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:

\[
\text{Signal } a \approx \text{Time } \rightarrow a
\]

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

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

\[
x(t) = \frac{1}{2} \int_{0}^{t} (v_r(t) + v_l(t)) \cos(\theta(t)) \, dt
\]

\[
\theta(t) = \frac{1}{l} \int_{0}^{t} (v_r(t) - v_l(t)) \, dt
\]

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

\[
x = (1 / 2) \times \text{integral} \ ((v_r + v_l) \times \cos \theta)
\]

\[
\theta = (1 / l) \times \text{integral} \ (v_r - v_l)
\]

Domain specific operators:

\[
(+) \quad :: \ Signal \ a \rightarrow \ Signal \ a \rightarrow \ Signal \ a
\]

\[
(*) \quad :: \ Signal \ a \rightarrow \ Signal \ a \rightarrow \ Signal \ a
\]

\[
\text{integral} :: \ Signal \ a \rightarrow \ Signal \ a
\]

...
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 $\text{Signal (Event (Signal a))}$ 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:

$$\text{SF } a \ b \approx \text{Signal } a \to \text{Signal } b$$

_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:

```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

(&&&)  :: (Arrow a) => a b c -> a b c' -> a b (c, c')
f &&& g  = arr (λx -> (x, x)) ≫≫ (f ⋆⋆⋆ g)
```
Picturing an Arrow

To model recursion, Paterson (2001) introduces \textit{ArrowLoop}:

\begin{verbatim}
class Arrow a ⇒ ArrowLoop a where
    loop :: a (b, d) (c, d) → a b c
\end{verbatim}
Robot Simulator Revisit

\[ \mathbf{x}_{SF} = (((\mathbf{vr}_{SF} \& \& \mathbf{vl}_{SF}) \gg arr (\text{uncurry} \, (+))) \& \& (\theta_{SF} \gg arr \cos)) \gg arr (\text{uncurry} \, (*) \gg \text{integral} \gg arr \, (/2)) \]
Robot Simulator Revisit

\[
x_{SF} = ((\text{vr}_{SF} \&\& \text{vl}_{SF}) \gg \text{arr} \ (\text{uncurry} \ (+))) \&\& (\text{theta}_{SF} \gg \text{arr} \ \cos) \\
\gg \text{arr} \ (\text{uncurry} \ (*) ) \gg \text{integral} \gg \text{arr} \ (/2)
\]
Robot Simulator Revisit

\[ x_{SF} = (((vr_{SF} && vl_{SF}) \gg arr (uncurry (+))) && (theta_{SF} \gg arr cos)) \gg arr (uncurry (\times)) \gg integral \gg arr (/2) \]
\[ x_{SF} = (((vr_{SF} \&\& vl_{SF}) \gg\gg arr (uncurry (+))) \&\& (theta_{SF} \gg\gg arr \cos)) \gg\gg arr (uncurry (\ast)) \gg\gg integral \gg\gg arr (/2) \]
Robot Simulator Revisit

$x_{SF} = (((vr_{SF} \&\& vl_{SF}) \gg arr (uncurry (+)))) \&\& (theta_{SF} \gg arr \ cos))

\gg arr (uncurry (*)) \gg integral \gg arr (/2)$
Robot Simulator Revisit

\[ x_{SF} = (((vr_{SF} \&\& vl_{SF}) \gg arr (uncurry (+))) \&\& (theta_{SF} \gg arr \cos)) \gg arr (uncurry (\ast)) \gg integral \gg arr (/2) \]
Robot Simulator Revisit

$$x_{SF} = (((v_{rSF} \& \& v_{lSF}) \ggg arr (uncurry (+))) \& \& (theta_{SF} \ggg arr \, cos)) \ggg arr (uncurry (*) \ggg integral \ggg arr (/2))$$
Robot Simulator Revisit

\[ x_{SF} = (((vr_{SF} \&\& vl_{SF}) \gg arr (uncurry (+))) \&\& (\theta_{SF} \gg arr \cos)) \gg arr (uncurry (\ast)) \gg integral \gg arr (/2) \]
Robot Simulator in Arrow Syntax

\[ x_{SF} = \textproc{proc } inp \to \text{do} \]

\[
\begin{align*}
vr & \leftarrow vr_{SF} \prec inp \\
vl & \leftarrow vl_{SF} \prec inp \\
\theta & \leftarrow \text{theta}_{SF} \prec inp \\
i & \leftarrow \text{integral} \prec (vr + vl) \times \cos \theta \\
returnA & \prec (i / 2)
\end{align*}
\]
Modeling Discrete Events

Events are *instantaneous* and have no duration.

\[
data \ Event \ a = Event \ a \mid NoEvent
\]

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 $\text{hold}$, $iPre$ puts an infinitesimal delay over the input signal, and initializes it with a new value.

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

We can implement $\text{hold}$ using $iPre$:

$$\text{hold } i = \text{proc } e \rightarrow \text{do}$$

$$\text{rec } y \leftarrow iPre \quad i \leftarrow z$$

$$\text{let } z = \text{case } e \text{ of } \text{Event } x \rightarrow x$$

$$\quad \text{NoEvent} \rightarrow y$$

$$\text{return } A \leftarrow z$$
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 \quad \text{arr id} \gg f = f
right identity \quad f \gg \text{arr id} = f
associativity \quad (f \gg g) \gg h = f \gg (g \gg h)
composition \quad \text{arr} (g \cdot f) = \text{arr} f \gg \text{arr} g
extension \quad \text{first} (\text{arr} f) = \text{arr} (f \times \text{id})
functor \quad \text{first} (f \gg g) = \text{first} f \gg \text{first} g
exchange \quad \text{first} f \gg \text{arr} (\text{id} \times g) = \text{arr} (\text{id} \times g) \gg \text{first} f
unit \quad \text{first} f \gg \text{arr} \text{fst} = \text{arr} \text{fst} \gg f
association \quad \text{first} (\text{first} f) \gg \text{arr} \text{assoc} = \text{arr} \text{assoc} \gg \text{first} f

where \text{assoc} ((a, b), c) = (a, (b, c))
Arrow Loop Laws

left tightening
\[ \text{loop}\left(\text{first } h \gg f\right) = h \gg \text{loop } f \]

right tightening
\[ \text{loop}\left(f \gg \text{first } h\right) = \text{loop } f \gg h \]

sliding
\[ \text{loop}\left(f \gg \text{arr } \left(\text{id } \times k\right)\right) = \text{loop}\left(\text{arr } \left(\text{id } \times k\right) \gg f\right) \]

vanishing
\[ \text{loop}\left(\text{loop } f\right) = \text{loop}\left(\text{arr } \text{assoc}^{-1} \gg f \gg \text{arr } \text{assoc}\right) \]

superposing
\[ \text{second}\left(\text{loop } f\right) = \text{loop}\left(\text{arr } \text{assoc} \gg \text{second } f \gg \text{arr } \text{assoc}^{-1}\right) \]

extension
\[ \text{loop}\left(\text{arr } f\right) = \text{arr } \left(\text{trace } f\right) \]
\[ \text{where } \text{trace } f \ b = \text{let } \left(c, d\right) = f \left(b, d\right) \text{ in } c \]
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 \textit{init}:

\begin{verbatim}
  class ArrowLoop a \Rightarrow ArrowInit a where
    init :: b \rightarrow a b b
\end{verbatim}

and two additional laws:

\begin{itemize}
  \item \textbf{commutativity} \quad \textit{first }f \ggg \textit{second }g = \textit{second }g \ggg \textit{first }f
  \item \textbf{product} \quad \textit{init }i \ggg \ggg \textit{init }j = \textit{init }\langle i, j \rangle
\end{itemize}

and still remain \textit{abstract}!
What’s Good about CCA

CCA provides a core set of operators for dataflow computations.

- The \textit{init} operator \textit{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 \textit{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?

\[
\text{init } i \gg arr f = arr f \gg \text{init } (f \ i)
\]
The CCA Language: Syntax

Variables \[ V ::= x \mid y \mid z \mid \ldots \]

Types \[ A, B, C ::= 1 \mid M \times N \mid A \to B \mid A \rightsquigarrow B \]

Expressions \[ M, N ::= () \mid V \mid (M, N) \mid \text{fst } M \mid \text{snd } M \mid \lambda V.M \mid M \; N \mid \text{trace } M \]

Programs \[ P, Q ::= \text{arr } M \mid P \gg Q \mid \text{first } P \mid \text{loop } P \mid \text{init } M \]

Environment \[ \Gamma ::= x_0 : A_0, \ldots, x_n : A_n \]

- Typed lambda calculus extended with unit, product, arrow and \textit{trace}.
- Instead of type classes, use \( A \rightsquigarrow B \) to denote arrow type.
- Programs and expressions are separated on purpose, so that programs are only \textit{finite compositions} of arrow combinators.
The CCA Language: Types

(UNIT) \( \Gamma \vdash () : 1 \)

(VAR) \( x : A \in \Gamma \) \( \Gamma \vdash x : A \)

(TRACE) \( \Gamma \vdash M : A \times C \rightarrow B \times C \)

\( \Gamma \vdash \text{trace} M : A \rightarrow B \)

(ABS) \( \Gamma, x : A \vdash M : B \) \( \Gamma \vdash \lambda x. M : A \rightarrow B \)

(APP) \( \Gamma \vdash M : A \rightarrow B \) \( \Gamma \vdash N : A \) \( \Gamma \vdash MN : B \)

(PAIR) \( \Gamma \vdash M : A \) \( \Gamma \vdash N : B \) \( \Gamma \vdash (M, N) : A \times B \)

(FST) \( \Gamma \vdash M : A \times B \) \( \Gamma \vdash \text{fst} M : A \)

(SND) \( \Gamma \vdash M : A \times B \) \( \Gamma \vdash \text{snd} M : B \)

(ARR) \( \vdash M : A \rightarrow B \) \( \vdash \text{arr} M : A \rightsquigarrow B \)

(SEQ) \( \vdash P : A \rightsquigarrow B \) \( \vdash Q : B \rightsquigarrow C \) \( \vdash P \gg Q : A \rightsquigarrow C \)

(LOOP) \( \vdash P : A \times C \rightsquigarrow B \times C \) \( \vdash \text{loop} P : A \rightsquigarrow B \)

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

$$\text{loopD } i f = \text{loop } (\text{arr } f \gg \text{second } (\text{init } i))$$
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):

\[
\begin{align*}
\text{ones} &= 1 \, \text{fby} \, \text{ones} \\
\text{sum} \, x &= x + 0 \, \text{fby} \, \text{sum} \, x \\
\text{nats} &= \text{sum} \, \text{ones}
\end{align*}
\]

\[
\begin{align*}
\text{fibs} &= \text{let} \ f = 0 \, \text{fby} \, g \\
&\quad \text{g} = 1 \, \text{fby} \, (f + g) \\
&\quad \text{in} \ f 
\end{align*}
\]

Compare to programs written in arrows:

\[
\begin{align*}
\text{ones} &= \text{arr} \, (\lambda_\_ \to 1) \\
\text{sum} &= \text{proc} \ x \to \text{do} \\
\quad &\quad \text{rec} \; s \leftarrow \text{init} \ 0 < s' \\
\quad &\quad \quad \text{let} \; s' = s + x \\
\quad &\quad \quad \text{returnA} < s' \\
\text{nats} &= \text{ones} \gg \text{sum}
\end{align*}
\]

\[
\begin{align*}
\text{fibs} &= \text{proc} \ _ \to \text{do} \\
\quad &\quad \text{rec} \; f \leftarrow \text{init} \ 0 < g \\
\quad &\quad \quad \text{g} \leftarrow \text{init} \ 1 < (f + g) \\
\quad &\quad \text{returnA} < f
\end{align*}
\]

Stream functions over discrete streams are arrows. We instantiate CCA by assigning \textit{init} the meaning of a \textit{unit delay}, just like \textit{fby}. 
Synchronous Dataflow: Normalization Example

Same \texttt{fibs} program written in arrow combinators:

\[
\text{\texttt{fibs}} = \text{\texttt{loop}} \left( \text{\texttt{arr}} \ \text{\texttt{snd}} \ \gg \ \text{\texttt{loop}} \left( \text{\texttt{arr}} \ (\text{\texttt{uncurry}} \ (\texttt{+})) \ \gg \ \text{\texttt{init}} \ 1 \ \gg \ \text{\texttt{arr}} \ \text{\texttt{dup}} \right) \ \gg \ \\
\text{\texttt{init}} \ 0 \ \gg \ \text{\texttt{arr}} \ \text{\texttt{dup}} \right)
\]

where \texttt{dup} \ \texttt{x} = (x, x)

Its normal form:

\[
\text{ccnf}_{\text{fibs}} = \text{loopD} \ (0, 1) \ (\lambda (_, (x, y)) \rightarrow (x, (y, x + y)))
\]
CCNF Tuple and Operational Semantics

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

\[
\text{run}_{\text{ccnf}} :: (d, (b, d) \rightarrow (c, d)) \rightarrow [b] \rightarrow [c]
\]

\[
\text{run}_{\text{ccnf}} (i, f) = g i
\]

\[
\text{where } g i (x : xs) = \text{let } (y, i') = f (x, i) \text{ in } y : g i' xs
\]

\(\text{run}_{\text{ccnf}}\) 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)

<table>
<thead>
<tr>
<th>Name</th>
<th>GHC&lt;sup&gt;1&lt;/sup&gt;</th>
<th>arrowp&lt;sup&gt;2&lt;/sup&gt;</th>
<th>CCNF&lt;sup&gt;3&lt;/sup&gt;</th>
<th>CCNF Tuple&lt;sup&gt;4&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>sine</td>
<td>1.0</td>
<td>2.40</td>
<td>17.05</td>
<td>470.56</td>
</tr>
<tr>
<td>fibonacci</td>
<td>1.0</td>
<td>1.87</td>
<td>16.48</td>
<td>123.15</td>
</tr>
<tr>
<td>factorial</td>
<td>1.0</td>
<td>3.09</td>
<td>15.84</td>
<td>22.62</td>
</tr>
<tr>
<td>bounded counter</td>
<td>1.0</td>
<td>3.22</td>
<td>44.48</td>
<td>98.91</td>
</tr>
</tbody>
</table>

- 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

<table>
<thead>
<tr>
<th>Function</th>
<th>Mathematics</th>
<th>Haskell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sine wave</td>
<td>( y'' = -y )</td>
<td>( y = \text{integral } y_0 ; y' )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( y' = \text{integral } y_1 ; (-y) )</td>
</tr>
<tr>
<td>Damped oscillator</td>
<td>( y'' = -c y' - y )</td>
<td>( y = \text{integral } y_0 ; y' )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( y' = \text{integral } y_1 ; (-c \ast y' - y) )</td>
</tr>
<tr>
<td>Lorenz attractor</td>
<td>( x' = \sigma (y - x) )</td>
<td>( x = \text{integral } x_0 ; (\sigma \ast (y - x)) )</td>
</tr>
<tr>
<td></td>
<td>( y' = x (\rho - z) - y )</td>
<td>( y = \text{integral } y_0 ; (x \ast (\rho - z) - y) )</td>
</tr>
<tr>
<td></td>
<td>( z' = xy - \beta z )</td>
<td>( z = \text{integral } z_0 ; (x \ast y - \beta \ast z) )</td>
</tr>
</tbody>
</table>

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

```haskell
data D a = D { val :: a, der :: D a }

(+) :: D a -> D a -> D a

(*) :: D a -> D a -> D a

integral :: a -> D a -> D a

integral v d = D v d
```
### DSL for ODE Using Arrows

<table>
<thead>
<tr>
<th>System</th>
<th>Equation</th>
<th>Proc Code</th>
</tr>
</thead>
</table>
| Sine wave            | $y'' = -y$        | **proc () → do**<br>  
  
  **rec** $y \leftarrow \text{integral } y_0 \leftarrow y'$<br>  
  $y' \leftarrow \text{integral } y_1 \leftarrow -y$<br>  
  return $A \leftarrow y$ |
| Damped oscillator    | $y'' = -cy' - y$  | **proc () → do**<br>  
  
  **rec** $y \leftarrow \text{integral } y_0 \leftarrow y'$<br>  
  $y' \leftarrow \text{integral } y_1 \leftarrow -c \ast y' - y$<br>  
  return $A \leftarrow y$ |
| Lorenz attractor     | $x' = \sigma(y - x)$<br>$y' = x(\rho - z) - y$<br>$z' = xy - \beta z$ | **proc () → do**<br>  
  
  **rec** $x \leftarrow \text{integral } x_0 \leftarrow \sigma \ast (y - x)$<br>  
  $y \leftarrow \text{integral } y_0 \leftarrow x \ast (\rho - z) - y$<br>  
  $z \leftarrow \text{integral } z_0 \leftarrow x \ast y - \beta \ast z$<br>  
  return $A \leftarrow (x, y, z)$ |
ODE Arrows are CCA

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

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

- 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:

\[ iPre \quad :: \quad a \rightarrow SF \ a \ a \]
\[ integral \quad :: \quad a \rightarrow SF \ a \ a \]

Both fit the type of \textit{init} combinator of CCA.
Extending CCA for Yampa Arrows

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

\[
\begin{align*}
\text{iPre} &:: a \to SF\ a\ a \\
\text{integral} &:: a \to SF\ a\ a
\end{align*}
\]

Both fit the type of \textit{init} combinator of CCA. \textbf{Solution: extend CCA with multi-sort inits!}

The CCNF for a Yampa arrow is either \textit{arr} \(f\), or

\[
\text{loopD}_2 \ (i, j) \ f = \text{loop} \ (\text{arr} \ f \ \gg\gg \text{second} \ (\text{iPre} \ i \ \ast\ast\ast \ \text{integral} \ j))
\]
Represent the CCNF for Yampa arrow as a generalized algebraic data type (GADT):

\[
data \text{CCNF}_{2} \ a \ b \ where
CCNF_{2} :: (\text{VectorSpace } DTime \ d, \text{Num } d) \Rightarrow
((c, d), (a, (c, d))) \rightarrow (b, (c, d))) \rightarrow \text{CCNF}_{2} \ a \ b
\]

Interact with the world with just \( \text{CCNF}_{2} \), no more arrows!

\[
\text{reactimate} :: IO (\text{DTime}, a) \rightarrow (b \rightarrow IO ()) \rightarrow \text{CCNF}_{2} \ a \ b \rightarrow IO ()
\]

\[
\text{reactimate sense actuate } (\text{CCNF}_{2} ((i, j), f)) = \text{run } i \ j
\]

\[
\text{where } \text{run } i \ j = \text{do}
\]

\[
(dt, x) \leftarrow \text{sense}
\]

\[
\text{let } (y, (i_{\text{new}}, j')) = f (x, (i, j))
\]

\[
j_{\text{new}} = \text{euler } dt \ j \ j'
\]

\[
\text{actuate } y
\]

\[
\text{run } i_{\text{new}} j_{\text{new}}
\]
Not All Yampa Arrows Are CCA

Yampa models *dynamic* systems with event switches:

\[
\text{switch} :: SF \ a \ (b, \ Event \ c) \rightarrow (c \rightarrow SF \ a \ b) \rightarrow SF \ a \ b
\]

Or alternatively:

\[
\text{switch} :: SF \ a \ (b, \ Event \ (SF \ a \ b)) \rightarrow SF \ a \ b
\]

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).
- 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)

<table>
<thead>
<tr>
<th>Name</th>
<th>Tagged</th>
<th>Arrow</th>
<th>CCA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponential</td>
<td>1</td>
<td>0.17</td>
<td>83.72</td>
</tr>
<tr>
<td>Sine wave</td>
<td>1</td>
<td>0.35</td>
<td>27.52</td>
</tr>
<tr>
<td>Damped oscillator</td>
<td>1</td>
<td>1.13</td>
<td>82.34</td>
</tr>
<tr>
<td>Lorenz attractor</td>
<td>1</td>
<td>3.55</td>
<td>159.54</td>
</tr>
</tbody>
</table>

- 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 Synthesis Example

Block diagram of Parry Cook’s Flute generator
\[ \text{flute0} \ \text{dur} \ \text{amp} \ \text{fqc} \ \text{press} \ \text{breath} = \]

\[
\begin{align*}
\text{let } & \text{en1 }= \text{arr} \ \$ \ \text{lineSeg} \ [0, 1.1 * \text{press}, \text{press}, \text{press}, 0] \ [0.06, 0.2, \text{dur} - 0.16, 0.02] \\
\text{en2 }&= \text{arr} \ \$ \ \text{lineSeg} \ [0, 1, 1, 0] \ [0.01, \text{dur} - 0.02, 0.01] \\
\text{enibr }&= \text{arr} \ \$ \ \text{lineSeg} \ [0, 0, 1, 1] \ [0.5, 0.5, \text{dur} - 1] \\
\text{emb }&= \text{delayt} \ (\text{mkBuf} 2 \ n) \ n \\
\text{bore }&= \text{delayt} \ (\text{mkBuf} 1 \ (n * 2)) \ (n * 2) \\
n &= \text{truncate} \ (1 / \text{fqc} / 2 \ * \text{fromIntegral sr})
\end{align*}
\]

\text{in} \ \text{proc} _- \rightarrow \ \text{do}

\[
\begin{align*}
\text{rec } & \text{tm }\leftarrow \text{timeA} \ \prec \ () \\
\text{env1 }&\leftarrow \text{en1} \ \prec \ \text{tm} \\
\text{env2 }&\leftarrow \text{en2} \ \prec \ \text{tm} \\
\text{envibr }&\leftarrow \text{enibr} \ \prec \ \text{tm} \\
\text{sin5 }&\leftarrow \text{sineA 5} \ \prec \ () \\
\text{rand }&\leftarrow \text{arr rand} \_f \ \prec \ () \\
\text{let } & \text{vibr }= \text{sin5} * \text{envibr} * 0.1 \\
\text{flow }&= \text{rand} * \text{env1} \\
\text{sum1 }&= \text{breath} * \text{flow} + \text{env1} + \text{vibr} \\
\text{flute }&\leftarrow \text{bore} \ \prec \ \text{out} \\
x &\leftarrow \text{emb} \ \prec \ \text{sum1} + \text{flute} * 0.4 \\
\text{out }&\leftarrow \text{lowpassA 0.27} \prec \ x - x * x * x + \text{flute} * 0.4 \\
\text{returnA }&\leftarrow \ \text{out} * \ \text{amp} * \ \text{env2}
\end{align*}
\]
loop \((\lambda (\_ \_ , \text{out}) \rightarrow (\_ \_ , \text{out})) \gg\)
   \((\text{first time} A \gg \lambda (\lambda (\text{tm}, \text{out}) \rightarrow (\text{tm}, (\text{out}, \text{tm}))) \gg\)
   \((\text{first en} 1 \gg \lambda (\text{env} 1, (\text{out}, \text{tm})) \rightarrow (\text{tm}, (\text{env} 1, \text{out}, \text{tm}))) \gg\)
   \((\text{first en} 2 \gg\)
   \arr (\lambda (\text{env} 2, (\text{env} 1, \text{out}, \text{tm})) \rightarrow (\text{tm}, (\text{env} 1, \text{env} 2, \text{out}))) \gg\)
   \((\text{first enibr} \gg\)
   \arr (\lambda (\text{envibr}, (\text{env} 1, \text{env} 2, \text{out})) \rightarrow ((\_ \_ , (\_ \_ , \text{out}, \text{envibr}, \text{out})))) \gg\)
   \((\text{first (sine} A 5) \gg\)
   \arr (\lambda (\text{sine} 5, (\text{env} 1, \text{env} 2, \text{envibr}, \text{out})) \rightarrow (\_ \_ , (\_ \_ , \text{env} 1, \text{env} 2, \text{envibr}, \text{out}, \text{sine} 5))) \gg\)
   \((\text{first (arr rand}_{-} f) \gg\)
   \arr (\lambda (\text{rand}, (\text{env} 1, \text{env} 2, \text{envibr}, \text{out}, \text{sine} 5)) \rightarrow \)
   \qquad \text{let} \text{vibr} = \text{sine} 5 \times \text{envibr} \times 0.1
   \qquad \text{flow} = \text{rand} \times \text{env} 1
   \qquad \text{sum} 1 = \text{breath} \times \text{flow} + \text{env} 1 + \text{vibr}
   \qquad \text{in} (\text{out}, (\text{env} 2, \text{sum} 1))) \gg\)
   \((\text{first bore} \gg\)
   \arr (\lambda (\text{flute}, (\text{env} 2, \text{sum} 1)) \rightarrow ((\_ \_ , \text{sum} 1), (\text{env} 2, \text{flute}))) \gg\)
   \((\text{first (arr (\_ \_ , \text{flute}) \rightarrow \text{sum} 1 + \text{flute} \times 0.4) \gg \text{emb}) \gg\)
   \arr (\lambda (\text{x}, (\text{env} 2, \text{flute})) \rightarrow ((\_ \_ , \text{x}), \text{env} 2))) \gg\)
   \((\text{first (arr (\_ \_ , \text{flute}) \rightarrow \_ \_ - \_ \_ \times \_ \_ \times \_ \_ + \text{flute} \times 0.4) \gg\)
   \quad \text{lowpass} A 0.27)
   \qquad \gg \arr (\lambda (\text{out}, \text{env} 2) \rightarrow ((\_ \_ , \text{out}), \text{out})))
   \gg \arr (\lambda (\text{env} 2, \text{out}) \rightarrow \_ \_ \times \text{amp} \times \text{env} 2)\)
\[
\text{fluteOpt\ dur\ amp\ fqc\ press\ breath} =
\]
\[
\text{let } env1 = \text{upSample}_f (\text{lineSeg\ am1\ du1}) 20
\]
\[
\text{env2} = \text{upSample}_f (\text{lineSeg\ am2\ du2}) 20
\]
\[
\text{env3} = \text{upSample}_f (\text{lineSeg\ am3\ du3}) 20
\]
\[
\text{omh} = 2 \times \pi / (\text{fromIntegral\ sr}) \times 5
\]
\[
c = 2 \times \cos \text{omh}
\]
\[
i = \sin \text{omh}
\]
\[
dt = 1 / \text{fromIntegral\ sr}
\]
\[
sr = \text{44100}
\]
\[
buf100 = \text{mkArr\ 100}
\]
\[
buf50 = \text{mkArr\ 50}
\]
\[
am1 = [0, 1.1 \times \text{press, press, press, 0}]
\]
\[
du1 = [0.06, 0.2, \text{dur} - 0.16, 0.02]
\]
\[
am2 = [0, 1, 1, 0]
\]
\[
du2 = [0.01, \text{dur} - 0.02, 0.01]
\]
\[
am3 = [0, 0, 1, 1]
\]
\[
du3 = [0.5, 0.5, \text{dur} - 1]
\]
\[
in\ \text{loopD}\ ((0, ((0, 0), 0), ((((buf100), 0), 0), ((0), (((buf50), 0), 0)))), (((0, i), (0, ((0, 0), 0))), ((0, ((0, 0), 0)), (0, ((0, 0), 0)))))))
\]
\[
\lambda(((\_a, \_f), \_c), \_d), \_e)\), (((\_g, \_j), (\_k, (\_m, \_n)))), (((\_i, \_q), (\_p, (\_r, \_s)))), (((\_o, (\_u, \_v)), (\_t, (\_w, \_x)))), (\_r, \_s)))))) \rightarrow
\]
\[
\text{let } \text{randf} = \text{rand}_f
\]
\[
(env1vu1, env1vu2) = env1 (\_v, \_u)
\]
\[
(env1xw1, env1xw2) = env1 (\_x, \_w)
\]
\[
(env3sr1, env3sr2) = env3 (\_s, \_r)
\]
\[
(env2ih1, env2ih2) = env2 (\_i, \_h)
\]
\[
d50nm = ((\text{delay}_f 50) (\_n, \_m))
\]
\[
d100lg = ((\text{delay}_f 100) (\_l, \_g))
\]
\[
foo = \_k + 0.27 \times (\_i) ((+(polyx) (\text{fstU\ d50nm})) \_eq)
\]
\[
bar = ((+(negate\ \_j)) ((c*\ \_q))
\]
\[
baz = ((+(+(+(breath) (*env1xw1\ randf)) \_env1vu1)) ((+((0.1) \_env3sr1)) \_bar)) + (\text{fstU\ d100lg} \times 0.4)
\]
\[
in\ (((+(\_amp\ foo)) \_env2ih1), (((\_p + \_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