Causal Commutative Arrows Revisited

Jeremy Yallop

University of Cambridge

Hai (Paul) Liu

Intel Labs

September 21, 2016

Normalization as an optimization technique?

Normalization as an optimization technique?

- Plausible, because it preserves semantics.

Normalization as an optimization technique?

- Plausible, because it preserves semantics.
- Effective, when conditions are met:

Normalization as an optimization technique?

- Plausible, because it preserves semantics.
- Effective, when conditions are met:
  - It has to *terminate*;
  - It gives *simpler* program as a result;
  - It enables *other* optimizations.

Normalization as an optimization technique?

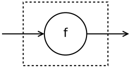
- Plausible, because it preserves semantics.
- Effective, when conditions are met:
  - It has to *terminate*;
  - It gives *simpler* program as a result;
  - It enables *other* optimizations.
- with a few catches:
  - Strongly normalizing can be too restrictive;
  - Sharing is hard to preserve;
  - Static or dynamic implementation?

Arrows

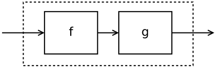
Arrows are a generalization of monads (Hughes 2000).

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

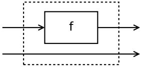
$$
\begin{array}{l}
\mathbf{class}\ \mathit{Arrow}\ \mathit{arr} \Rightarrow \mathit{ArrowLoop}\ \mathit{arr}\ \mathbf{where} \\
\quad \mathit{loop} \;\; :: \mathit{arr}\ (a, c)\ (b, c) \to \mathit{arr}\ a\ b
\end{array}
$$



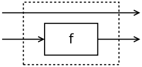
(a) *arr f*



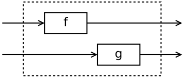
(b) $f \ggg g$



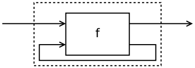
(c) *first f*



(d) *second f*



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



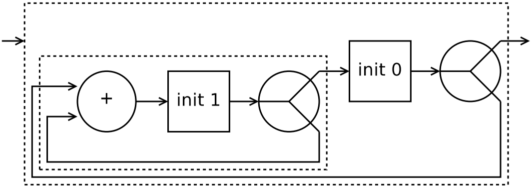
(f) *loop f*

Arrow and ArrowLoop laws

$$
\begin{array}{rcl}
arr\ id \ggg f & \equiv & f \\
f \ggg arr\ id & \equiv & f \\
(f \ggg g) \ggg h & \equiv & f \ggg (g \ggg h) \\
arr\ (g\ .\ f) & \equiv & arr\ f \ggg arr\ g \\
first\ (arr\ f) & \equiv & arr\ (f \times id) \\
first\ (f \ggg g) & \equiv & first\ f \ggg first\ g \\
first\ f \ggg arr\ (id \times g) & \equiv & arr\ (id \times g) \ggg first\ f \\
first\ f \ggg arr\ fst & \equiv & arr\ fst \ggg f \\
first\ (first\ f) \ggg arr\ assoc & \equiv & arr\ assoc \ggg first\ f
\end{array}
$$

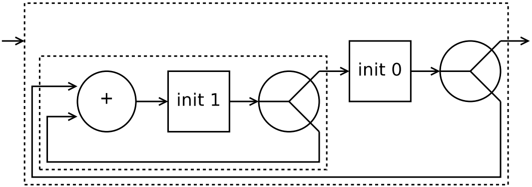
$$
\begin{array}{rcl}
loop\ (first\ h \ggg f) & \equiv & h \ggg loop\ f \\
loop\ (f \ggg first\ h) & \equiv & loop\ f \ggg h \\
loop\ (f \ggg arr\ (id \times k)) & \equiv & loop\ (arr\ (id \times k) \ggg f) \\
loop\ (loop\ f) & \equiv & loop\ (arr\ assoc^{-1}\ .\ f\ .\ arr\ assoc) \\
second\ (loop\ f) & \equiv & loop\ (arr\ assoc\ .\ second\ f\ .\ arr\ assoc^{-1}) \\
loop\ (arr\ f) & \equiv & arr\ (trace\ f)
\end{array}
$$

Normalizing arrows (a dataflow example)

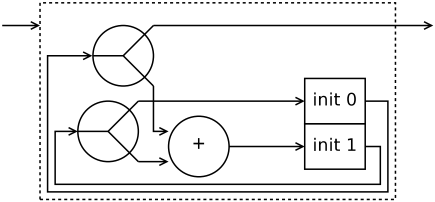


(a) original

Normalizing arrows (a dataflow example)



(a) original



(b) normalized

Causal Commutative Arrows

CCA is a more restricted arrow with an additional *init* combinator:

$$\textbf{class}\ \mathit{ArrowLoop\ arr} \Rightarrow \mathit{ArrowInit\ arr}\ \textbf{where}$$
$$\quad \mathit{init} :: a \to \mathit{arr}\ a\ a$$

and two additional arrow laws:

$$\begin{aligned}
\mathit{first}\ f \ggg \mathit{second}\ g &\equiv \mathit{second}\ g \ggg \mathit{first}\ f \\
\mathit{init}\ i \star\!\!\star\!\!\star\ \mathit{init}\ j &\equiv \mathit{init}\ (i, j)
\end{aligned}$$

Causal Commutative Arrows

CCA is a more restricted arrow with an additional *init* combinator:

$\mathbf{class}\ \mathit{ArrowLoop\ arr} \Rightarrow \mathit{ArrowInit\ arr}\ \mathbf{where}$
$\quad \mathit{init} :: a \rightarrow \mathit{arr}\ a\ a$

and two additional arrow laws:

$$\begin{aligned} \mathit{first}\ f \ggg \mathit{second}\ g &\equiv \mathit{second}\ g \ggg \mathit{first}\ f \\ \mathit{init}\ i \star\!\star\!\star\ \mathit{init}\ j &\equiv \mathit{init}\ (i, j) \end{aligned}$$

Causal Commutative Normal Form (CCNF) is either a pure arrow, or a single loop containing a pure arrow and an initial state:

$\mathit{loopD} :: \mathit{ArrowInit\ arr} \Rightarrow c \rightarrow ((a, c) \rightarrow (b, c)) \rightarrow \mathit{arr}\ a\ b$
$\mathit{loopD}\ i\ f = \mathit{loop}\ (\mathit{arr}\ f \ggg \mathit{second}\ (\mathit{init}\ i))$

Proved by algebraic arrow laws. (Liu et al. ICFP2009, JFP2010)

Application: stream transformers as arrows

$$
\begin{aligned}
&\textbf{newtype}\ \mathit{SF}\ a\ b = \mathit{SF}\ \{\mathit{unSF} :: a \to (b, \mathit{SF}\ a\ b)\}\\
&\textbf{instance}\ \mathit{Arrow}\ \mathit{SF}\ \textbf{where}\\
&\quad \mathit{arr}\ f \quad = g\ \textbf{where}\ g = \mathit{SF}\ (\lambda x \to (f\ x, g))\\
&\quad f \ggg g = \ldots\\
&\quad \mathit{first}\ f \ = \ldots\\
&\textbf{instance}\ \mathit{ArrowLoop}\ \mathit{SF}\ \textbf{where}\ \ldots\\
&\textbf{instance}\ \mathit{ArrowInit}\quad \mathit{SF}\ \textbf{where}\ \ldots
\end{aligned}
$$

Application: stream transformers as arrows

$$\textbf{newtype}\ \mathit{SF}\ a\ b = \mathit{SF}\ \{\mathit{unSF} :: a \to (b, \mathit{SF}\ a\ b)\}$$

$$\begin{array}{l}
\textbf{instance}\ \mathit{Arrow}\ \mathit{SF}\ \textbf{where} \\
\quad \mathit{arr}\ f \quad = g\ \textbf{where}\ g = \mathit{SF}\ (\lambda x \to (f\ x, g)) \\
\quad f \ggg g = \ldots \\
\quad \mathit{first}\ f \quad = \ldots
\end{array}$$

$$\begin{array}{l}
\textbf{instance}\ \mathit{ArrowLoop}\ \mathit{SF}\ \textbf{where}\ \ldots \\
\textbf{instance}\ \mathit{ArrowInit}\quad \mathit{SF}\ \textbf{where}\ \ldots
\end{array}$$

We can run a stream transformer over an input stream:

$$\begin{array}{l}
\mathit{run}_{\mathit{SF}} :: \mathit{SF}\ a\ b \to [a] \to [b] \\
\mathit{run}_{\mathit{SF}}\ (\mathit{SF}\ f)\ (x : \mathit{xs}) = \textbf{let}\ (y, f') = f\ x\ \textbf{in}\ y : \mathit{run}_{\mathit{SF}}\ f'\ \mathit{xs} \\
\mathit{nth}_{\mathit{SF}} :: \mathit{Int} \to \mathit{SF}\ ()\ a \to a \\
\mathit{nth}_{\mathit{SF}}\ n\ \mathit{sf} = \mathit{run}_{\mathit{SF}}\ \mathit{sf}\ (\mathit{repeat}\ ())\ !!\ n
\end{array}$$

Performance Comparison

Orders of magnitude speedup (JFP2010):

Name	SF	$CCNF_{sf}$	$CCNF_{tuple}$
exp	1.0	30.84	672.79
sine	1.0	18.89	442.48
oscSine	1.0	14.28	29.53
50's sci-fi	1.0	18.72	21.37
robotSim	1.0	24.67	34.93

Table : Performance Ratio (greater is better)

Normalization of CCA programs seems very effective!

Performance Comparison

Orders of magnitude speedup (JFP2010):

Name	*SF*	$CCNF_{sf}$	$CCNF_{tuple}$
exp	1.0	30.84	672.79
sine	1.0	18.89	442.48
oscSine	1.0	14.28	29.53
50's sci-fi	1.0	18.72	21.37
robotSim	1.0	24.67	34.93

Table : Performance Ratio (greater is better)

Normalization of CCA programs seems very effective!
But why is everyone not using it??

Performance Comparison

Orders of magnitude speedup (JFP2010):

Name	*SF*	$CCNF_{sf}$	$CCNF_{tuple}$
exp	1.0	30.84	672.79
sine	1.0	18.89	442.48
oscSine	1.0	14.28	29.53
50's sci-fi	1.0	18.72	21.37
robotSim	1.0	24.67	34.93

Table : Performance Ratio (greater is better)

Normalization of CCA programs seems very effective!
But why is everyone not using it??
Not even used by *Euterpea*, the music and sound synthesis framework from the same research group!

Pitfalls of the CCA implementation

The initial CCA library was implemented using Template Haskell, because:

- Normalization is a syntactic transformation;
- Meta-level implementation guarantees normal form at compile time;
- TH is less work than a full-blown pre-processor.

Pitfalls of the CCA implementation

The initial CCA library was implemented using Template Haskell, because:

- Normalization is a syntactic transformation;
- Meta-level implementation guarantees normal form at compile time;
- TH is less work than a full-blown pre-processor.

However, TH based static normalization is:

- restricted to first-order, no reactivity, etc.

Pitfalls of the CCA implementation

The initial CCA library was implemented using Template Haskell, because:

- Normalization is a syntactic transformation;
- Meta-level implementation guarantees normal form at compile time;
- TH is less work than a full-blown pre-processor.

However, TH based static normalization is:

- restricted to first-order, no reactivity, etc.
- hard to program with:

$$f\ x = \ldots[|\ \ldots x \ldots\ |]\ \ldots$$
$$\ldots\ \$(norm\ g)\ \ldots$$

Pitfalls of the CCA implementation

The initial CCA library was implemented using Template Haskell, because:

- Normalization is a syntactic transformation;
- Meta-level implementation guarantees normal form at compile time;
- TH is less work than a full-blown pre-processor.

However, TH based static normalization is:

- restricted to first-order, no reactivity, etc.
- hard to program with:

  $f\ x = \ldots[|\ \ldots x\ldots\ |]\ \ldots$
  $\ldots\ \$(norm\ g)\ \ldots$

- perhaps not as effective as we had thought for "real" applications?

How about run-time normalization?

How about run-time normalization?

```
from:  Paul Liu
to:  Jeremy Yallop
cc:  Paul Hudak, Eric Cheng
date:  18 June 2009

I wonder if there is any way to optimize GHC's
output based on your code since the CCNF is
actually running slower
```

How about run-time normalization?

```
from:  Paul Liu
to:  Jeremy Yallop
cc:  Paul Hudak, Eric Cheng
date:  18 June 2009

I wonder if there is any way to optimize GHC's
output based on your code since the CCNF is
actually running slower
```

"*...that the actual construction of CCNF is now at run-time rather than compile-time. Therefore, we cannot rely on GHC to take the pure function and state captured in a CCNF and produce optimized code...*" (Liu 2011)

Normalization by construction

1. Define normal form as a data type:

$$
\begin{array}{l}
\mathbf{data}\ \mathit{CCNF}\ a\ b\ \mathbf{where} \\
\quad \mathit{Arr} \quad\ \, :: (a \rightarrow b) \rightarrow \mathit{CCNF}\ a\ b \\
\quad \mathit{LoopD} :: c \rightarrow ((a, c) \rightarrow (b, c)) \rightarrow \mathit{CCNF}\ a\ b
\end{array}
$$

Normalization by construction

1. Define normal form as a data type:

   $$
   \begin{array}{l}
   \textbf{data}\ \mathit{CCNF}\ a\ b\ \textbf{where} \\
   \quad \mathit{Arr} \quad\ :: (a \to b) \to \mathit{CCNF}\ a\ b \\
   \quad \mathit{LoopD} :: c \to ((a, c) \to (b, c)) \to \mathit{CCNF}\ a\ b
   \end{array}
   $$

2. Observation function:

   $$
   \begin{array}{l}
   \mathit{observe} :: \mathit{ArrowInit}\ \mathit{arr} \Rightarrow \mathit{CCNF}\ a\ b \to \mathit{arr}\ a\ b \\
   \mathit{observe}\ (\mathit{Arr}\ f) = \mathit{arr}\ f \\
   \mathit{observe}\ (\mathit{LoopD}\ i\ f) = \mathit{loop}\ (\mathit{arr}\ f \ggg \mathit{second}\ (\mathit{init}\ i))
   \end{array}
   $$

Normalization by construction

1. Define normal form as a data type:

   $$\begin{array}{l} \textbf{data}\ \mathit{CCNF}\ a\ b\ \textbf{where} \\ \quad \mathit{Arr}\quad\ \ :: (a \to b) \to \mathit{CCNF}\ a\ b \\ \quad \mathit{LoopD} :: c \to ((a, c) \to (b, c)) \to \mathit{CCNF}\ a\ b \end{array}$$

2. Observation function:

   $$\begin{array}{l} \mathit{observe} :: \mathit{ArrowInit}\ \mathit{arr} \Rightarrow \mathit{CCNF}\ a\ b \to \mathit{arr}\ a\ b \\ \mathit{observe}\ (\mathit{Arr}\ f) = \mathit{arr}\ f \\ \mathit{observe}\ (\mathit{LoopD}\ i\ f) = \mathit{loop}\ (\mathit{arr}\ f \ggg \mathit{second}\ (\mathit{init}\ i)) \end{array}$$

3. Instances for the data type:

   $$\begin{array}{lll} \textbf{instance}\ \mathit{Arrow} & \mathit{CCNF}\ \textbf{where} \ldots \\ \textbf{instance}\ \mathit{ArrowLoop} & \mathit{CCNF}\ \textbf{where} \ldots \\ \textbf{instance}\ \mathit{ArrowInit} & \mathit{CCNF}\ \textbf{where} \ldots \end{array}$$

Optimize the observe function

1. Specialize *observe* to a concrete instance.

$$
\begin{aligned}
&\mathit{observe} :: \mathit{ArrowInit}\ \mathit{arr} \Rightarrow \mathit{CCNF}\ a\ b \rightarrow \mathit{arr}\ a\ b \\
&\mathit{observe}_{SF} :: \mathit{CCNF}\ a\ b \rightarrow \mathit{SF}\ a\ b \\
&\mathit{observe}_{SF}\ (\mathit{Arr}\ f) = \mathit{arr}_{SF}\ f \\
&\mathit{observe}_{SF}\ (\mathit{LoopD}\ i\ f) = \mathit{loop}_{SF}\ (\mathit{arr}_{SF}\ f \ggg_{SF} \mathit{second}_{SF}\ (\mathit{init}_{SF}\ i))
\end{aligned}
$$

Optimize the observe function

1. Specialize *observe* to a concrete instance.

$$
\begin{aligned}
&\mathit{observe} :: \mathit{ArrowInit}\ \mathit{arr} \Rightarrow \mathit{CCNF}\ a\ b \rightarrow \mathit{arr}\ a\ b \\
&\mathit{observe}_{SF} :: \mathit{CCNF}\ a\ b \rightarrow \mathit{SF}\ a\ b \\
&\mathit{observe}_{SF}\ (\mathit{Arr}\ f) = \mathit{arr}_{SF}\ f \\
&\mathit{observe}_{SF}\ (\mathit{LoopD}\ i\ f) = \mathit{loop}_{SF}\ (\mathit{arr}_{SF}\ f \ggg_{SF} \mathit{second}_{SF}\ (\mathit{init}_{SF}\ i))
\end{aligned}
$$

2. Derive an optimized definition.

$$
\begin{aligned}
&\mathit{observe}_{SF}\ (\mathit{LoopD}\ i\ f) = \mathit{loopD}\ i\ f \\
&\quad \textbf{where} \\
&\qquad \mathit{loopD} :: c \rightarrow ((a, c) \rightarrow (b, c)) \rightarrow \mathit{SF}\ a\ b \\
&\qquad \mathit{loopD}\ i\ f = \mathit{SF}\ (\lambda x \rightarrow \textbf{let}\ (y, i') = f\ (x, i)\ \textbf{in}\ (y, \mathit{loopD}\ i'\ f))
\end{aligned}
$$

Optimize the observe function

1. Specialize *observe* to a concrete instance.

$$
\begin{aligned}
&observe :: ArrowInit\ arr \Rightarrow CCNF\ a\ b \rightarrow arr\ a\ b \\
&observe_{SF} :: CCNF\ a\ b \rightarrow SF\ a\ b \\
&observe_{SF}\ (Arr\ f) = arr_{SF}\ f \\
&observe_{SF}\ (LoopD\ i\ f) = loop_{SF}\ (arr_{SF}\ f \ggg_{SF} second_{SF}\ (init_{SF}\ i))
\end{aligned}
$$

2. Derive an optimized definition.

$$
\begin{aligned}
&observe_{SF}\ (LoopD\ i\ f) = loopD\ i\ f \\
&\quad \textbf{where} \\
&\qquad loopD :: c \rightarrow ((a, c) \rightarrow (b, c)) \rightarrow SF\ a\ b \\
&\qquad loopD\ i\ f = SF\ (\lambda x \rightarrow \textbf{let}\ (y, i') = f\ (x, i)\ \textbf{in}\ (y, loopD\ i'\ f))
\end{aligned}
$$

3. Fuse *observe* with the context in which it is used.

$$
\begin{aligned}
&nth_{CCNF} :: Int \rightarrow CCNF\ ()\ a \rightarrow a \\
&nth_{CCNF}\ n = nth_{SF}\ n\ .\ observe_{SF} = \ldots
\end{aligned}
$$

Performance comparison

- Compute $44100 \times 5 = 2,205,000$ samples ($\approx$ 5 seconds of audio)
- GHC 7.10.3 using the flags `-O2 -funfolding-use-limit=512`
- 64-bit Linux, Intel Xeon CPU E5-2680 2.70GHz

Benchmark			**Unnormalized**	**Normalized**	
Name	States	Loops	*SF*	*CCNF*	TH
fib	2	1	1.0	2.29	2.30
exp	1	2	1.0	242	242
sine	2	1	1.0	124	146
oscSine	1	1	1.0	60.6	60.6
sci-fi	3	3	1.0	27.7	27.4
robot	5	4	1.0	104	96.7
flute	16	7	1.0	5.10	16.2
shepard	80	30	1.0	7.47	12.9
			baseline	speedup ratio	

Why it works

- Unlike SF, $CCNF$ is not recursively defined.

$$
\begin{aligned}
&\textbf{data}\ SF\ a\ b\ \textbf{where}\\
&\quad SF :: a \to (b, SF\ a\ b) \to SF\ a\ b\\
&\textbf{data}\ CCNF\ a\ b\ \textbf{where}\\
&\quad Arr \quad\ \ :: (a \to b) \to CCNF\ a\ b\\
&\quad LoopD :: c \to ((a, c) \to (b, c)) \to CCNF\ a\ b
\end{aligned}
$$

Why it works

- Unlike SF, $CCNF$ is not recursively defined.

$$
\begin{array}{l}
\textbf{data}\ SF\ a\ b\ \textbf{where} \\
\quad SF :: a \to (b, SF\ a\ b) \to SF\ a\ b \\
\textbf{data}\ CCNF\ a\ b\ \textbf{where} \\
\quad Arr\ \ \ \ \ :: (a \to b) \to CCNF\ a\ b \\
\quad LoopD :: c \to ((a, c) \to (b, c)) \to CCNF\ a\ b
\end{array}
$$

- Hand optimized observe function is the key to get performance.

$$
\begin{array}{l}
nth_{CCNF} :: Int \to CCNF\ ()\ a \to a \\
nth_{CCNF}\ n = nth_{SF}\ n\ .\ observe_{SF} = \ldots
\end{array}
$$

Why it works

- Unlike SF, $CCNF$ is not recursively defined.

  $$\begin{array}{l}\textbf{data}\ SF\ a\ b\ \textbf{where} \\ \quad SF :: a \to (b, SF\ a\ b) \to SF\ a\ b\end{array}$$

  $$\begin{array}{l}\textbf{data}\ CCNF\ a\ b\ \textbf{where} \\ \quad Arr \quad\ :: (a \to b) \to CCNF\ a\ b \\ \quad LoopD :: c \to ((a, c) \to (b, c)) \to CCNF\ a\ b\end{array}$$

- Hand optimized observe function is the key to get performance.

  $$\begin{array}{l} nth_{CCNF} :: Int \to CCNF\ ()\ a \to a \\ nth_{CCNF}\ n = nth_{SF}\ n\ .\ observe_{SF} = ... \end{array}$$

- GHC has improved! GHC 6.10 fails to optimize our program.

Why it works

- Unlike SF, $CCNF$ is not recursively defined.

  $$\begin{aligned}&\textbf{data}\ SF\ a\ b\ \textbf{where}\\&\quad SF :: a \to (b, SF\ a\ b) \to SF\ a\ b\\&\textbf{data}\ CCNF\ a\ b\ \textbf{where}\\&\quad Arr\ \ \ \ \ :: (a \to b) \to CCNF\ a\ b\\&\quad LoopD :: c \to ((a, c) \to (b, c)) \to CCNF\ a\ b\end{aligned}$$

- Hand optimized observe function is the key to get performance.

  $$\begin{aligned}&nth_{CCNF} :: Int \to CCNF\ ()\ a \to a\\&nth_{CCNF}\ n = nth_{SF}\ n\ .\ observe_{SF} = \ldots\end{aligned}$$

- GHC has improved! GHC 6.10 fails to optimize our program.

Compilers help those who help compilers!

Levels of abstraction

Axiomatic . . . Type class (Arrow laws)

Levels of abstraction

Axiomatic	...	Type class (Arrow laws)
$\downarrow$		$\downarrow$
Denotational	...	Data type (Interpretation)

Levels of abstraction

Axiomatic	...	Type class (Arrow laws)
$\downarrow$		$\downarrow$
Denotational	...	Data type (Interpretation)
$\downarrow$		$\downarrow$
Operational	...	Mealy machine (state and transition)

Levels of abstraction

$$\begin{array}{ccc} \text{Axiomatic} & \ldots & \text{Type class (Arrow laws)} \\ \downarrow & & \downarrow \\ \text{Denotational} & \ldots & \text{Data type (Interpretation)} \\ \downarrow & & \downarrow \\ \text{Operational} & \ldots & \text{Mealy machine (state and transition)} \end{array}$$

$$\begin{array}{l} \mathit{nth}_{CCNF}\ n\ (\mathit{LoopD}\ i\ f) = \mathit{next}\ n\ i \\ \quad \textbf{where} \\ \qquad \mathit{next}\ n\ i = \textbf{if}\ n \equiv 0\ \textbf{then}\ x\ \textbf{else}\ \mathit{next}\ (n-1)\ i' \\ \qquad\quad \textbf{where}\ (x, i') = f\ ((), i) \end{array}$$

That is not all (performance we could have)

- CCA normalization clusters all states as one nested tuple.

$$
\begin{aligned}
&\mathit{LoopD}\ ((0,((0,0),0)),\\
&\qquad(((((\mathit{buf100}),0),0),((0),(((\mathit{buf50}),0),0))),\\
&\qquad\quad(((0,i),(0,((0,0),0))),((0,((0,0),0)),(0,((0,0),0))))))\\
&\quad(\lambda(((((a,f),e),d),c),\ldots)\rightarrow\ldots)
\end{aligned}
$$

That is not all (performance we could have)

- CCA normalization clusters all states as one nested tuple.

$$
\begin{array}{l}
\mathit{LoopD}\ ((0,((0,0),0)),\\
\qquad\qquad (((((\mathit{buf100}),0),0),((0),(((\mathit{buf50}),0),0))),\\
\qquad\qquad\quad (((0,i),(0,((0,0),0))),((0,((0,0),0)),(0,((0,0),0))))))\\
\quad (\lambda(((((a,f),e),d),c),\ldots) \rightarrow \ldots)
\end{array}
$$

- Transition function destructs/constructs tuples at every iteration!

$$
\begin{array}{l}
\mathit{next}\ n\ i = \mathbf{if}\ n \equiv 0\ \mathbf{then}\ x\ \mathbf{else}\ \mathit{next}\ (n-1)\ i'\\
\quad \mathbf{where}\ (x,i') = f\ ((),i)
\end{array}
$$

That is not all (performance we could have)

- CCA normalization clusters all states as one nested tuple.

$$
\begin{aligned}
&\mathit{LoopD}\ ((0,((0,0),0)),\\
&\qquad\quad (((((\mathit{buf100}),0),0),((0),(((\mathit{buf50}),0),0))),\\
&\qquad\qquad (((0,i),(0,((0,0),0))),((0,((0,0),0)),(0,((0,0),0))))))\\
&\quad (\lambda(((((a,f),e),d),c),\ldots) \to \ldots)
\end{aligned}
$$

- Transition function destructs/constructs tuples at every iteration!

$$
\begin{aligned}
&\mathit{next}\ n\ i = \textbf{if}\ n \equiv 0\ \textbf{then}\ x\ \textbf{else}\ \mathit{next}\ (n-1)\ i'\\
&\quad \textbf{where}\ (x,i') = f\ ((),i)
\end{aligned}
$$

- GHC can only help us so far.

That is not all (performance we could have)

- CCA normalization clusters all states as one nested tuple.

$$\begin{array}{l} \mathit{LoopD}\ ((0,((0,0),0)), \\ \qquad\quad (((((\mathit{buf100}),0),0),((0),(((\mathit{buf50}),0),0))), \\ \qquad\qquad (((0,i),(0,((0,0),0))),((0,((0,0),0)),(0,((0,0),0)))))) \\ \quad (\lambda(((((a,f),e),d),c),...) \to ...) \end{array}$$

- Transition function destructs/constructs tuples at every iteration!

$$\begin{array}{l} \mathit{next}\ n\ i = \textbf{if}\ n \equiv 0\ \textbf{then}\ x\ \textbf{else}\ \mathit{next}\ (n-1)\ i' \\ \quad \textbf{where}\ (x,i') = f\ ((),i) \end{array}$$

- GHC can only help us so far.
- Real applications demand mutable states (for arrays and so on).

Local mutable state via ST Monad

ST Monad in Haskell:

$$
\begin{aligned}
&\mathbf{data}\ ST\ s\ a = \ldots \\
&runST :: (forall\ s\ .\ ST\ s\ a) \rightarrow a \\
&fixST\ \ :: (a \rightarrow ST\ s\ a) \rightarrow a
\end{aligned}
$$

Local mutable state via ST Monad

ST Monad in Haskell:

$$\begin{aligned}
&\textbf{data}\ ST\ s\ a = \ldots \\
&runST :: (forall\ s\ .\ ST\ s\ a) \rightarrow a \\
&fixST\ \ :: (a \rightarrow ST\ s\ a) \rightarrow a
\end{aligned}$$

Use ST type as our state:

$$\begin{aligned}
&\textbf{data}\ CCNF_{ST}\ s\ a\ b\ \textbf{where} \\
&\quad Arr_{ST}\quad\ \ :: (a \rightarrow b) \rightarrow CCNF_{ST}\ s\ a\ b \\
&\quad LoopD_{ST} :: ST\ s\ c \rightarrow (c \rightarrow a \rightarrow ST\ s\ b) \rightarrow CCNF_{ST}\ s\ a\ b
\end{aligned}$$

Local mutable state via ST Monad

ST Monad in Haskell:

$$\begin{aligned}
&\mathbf{data}\ ST\ s\ a = \ldots \\
&runST :: (forall\ s\ .\ ST\ s\ a) \rightarrow a \\
&fixST\ \ :: (a \rightarrow ST\ s\ a) \rightarrow a
\end{aligned}$$

Use ST type as our state:

$$\begin{aligned}
&\mathbf{data}\ CCNF_{ST}\ s\ a\ b\ \mathbf{where} \\
&\quad Arr_{ST}\ \ \ \ \ :: (a \rightarrow b) \rightarrow CCNF_{ST}\ s\ a\ b \\
&\quad LoopD_{ST} :: ST\ s\ c \rightarrow (c \rightarrow a \rightarrow ST\ s\ b) \rightarrow CCNF_{ST}\ s\ a\ b
\end{aligned}$$

The fused observe function:

$$\begin{aligned}
&nth'_{ST} :: Int \rightarrow CCNF_{ST}\ s\ ()\ a \rightarrow ST\ s\ a \\
&nth'_{ST}\ n\ (LoopD_{ST}\ i\ f) = \mathbf{do} \\
&\quad g \leftarrow fmap\ f\ i \\
&\quad \mathbf{let}\ next\ n = \mathbf{do}\ x \leftarrow g\ () \\
&\qquad\qquad\qquad\qquad \mathbf{if}\ n \leqslant 0\ \mathbf{then}\ return\ x\ \mathbf{else}\ next\ (n-1) \\
&\quad next\ n
\end{aligned}$$

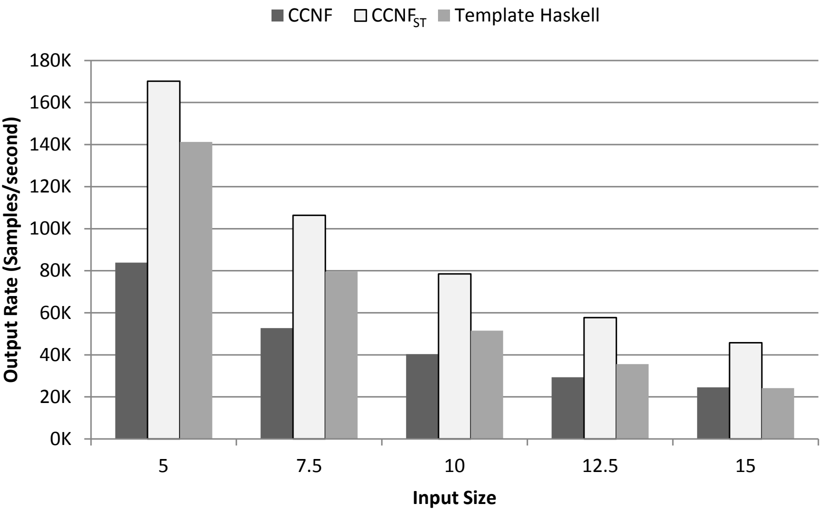
A (recursively defined) sound synthesis example

$$
\begin{array}{l}
\mathit{shepard} :: \mathit{BufferedCircuit}\ a \Rightarrow \mathit{Time} \rightarrow a\ ()\ \mathit{Double} \\
\mathit{shepard}\ \mathit{seconds} = \mathbf{if}\ \mathit{seconds} \leqslant 0.0 \\
\quad \mathbf{then}\ \mathit{arr}\ (\mathit{const}\ 0.0) \\
\quad \mathbf{else}\ \mathit{proc}\ \_ \rightarrow \mathbf{do} \\
\qquad f \leftarrow \mathit{envLineSeg}\ [800, 100, 100]\ [4.0, \mathit{seconds}] \prec ()\\
\qquad e \leftarrow \mathit{envLineSeg}\ [0, 1, 0, 0]\ [2.0, 2.0, \mathit{seconds}] \prec ()\\
\qquad s \leftarrow \mathit{osc}\ \mathit{sineTable}\ 0 \prec f \\
\qquad r \leftarrow \mathit{delayLine}\ 0.5 \lll \mathit{shepard}\ (\mathit{seconds} - 0.5) \prec () \\
\qquad \mathit{returnA} \prec (e * s * 0.1) + r
\end{array}
$$

Challenges of optimizing a recursively defined arrow:

- Static normalization blows up code size.
- Nested states builds up quickly and deeply.

Shepard performance (higher is better)



That is still not all (performance we would like to have)

- The definition of *loop* requires recursive monad:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{ArrowLoop}\ (\mathit{CCNF}_{ST}\ s)\ \textbf{where} \\
\quad \mathit{loop}\ (\mathit{LoopD}_{ST}\ i\ f) = \mathit{LoopD}_{ST}\ i\ h \\
\qquad \textbf{where}\ h\ i\ x = \textbf{do} \\
\qquad\qquad\qquad \textbf{rec}\ (y, j) \leftarrow f\ i\ (x, j) \\
\qquad\qquad\qquad \mathit{return}\ y
\end{array}
$$

- Although in the end all loops are de-coupled, the overhead of ST type remains in compiled code.

$$
\begin{array}{l}
\mathit{fixST} :: (a \rightarrow ST\ s\ a) \rightarrow ST\ s\ a \\
\mathit{fixST}\ k = ST\ \$\ \lambda s \rightarrow \\
\quad \textbf{let}\ \mathit{ans} \qquad\quad = \mathit{liftST}\ (k\ r)\ s \\
\qquad\ \ \mathit{STret}\ \_\ r = \mathit{ans} \\
\quad \textbf{in case}\ \mathit{ans}\ \textbf{of}\ \mathit{STret}\ s'\ x \rightarrow (\#\ s', x\ \#)
\end{array}
$$

Related work

- Representing arrow computation as data (Hughes 2005, Nilsson 2005, Yallop 2010)
- Generalized arrows (Joseph 2014)
- Deriving implementation by equational reasoning (Birds 1988, Hinze 2000)
- Free representation used in optimization (Voigtländer 2008, Kiselyov and Ishii 2015)

More in the paper

- Normalization by construction in steps.
- Equational derivation of *observe* function.
- Embedding mutable states with ST monad.
- Proving $CCNF_{ST}$ is an instance of CCA.
- Detailed performance analysis.

More in the paper

- Normalization by construction in steps.
- Equational derivation of *observe* function.
- Embedding mutable states with ST monad.
- Proving $CCNF_{ST}$ is an instance of CCA.
- Detailed performance analysis.

https://github.com/yallop/causal-commutative-arrows-revisited

Thank you!