$(\text{FIRST}) \quad \frac{f \Downarrow f' \quad \mathit{first}\ f' \mapsto e \quad e \Downarrow e'}{\mathit{first}\ f \Downarrow e'} \qquad (\text{INIT}) \quad \frac{\mathit{init}\ i \mapsto e \quad e \Downarrow e'}{\mathit{init}\ i \Downarrow e'}$$

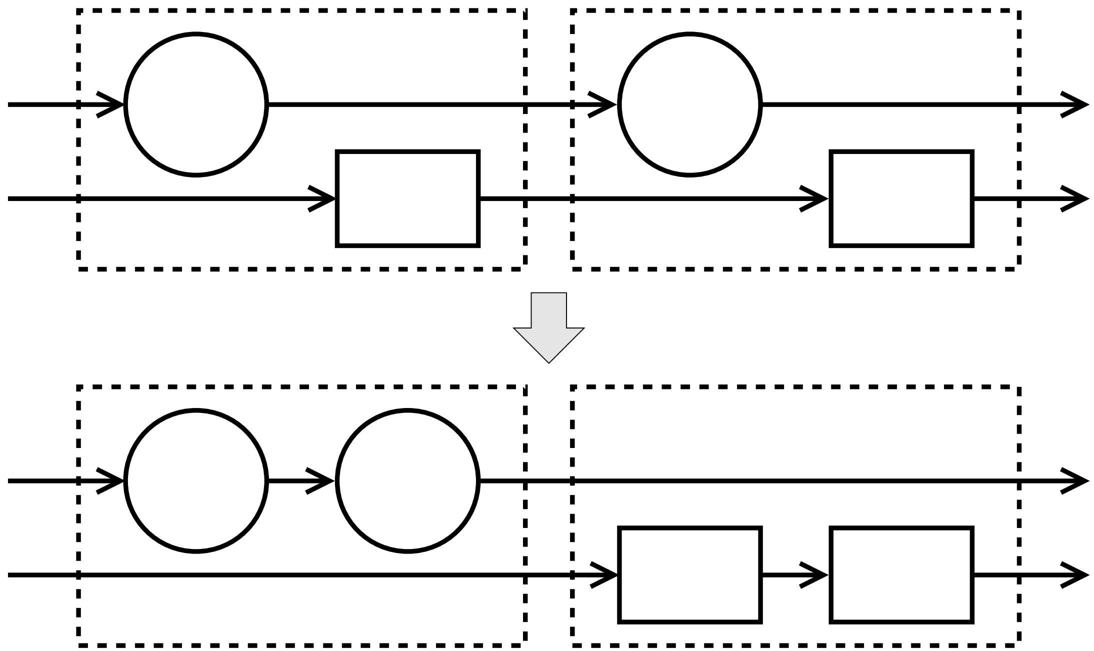
$$(\text{LOOP}) \quad \frac{\mathit{loop}\ f \mapsto e \quad e \Downarrow e'}{\mathit{loop}\ f \Downarrow e'} \qquad (\text{LOOPB}) \quad \frac{f \Downarrow f' \quad \mathit{loopB}\ i\ f' \mapsto e \quad e \Downarrow e'}{\mathit{loopB}\ i\ f \Downarrow e'}$$

- Big step reduction following an inner most strategy.
- Always terminating.

Normalization Explained

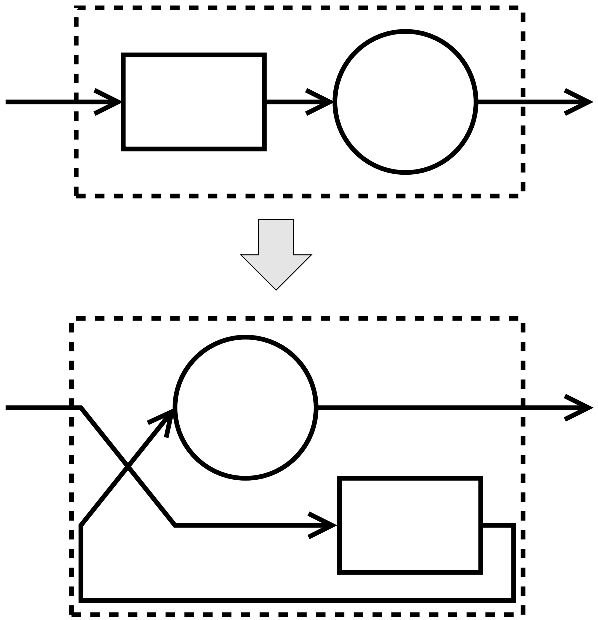
- Based on arrow laws, but directed.
- The two new laws, commutativity and product, are essential.
- Best illustrated by pictures...

Re-order Parallel pure and stateful arrows



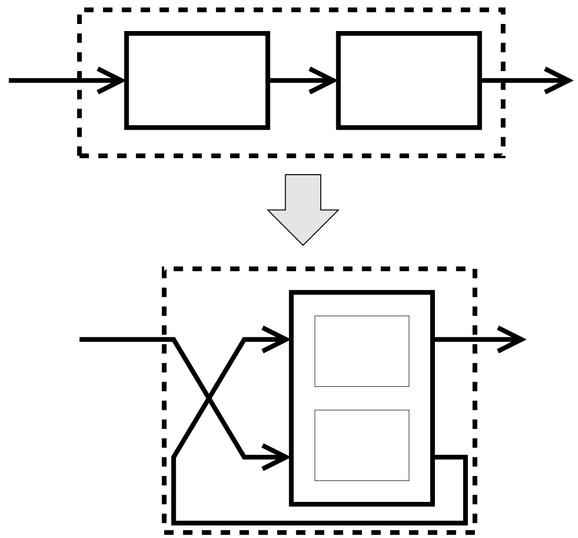
Related law: exchange (a special case of commutativity).

Re-order sequential pure and stateful arrows



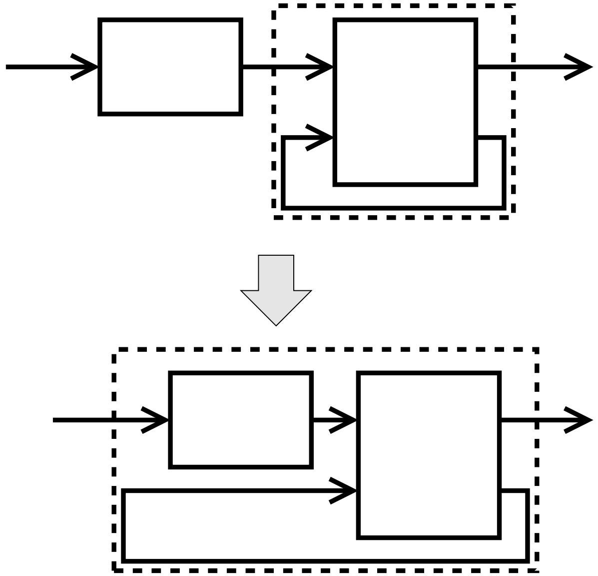
Related laws: tightening, sliding, and definition of second.

Change sequential to parallel



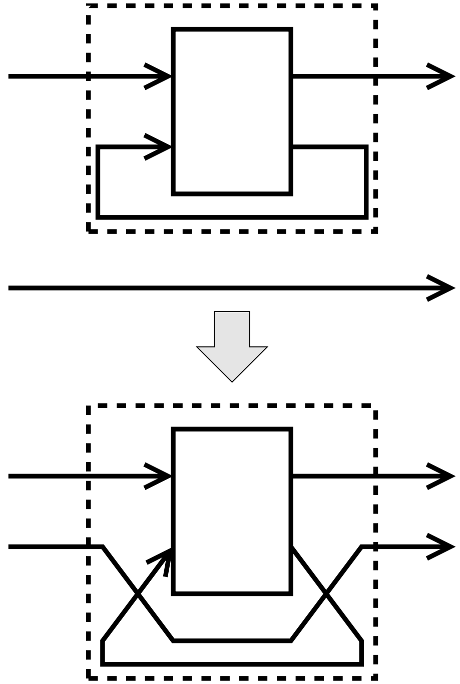
Related laws: product, tightening, sliding, and definition of second.

Move sequential into loop



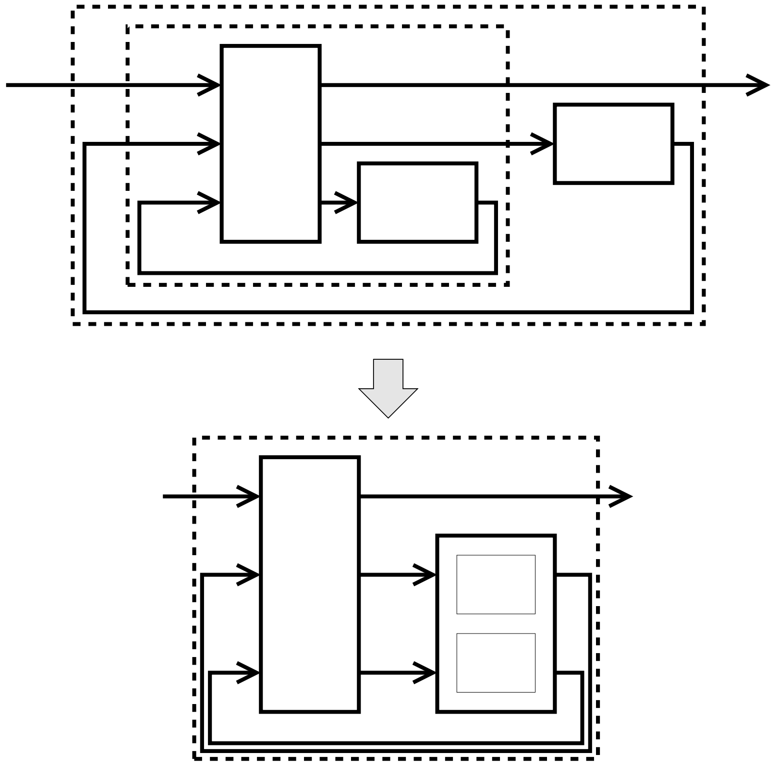
Related law: tightening.

Move parallel into loop



Related laws: superposing, and definition of second.

Fuse nested loops



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

Further Optimization

Optimized CCNF.

$$loopB :: \theta \rightarrow (\alpha \times (\gamma \times \theta) \rightsquigarrow \beta \times (\gamma \times \theta)) \rightarrow (\alpha \rightsquigarrow \beta)$$

$$loopD :: \theta \rightarrow (\alpha \times \theta \rightsquigarrow \beta \times \theta) \rightarrow (\alpha \rightsquigarrow \beta)$$

$$loopB\ i\ f === loopD\ i\ g$$

$$\quad \textbf{where}\ g\ (x, i) = \textbf{let}\ (y, (z, i')) = f\ (x, (z, i'))$$

$$\qquad \textbf{in}\ \ (y, i')$$

Further Optimization

Optimized CCNF.

$$loopB :: \theta \to (\alpha \times (\gamma \times \theta) \rightsquigarrow \beta \times (\gamma \times \theta)) \to (\alpha \rightsquigarrow \beta)$$

$$loopD :: \theta \to (\alpha \times \theta \rightsquigarrow \beta \times \theta) \to (\alpha \rightsquigarrow \beta)$$

$$loopB\ i\ f === loopD\ i\ g$$

$$\textbf{where}\ g\ (x, i) = \textbf{let}\ (y, (z, i')) = f\ (x, (z, i'))$$

$$\textbf{in}\ \ (y, i')$$

Inline the pair, no more arrows!

$$runCCNF \quad :: \theta \to (\alpha \times \theta \to \beta \times \theta) \to [\alpha] \to [\beta]$$

$$runCCNF\ i\ f = g\ i$$

$$\textbf{where}\ g\ i\ (x : xs) = \textbf{let}\ (y, i') = f\ (x, i)$$

$$\textbf{in}\ \ y : g\ i'\ xs$$

Combine with Stream Fusion

Stream Fusion (Coutts, et. al., 2007) gets rid of intermediate structure.

$$\mathbf{data}\ \mathit{Stream}\ a \quad = \forall\ s\ .\ \mathit{Stream}\ (s \rightarrow \mathit{Step}\ a\ s)\ s$$

$$\mathbf{data}\ \mathit{Step} \quad a\ s = \mathit{Yield}\ a\ s$$

$$\mathit{loopS} :: \theta \rightarrow (\alpha \times \theta \rightarrow \beta \times \theta) \rightarrow \mathit{Stream}\ \alpha \rightarrow \mathit{Stream}\ \beta$$

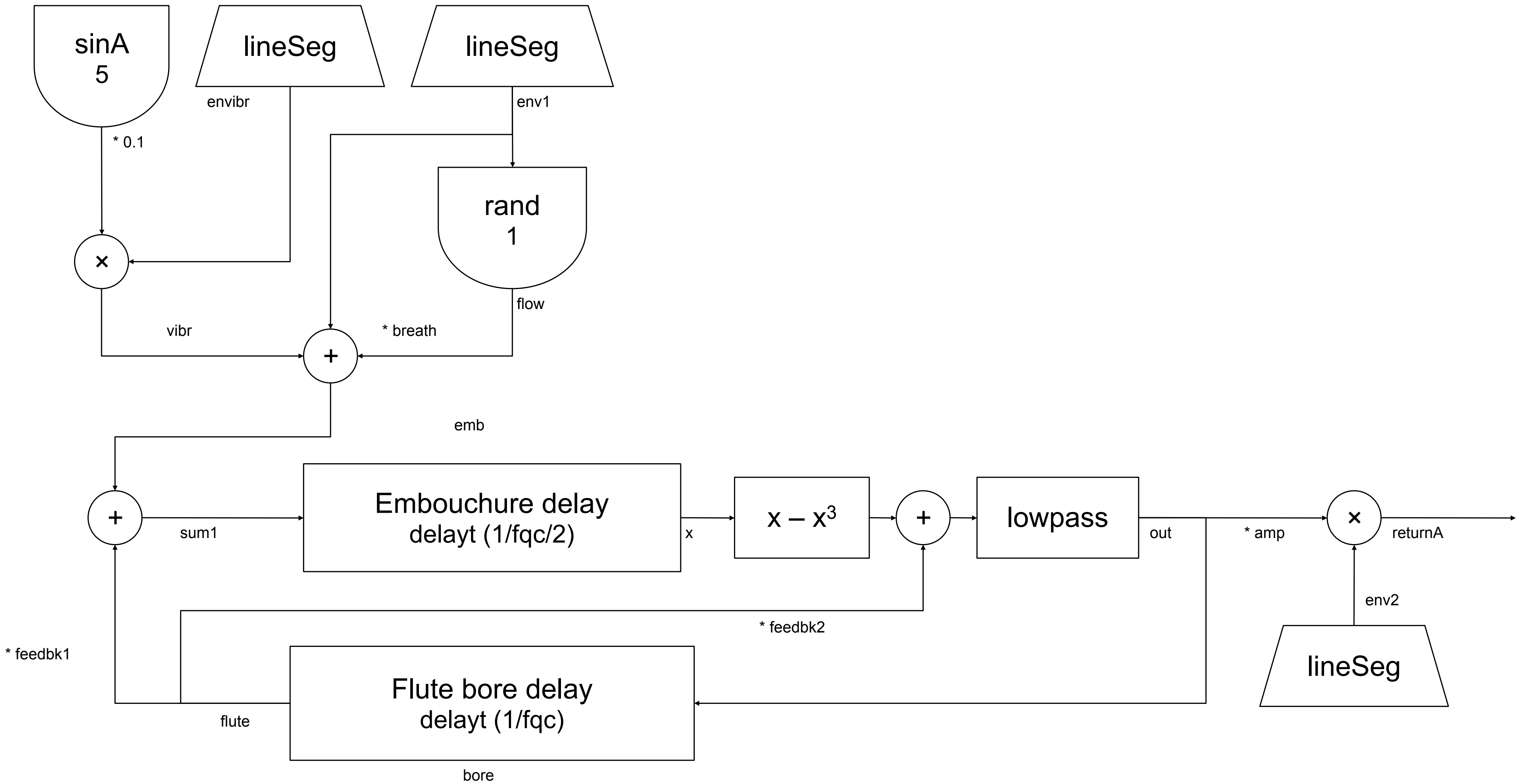
- Stream producers written in terms of non-recursive stepper functions.
- Compiler fuses all into a tail recursive loop, unboxing types if possible.
- CCA normalization helps translating recursion into stepper function!

Benchmarks (Speed Ratio, Greater is Better)

Name (LOC)	1. GHC	2. arrowp	3. CCNF	4. Fusion
exp (4)	1.0	2.4	13.9	190.9
sine (6)	1.0	2.66	12.0	284.0
oscSine (4)	1.0	1.75	4.1	13.0
50′s sci-fi (5)	1.0	1.28	10.2	19.2
robotSim (8)	1.0	1.48	8.9	36.8

- No more arrows. No more interpretation overhead.
- No intermediate structure. Tight loop. Unboxed type.

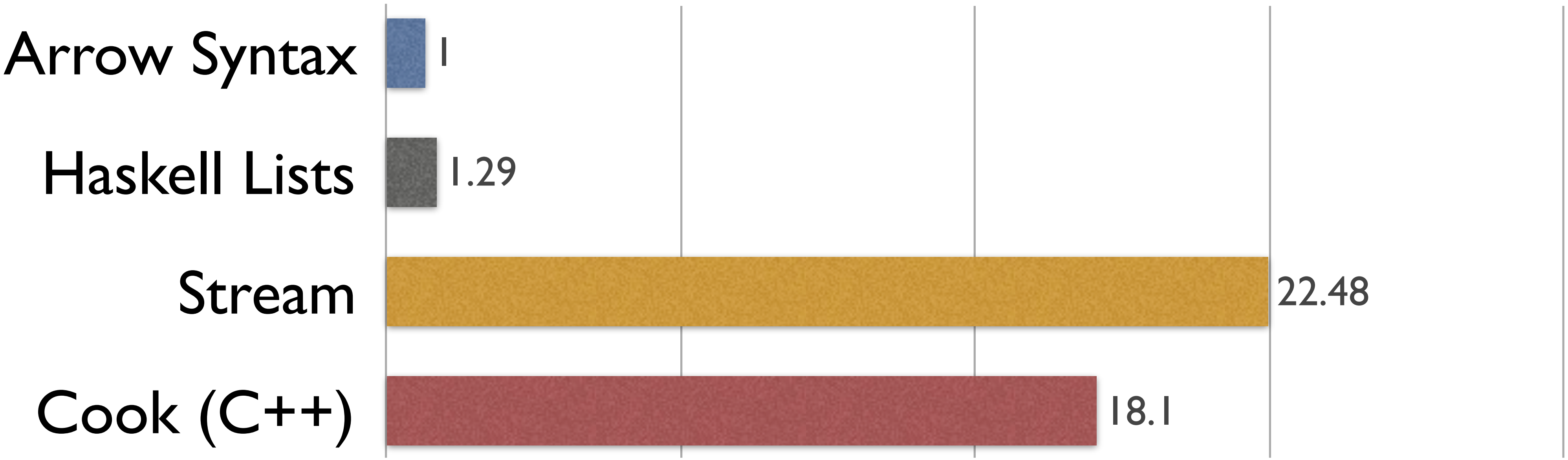
Demo: Real-time Sound Synthesis


sinA
5
lineSeg
lineSeg
envibr
env1
* 0.1
rand
1
flow
vibr
* breath
emb
Embouchure delay
delayt (1/fqc/2)
x
x – x3
lowpass
out
* amp
returnA
sum1
env2
* feedbk2
* feedbk1
lineSeg
Flute bore delay
delayt (1/fqc)
flute
bore

```
flute0 dur amp fqc press breath =
  let en1   = arr $ lineSeg [0, 1.1 ∗ press, press, press, 0] [0.06, 0.2, dur − 0.16, 0.02]
      en2   = arr $ lineSeg [0, 1, 1, 0] [0.01, dur − 0.02, 0.01]
      enibr = arr $ lineSeg [0, 0, 1, 1] [0.5, 0.5, dur − 1]
      emb   = delayt (mkBuf 2 n) n
      bore  = delayt (mkBuf 1 (n ∗ 2)) (n ∗ 2)
      n     = truncate (1 / fqc / 2 ∗ fromIntegral sr)
  in proc _ → do
    rec tm     ← timeA          ≺ ()
        env1   ← en1            ≺ tm
        env2   ← en2            ≺ tm
        envibr ← enibr          ≺ tm
        sin5   ← sineA 5        ≺ ()
        rand   ← arr rand_f     ≺ ()
        let vibr = sin5 ∗ envibr ∗ 0.1
            flow = rand ∗ env1
            sum1 = breath ∗ flow + env1 + vibr
        flute ← bore            ≺ out
        x     ← emb             ≺ sum1           + flute ∗ 0.4
        out   ← lowpassA 0.27   ≺ x − x ∗ x ∗ x + flute ∗ 0.4
    returnA ≺ out ∗ amp ∗ env2
```

```
loop (arr (λ(_, out) → ((), out)) ⋙
  (first timeA ⋙ arr (λ(tm, out) → (tm, (out, tm)))) ⋙
    (first en1 ⋙ arr (λ(env1, (out, tm)) → (tm, (env1, out, tm)))) ⋙
      (first en2 ⋙
        arr (λ(env2, (env1, out, tm)) → (tm, (env1, env2, out)))) ⋙
        (first enibr ⋙
          arr (λ(envibr, (env1, env2, out)) → ((), (env1, env2, envibr, out)))) ⋙
          (first (sineA 5) ⋙
            arr (λ(sin5, (env1, env2, envibr, out)) →
                  ((), (env1, env2, envibr, out, sin5)))) ⋙
            (first (arr rand_f) ⋙
              arr (λ(rand, (env1, env2, envibr, out, sin5)) →
                let vibr = sin5 ∗ envibr ∗ 0.1
                    flow = rand ∗ env1
                    sum1 = breath ∗ flow + env1 + vibr
                in (out, (env2, sum1)))) ⋙
              (first bore ⋙
                arr (λ(flute, (env2, sum1)) → ((flute, sum1), (env2, flute)))) ⋙
                (first (arr (λ(flute, sum1) → sum1 + flute ∗ 0.4) ⋙ emb) ⋙
                  arr (λ(x, (env2, flute)) → ((flute, x), env2))) ⋙
                  (first (arr (λ(flute, x) → x − x ∗ x ∗ x + flute ∗ 0.4) ⋙
                    lowpassA 0.27)
                     ⋙ arr (λ(out, env2) → ((env2, out), out))))
   ⋙ arr (λ(env2, out) → out ∗ amp ∗ env2)
```

```
fluteOpt dur amp fqc press breath =
  let env1 = upSample_f (lineSeg am1 du1) 20
      env2    = upSample_f (lineSeg am2 du2) 20
      env3    = upSample_f (lineSeg am3 du3) 20
      omh     = 2 ∗ pi / (fromIntegral sr) ∗ 5
      c       = 2 ∗ cos omh
      i       = sin omh
      dt      = 1 / fromIntegral sr
      sr      = 44100
      buf100  = mkArr 100
      buf50   = mkArr 50
      am1     = [0, 1.1 ∗ press, press, press, 0]
      du1     = [0.06, 0.2, dur − 0.16, 0.02]
      am2     = [0, 1, 1, 0]
      du2     = [0.01, dur − 0.02, 0.01]
      am3     = [0, 0, 1, 1]
      du3     = [0.5, 0.5, dur − 1]
  in loopS ((0, ((0, 0), 0)), (((((buf100), 0), 0), ((0), (((buf50), 0), 0))), (((0, i), (0, ((0, 0), 0))), ((0, ((0, 0), 0)), (0, ((0, 0), 0))))))
    ((λ(((((_a, _f), _e), _d), _c), ((_b, (_h, _i)), (((_g, _l), (_k, (_m, _n))), (((_j, _q), (_p, (_r, _s))), ((_o, (_u, _v)), (_t, (_w, _x))))))) →
      let randf                    = rand_f _f
          (env1vu1, env1vu2)       = env1 (_v, _u)
          (env1xw1, env1xw2)       = env1 (_x, _w)
          (env3sr1, env3sr2)       = env3 (_s, _r)
          (env2ih1, env2ih2)       = env2 (_i, _h)
          d50nm                    = ((delay_f 50) (_n, _m))
          d100lg                   = ((delay_f 100) (_l, _g))
          foo                      = _k + 0.27 ∗ (((−) ((+((polyx) (fstU d50nm))) baz)) _k)
          bar                      = (((+) (negate _j)) ((c∗) _q))
          baz                      = (((+((+((∗breath) ((∗env1xw1) randf))) env1vu1)) ((∗((∗0.1) env3sr1)) bar))) + (fstU d100lg ∗ 0.4)
      in (((∗((∗amp) foo)) env2ih1), (((_b + dt), (env2ih2, _b)), ((((sndU d100lg), foo), (foo, ((sndU d50nm), baz))),
          (((_q, bar), ((_p + dt), (env3sr2, _p))), (((_o + dt), (env1vu2, _o)), ((_t + dt), (env1xw2, _t))))))))
```



Flute Performance Comparison

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.

Thank You!