The Theory and Practice of Causal Commutative Arrows

Hai (Paul) Liu

Advisor: Paul Hudak

Computer Science Department
Yale University

October 2010

Contributions

1. Formalization of Causal Commutative Arrows (CCA):
   - Definition of CCA and its laws.
   - Definition of a CCA language that is strongly normalizing.
   - Proof of the soundness and termination of CCA normalization.
2. Implementation of CCA normalization/optimization:
   - Compile-time normalization through meta-programming.
   - Run-time performance improvement by orders of magnitude.
3. Applications of CCA:
   - Synchronous Dataflow
     - relating CCA normal form to an operational semantics.
   - Ordinary Differential Equations (ODE)
     - designing embedded DSLs, solving space leaks.
   - Functional Reactive Programming (FRP)
     - solving space leaks, extending CCA for hybrid modeling.

Contributions

1. Formalization of Causal Commutative Arrows (CCA):
   - Definition of CCA and its laws.
   - Definition of a CCA language that is strongly normalizing.
   - *Proof of the soundness and termination of CCA normalization.*
2. Implementation of CCA normalization/optimization:
   - *Compile-time normalization through meta-programming.*
   - Run-time performance improvement by orders of magnitude.
3. Applications of CCA:
   - Synchronous Dataflow
     - relating CCA normal form to an operational semantics.
   - Ordinary Differential Equations (ODE)
     - designing embedded DSLs, *solving space leaks*.
   - Functional Reactive Programming (FRP)
     - *solving space leaks,* extending CCA for hybrid modeling.

Motivation

What is a good abstraction for *Functional Reactive Programming (FRP)*?

Motivation

What is a good abstraction for *Functional Reactive Programming (FRP)*?

What is a good abstraction?

Motivation

What is a good abstraction for *Functional Reactive Programming (FRP)*?

What is a good abstraction?

- Abstract, high-level, more focus, less detail.
- General enough to express interesting programs.
- Specific enough to make use of domain knowledge.

Motivation

What is a good abstraction for *Functional Reactive Programming (FRP)*?

What is a good abstraction?

- Abstract, high-level, more focus, less detail.
- General enough to express interesting programs.
- Specific enough to make use of domain knowledge.

What is FRP?

Part I: FRP

Functional Reactive Programming

FRP is a paradigm for programming time based *hybrid systems*, with applications in graphics, animation, robotics, GUI, vision, etc.

FRP belongs to a larger family of *synchronous dataflow* languages.

Functional Reactive Programming

FRP is a paradigm for programming time based *hybrid systems*, with applications in graphics, animation, robotics, GUI, vision, etc.

FRP belongs to a larger family of *synchronous dataflow* languages.

- Dataflow: data flow (along edges) between instructions (nodes).
- Synchronous: computation in each cycle is instantaneous.
- Hybrid: FRP models both continuous and discrete components.

Functional Reactive Programming

FRP is a paradigm for programming time based *hybrid systems*, with applications in graphics, animation, robotics, GUI, vision, etc.

FRP belongs to a larger family of *synchronous dataflow* languages.

- Dataflow: data flow (along edges) between instructions (nodes).
- Synchronous: computation in each cycle is instantaneous.
- Hybrid: FRP models both continuous and discrete components.

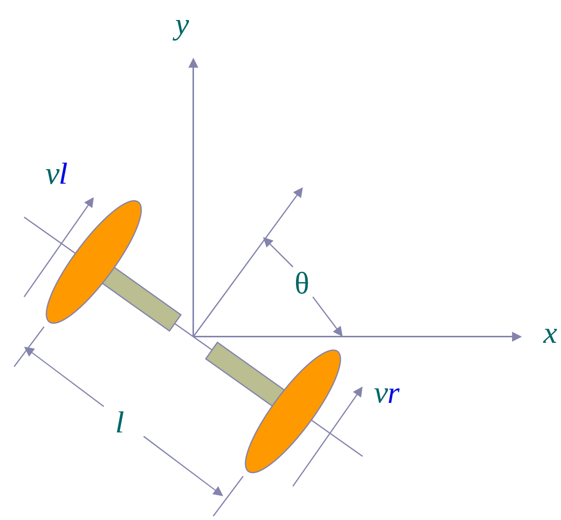
How do we program such systems?

First-class Signals

Represent time *changing* quantities as an abstract data type:

$$Signal\ a \ \approx\ Time \rightarrow a$$

Example: a robot simulator. Its robots have a differential drive.



Example: Robot Simulator

The equations governing the x position of a differential drive robot:

$$
\begin{aligned}
x(t) &= \frac{1}{2}\int_0^t (v_r(t) + v_l(t))\cos(\theta(t))dt \\
\theta(t) &= \frac{1}{l}\int_0^t (v_r(t) - v_l(t))dt
\end{aligned}
$$

The corresponding FRP program: (Note the lack of explicit time)

$$
\begin{aligned}
&x = (1 \,/\, 2) * integral\ ((vr + vl) * cos\ \theta) \\
&\theta = (1 \,/\, l) * integral\ (vr - vl)
\end{aligned}
$$

Domain specific operators:

$$
\begin{aligned}
&(+) &&:: Signal\ a \rightarrow Signal\ a \rightarrow Signal\ a \\
&(*) &&:: Signal\ a \rightarrow Signal\ a \rightarrow Signal\ a \\
&integral &&:: Signal\ a \rightarrow Signal\ a \\
&\ldots
\end{aligned}
$$

First-class Signals: Good or Bad?

Good:

- Conceptually simple and concise.
- Easy to program with, no clutter.
- The basis for a large number of FRP implementations.

First-class Signals: Good or Bad?

Good:

- Conceptually simple and concise.
- Easy to program with, no clutter.
- The basis for a large number of FRP implementations.

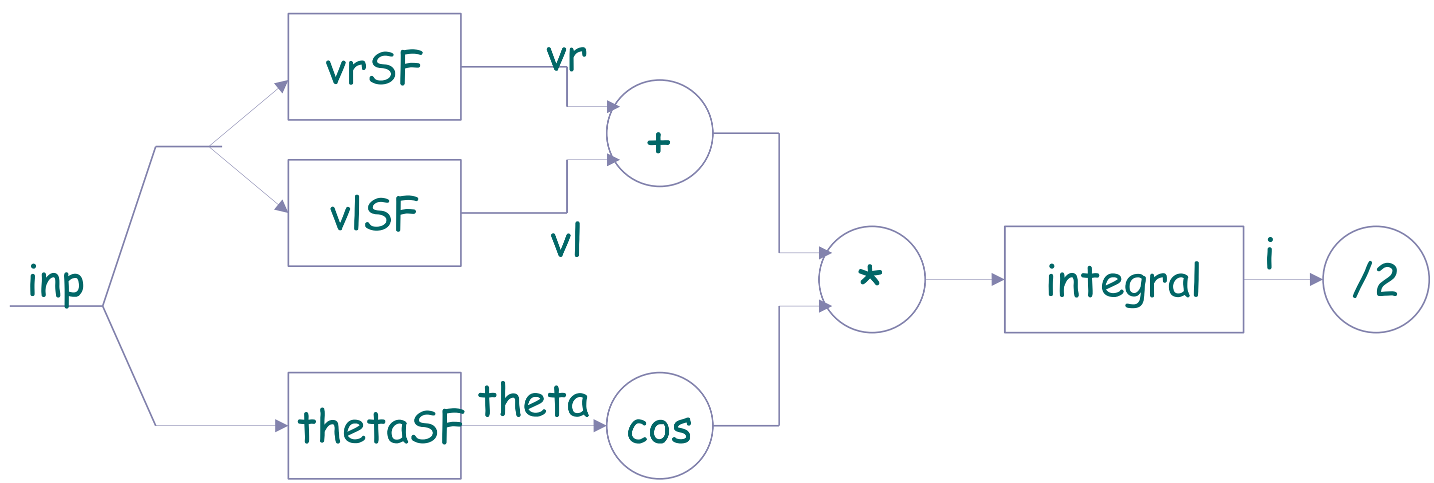
Bad:

- Higher-order signals $Signal\ (Event\ (Signal\ a))$ are *ambiguous*.
- *Time and space leak*: program slows down and consumes memory at an unexpected rate.

Improving the Abstraction with Signal Functions

Instead of first-class signals, use first-class *signal functions*:

$$SF\ a\ b \approx Signal\ a \rightarrow Signal\ b$$



Yampa is a FRP language that models signal functions using arrows.

Signal Functions are Arrows

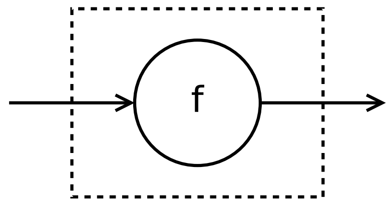
Arrows (Hughes 2000) are a generalization of monads. In Haskell:

$$
\begin{array}{l}
\mathbf{class}\ \mathit{Arrow}\ a\ \mathbf{where} \\
\quad \mathit{arr} \quad :: (b \to c) \to a\ b\ c \\
\quad (\ggg) :: a\ b\ c \to a\ c\ d \to a\ b\ d \\
\quad \mathit{first} \ :: a\ b\ c \to a\ (b, d)\ (c, d)
\end{array}
$$

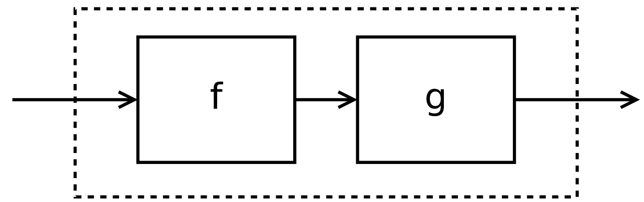
Support both sequential and parallel composition.

$$
\begin{array}{ll}
\mathit{second} & :: (\mathit{Arrow}\ a) \Rightarrow a\ b\ c \to a\ (d, b)\ (d, c) \\
\multicolumn{2}{l}{\mathit{second}\ f = \mathit{arr}\ \mathit{swap} \ggg \mathit{first}\ f \ggg \mathit{arr}\ \mathit{swap}} \\
\multicolumn{2}{l}{\quad \mathbf{where}\ \mathit{swap}\ (a, b) = (b, a)} \\
(\star\!\star\!\star) & :: (\mathit{Arrow}\ a) \Rightarrow a\ b\ c \to a\ b'\ c' \to a\ (b, b')\ (c, c') \\
f \star\!\star\!\star g & = \mathit{first}\ f \ggg \mathit{second}\ g \\
(\&\&\&) & :: (\mathit{Arrow}\ a) \Rightarrow a\ b\ c \to a\ b\ c' \to a\ b\ (c, c') \\
f \,\&\&\&\, g & = \mathit{arr}\ (\lambda x \to (x, x)) \ggg (f \star\!\star\!\star g)
\end{array}
$$

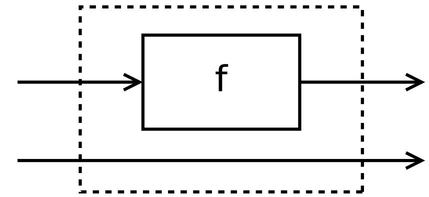
Picturing an Arrow



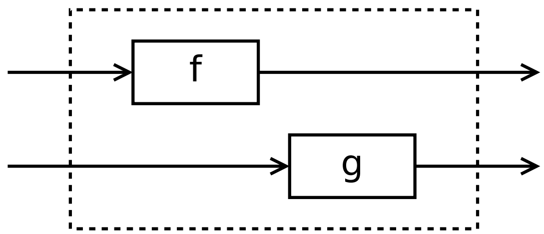
(a) $arr\ f$



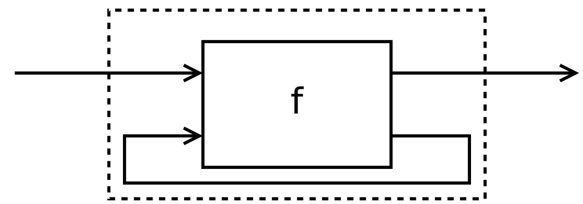
(b) $f \ggg g$



(c) $first\ f$



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

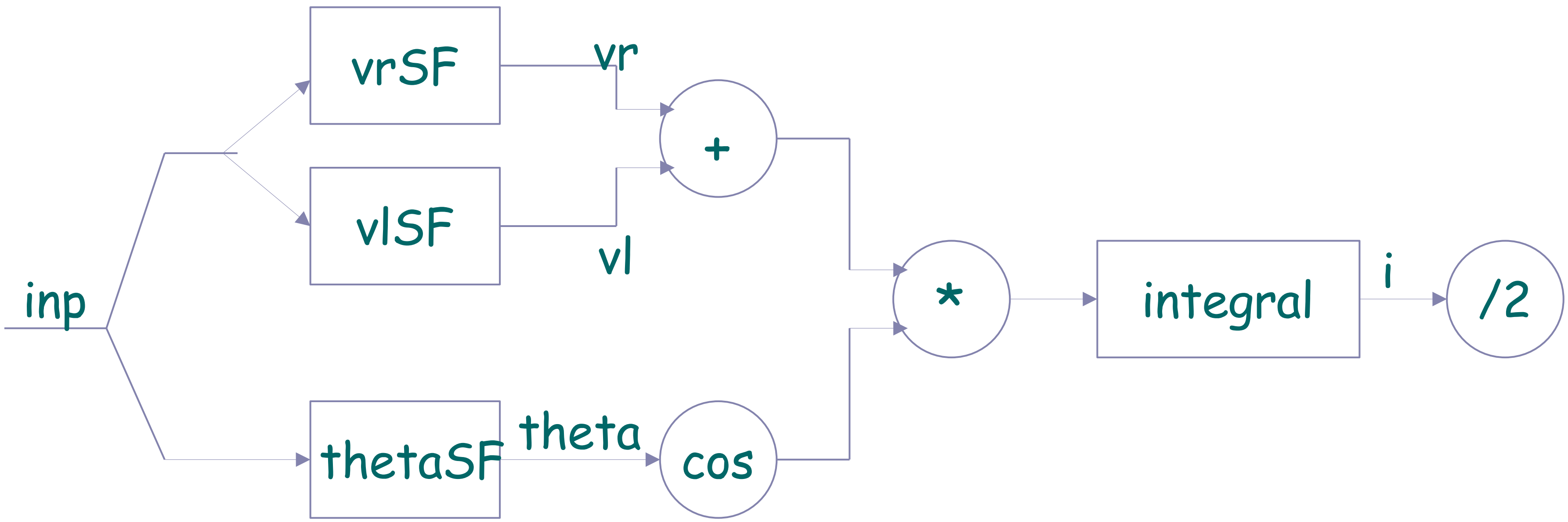


(e) $loop\ f$

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

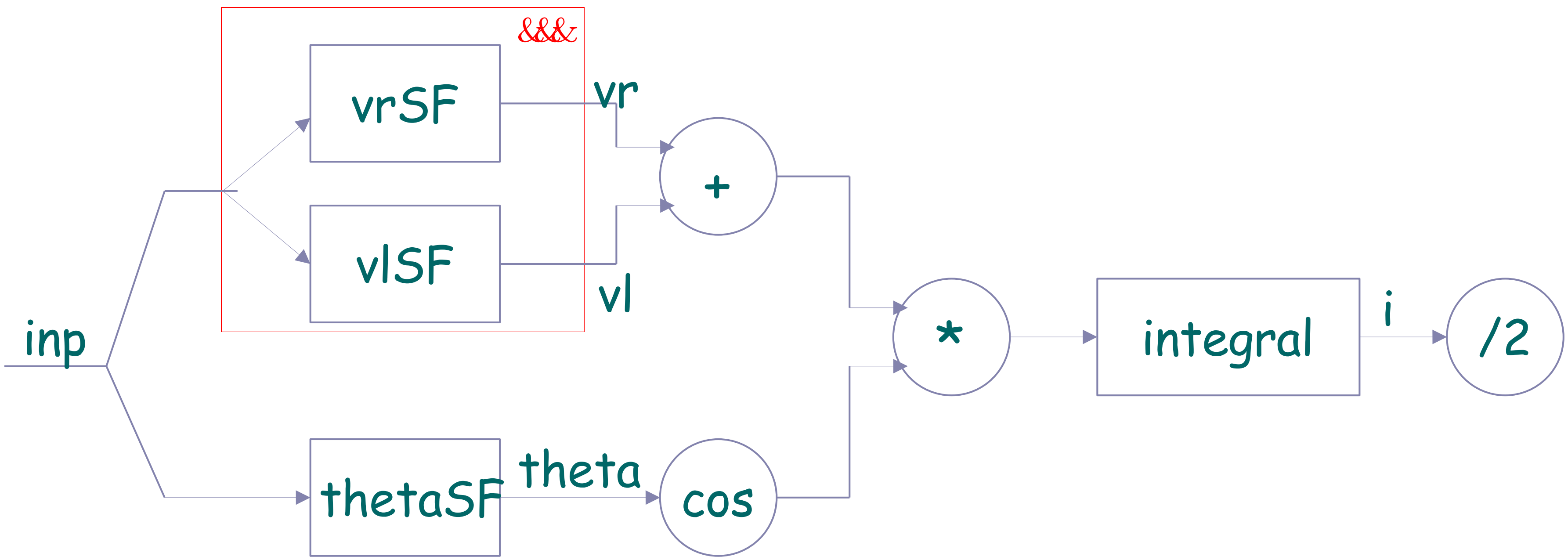
$$\textbf{class}\ Arrow\ a \Rightarrow ArrowLoop\ a\ \textbf{where}$$
$$\quad loop :: a\ (b, d)\ (c, d) \rightarrow a\ b\ c$$

Robot Simulator Revisit



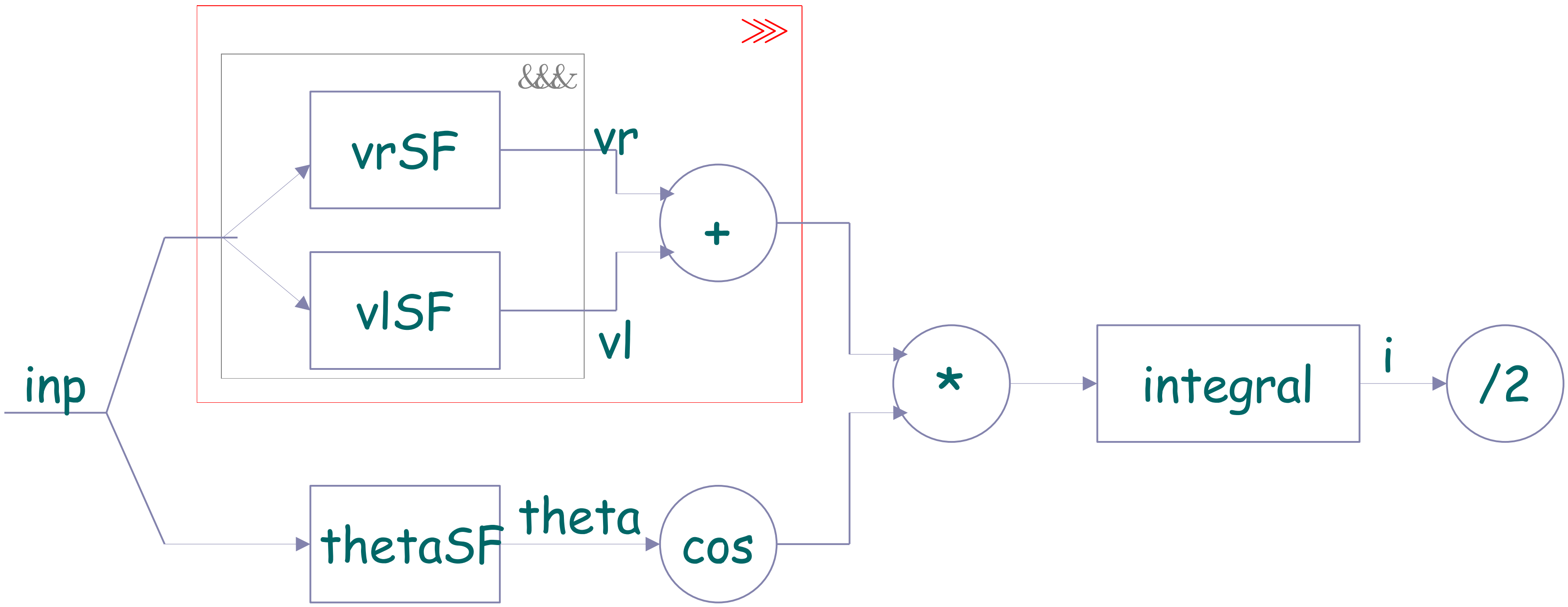
$$xSF = (((vrSF \&\&\& vlSF) \ggg arr\ (uncurry\ (+))) \&\&\& (thetaSF \ggg arr\ cos))$$
$$\ggg arr\ (uncurry\ (*)) \ggg integral \ggg arr\ (/2)$$

Robot Simulator Revisit



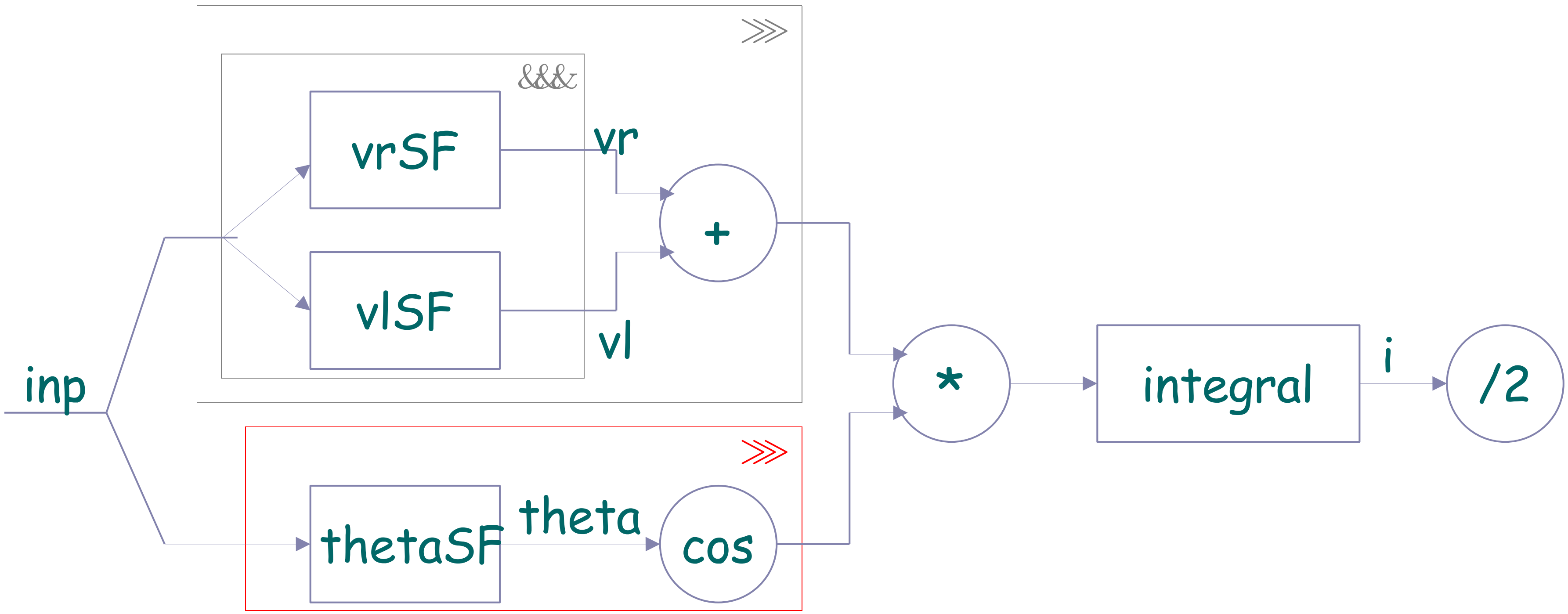
$$xSF = ((\boxed{(vrSF \,\&\&\&\, vlSF)} \ggg arr\ (uncurry\ (+)))\&\&\&(thetaSF \ggg arr\ cos))$$
$$\ggg arr\ (uncurry\ (*)) \ggg integral \ggg arr\ (/2)$$

Robot Simulator Revisit



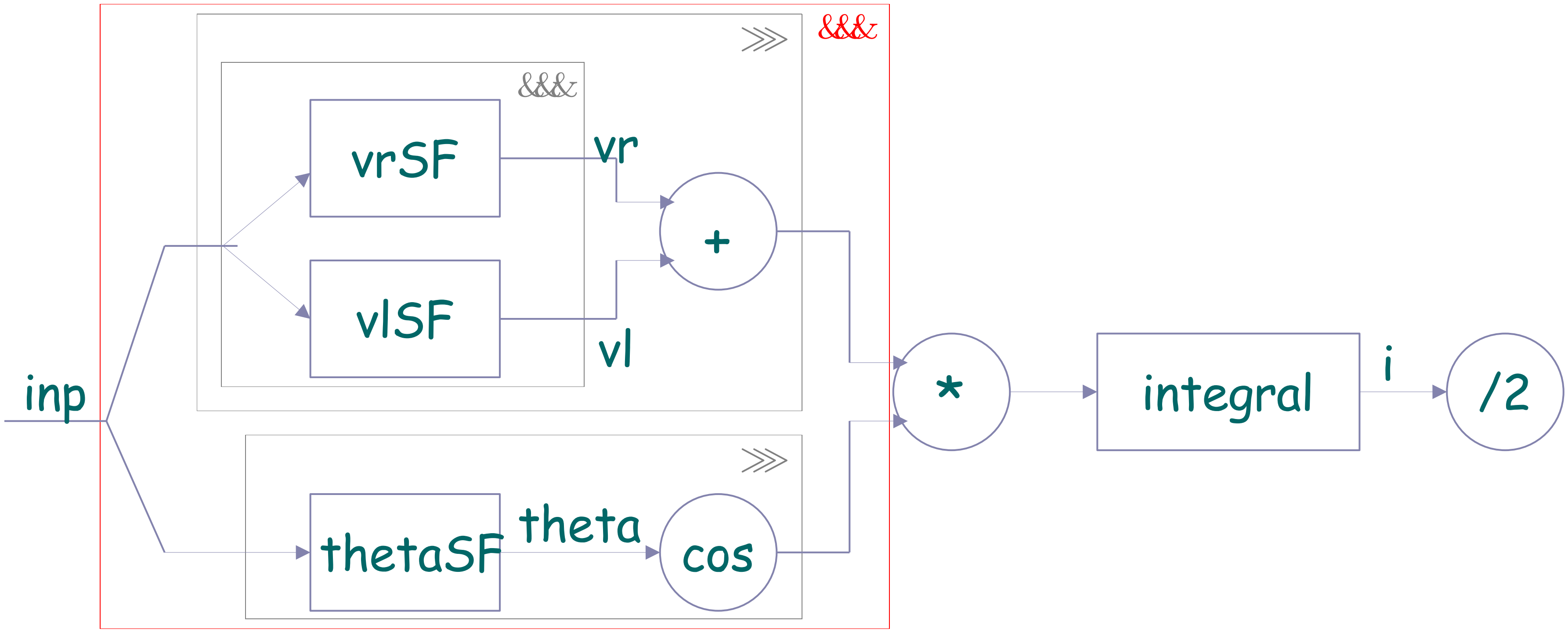
$$xSF = (\boxed{((vrSF \mathbin{\&\&\&} vlSF) \ggg arr\ (uncurry\ (+)))} \mathbin{\&\&\&} (thetaSF \ggg arr\ cos))$$
$$\ggg arr\ (uncurry\ (*)) \ggg integral \ggg arr\ (/2)$$

Robot Simulator Revisit



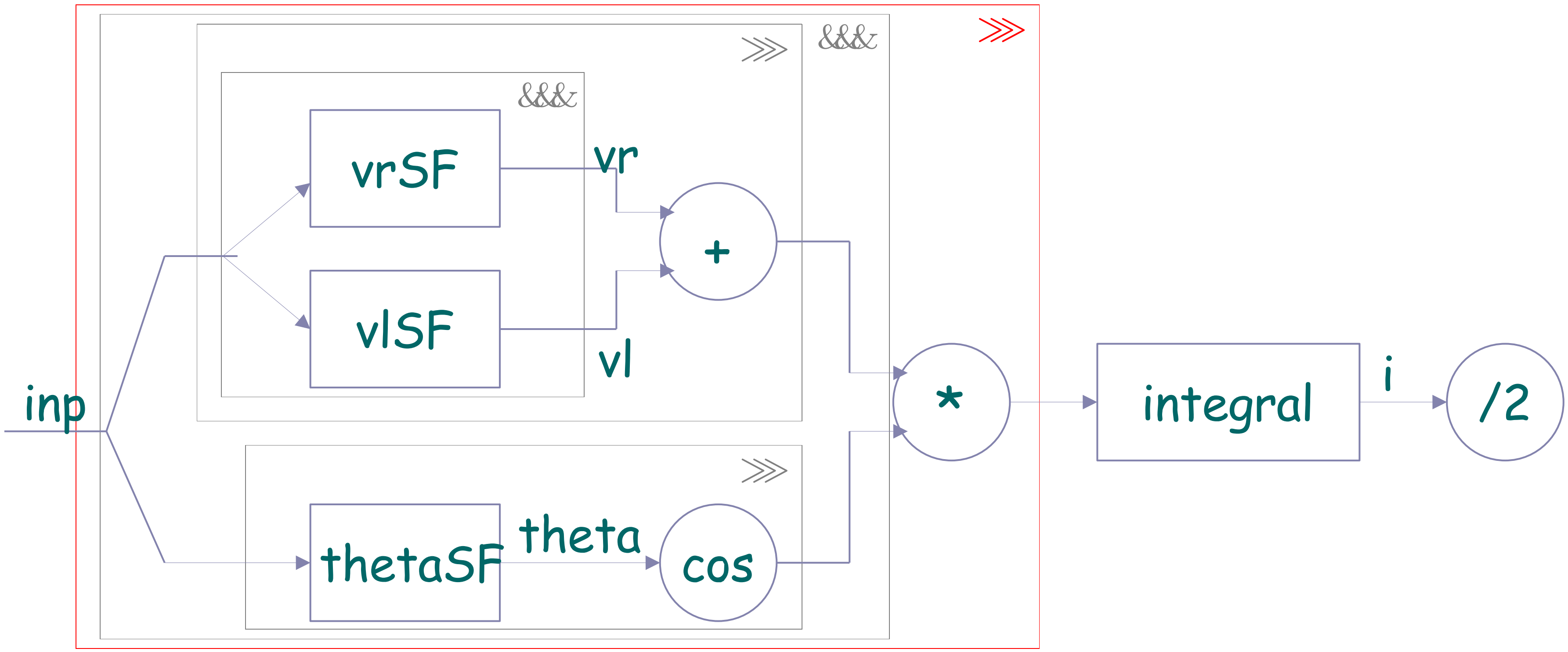
$$xSF = (((vrSF \&\&\& vlSF) \ggg arr\ (uncurry\ (+))) \&\&\& \boxed{(thetaSF \ggg arr\ cos)})$$
$$\ggg arr\ (uncurry\ (*)) \ggg integral \ggg arr\ (/2)$$

Robot Simulator Revisit



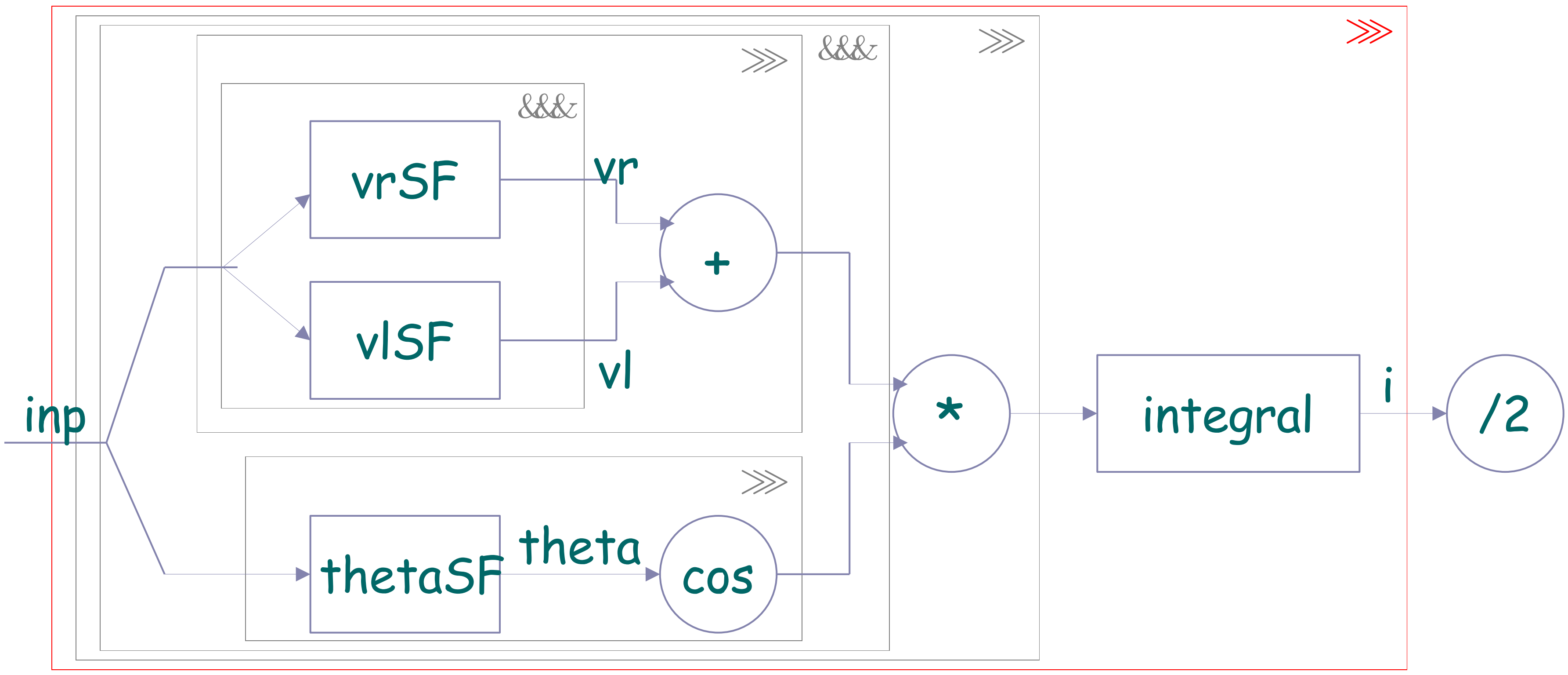
$$xSF = \boxed{(((vrSF \,\&\&\&\, vlSF) \ggg arr\ (uncurry\ (+)))\&\&\&(thetaSF \ggg arr\ cos))}$$
$$\ggg arr\ (uncurry\ (*)) \ggg integral \ggg arr\ (/2)$$

Robot Simulator Revisit



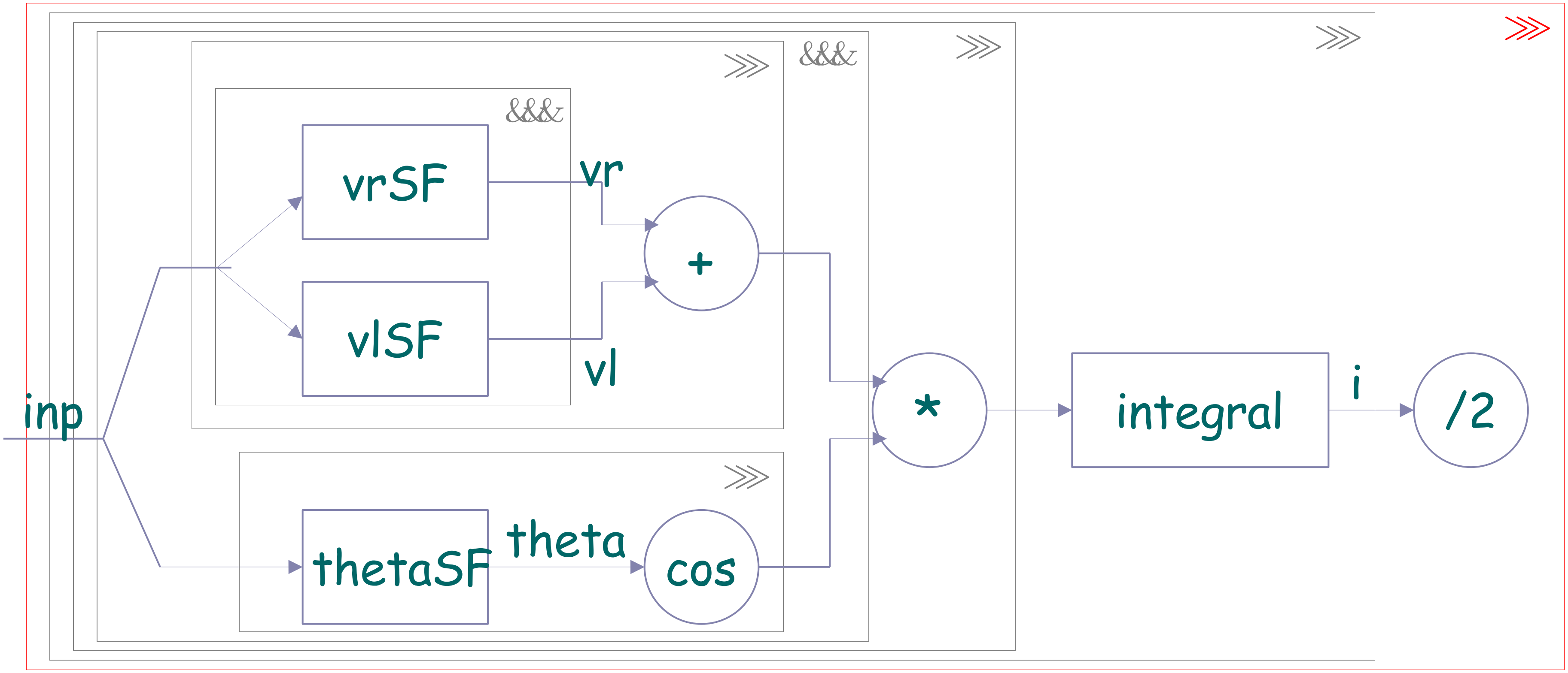
$$xSF = (((vrSF \&\&\& vlSF) \ggg arr\ (uncurry\ (+))) \&\&\& (thetaSF \ggg arr\ cos))$$
$$\ggg arr\ (uncurry\ (*)) \ggg integral \ggg arr\ (/2)$$

Robot Simulator Revisit



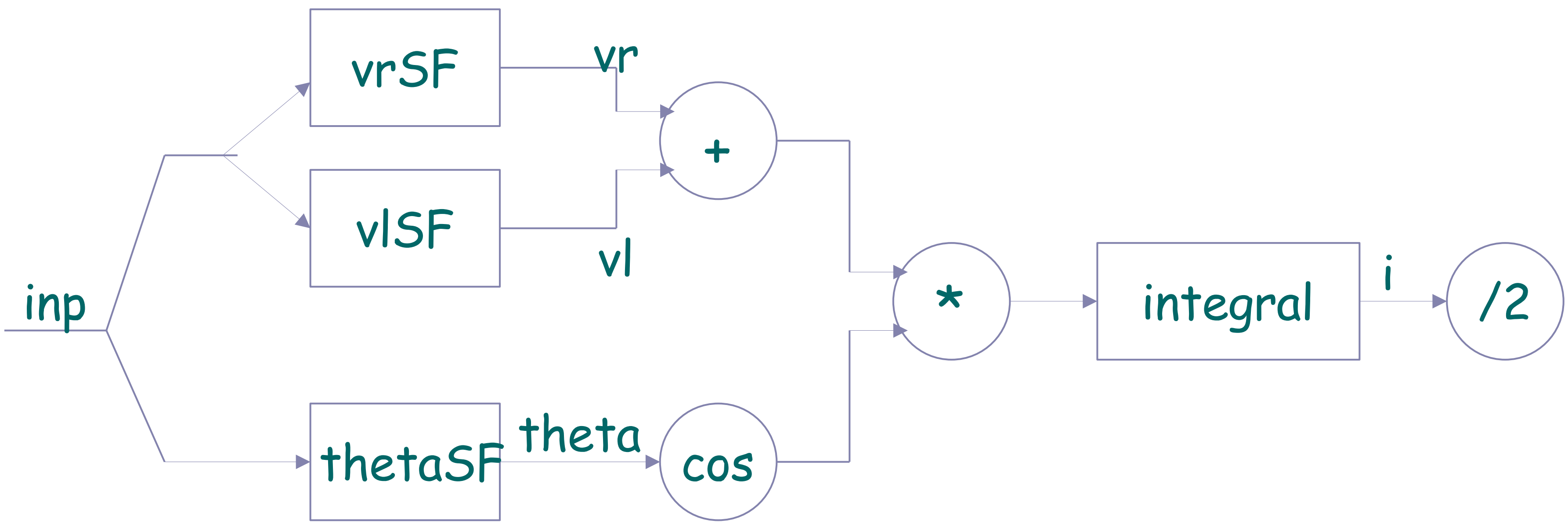
$$xSF = \boxed{(((vrSF \,\&\&\&\, vlSF) \ggg arr\ (uncurry\ (+)))\&\&\&(thetaSF \ggg arr\ cos)) \ggg arr\ (uncurry\ (*)) \ggg integral} \ggg arr\ (/2)$$

Robot Simulator Revisit



$$xSF = \boxed{(((vrSF \&\&\& vlSF) \ggg arr\ (uncurry\ (+)))\&\&\&(thetaSF \ggg arr\ cos)) \ggg arr\ (uncurry\ (*)) \ggg integral \ggg arr\ (/2)}$$

Robot Simulator in Arrow Syntax



$$
\begin{array}{l}
xSF = \textbf{proc}\ inp \rightarrow \textbf{do} \\
\qquad vr \leftarrow vrSF \prec inp \\
\qquad vl \leftarrow vlSF \prec inp \\
\qquad \theta \leftarrow thetaSF \prec inp \\
\qquad i \leftarrow integral \prec (vr + vl) * cos\ \theta \\
\qquad returnA \prec (i\ /\ 2)
\end{array}
$$

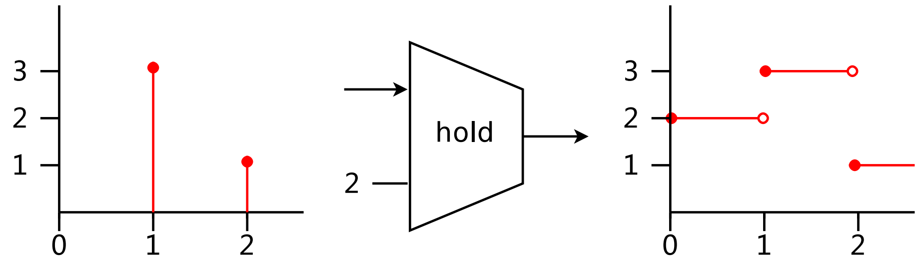
Modeling Discrete Events

Events are *instantaneous* and have no duration.

$$\textbf{data}\ Event\ a = Event\ a \mid NoEvent$$

Example: coerce from an discrete-time event stream to continuous-time signal by "holding" a previous event value.

$$hold :: a \rightarrow SF\ (Event\ a)\ a$$



Infinitesimal Delay with $iPre$

As a more primitive operator than $hold$, $iPre$ puts an infinitesimal delay over the input signal, and initializes it with a new value.

$$iPre :: a \rightarrow SF\ a\ a$$

We can implement $hold$ using $iPre$:

$$
\begin{array}{l}
hold\ i = \textbf{proc}\ e \rightarrow \mathbf{do} \\
\quad \textbf{rec}\ y \leftarrow iPre\ i \prec z \\
\qquad \mathbf{let}\ z = \mathbf{case}\ e\ \mathbf{of}\ \begin{array}{ll} Event\ x & \rightarrow x \\ NoEvent & \rightarrow y \end{array} \\
\quad returnA \prec z
\end{array}
$$

What's Good About Using Arrows in FRP

- Highly *abstract*, and yet allow domain specific extensions.
- Like monads, they are *composable* and can be stateful.
- Modular: both input and output are explicit.
- Higher-order signal function $SF\ a\ (b, Event\ (SF\ a\ b))$ as event switch.
- *Formal properties* expressed as 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	$first\ (first\ f) \ggg arr\ assoc = arr\ assoc \ggg first\ f$
	$\mathbf{where}\ assoc\ ((a, b), c) = (a, (b, c))$

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

FRP as a Domain Specific Language

What makes a good abstraction for FRP?

FRP as a Domain Specific Language

What makes a good abstraction for FRP?

Signals?

FRP as a Domain Specific Language

What makes a good abstraction for FRP?

Signals? *flexible, but ... not enough discipline.*

FRP as a Domain Specific Language

What makes a good abstraction for FRP?

Signals? *flexible, but ... not enough discipline.*

Arrows?

FRP as a Domain Specific Language

What makes a good abstraction for FRP?

Signals? *flexible, but ... not enough discipline.*

Arrows? *Disciplined, but ... not specific enough.*

FRP as a Domain Specific Language

What makes a good abstraction for FRP?

Signals? *flexible, but ... not enough discipline.*

Arrows? *Disciplined, but ... not specific enough.*

What is domain specific about FRP?

FRP as a Domain Specific Language

What makes a good abstraction for FRP?

Signals? *flexible, but ... not enough discipline.*

Arrows? *Disciplined, but ... not specific enough.*

What is domain specific about FRP? *Causality.*

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

FRP as a Domain Specific Language

What makes a good abstraction for FRP?

Signals? *flexible, but ... not enough discipline.*

Arrows? *Disciplined, but ... not specific enough.*

What is domain specific about FRP? *Causality.*

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

Can we refine the arrow abstraction to capture causality?

Part II. CCA

Causal Commutative Arrows (CCA)

Introduce a new operator $init$:

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

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!*

What's Good about CCA

CCA provides a *core set* of operators for dataflow computations.

- The $init$ operator *does not talk about time*, and the product law puts little restriction over its actual semantics.
- The commutativity law states an important non-interference property so that *side effects can only be local*.

What's Good about CCA

CCA provides a *core set* of operators for dataflow computations.

- The $init$ operator *does not talk about time*, and the product law puts little restriction over its actual semantics.
- The commutativity law states an important non-interference property so that *side effects can only be local*.

Quiz: why not make this a law?

$$init\ i \ggg arr\ f = arr\ f \ggg init\ (f\ i)$$

The CCA Language: Syntax

Variables	V	$::=$	$x \mid y \mid z \mid \ldots$
Types	A, B, C	$::=$	$1 \mid M \times N \mid A \rightarrow B \mid A \leadsto B$
Expressions	M, N	$::=$	$() \mid V \mid (M, N) \mid \mathit{fst}\ M \mid \mathit{snd}\ M \mid$
			$\lambda V.M \mid M\ N \mid \mathit{trace}\ M$
Programs	P, Q	$::=$	$\mathit{arr}\ M \mid P \ggg Q \mid \mathit{first}\ P \mid \mathit{loop}\ P \mid \mathit{init}\ M$
Environment	Γ	$::=$	$x_0 : A_0, \ldots, x_n : A_n$

- Typed lambda calculus extended with unit, product, arrow and trace.
- Instead of type classes, use $A \leadsto B$ to denote arrow type.
- Programs and expressions are separated on purpose, so that programs are only *finite compositions* of arrow combinators.

The CCA Language: Types

$$(\text{UNIT}) \quad \Gamma \vdash () : 1 \qquad (\text{VAR}) \; \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad (\text{TRACE}) \; \frac{\Gamma \vdash M : A \times C \to B \times C}{\Gamma \vdash \mathit{trace}\ M : A \to B}$$

$$(\text{ABS}) \; \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B} \qquad (\text{APP}) \; \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}$$

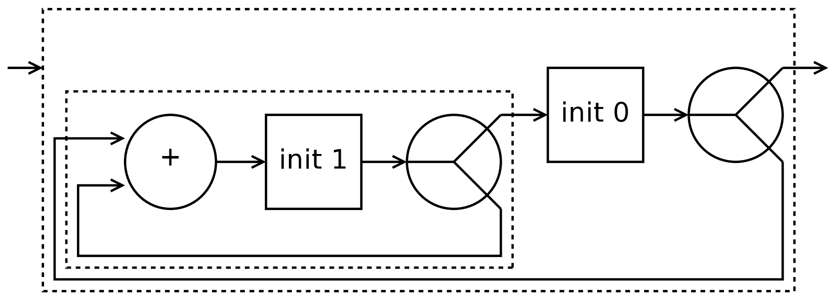
$$(\text{PAIR}) \; \frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash (M, N) : A \times B} \qquad (\text{FST}) \; \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathit{fst}\ M : A} \qquad (\text{SND}) \; \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathit{snd}\ M : B}$$

$$(\text{ARR}) \; \frac{\vdash M : A \to B}{\vdash \mathit{arr}\ M : A \rightsquigarrow B} \qquad (\text{SEQ}) \; \frac{\vdash P : A \rightsquigarrow B \quad \vdash Q : B \rightsquigarrow C}{\vdash P \ggg Q : A \rightsquigarrow C}$$

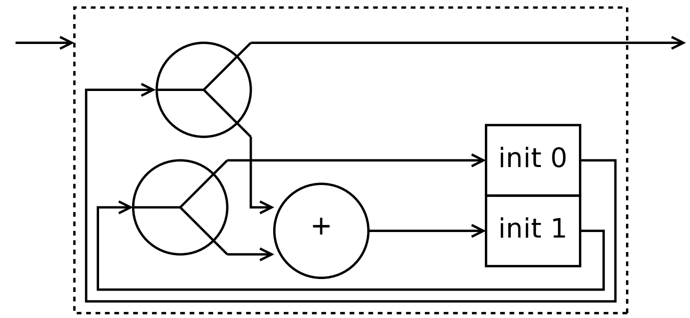
$$(\text{FIRST}) \; \frac{\vdash P : A \rightsquigarrow B}{\vdash \mathit{first}\ P : A \times C \rightsquigarrow B \times C} \qquad (\text{LOOP}) \; \frac{\vdash P : A \times C \rightsquigarrow B \times C}{\vdash \mathit{loop}\ P : A \rightsquigarrow B}$$

$$(\text{INIT}) \; \frac{\vdash M : A}{\vdash \mathit{init}\ M : A \rightsquigarrow A}$$

Causal Commutative Normal Form (CCNF)



(f) Original



(g) Normalized

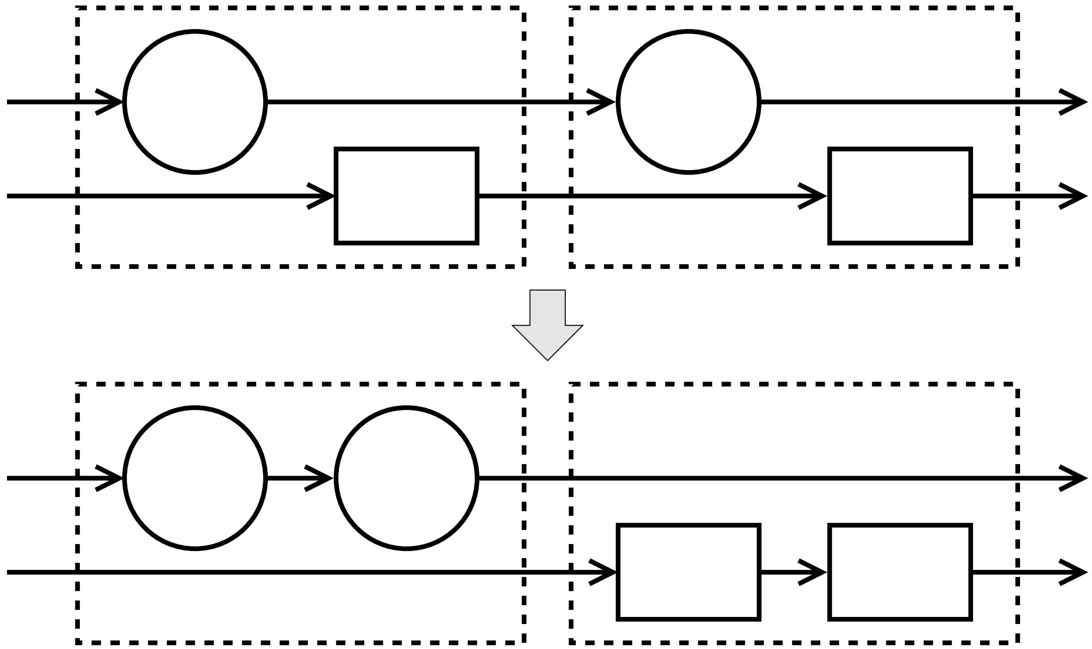
Theorem (CCNF) For all well typed CCA program $p : A \rightsquigarrow B$, there exists a normal form p_{norm}, called the Causal Commutative Normal Form, which is either of the form $arr\ f$, or $loopD\ i\ f$ for some i and f, such that $p_{norm} : A \rightsquigarrow B$, and $p \Downarrow p_{norm}$. In unsugared form, the second form is equivalent to

$$loopD\ i\ f = loop\ (arr\ f \ggg second\ (init\ i))$$

Normalization Explained

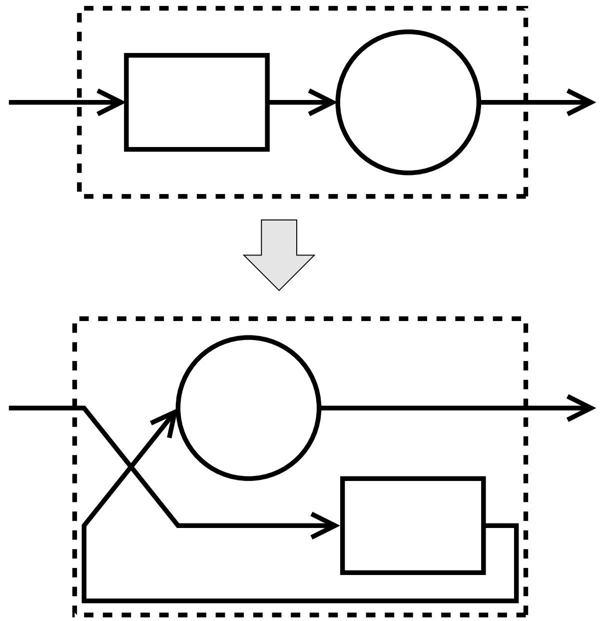
- Based on arrow laws, but directed.
- The two new laws, commutativity and product, are essential.
- Best illustrated by pictures...

Re-order Parallel Pure and Stateful Arrows



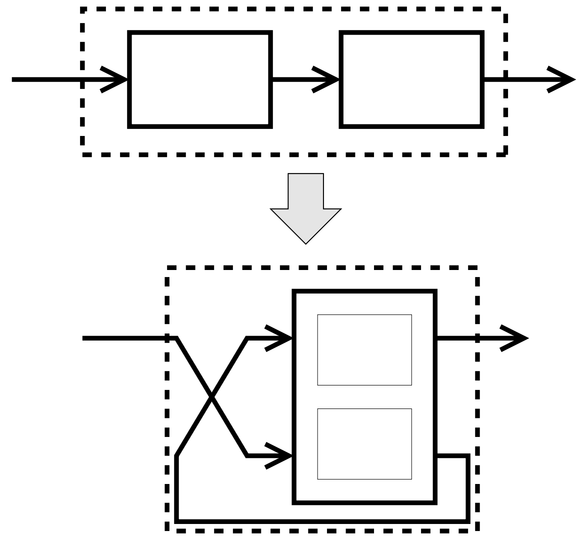
Related law: exchange (a special case of commutativity).

Re-order Sequential Pure and Stateful Arrows



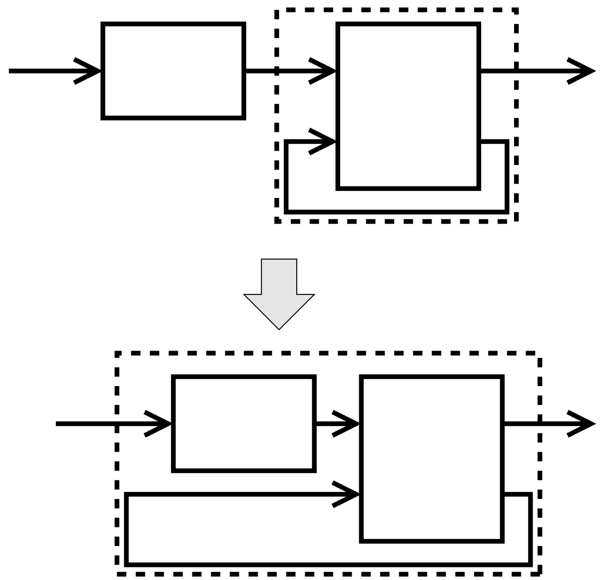
Related laws: tightening, sliding, and definition of second.

Change Sequential to Parallel



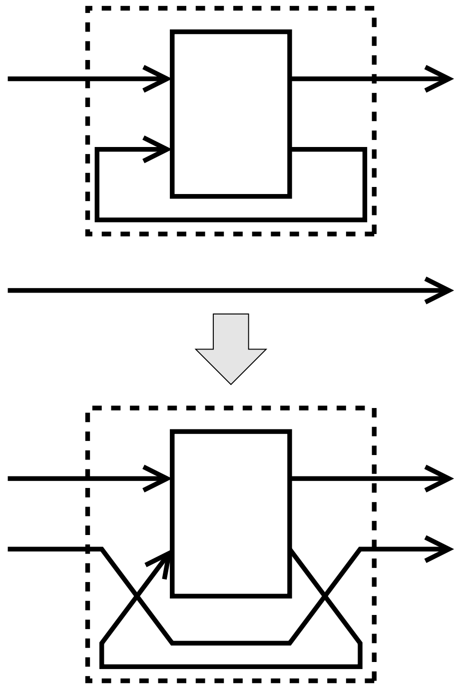
Related laws: product, tightening, sliding, and definition of second.

Move Sequential into Loop



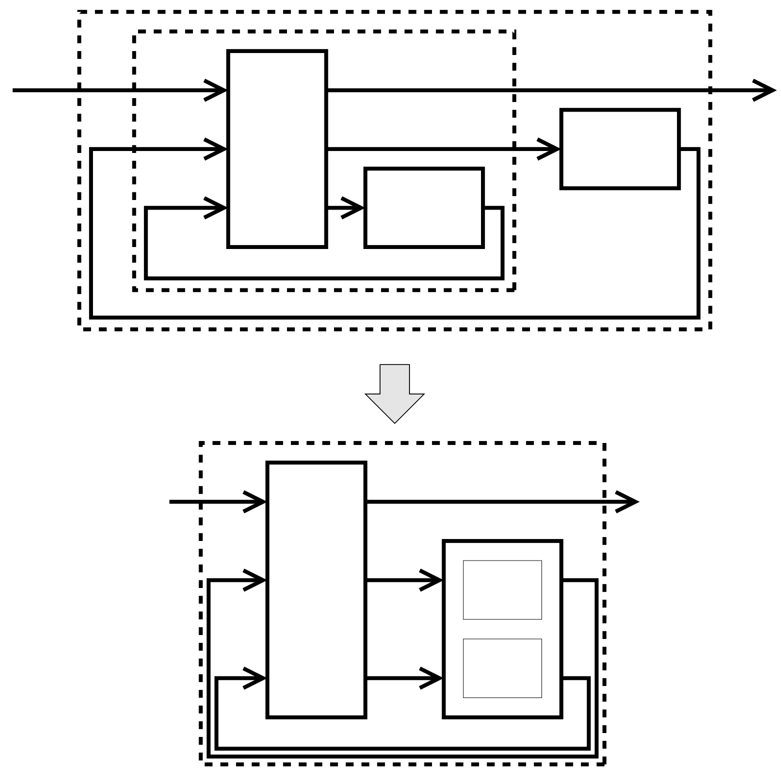
Related law: tightening.

Move Parallel into Loop



Related laws: superposing, and definition of second.

Fuse Nested Loops



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

Part III. Applications

Synchronous Dataflow

Programs written in a stream based dataflow language (Lucid):

$$
\begin{aligned}
ones &= 1 \text{ `}\mathit{fby}\text{` } ones \\
sum\ x &= x + 0 \text{ `}\mathit{fby}\text{` } sum\ x \\
nats &= sum\ ones
\end{aligned}
\qquad
\begin{aligned}
fibs = \mathbf{let}\ f &= 0 \text{ `}\mathit{fby}\text{` } g \\
g &= 1 \text{ `}\mathit{fby}\text{` } (f + g) \\
\mathbf{in}\ f &
\end{aligned}
$$

Compare to programs written in arrows:

$$
\begin{aligned}
ones &= arr\ (\lambda\_ \rightarrow 1) \\
sum &= \textbf{proc}\ x \rightarrow \mathbf{do} \\
&\qquad \textbf{rec}\ s \leftarrow init\ 0 \prec s' \\
&\qquad\quad \mathbf{let}\ s' = s + x \\
&\qquad returnA \prec s' \\
nats &= ones \ggg sum
\end{aligned}
\qquad
\begin{aligned}
fibs &= \textbf{proc}\ \_ \rightarrow \mathbf{do} \\
&\qquad \textbf{rec}\ f \leftarrow init\ 0 \prec g \\
&\qquad\quad g \leftarrow init\ 1 \prec (f + g) \\
&\qquad returnA \prec f
\end{aligned}
$$

Stream functions over discrete streams are arrows. We instantiate CCA by assigning $init$ the meaning of a *unit delay*, just like ‘fby‘.

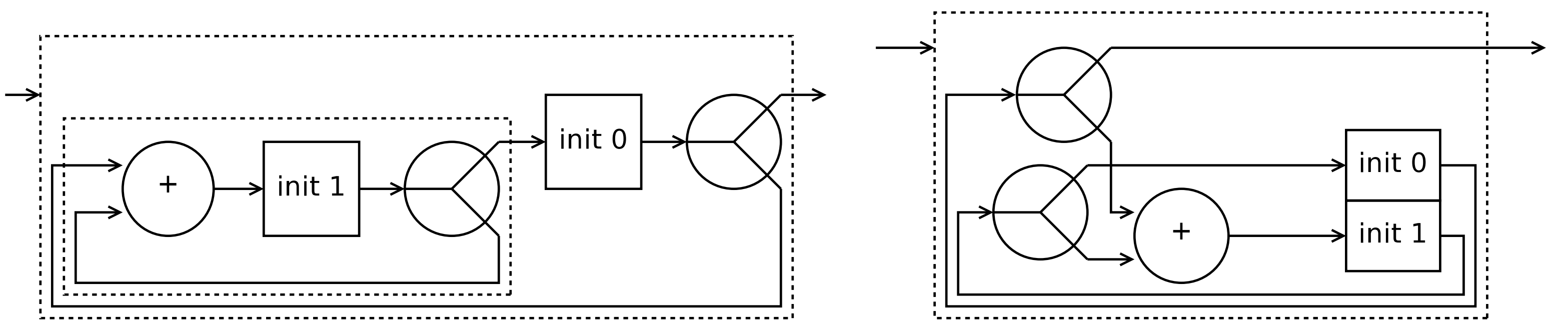
Synchronous Dataflow: Normalization Example

Same fibs program written in arrow combinators:

$$\begin{aligned} \mathit{fibs} = {} & \mathit{loop}\ (\mathit{arr}\ \mathit{snd} \ggg \mathit{loop}\ (\mathit{arr}\ (\mathit{uncurry}\ (+)) \ggg \mathit{init}\ 1 \ggg \mathit{arr}\ \mathit{dup}) \ggg \\ & \qquad \mathit{init}\ 0 \ggg \mathit{arr}\ \mathit{dup}) \\ & \textbf{where}\ \mathit{dup}\ x = (x, x) \end{aligned}$$

Its normal form:

$$\mathit{ccnf}_{\mathit{fibs}} = \mathit{loopD}\ (0,1)\ (\lambda(\_,(x,y)) \rightarrow (x,(y,x+y)))$$



(a) Original

(b) Normalized

CCNF Tuple and Operational Semantics

We call the pair (i, f) a CCNF tuple for a CCNF in the form $loopD\ i\ f$.

$$
\begin{array}{l}
run_{ccnf} :: (d, (b, d) \rightarrow (c, d)) \rightarrow [\,b\,] \rightarrow [\,c\,] \\
run_{ccnf}\ (i, f) = g\ i \\
\quad \textbf{where}\ g\ i\ (x : xs) = \textbf{let}\ (y, i') = f\ (x, i)\ \textbf{in}\ y : g\ i'\ xs
\end{array}
$$

run_{ccnf} implements an *operational semantics for causal stream functions* that is also known as a Mealy machine, a form of automata.

By using CCNF tuples directly, we avoid all arrow structures!

Dataflow Benchmarks (Speed Ratio)

Name	GHC^1	$arrowp^2$	$CCNF^3$	$CCNF\ Tuple^4$
sine	1.0	2.40	17.05	470.56
fibonacci	1.0	1.87	16.48	123.15
factorial	1.0	3.09	15.84	22.62
bounded counter	1.0	3.22	44.48	98.91

- 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. Arrow combinator after CCA normalization.
  4. CCNF tuple after CCA normalization.

Representing Autonomous ODE

An ordinary differential equation (ODE) of order n is of the form:

$$f^{(n)} = F(t, f, f', \ldots, f^{(n-1)})$$

for an unknown function $f(t)$, with its n^{th} derivative described by $f^{(n)}$, where $f \in \mathbb{R} \to \mathbb{R}$ and $t \in \mathbb{R}$.

An *initial value problem* of a first order autonomous ODE is of the form:

$$f' = F(f) \quad s.t. \quad f(t_0) = f_0$$

The given pair $(t_0, f_0) \in \mathbb{R} \times \mathbb{R}$ is called the *initial condition*.

DSL for ODE Using Tower of Derivatives

Function	**Mathematics**	**Haskell**
Sine wave	$y'' = -y$	$y = integral\ y_0\ y'$ $y' = integral\ y_1\ (-y)$
Damped oscillator	$y'' = -cy' - y$	$y = integral\ y_0\ y'$ $y' = integral\ y_1\ (-c * y' - y)$
Lorenz attractor	$x' = \sigma(y - x)$ $y' = x(\rho - z) - y$ $z' = xy - \beta z$	$x = integral\ x_0\ (\sigma * (y - x))$ $y = integral\ y_0\ (x * (\rho - z) - y)$ $z = integral\ z_0\ (x * y - \beta * z)$

ODE represented as a tower-of-derivatives (Karczmarczuk 1998):

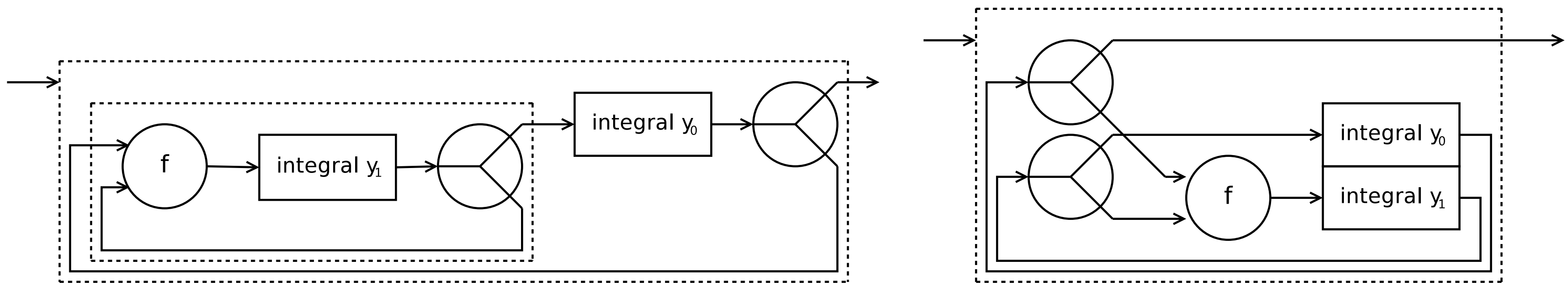
$$\begin{aligned}
&\mathbf{data}\ D\ a = D\ \{val :: a, der :: D\ a\} \\
&(+) \qquad :: D\ a \rightarrow D\ a \rightarrow D\ a \\
&(*) \qquad :: D\ a \rightarrow D\ a \rightarrow D\ a \\
&integral :: a \rightarrow D\ a \rightarrow D\ a \\
&integral\ v\ d = D\ v\ d
\end{aligned}$$

DSL for ODE Using Arrows

Sine wave	$y'' = -y$	**proc** $() \rightarrow \mathbf{do}$ **rec** $y \leftarrow integral\ y_0 \prec y'$ $y' \leftarrow integral\ y_1 \prec -y$ $returnA \prec y$
Damped oscillator	$y'' = -cy' - y$	**proc** $() \rightarrow \mathbf{do}$ **rec** $y \leftarrow integral\ y_0 \prec y'$ $y' \leftarrow integral\ y_1 \prec -c * y' - y$ $returnA \prec y$
Lorenz attractor	$x' = \sigma(y - x)$ $y' = x(\rho - z) - y$ $z' = xy - \beta z$	**proc** $() \rightarrow \mathbf{do}$ **rec** $x \leftarrow integral\ x_0 \prec \sigma * (y - x)$ $y \leftarrow integral\ y_0 \prec x * (\rho - z) - y$ $z \leftarrow integral\ z_0 \prec x * y - \beta * z$ $returnA \prec (x, y, z)$

ODE Arrows are CCA

The *integral* function is indeed just the *init* operator in CCA.



(c) Original (d) Normalized

After normalization to an CCNF tuple $(i, f) :: (s, (a, s) \to (b, s))$

- The state i is a nested tuple that represents a *vector of initial values*.
- The pure function f *computes the value of derivatives*.

ODEs can be numerically solved by using just CCNF tuples!

Extending CCA for Yampa Arrows

Yampa models both discrete-time and continuous-time signals with two essential arrow combinators:

$$
\begin{aligned}
iPre &:: a \rightarrow SF\ \ a\ \ a \\
integral &:: a \rightarrow SF\ \ a\ \ a
\end{aligned}
$$

Both fit the type of $init$ combinator of CCA.

Extending CCA for Yampa Arrows

Yampa models both discrete-time and continuous-time signals with two essential arrow combinators:

$$\begin{aligned} iPre &:: a \rightarrow SF\ a\ a \\ integral &:: a \rightarrow SF\ a\ a \end{aligned}$$

Both fit the type of $init$ combinator of CCA. *Solution: extend CCA with multi-sort inits!*

The CCNF for a Yampa arrow is either $arr\ f$, or

$$loopD_2\ (i, j)\ f = loop\ (arr\ f \ggg second\ (iPre\ i \star\!\star\!\star\ integral\ j))$$

Animate Yampa Arrow with $CCNF_2$

Represent the CCNF for Yampa arrow as a generalized algebraic data type (GADT):

$$
\begin{aligned}
&\mathbf{data}\ CCNF_2\ a\ b\ \mathbf{where}\\
&\quad CCNF_2 :: (VectorSpace\ DTime\ d, Num\ d) \Rightarrow\\
&\qquad\qquad ((c, d), (a, (c, d)) \rightarrow (b, (c, d))) \rightarrow CCNF_2\ a\ b
\end{aligned}
$$

Interact with the world with just $CCNF_2$, no more arrows!

$$
\begin{aligned}
&reactimate :: IO\ (DTime, a) \rightarrow (b \rightarrow IO\ ()) \rightarrow CCNF_2\ a\ b \rightarrow IO\ ()\\
&reactimate\ sense\ actuate\ (CCNF_2\ ((i, j), f)) = run\ i\ j\\
&\quad \mathbf{where}\ run\ i\ j = \mathbf{do}\\
&\qquad\qquad (dt, x) \leftarrow sense\\
&\qquad\qquad \mathbf{let}\ (y, (i_{new}, j')) = f\ (x, (i, j))\\
&\qquad\qquad\qquad j_{new} = euler\ dt\ j\ j'\\
&\qquad\qquad actuate\ y\\
&\qquad\qquad run\ i_{new}\ j_{new}
\end{aligned}
$$

Not All Yampa Arrows Are CCA

Yampa models *dynamic* systems with event switches:

$$switch :: SF\ a\ (b, Event\ c) \rightarrow (c \rightarrow SF\ a\ b) \rightarrow SF\ a\ b$$

Or alternatively:

$$switch :: SF\ a\ (b, Event\ (SF\ a\ b)) \rightarrow SF\ a\ b$$

But the normal form of CCA is *static*: both the state i and the function f in a CCNF tuple are of a *fixed structure*.

Workaround: do not use CCNF tuple directly, but use switches on top of normalized arrows.

Related Work

- Single while loop (Harel 1980).
- Compilation of synchronous dataflow (Halbwachs et al. 1991, Amagbagnon et al. 1995).
- Functional representation of streams (Caspi and Pouzet 1998).
- Functional stream derivatives (Rutten 2006).
- Stream Fusion (Coutts et al. 2007).
- FRP and arrow optimizations (Burchett et al. 2007, Nilsson 2005).

Why We Love Arrows

CCA is a fine example demonstrating the power of abstraction through arrows:

- High-level abstraction != sluggish performance.
- CCA extends generic arrows with domain knowledge. (ICFP2009)
- Use arrow for embedded DSLs and preserve sharing. (PADL2010)
- Arrows eliminate a certain form of space leaks in FRP. (ENTCS2007)

Future Work

- Improve CCA implementation with a new meta-programming tool.
- Optimize CCNF code with a custom code inliner/generator.
- Extend CCA to handle concurrent I/O.

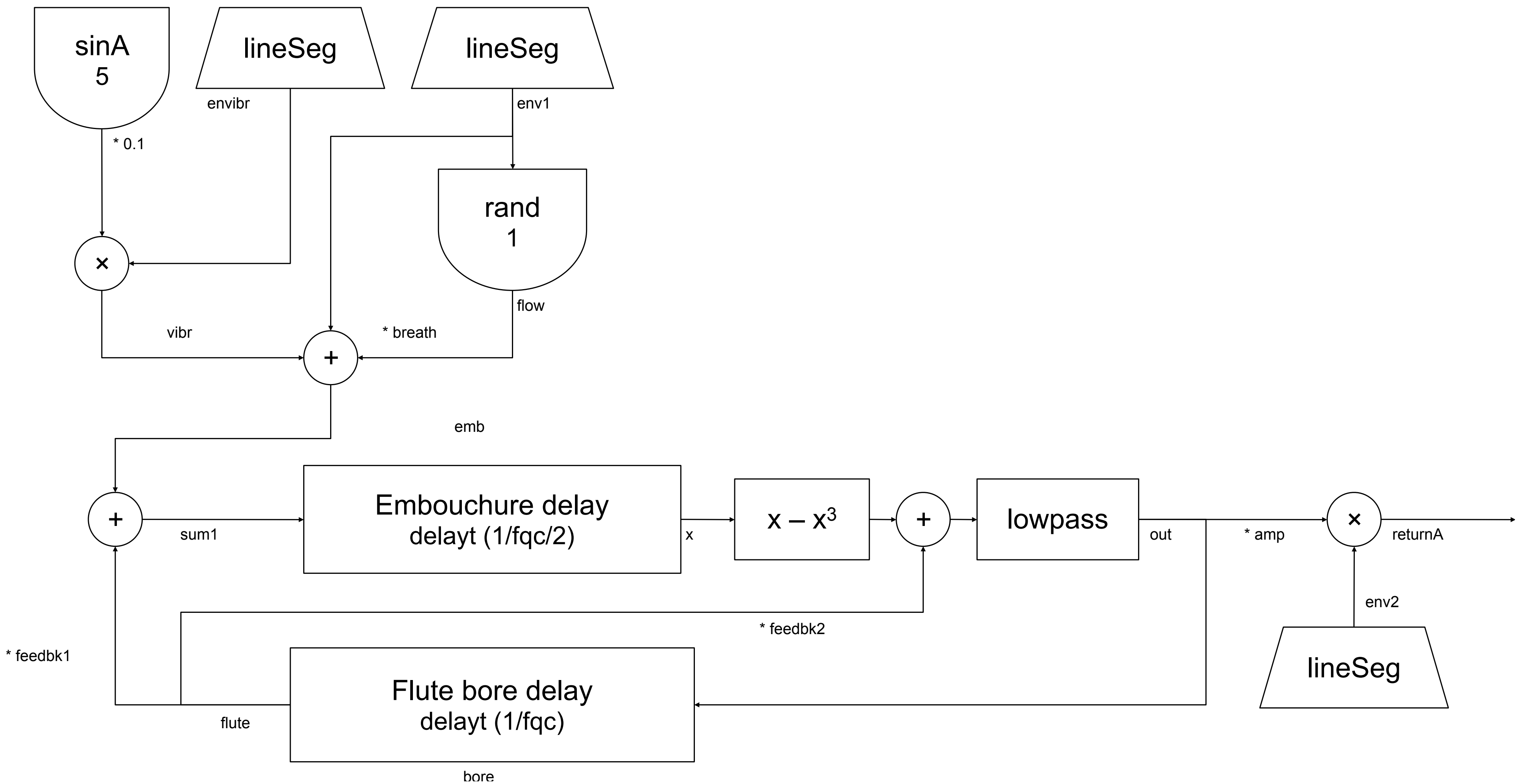
Thank you!

ODE Benchmarks (Speed Ratio)

Name	Tagged	Arrow	CCA
Exponential	1	0.17	83.72
Sine wave	1	0.35	27.52
Damped oscillator	1	1.13	82.34
Lorenz attractor	1	3.55	159.54

- Tagged version gets slower as program gets more complex.
- Arrow version still has some overhead.
- CCA version generates very efficient code in a tight loop.

Sound Sythesis Example

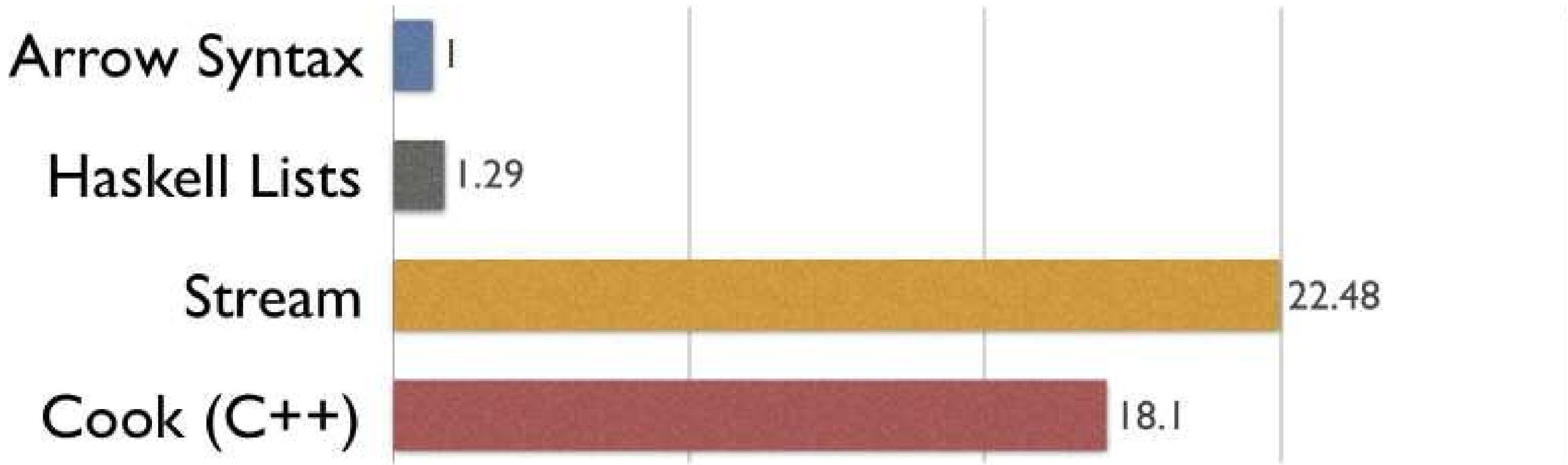


Block diagram of Parry Cook’s Flute generator

```
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
```

$$
\begin{array}{l}
loop\ (arr\ (\lambda(\_, out) \rightarrow ((), out)) \ggg \\
\quad (first\ timeA \ggg arr\ (\lambda(tm, out) \rightarrow (tm, (out, tm)))) \ggg \\
\quad\quad (first\ en1 \ggg arr\ (\lambda(env1, (out, tm)) \rightarrow (tm, (env1, out, tm)))) \ggg \\
\quad\quad\quad (first\ en2 \ggg \\
\quad\quad\quad\quad arr\ (\lambda(env2, (env1, out, tm)) \rightarrow (tm, (env1, env2, out)))) \ggg \\
\quad\quad\quad\quad (first\ enibr \ggg \\
\quad\quad\quad\quad\quad arr\ (\lambda(envibr, (env1, env2, out)) \rightarrow ((), (env1, env2, envibr, out)))) \ggg \\
\quad\quad\quad\quad\quad (first\ (sineA\ 5) \ggg \\
\quad\quad\quad\quad\quad\quad arr\ (\lambda(sin5, (env1, env2, envibr, out)) \rightarrow \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad ((), (env1, env2, envibr, out, sin5)))) \ggg \\
\quad\quad\quad\quad\quad\quad (first\ (arr\ rand\_f) \ggg \\
\quad\quad\quad\quad\quad\quad\quad arr\ (\lambda(rand, (env1, env2, envibr, out, sin5)) \rightarrow \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{let}\ vibr = sin5 * envibr * 0.1 \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad flow = rand * env1 \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad sum1 = breath * flow + env1 + vibr \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{in}\ (out, (env2, sum1)))) \ggg \\
\quad\quad\quad\quad\quad\quad\quad (first\ bore \ggg \\
\quad\quad\quad\quad\quad\quad\quad\quad arr\ (\lambda(flute, (env2, sum1)) \rightarrow ((flute, sum1), (env2, flute)))) \ggg \\
\quad\quad\quad\quad\quad\quad\quad\quad (first\ (arr\ (\lambda(flute, sum1) \rightarrow sum1 + flute * 0.4) \ggg emb) \ggg \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad arr\ (\lambda(x, (env2, flute)) \rightarrow ((flute, x), env2))) \ggg \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad (first\ (arr\ (\lambda(flute, x) \rightarrow x - x * x * x + flute * 0.4) \ggg \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad lowpassA\ 0.27) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \ggg arr\ (\lambda(out, env2) \rightarrow ((env2, out), out)))) \\
\quad \ggg arr\ (\lambda(env2, out) \rightarrow out * amp * env2)
\end{array}
$$

```
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 loopD ((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