Causal Commutative Arrows and Their Optimization

Hai Liu Eric Cheng Paul Hudak
Department of Computer Science
Yale University
{hai.liu,eric.cheng,paul.hudak}@yale.edu

Abstract

Arrows are a popular form of abstract computation. Being more general than monads, they are more broadly applicable, and in particular are a good abstraction for signal processing and dataflow computations. Most notably, arrows form the basis for a domain specific language called *Yampa*, which has been used in a variety of concrete applications, including animation, robotics, sound synthesis, control systems, and graphical user interfaces.

Our primary interest is in better understanding the class of abstract computations captured by Yampa. Unfortunately, arrows are not concrete enough to do this with precision. To remedy this situation we introduce the concept of *commutative arrows* that capture a kind of non-interference property of concurrent computations. We also add an *init* operator, and identify a crucial law that captures the causal nature of arrow effects. We call the resulting computational model *causal commutative arrows*.

To study this class of computations in more detail, we define an extension to the simply typed lambda calculus called *causal commutative arrows* (CCA), and study its properties. Our key contribution is the identification of a normal form for CCA called *causal commutative normal form* (CCNF). By defining a normal-

ization procedure we have developed an optimization strategy that yields dramatic improvements in performance over conventional implementations of arrows. We have implemented this technique in Haskell, and conducted benchmarks that validate the effectiveness of our approach. When combined with stream fusion, the overall methodology can result in speed-ups of greater than two orders of magnitude.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages, Performance, Theory

Keywords Functional Programming, Arrows, Functional Reactive Programming, Dataflow Language, Stream Processing, Program Optimization

ICFP'09, August 31–September 2, 2009, Edinburgh, Scotland, UK.

1. Introduction

Consider the following recursive mathematical definition of the exponential function:

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

In Yampa [35, 21], a domain-specific language embedded in Haskell [36], we can write this using arrow syntax [32] as follows:

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

Even for those not familiar with arrow syntax or Haskell, the close correspondence between the mathematics and the Yampa program should be clear. As in most high-level language designs, this is the primary motivation for developing a language such as Yampa: reducing the gap between program and specification.

Yampa has been used in a variety of applications, including robotics [21, 34, 33], sound synthesis [15, 6], animation [35, 21], video games [11, 7], bio-chemical processes [22], control systems [31], and graphical user interfaces [10, 9]. There are several reasons that we prefer a language design based on arrows over, for example, an approach such as that used in Fran [13]. First, arrows are more *modular* – they convey information about input as well as output, whereas Fran's inputs are implicit and global. Second, the use of arrows eliminates a subtle but devastating form of *space leak*, as described in [27]. Third, arrows introduce a meta-level of compu-

tation that aids in reasoning about program correctness, transformation, and optimization.

But in fact, conventional arrows (or to borrow a phrase from [26], "classic arrows") are not strong enough to capture the family of computations that we are interested in – more laws are needed to constrain the computation space. Unfortunately, more constrained forms of computation – such as monads [29] and applicative functors [28] – are not general enough. In addition, there are not enough operators. In particular, we find the need for an abstract *initialization* operator and its associated laws.

In this paper we give a precise abstract characterization of a class of arrow computations that we call *causal commutative arrows*, or just CCA for short. More precisely, the contributions in this paper can be summarized as follows:

1. We define a notion of *commutative arrow* by extending the conventional set of arrow laws to include a commutativity law.
2. We define an `ArrowInit` type class with an *init* operator and an associated law that captures the essence of causal computation.
3. We define a small language called *CCA*, an extension of the simply typed lambda calculus, in which the above ideas are manifest. For this language we establish:
   (a) a *normal form*, and
   (b) a *normalization procedure*.

   We achieve this result using only CCA laws, without referring to any concrete semantics or implementation.
4. We define an *optimization technique* for causal commutative arrows that yields substantial improvements in performance over

previous attempts to optimize arrow combinators and arrow syntax.

5. Finally, we show how to combine our ideas with those of *stream fusion* to yield speed-ups that can exceed two orders of magnitude.

We begin the presentation with a brief overview of arrows in Section 2. The knowledgeable reader may prefer to skip directly to Section 3, where we give the definition and laws for CCA. In Section 4 we define an extension of the simply-typed lambda calculus that captures CCA, and show in Section 5 that any CCA program can be transformed into a uniform representation that we call *Causal Commutative Normal Form* (CCNF). We show that the normalization procedure is sound, based on equational reasoning using only the CCA laws. In Section 6 we discuss further optimizations, and in Section 7 we present benchmarks showing the effectiveness of our approach. We conclude in Section 8 with a discussion of related work.

2. An Introduction to Arrows

Arrows [23] are a generalization of monads that relax the stringent linearity imposed by monads, while retaining a disciplined style of composition. Arrows have enjoyed a wide range of applications, often as a domain-specific embedded language (DSEL [19, 20]), including the many Yampa applications cited earlier, as well as parsers and printers [25], parallel computing [18], and so on. Arrows also have a theoretical foundation in category theory, where they are strongly related to (but not precisely the same as) *Freyd categories* [2, 37].

2.1 Conventional Arrows

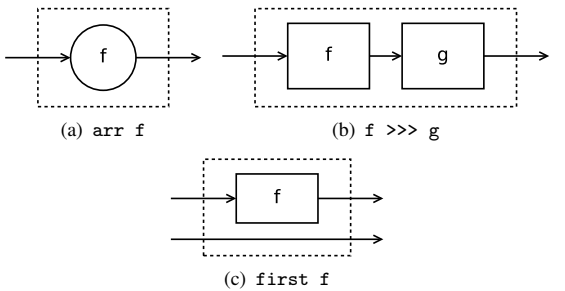
Like monads, arrows capture a certain class of abstract computations, and offer a way to structure programs. In Haskell this is achieved through the `Arrow` type class:

```
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)
```

The combinator `arr` lifts a function from `b` to `c` to a "pure" arrow computation from `b` to `c`, namely `a b c` where `a` is the arrow type. The output of a pure arrow entirely depends on the input (it is analogous to `return` in the `Monad` class). `>>>` composes two arrow computations by connecting the output of the first to the input of the second (and is analogous to bind (`(>>=)`) in the `Monad` class). But in addition to composing arrows linearly, it is desirable to compose them in parallel – i.e. to allow "branching" and "merging" of inputs and outputs. There are several ways to do this, but by simply defining the `first` combinator in the `Arrow` class, all other combinators can be defined. `first` converts an arrow computation taking one input and one result, into an arrow computation taking two inputs and two results. The original arrow is applied to the first part of the input, and the result becomes the first part of the output. The second part of the input is fed directly to the second part of the output.

Other combinators can be defined using these three primitives. For example, the dual of `first` can be defined as:

```
arr     :: Arrow a ⇒ (b → c) → a b c
(>>>)   :: Arrow a ⇒ a b c → a c d → a b d
first   :: Arrow a ⇒ a b c → a (b, d) (c, d)
(***)   :: Arrow a ⇒ a b c → a b' c' →
                      a (b, b') (c, c')
loop    :: Arrow a ⇒ a (b,d) (c,d) → a b c
```



(a) `arr f` (b) `f >>> g`

(c) `first f`

(d) `f *** g` (e) `loop f`

Figure 1. Commonly Used Arrow Combinators

```
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)
```

Parallel composition can be defined as a sequence of first and second:

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

A mere implementation of the arrow combinators, of course, does not make it an arrow – the implementation must additionally satisfy a set of *arrow laws*, which are shown in Figure 2.

2.2 Looping Arrows

To model recursion, we can introduce a `loop` combinator [32]. The exponential example given in the introduction requires recursion, as do many applications in signal processing, for example. In Haskell this combinator is captured in the `ArrowLoop` class:

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

A valid instance of this class should satisfy the additional laws shown in Figure 3. This class and its associated laws are related to the trace operator in [40, 17], which was generalized to arrows in [32].

We find that arrows are best viewed pictorially, especially for applications such as signal processing, where domain experts commonly draw signal flow diagrams. Figure 1 shows some of the basic combinators in this manner, including `loop`.

2.3 Arrow Syntax

Recall the Yampa definition of the exponential function given earlier:

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

This program is written using *arrow syntax*, introduced by Paterson

[32] and adopted by GHC (the predominant Haskell implementation) because it ameliorates the cumbersome nature of writing in

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

Figure 2. Conventional Arrow Laws

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

Figure 3. Arrow Loop Laws

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

Figure 4. Causal Commutative Arrow Laws

the point-free style demanded by arrows. The above program is equivalent to the following sugar-free program:

```
exp = fixA (integral >>> arr (+1))
  where fixA f = loop (second f >>>
                       arr (λ (_, y) → (y, y)))
```

Although more cumbersome, we will use this program style in the remainder of the paper, in order to avoid having to explain the meaning of arrow syntax in more detail.

3. Causal Commutative Arrows

In this section we introduce two key extensions to conventional arrows, and demonstrate their use by implementing a stream transformer in Haskell.

First, as mentioned in the introduction, the set of arrow and arrow loop laws is not strong enough to capture stream computations. In particular, the *commutativity law* shown in Figure 4 establishes a non-interference property for concurrent computations – effects are still allowed, but this law guarantees that concurrent effects cannot interfere with each other. We say that an arrow is *commutative* if it satisfies the conventional laws as well as this critical additional law. Yampa is in fact based on commutative arrows.

Second, we note that Yampa has a primitive operator called `iPre` that is used to inject a delay into a computation; indeed it is the primary effect imposed by the Yampa arrow [35, 21]. Similar operators, often called `delay`, also appear in dataflow programming [43], stream processing [39, 41], and synchronous languages [4, 8]. In all cases, the operator introduces stateful computation into an otherwise stateless setting.

In an effort to make this operation more abstract, we rename it `init` and capture it in the following type class:

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

Intuitively, the argument to `init` is the initial output; subsequent output is a copy of the input to the arrow. It captures the essence

```
newtype SF a b = SF { unSF :: a → (b, SF a b) }

instance Arrow SF where
  arr f = SF h
    where h x = (f x, SF h)
  first f = SF (h f)
```

```
    where h f (x, z) = let (y, f') = unSF f x
                       in ((y, z), SF (h f'))
  f >>> g = SF (h f g)
   where h f g x = let (y, f') = unSF f x
                       (z, g') = unSF g y
                   in (z, SF (h f' g'))

instance ArrowLoop SF where
  loop f = SF (h f)
    where h f x = let ((y, z), f') = unSF f (x, z)
                  in (y, SF (h f'))

instance ArrowInit SF where
  init i = SF (h i)
    where h i x = (i, SF (h x))

runSF :: SF a b → [a] → [b]
runSF f = g f
  where g f (x:xs) = let (y, f') = unSF f x
                     in y : g f' xs
```

Figure 5. Causal Stream Transformer

of causal computations, namely that the current output depends only on the current as well as previous inputs. Besides causality, we make no other assumptions about the nature of these values: they may or may not vary with time, and the increment of change may be finite or infinitesimally small.

More importantly, a valid instance of the `ArrowInit` class must satisfy the *product* law shown in Figure 4. This law states that

two `init`s paired together are equivalent to one `init` of a pair. Here we use the `***` operator instead of its expanded definition `first` ... >>> `second` ... to imply that the product law assumes commutativity.

We will see in a later section that `init` and the product law are critical to our normalization and optimization strategies. But `init`

Syntax

$$
\begin{array}{rrcl}
\text{Variables} & V & ::= & x \mid y \mid z \mid \ldots \\
\text{Primitive Types} & t & ::= & 1 \mid \mathit{Int} \mid \mathit{Bool} \mid \ldots \\
\text{Types} & \alpha, \beta, \theta & ::= & t \mid \alpha \times \beta \mid \alpha \to \beta \mid \alpha \rightsquigarrow \beta \\
\text{Expressions} & E & ::= & V \mid (E_1, E_2) \mid \mathit{fst}\ E \mid \mathit{snd}\ E \mid \lambda x : \alpha . E \mid E_1\ E_2 \mid () \mid \ldots \\
\text{Environment} & \Gamma & ::= & x_1 : \alpha_1, \ldots, x_n : \alpha_n
\end{array}
$$

Typing Rules

$$
\text{(UNIT)}\quad \Gamma \vdash () : 1 \qquad \text{(VAR)}\quad \frac{(x : \alpha) \in \Gamma}{\Gamma \vdash x : \alpha} \qquad \text{(ABS)}\quad \frac{\Gamma, x : \alpha \vdash E : \beta}{\Gamma \vdash \lambda x : \alpha . E : \alpha \to \beta}
$$

$$
\text{(APP)}\quad \frac{\Gamma \vdash E_1 : \alpha \to \beta \quad \Gamma \vdash E_2 : \alpha}{\Gamma \vdash E_1\ E_2 : \beta} \qquad \text{(PAIR)}\quad \frac{\Gamma \vdash E_1 : \alpha \quad \Gamma \vdash E_2 : \beta}{\Gamma \vdash (E_1, E_2) : \alpha \times \beta}
$$

Constants

$$
\begin{array}{rcl}
\mathit{arr}_{\alpha,\beta} & : & (\alpha \to \beta) \to (\alpha \rightsquigarrow \beta) \\
\ggg_{\alpha,\beta,\theta} & : & (\alpha \rightsquigarrow \beta) \to (\beta \rightsquigarrow \theta) \to (\alpha \rightsquigarrow \theta) \\
\mathit{first}_{\alpha,\beta,\theta} & : & (\alpha \rightsquigarrow \beta) \to (\alpha \times \theta \rightsquigarrow \beta \times \theta)
\end{array}
$$

Definitions

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

$$\text{(FST)}\ \frac{\Gamma \vdash E : \alpha \times \beta}{\Gamma \vdash \mathit{fst}\ E : \alpha} \qquad \text{(SND)}\ \frac{\Gamma \vdash E : \alpha \times \beta}{\Gamma \vdash \mathit{snd}\ E : \beta}$$

$$\begin{array}{rcl} \mathit{loop}_{\alpha,\beta,\theta} & : & (\alpha \times \theta \rightsquigarrow \beta \times \theta) \to (\alpha \rightsquigarrow \beta) \\ \mathit{init}_{\alpha} & : & \alpha \to (\alpha \rightsquigarrow \alpha) \end{array}$$

$$\begin{array}{rcl} \mathit{id} & : & \alpha \to \alpha \\ \mathit{id} & = & \lambda x.x \\ (\cdot) & : & (\beta \to \theta) \to (\alpha \to \beta) \to (\alpha \to \theta) \\ (\cdot) & = & \lambda f.\lambda g.\lambda x.f\,(g\ x) \\ (\times) & : & (\alpha \to \beta) \to (\theta \to \gamma) \to (\alpha \times \theta \to \beta \times \gamma) \\ (\times) & : & \lambda f.\lambda g.\lambda z.(f\ (\mathit{fst}\ z), g\ (\mathit{snd}\ z)) \\ \mathit{dup} & : & \alpha \to \alpha \times \alpha \\ \mathit{dup} & = & \lambda x.(x, x) \end{array}$$

$$\begin{array}{rcl} \mathit{shuffle} & : & \alpha \times ((\beta \times \delta) \times (\theta \times \eta)) \to (\alpha \times (\beta \times \theta)) \times (\delta \times \eta) \\ \mathit{shuffle} & = & \mathit{assoc}^{-1} \cdot (\mathit{id} \times \mathit{transpose}) \\ \mathit{shuffle}^{-1} & : & (\alpha \times (\beta \times \theta)) \times (\delta \times \eta) \to \alpha \times ((\beta \times \delta) \times (\theta \times \eta)) \\ \mathit{shuffle}^{-1} & = & (\mathit{id} \times \mathit{transpose}) \cdot \mathit{assoc} \end{array} \qquad \begin{array}{rcl} \mathit{swap} & : & \alpha \times \beta \to \beta \times \alpha \\ \mathit{swap} & = & \lambda z.(\mathit{snd}\ z, \mathit{fst}\ z) \\ \mathit{second} & : & (\alpha \rightsquigarrow \beta) \to (\theta \times \alpha \rightsquigarrow \theta \times \beta) \\ \mathit{second} & = & \lambda f.\mathit{arr}\ \mathit{swap} \ggg \mathit{first}\ f \ggg \mathit{arr}\ \mathit{swap} \end{array}$$

Figure 6. *CCA*: a language of Causal Commutative Arrows

is also important in allowing us to define operators that were previously taken as domain-specific primitives. In particular, consider the `integral` operator used in the exponentiation examples. With `init`, we can define `integral` using the Euler integration method and a fixed global step `dt` as follows:

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

To complete the picture, we give an instance (i.e. an implementation) of CCA that captures a causal *stream transformer*, as shown in Figure 5, where:

- `SF a b` is an arrow representing functions (transformers) from streams of type `a` to streams of type `b`. It is essentially a recursively defined data type consisting of a function with its continuation, a concept closely related to a form of finite state automaton called a *Mealy Machine* [14]. *Yampa* enjoys a similar implementation, and the same data type was called *Auto* in [32].
- `SF` is declared an instance of type classes `Arrow`, `ArrowLoop` and `ArrowInit`. For example, `exp` can be instantiated as type `exp :: SF () Double`. These instances obey all of the arrow laws, including the two additional laws that we introduced.
- `runSF :: SF a b -> [a] -> [b]` converts an `SF` arrow into a stream transformer that maps an input stream of type `[a]` to an output stream of type `[b]`.

As a demonstration, we can sample the exponential function at a fixed time interval by running the `exp` arrow over an uniform input stream `inp`:

```
dt = 0.01          :: Double
inp = () : inp     :: [()]

∗Main> runSF exp inp
```

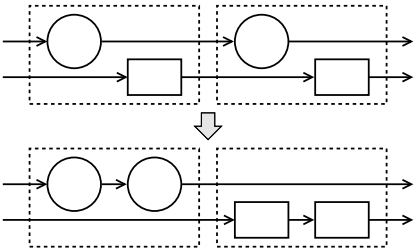
```
[1.0,1.01,1.0201,1.030301,1.04060401,1.0510100501,
...
```

We must stress that the `SF` type is but one instance of a causal commutative arrow, and alternative implementations such as the synchronous circuit type `SeqMap` in [32] and the stream function type (incidentally also called) `SF` in [24] also qualify as valid instances. The abstract properties such as normal forms that we develop in the next section are applicable to any of these instances, and thus are more broadly applicable than optimization techniques based on a specific semantic model, such as the one considered in [5].

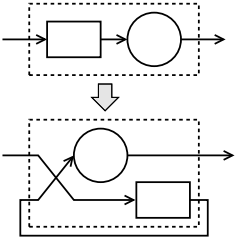
4. A Language of Causal Commutative Arrows

To study the properties of CCA more rigorously, we first introduce a language of CCA terms in Figure 6. which is an extension of the simply-typed lambda calculus with a few primitives types, tuples, and arrows. Note that:

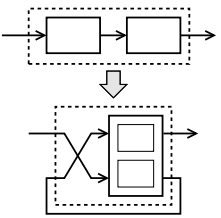
- Although the syntax requires that we write type annotations for variables in lambda abstraction, we often omit them and instead give the type of an entire expression.



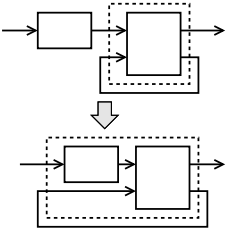
(a) Reorder parallel pure and stateful



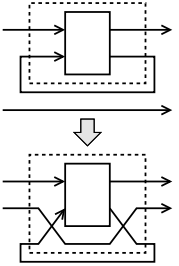
(b) Reorder sequential pure and stateful



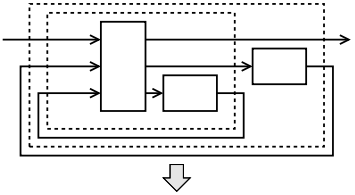
(c) Change sequential composition to parallel

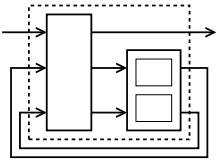


(d) Move sequential composition into loop



(e) Move parallel composition into loop





(f) Fuse nested loops

Figure 7. Arrow Transformations

- In previous examples we used the Haskell type `Arrow a => a b c` to represent an arrow type `a` mapping from type `b` to type `c`. However, CCA does not have type classes, and thus we write $\alpha \rightsquigarrow \beta$ instead.
- Each arrow constant represents a family of constant arrow functions indexed by types. We'll omit the type subscripts when they are obvious from context.

The figure also defines a set of commonly used auxiliary functions.

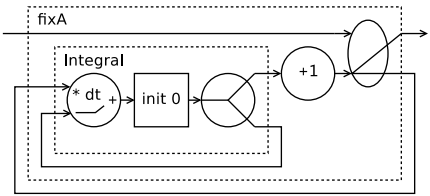
Besides satisfying the usual beta law for lambda expressions, arrows in CCA also satisfy the nine conventional arrow laws (Figure 2), the six arrow loop laws (Figure 3), and the two causal commutative arrow laws (Figure 4).

Due to the existence of immediate feedback in loops, CCA is able to make use of general recursion that is not allowed in the simply typed lambda calculus. To see why immediate feedback is necessary, we can look back at the `fixA` function used to define the

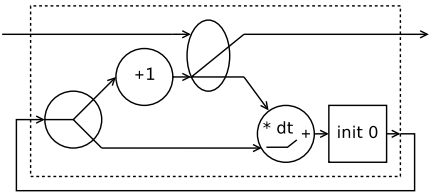
combinator version of `exp`. We rewrite it using CCA syntax below:

$$fixA : (\alpha \rightsquigarrow \alpha) \rightarrow (\beta \rightsquigarrow \alpha)$$
$$fixA = \lambda f.loop(second\ f \ggg arr(\lambda x.(snd\ x, snd\ x)))$$

It computes a fixed point of an arrow at the value level, and contains no *init* in its definition. We consider the ability to model general recursion a strength of our work that is often lacking in other stream or dataflow programming languages.



(a) Original



(b) Reorganized

Figure 8. Diagrams for `exp`

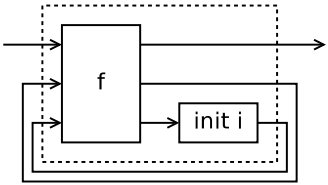


Figure 9. Diagram for $loopB$

5. Normalization of CCA

In most implementations, programs written using arrows carry a runtime overhead, primarily due to the extra tupling forced onto functions' arguments and return values. There have been several attempts [30, 24] to optimize arrow-based programs using arrow laws, but the results have not been entirely satisfactory. Although conventional arrow and arrow loop laws offer ways to combine pure arrows or collapse nested loops, they are not powerful enough to deal with *effectful* arrows, such as the $init$ combinator.

5.1 Intuition

Our new optimization strategy is based on the following rather striking observation: *any CCA program can be transformed into a single loop containing one pure arrow and one initial state value*. More precisely, any CCA program can be normalized into either the form $arr\ f$ or:

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

where f is a pure function and i is an initial state. Note that

all other arrow combinators, and therefore all of the overheads associated with them (tupling, etc.) are completely eliminated. Not surprisingly, the resulting improvement in performance is rather dramatic, as we will see later.

We treat the loop combinator not just as a way to provide feedback from output to input, but also as a way to reorganize a complex composition of arrows. To see how this works, it is helpful to visualize a few examples, as shown in Figure 7, and explained below. This should help explain the intuition behind our normalization process, which is treated formally in the next section.

The diagrams in Figure 7 can be explained as follows:

(a) *Re-order parallel pure and stateful arrows*. Figure 7(a) shows the exchange law for arrows, which is a special case of the commutativity law, and useful for re-ordering pure and stateful arrows.

(b) *Re-order sequential pure and stateful arrows*. Figure 7(b) shows how the immediate feedback of the loop combinator

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

Figure 10. One Step Reduction for CCA

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

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

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

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

Figure 11. Normalization Procedure for CCA

helps to re-order arrows. This follows from the definition of $second$, and the tightening and sliding laws for loops.

(c) *Change sequential composition to parallel*. Figure 7(c) shows that in addition to the sequential re-ordering we can use the product law to fuse two stateful computations into one.

(d) *Move sequential composition into loop*. Figure 7(d) shows the left-tightening law for loops. Because the first arrow can also be a loop, we are able to combine sequential compositions of two loops into a nested one.

(e) *Move parallel composition into loop*. Figure 7(e) shows a variant of the superposing law for loops using $first$ instead of $second$. Since we know that parallel composition can be decomposed into first and second, and if each of them can be transformed into a loop, they will eventually be combined into a nested loop as shown.

(f) *Fuse nested loops*. Figure 7(f) shows an extension of the vanishing law for loops to handle stateful computations. Its proof requires the commutative law and product law to switch the position of two stateful arrows and join them together.

As a concrete example, Figure 8(a) is a diagram of the original `exp` example given earlier. In Figure 8(b) we have unfolded the definition of `integral` and applied the optimization strategy. The result is a single loop, where all pure functions can be combined together to minimize arrow implementation overheads.

5.2 Algorithm

In this section we give a formal definition of the normalization procedure. First we define a combinator called $loopB$ that can be viewed as syntactic sugar for handling both immediate and delayed feedback:

$$loopB : \theta \to (\alpha \times (\gamma \times \theta) \rightsquigarrow \beta \times (\gamma \times \theta)) \to (\alpha \rightsquigarrow \beta)$$
$$loopB = \lambda i.\lambda f.loop\ (f \ggg second(second(init\ i)))$$

A pictorial view of $loopB$ is given in Figure 9. The second argument to $loopB$ is an arrow mapping from an input of type α to output β, while looping over a pair $\gamma \times \theta$. The value of type θ is initialized before looping back, and is often regarded as an internal state. The value of type γ is immediately fed back and often used for general recursions at the value level.

We define a single step reduction $\mapsto$ as a set of rules in Figure 10, and a normalization procedure in Figure 11. The normalization relation $\Downarrow$ can be seen as a big step reduction following an innermost strategy, and is indeed a function.

Note that some of the reduction rules resemble the arrow laws of the same name. However, there are some subtle but important differences: First, unlike the laws, reduction is directed. Second, the rules are extended to handle $loopB$ instead of $loop$. Finally, they are adjusted to avoid overlaps.

Theorem 5.1 (CCNF) *For all $\vdash e : \alpha \rightsquigarrow \beta$, there exists a normal form e_{norm}, called the Causal Commutative Normal Form, which is either of the form $arr\ f$, or $loopB\ i\ (arr\ f)$ for some i and f, such that $\vdash e_{norm} : \alpha \rightsquigarrow \beta$, and $e \Downarrow e_{norm}$. In unsugared form,*

the second form is equivalent to:

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

Proof: Follows directly from Lemmas 5.1 and 5.2. □

Note that we only consider closed terms with empty type environments in Theorem 5.1, otherwise we would have to include lambda normal forms as part of CCNF. For example, $x : \alpha \rightsquigarrow \beta \vdash x : \alpha \rightsquigarrow \beta$ would qualify as a valid CCNF since x is of an arrow type, and there is no further reduction possible. Although this addition may be needed in real implementations, it would unnecessarily complicate the discussion, so we disallow open terms for simplicity.

Lemma 5.1 (Soundness) *The reduction rules given in Figure 10 are both type and semantics preserving, i.e., if $e \mapsto e'$ then $e = e'$ is syntactically derivable from the set of CCA laws.*

Proof: By equational reasoning using arrow laws. The **loop** and **init** rules follow from the definition of $loopB$; **composition** and **extension** are directly based on the arrow laws with the same name; **left** and **right tightening** and **superposing** rules follow the definition of $loopB$, the commutativity law and the arrow loop laws with the same name. The proof of the **vanishing** rule is more involved, and is given in Appendix B. □

Note that the set of reduction rules is sound but not complete, because the loop combinator can introduce general recursion at the value level.

Lemma 5.2 (Termination) *The normalization procedure for CCA given in Figure 11 terminates for all well-typed arrow expressions* $\vdash e : \alpha \rightsquigarrow \beta$.

Proof: By structural induction over all possible combinations of well-typed arrow terms. See Appendix A for details. □

6. Further Optimization

We have implemented the normalization procedure of CCA in Haskell. In fact the normalization of an arrow term does not have to stop at CCNF, because pure functions in the language are of simply typed lambda calculus, which is strongly normalizing. Extra care was taken to preserve sharing of lambda terms, to eliminate redundant variables, and so on.

In the remainder of this section we describe a simple sequence of other optimizations that ultimately leads to a single imperative loop that can be implemented extremely efficiently.

Optimized Loop In addition to `loopB`, for optimization purposes we introduce another looping combinator, `loopD`, for loops with only delayed feedback. For comparison, the Haskell definitions of both are given below:

```
loopB :: ArrowInit a ⇒
         e → a (b, (d, e)) (c, (d, e)) → a b c
loopD :: ArrowInit a ⇒
         e → a (b, e) (c, e) → a b c
loopB i f = loop (f >>> second (second (init i)))
loopD i f = loop (f >>> second (init i))
```

The reason to introduce `loopD` is that many applications of CCA re-

sult in an arrow in which all loops only have delayed feedback. For example, after removing redundant variables, normalizing lambdas, and eliminating common sub-expressions, the CCNF for `exp` is:

```
exp' = loopB 0 (arr (λ (x, (z, y)) →
         let i = y + 1 in (i, (z, y + dt * i))))
```

Clearly the variable `z` here is redundant, and it can be removed by changing `loopB` to `loopD`:

```
exp'' = loopD 0 (arr (λ (x, y) →
         let i = y + 1 in (i, y + dt * i)))
```

The above function corresponds nicely with the diagram shown in Figure 8(b). We call this result *optimized CCNF*.

Inlining Implementation In fact `loopD` can be made even more efficient if we expose the underlying arrow implementation. For example, using the `SF` data type shown in Figure 5, `loopD` can be defined as:

```
loopD i f = SF (g i f)
  where g i f x =
          let ((y, i'), f') = unSF f (x, i)
          in (y, SF (g i' f'))
```

Also, if we examine the use of `loopD` in optimized CCNF, we notice that the arrow it takes is always a pure arrow, and hence we can drop the arrow and use the pure function instead. Furthermore, if our interest is just in computing from an input stream to an output, we can drop the intermediate `SF` data structure altogether, thus yielding:

```
runCCNF :: e → ((b, e) → (c, e)) → [b] → [c]
runCCNF i f = g i
```

```
    where g i (x:xs) = let (y, i') = f (x, i)
                       in y : g i' xs
```

`runCCNF` essentially converts an optimized CCNF term directly into a stream transformer. In doing so, we have successfully transformed away all arrow instances, including the data structure used to implement them! The result is of course no longer abstract, and is closely tied to the low-level representation of streams.

Combining CCA With Stream Fusion We can perform even more aggressive optimizations on CCNF by borrowing the stream representation and optimization techniques introduced by Coutts et al. [12]. First, we define a datatype to encapsulate a stream as a product of a stepper function and an initial state:

```
data Stream a = forall s. Stream (s → Step a s) s
data Step a s = Yield a s
```

Here `a` is the element type and `s` is an existentially quantified state type. For our purposes, we have simplified the return type of the original stepper function in [12]. Our stepper function essentially consumes a state and yields an element in the stream paired with a new state.

The key to effective fusion is that all stream producers must be non-recursive. In other words, a recursively defined stream such as `exp` should be written in terms of non-recursive stepper functions, with recursion deferred until the stream is unfolded. Programs written in this style can then be fused by the compiler into a tail-recursive loop, at which point tail-call eliminations and various unboxing optimizations can be easily applied.

This is where CCA and our normalization procedure fit together so nicely. We can take advantage of the arrow syntax to write

recursive code, and rely on the arrow translator to express it non-recursively using the `loop` combinator. We then normalize it into CCNF, and rewrite it in terms of streams.

The last step is surprisingly straightforward. We introduce yet another loop combinator `loopS` that closely resembles `loopD`:

```
loopS :: t → ((a, t) → (b, t)) →
          Stream a → Stream b
loopS z f (Stream next0 s0) = Stream next (z, s0)
    where
      next (i, s) = case next0 s of
                      Yield x s' → Yield y (z', s')
                          where (y, z') = f (x, i)
```

Intuitively, `loopS` is the unlifted version of `loopD`. The initial state of the output stream consists of the initial feedback value `z` and the state of the input stream. As the resulting stream gets unfolded, it supplies `f` with an input tuple and carries the output along with the next state of the input stream. In general, we can rewrite terms of the form $loopD\ i\ (arr\ f)$ into $loopS\ i\ f$ for some i and f.

To illustrate this, let us revisit the `exp` example. We take the optimized CCNF `exp''` and rewrite it in terms of `loopS` as `expOpt`:

```
expOpt :: Stream Double
expOpt sr = loopS 0 (λ (x, y) → let i = y + 1
                                in (i, y + dt * i))
                    (constS ())

constS :: a → Stream a
constS c = Stream next () where next _ = Yield c ()
```

Since the resulting stream producer ignores any input, we define `constS` to supply a stream of unit values. This does not negatively

impact performance, as the compiler is able to remove the dummy values eventually.

To extract elements from a stream, we can write a tail-recursive function to unfold it. For example, the function `nth` extracts the nth element from a stream:

```
nth :: Int → Stream a → a
nth n (Stream next0 s0) = go n s0 where
      go n s = case next0 s of
                 Yield x s' → if n == 0
                                then x
                                else go (n-1) s'

e2 :: Double
e2 = nth 2 expOpt  -- 1.0201
```

We can define unfolding functions other than `nth` in a similar manner.

With the necessary optimization options turned on, GHC fuses `nth` and `expOpt` into a tail-recursive loop. The code below shows the equivalent intermediate representation extracted from GHC after optimization. It uses only strict and unboxed types (`Int#` and `Double#`).

```
go :: Int# → Double# → Double#
go n y =
    case n of
      __DEFAULT  → go (n - 1) (y + dt * (y + 1.0))
      0          → y + 1.0

e2 :: Double
e2 = D# (go 2 0.0)
```

In summary, employing stream fusion, the GHC compiler can turn any CCNF into a tight imperative loop that is free of all cons cell and closure allocations. This results in a dramatic speedup for CCA programs and eliminates the need for heap allocation and garbage collection. In the next section we quantify this claim via benchmarks.

7. Benchmarks

We ran a set of benchmarks to measure the performance of several programs written in arrow syntax, but compiled and optimized in different ways. For each program, we:

1. Compiled with GHC, which has a built-in translator for arrow syntax.
2. Translated using Paterson's *arrowp* pre-processor to arrow combinators, and then compiled with GHC.
3. Normalized into CCNF combinators, and compiled with GHC.
4. Normalized into CCNF combinators, rewritten in terms of streams, and compiled with GHC using stream fusion.

The five benchmarks we used are: the exponential function given earlier, a sine wave with fixed frequency using Goertzel's method, a sine wave with variable frequency, "50's sci-fi" sound synthesis program taken from [15], and a robot simulator taken from [21]. The programs were compiled and run on an Intel Core 2 Duo machine with GHC version 6.10.1, using the C backend code generator and `-O2` optimization. We measured the CPU time used to run a program through 10^6 samples. The results are shown in

Figure 12, where the numbers represent normalized speedup ratios, and we also include the lines of code (LOC) for the source program.

The results show dramatic performance improvements using normalized arrows. We note that:

1. Based on the same arrow implementation, the performance gain of CCNF over the first two approaches is entirely due to program transformations at the source level. This means that the runtime overhead of arrows is significant, and cannot be neglected for real applications.
2. The stream representation of CCNF produces high-performance code that is completely free of dynamic memory allocation and intermediate data structures, and can be orders of magnitude faster than its arrow-based predecessors.
3. GHC's arrow syntax translator does not do as well as Paterson's original translator for the sample programs we chose, though both are significantly outperformed by our normalization techniques.

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

Figure 12. Performance Ratio (greater is better)

8. Discussion

Our key contribution is the discovery of a normal form for core Yampa, or CCA, programs: any CCA program can be transformed into a single loop with just one pure (and strongly normalizing) function and a set of initial states. This discovery is new and original, and has practical implications in implementing not just Yampa, but a broader class of synchronous dataflow languages and stream computations because this property is entirely based on axiomatic laws, not any particular semantic model. We discuss such relevance and related topics to our approach below.

8.1 Alternative Formalisms

Apart from arrows, other formalisms such as monads, comonads and applicative functors have been used to model computations over data streams [3, 42, 28]. Central to many of these approaches are the representation of streams and computations about them. However, notably missing are the connections between stream computation and the related laws. For example, Uustalu's work [42] concluded that comonad is a suitable model for dataflow computation, but it lacks any discussion of whether the comonadic laws are of any relevance.

In contrast, it is the very idea of making sense out of arrow and arrow loop laws that motivated our work. We argue that arrows are a suitable abstract model for stream computation not only because we can implement stream functions as arrows, but also because abstract properties like the arrow laws help to bring more insights to our target application domain.

Besides having to satisfy respective laws for these formalisms, each abstraction has to introduce domain specific operators, otherwise it would be too general to be useful. With respect to causal streams, many have introduced $init$ (also known as $delay$) as a primitive to enable stateful computation, but few seem to have made the connection of its properties to program optimizations.

Notably the product law we introduced for CCA relates to a bisimilarity property of co-algebraic streams, i.e., the product of two initialized streams are bisimilar to one initialized stream of product.

8.2 Co-algebraic streams

The co-algebraic property of streams is well known, and most relevant to our work is Caspi and Pouzet's representation of stream and stream functions in a functional language setting [5], which also uses a primitive similar to the trace operator (and hence the arrow loop combinator) to model recursion. Their compilation technique, however, lacks a systematic approach to optimize nested recursions. We consider our technique more effective and more abstract.

Most synchronous languages, including the one introduced in [5], are able to compile stream programs into a form called *single loop code* by performing a causality analysis to break the feedback loop of recursively defined values. Many efforts have been made to generate efficient single loop code [16, 1], but to our best knowledge there has not been a strong result like normal forms. Our discovery of CCNF is original, and the optimization by normalization approach is both systematic and deterministic. Together with stream fusion, we produce a result that is not just a single loop, but a highly optimized one.

Also relevant is Rutten's work on high-order functional stream derivatives [38]. We believe that arrows are a more general abstraction than functional stream derivatives, because the latter still exposes the structure of a stream. Moreover, arrows give rise to a high-level language with richer algebraic properties than the 2-adic calculus considered in [38].

8.3 Expressiveness

It is known that operationally a Mealy machine is able to represent all causal stream functions [38], while the CCA language defined in Figure 6 represents only a subset. For example, the switch combinator introduced in Yampa [21] is able to dynamically replace a running arrow with a new one depending on an input event, and hence to switch the system behavior completely. With CCA, there is no way to change the compositional structure of the arrow program itself at run time. For another example, many dataflow and stream programming languages also provide conditionals, such as `if-then-else`, as part of the language [43, 4]. To enable conditionals at the arrow level, we need to further extend CCA to be an instance of the `ArrowChoice` class. Both are worthy extensions to consider for future work.

It should also be noted that the local state introduced by `init` is one of the minimal side effects one can introduce to arrow programs. The commutativity law for CCA ensures that the effect of one arrow cannot interfere with another when composed together, and it is no longer satisfiable when such ordering becomes important, e.g., when arrows are used to model parsers and printers [25].

On the other hand, because the language for CCA remains highly abstract, it could be applicable to domains other than FRP

or dataflow. We'll leave such findings to future work.

8.4 Stream fusion

Stream fusion can help fuse zips, left folds, and nested lists into efficient loops. But on its own, it does not optimize recursively and lazily defined streams effectively.

Consider a stream generating the Fibonacci sequence. It is one of the simplest classic examples that characterizes stateful stream computation. One way of writing it in Haskell is to exploit laziness and zip the stream with itself:

```
fibs :: [Int]
fibs = 0:1:zipWith (+) fibs (tail fibs)
```

While the code is concise and elegant, such programming style relies too much on the definition of an inductively defined structure. The explicit sharing of the stream `fibs` in the definition is a blessing and a curse. On one hand, it runs in linear time and constant space. On the other hand, the presence of the stream structure gets in the way of optimization. None of the current fusion or deforestation techniques are able to effectively eliminate cons cell allocations in this example. Real-world stream programs are usually much more complex and involve more feedback, and the time spent in allocating intermediate structure and by the garbage collector could degrade performance significantly.

We can certainly write a stream in stepper style that generates the Fibonacci sequence:

```
fib_stream :: Stream Int
fib_stream = Stream next (0, 1) where
```

```
        next (a, b) = Yield r (b, r)
            where r = a + b

f1 :: Int
f1 = nth 5 fib_stream  -- 13
```

Stream fusion will fuse `nth` and `fib_stream` to produce an efficient loop. For a comparison, with our technique the arrow version of the Fibonacci sequence shown below compiles to the same efficient loop as `f1` above, and yet retains the benefit of being abstract and concise.

```
fibA = proc _ → do
  rec let r = d2 + d1
      d1 ← init 0 ⤙ d2
      d2 ← init 1 ⤙ r
  returnA ⤙ r
```

We must stress that writing stepper functions is not always as easy as in trivial examples like `fib` and `exp`. Most non-trivial stream programs that we are concerned with contain many recursive parts, and expressing them in terms of combinators in a non-recursive way can get unwieldy. Moreover, this kind of coding style exposes a lot of operational details which are arguably unnecessary for representing the underlying algorithm. In contrast, arrow syntax relieves the burden of coding in combinator form and allows recursion via the `rec` keyword. It also completely hides the actual implementation of the underlying stream structure and is therefore more abstract.

The strength of CCA is the ability to normalize any causal and recursive stream function. Combining both fusion and our normalization algorithm, any CCA program can be reliably and pre-

dictably optimized into an efficient machine-friendly loop. The process can be fully automated, allowing programmers to program at an abstract level while getting performance competitive to programs written in low-level imperative languages.

Acknowledgements This research was supported in part by NSF grants CCF-0811665 and CNS-0720682, and a grant from Microsoft Research.

References

[1] Pascalin Amagbegnon, Loc Besnard, and Paul Le Guernic. Implementation of the data-flow synchronous language signal. In *In Conference on Programming Language Design and Implementation*, pages 163–173. ACM Press, 1995.

[2] Robert Atkey. What is a categorical model of arrows? In *Mathematically Structured Functional Programming*, 2008.

[3] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in haskell. In *ICFP*, pages 174–184, 1998.

[4] P. Caspi, N. Halbwachs, D. Pilaud, and J.A. Plaice. Lustre: A declarative language for programming synchronous systems. In *14th ACM Symp. on Principles of Programming Languages*, January 1987.

[5] Paul Caspi and Marc Pouzet. A Co-iterative Characterization of Synchronous Stream Functions. In *Coalgebraic Methods in Computer Science (CMCS'98)*, Electronic Notes in Theoretical Computer Science, March 1998. Extended version available as a VERIMAG tech. report no. 97–07 at `www.lri.fr/~pouzet`.

[6] Eric Cheng and Paul Hudak. Look ma, no arrows – a functional reactive real-time sound synthesis framework. Technical Report YALEU/DCS/RR-1405, Yale University, May 2008.

[7] Mun Hon Cheong. Functional programming and 3d games, November 2005. also see www.haskell.org/haskellwiki/Frag.

[8] Jean-Louis Colaço, Alain Girault, Grégoire Hamon, and Marc Pouzet. Towards a higher-order synchronous data-flow language. In *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*, pages 230–239, New York, NY, USA, 2004. ACM.

[9] Antony Courtney. *Modelling User Interfaces in a Functional Language*. PhD thesis, Department of Computer Science, Yale University, May 2004.

[10] Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Proc. of the 2001 Haskell Workshop*, September 2001.

[11] Antony Courtney, Henrik Nilsson, and John Peterson. The Yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN Haskell Workshop (Haskell'03)*, pages 7–18, Uppsala, Sweden, August 2003. ACM Press.

[12] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2007*, April 2007.

[13] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 263–273, June 1997.

[14] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[15] George Giorgidze and Henrik Nilsson. Switched-on yampa. In Paul Hudak and David Scott Warren, editors, *Practical Aspects of Declarative Languages, 10th International Symposium, PADL 2008, San Francisco, CA, USA, January 7-8, 2008*, volume 4902 of *Lecture*

Notes in Computer Science, pages 282–298. Springer, 2008.

[16] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In J. Maluszyński and M. Wirsing, editors, *Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming*, number 528, pages 1–13207–218. Springer Verlag, 1991.

[17] Masahito Hasegawa. Recursion from cyclic sharing: traced monoidal categories and models of cyclic lambda calculi. pages 196–213. Springer Verlag, 1997.

[18] L. Huang, P. Hudak, and J. Peterson. Hporter: Using arrows to compose parallel processes. In *Proc. Practical Aspects of Declarative Languages*, pages 275–289. Springer Verlag LNCS 4354, January 2007.

[19] P. Hudak. Building domain specific embedded languages. *ACM Computing Surveys*, 28A:(electronic), December 1996.

[20] Paul Hudak. Modular domain specific languages and tools. In *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society, June 1998.

[21] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Summer School on Advanced Functional Programming 2002, Oxford University*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer-Verlag, 2003.

[22] Paul Hudak, Paul Liu, Michael Stern, and Ashish Agarwal. Yampa meets the worm. Technical Report YALEU/DCS/RR-1408, Yale University, July 2008.

[23] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.

[24] John Hughes. Programming with arrows. In *Advanced Functional Programming*, pages 73–129, 2004.

[25] Patrik Jansson and Johan Jeuring. Polytypic compact printing and parsing. In *ESOP*, pages 273–287, 1999.

[26] Sam Lindley, Philip Wadler, and Jeremy Yallop. The arrow calculus (functional pearl). *Draft*, 2008.

[27] Hai Liu and Paul Hudak. Plugging a space leak with an arrow. *Electronic Notes in Theoretical Computer Science*, 193:29–45, nov 2007.

[28] Conor McBride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, 2008.

[29] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[30] Henrik Nilsson. Dynamic optimization for functional reactive programming using generalized algebraic data types. In *ICFP*, pages 54–65, 2005.

[31] Clemens Oertel. *RatTracker: A Functional-Reactive Approach to Flexible Control of Behavioural Conditioning Experiments*. PhD thesis, Wilhelm-Schickard-Institute for Computer Science at the University of Tübingen, May 2006.

[32] Ross Paterson. A new notation for arrows. In *ICFP'01: International Conference on Functional Programming*, pages 229–240, Firenze, Italy, 2001.

[33] John Peterson, Gregory Hager, and Paul Hudak. A language for declarative robotic programming. In *International Conference on Robotics and Automation*, 1999.

[34] John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with Haskell. In *First International Workshop on*

Practical Aspects of Declarative Languages. SIGPLAN, Jan 1999.

[35] John Peterson, Zhanyong Wan, Paul Hudak, and Henrik Nilsson. *Yale FRP User's Manual*. Department of Computer Science, Yale University, January 2001. Available at `http://www.haskell.org/frp/manual.html`.

[36] Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. http://www.haskell.org/definition/.

[37] John Power and Hayo Thielecke. Closed freyd- and kappa-categories. In *ICALP*, pages 625–634, 1999.

[38] Jan J. M. M. Rutten. Algebraic specification and coalgebraic synthesis of mealy automata. *Electr. Notes Theor. Comput. Sci*, 160:305–319, 2006.

[39] Robert Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[40] Ross Howard Street, A. Joyal, and D. Verity. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society*, 119(3):425–446, 1996.

[41] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, London, UK, 2002. Springer-Verlag.

[42] Tarmo Uustalu and Varmo Vene. The essence of dataflow programming. In Zoltán Horváth, editor, *CEFP*, volume 4164 of *Lecture Notes in Computer Science*, pages 135–167. Springer, 2005.

[43] William W. Wadge and Edward A. Ashcroft. *LUCID, the dataflow programming language*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

A. Proof for the termination lemma

Proof: We will show that the there always exists a e_{norm} for well formed arrow expression $\vdash\ e\ :\ \alpha \rightsquigarrow \beta$, and the normalization procedure always terminates. This is done by structural induction over all possible arrow terms, and any closed expression e that's not already in arrow terms shall be first beta reduced.

1. $e = arr\ f$
   It already satisfies the termination condition.
2. $e = first\ f$
   By induction hypothesis, $f \Downarrow arr\ f'$, or $f \Downarrow loopB\ i\ (arr\ f'')$, where f' and f'' are pure functions.

   In the first case by extension rule $first\ f \mapsto arr(f' \times id)$ and terminates; In the second case

   $$\begin{aligned} & first\ f \\ \overset{*}{\mapsto} & first(loopB\ i(arr\ f'')) \\ & \text{superposing} \\ \mapsto & loopB\ i\ (arr\ juggle \ggg arr\ f'' \ggg arr\ juggle) \\ & \text{composition} \\ \mapsto & loopB\ i\ (arr\ (juggle \cdot f'' juggle)) \end{aligned}$$

   and terminates.
3. $e = f \ggg g$
   By induction hypothesis, $f \Downarrow arr\ f'$ or $f \Downarrow loopB\ i\ (arr\ f'')$, and $g \Downarrow arr\ g'$ or $g \Downarrow loopB\ i\ (arr\ g'')$. So there are 4 combinations, and in all cases they terminate.

   1)

$f \ggg g$
$\mapsto^{*} arr\ f' \ggg arr\ g'$
composition
$\mapsto arr(g' \cdot f')$

2)
$f \ggg g$
$\mapsto^{*} arr\ f' \ggg loopB\ i\ (arr\ g'')$
left tightening
$\mapsto loopB\ i\ (first\ (arr\ f') \ggg arr\ g'')$
extension
$\mapsto loopB\ i\ (arr\ (f' \times id) \ggg arr\ g'')$
composition
$\mapsto loopB\ i\ (arr\ (g'' \cdot (f' \times id)))$

3)
$f \ggg g$
$\mapsto^{*} loopB\ i\ (arr\ f'') \ggg arr\ g'$
right tightening
$\mapsto loopB\ i\ (arr\ f'' \ggg first(arr\ g'))$
extension
$\mapsto loopB\ i\ (arr\ f'' \ggg arr(g' \times id))$
composition
$\mapsto loopB\ i\ (arr\ ((g' \times id) \cdot f''))$

4)
$f \ggg g$
$\mapsto^{*} loopB\ i\ (arr\ f'') \ggg loopB\ i\ (arr\ g'')$
left tightening

$\mapsto loopB\ j\ (first(loopB\ i\ (arr\ f''))) \ggg arr\ g'')$
superposing
$\mapsto loopB\ j\ (loopB\ i\ (arr\ juggle \ggg arr\ f'' \ggg arr\ juggle) \ggg arr\ g'')$
composition
$\overset{*}{\mapsto} loopB\ j\ (loopB\ i\ (arr\ (juggle \cdot f'' \cdot juggle)) \ggg arr\ g'')$
right tightening
$\mapsto loopB\ j\ (loopB\ i\ (arr\ (juggle \cdot f'' \cdot juggle) \ggg first(arr\ g'')))$
extension
$\mapsto loopB\ j\ (loopB\ i\ (arr\ (juggle \cdot f'' \cdot juggle) \ggg arr\ (g'' \times id)))$
composition
$\mapsto loopB\ j\ (loopB\ i\ (arr\ ((g'' \times id) \cdot juggle \cdot f'' \cdot juggle)))$
vanishing
$\mapsto loopB\ (j,i)\ (arr\ shuffle \ggg arr\ ((g'' \times id) \cdot juggle \cdot f'' \cdot juggle) \ggg arr\ shuffle^{-1})$
composition
$\overset{*}{\mapsto} loopB\ (j,i)\ (arr\ (shuffle^{-1} \cdot (g'' \times id) \cdot juggle \cdot f'' \cdot juggle \cdot shuffle))$

4. $e = loop\ f$
By induction hypothesis, $f \Downarrow arr\ f'$ or $f \Downarrow loopB\ i\ (arr\ f'')$. In the first case

$loop\ f$
$\overset{*}{\mapsto} loop\ (arr\ f')$
loop

$\mapsto loopB\ ()(arr\ assoc^{-1} \ggg arr\ f' \ggg arr\ assoc)$
composition
$\mapsto loopB\ ()(arr\ (assoc \cdot f' \cdot assoc^{-1}))$

and terminates. In the second case

$loop\ f$
$\overset{*}{\mapsto} loop\ (loopB\ i\ (arr\ f''))$
loop
$\mapsto loopB\ ()(arr\ assoc^{-1} \ggg loopB\ i\ (arr\ f'') \ggg arr\ assoc)$
left and right tightening
$\overset{*}{\mapsto} loopB\ ()(loopB\ i\ (first(arr\ assoc^{-1}) \ggg arr\ f'' \ggg first(arr\ assoc)))$
extension and composition
$\overset{*}{\mapsto} loopB\ ()(loopB\ i\ (arr\ ((assoc \times id) \cdot f'' \cdot (assoc^{-1} \times id))))$
vanishing
$\mapsto loopB\ ((), i)(arr\ shuffle \ggg arr\ ((assoc \times id) \cdot f'' \cdot (assoc^{-1} \times id)) \ggg arr\ shuffle^{-1})$
composition
$\mapsto loopB\ ((), i)(arr(shuffle^{-1} \cdot (assoc \times id) \cdot f'' \cdot (assoc^{-1} \times id) \cdot shuffle))$

and terminates.

5. $e = init\ i$
By init rule, $init\ i \mapsto loopB\ i\ (arr\ (swap \cdot juggle \cdot swap))$ and terminates.

□

B. Proof for the *vanishing* rule of $loopB$

Proof: We will show that

$$\begin{aligned} & loopB\ i\ (loopB\ j\ f) \\ = \ & loopB\ (i,j)\ (arr\ shuffle \ggg f \ggg arr\ shuffle^{-1}) \end{aligned}$$

by equational reasoning.

$$\begin{aligned} & loopB\ i\ (loopB\ j\ f) \\ & \text{definition of loopB} \\ = \ & loop\ (loopB\ j\ f \ggg second\ (second\ (init\ i))) \\ & \text{definition of loopB} \\ = \ & loop\ (loop\ (f \ggg second\ (second\ (init\ j))) \ggg \\ & \qquad second\ (second\ (init\ i))) \\ & \text{right tightening of loop} \\ = \ & loop\ (loop\ (f \ggg second\ (second\ (init\ j)) \ggg \\ & \qquad first(second\ (second\ (init\ i))))) \\ & \text{commutativity} \\ = \ & loop\ (loop\ (f \ggg first(second\ (second\ (init\ i))) \ggg \\ & \qquad second\ (second\ (init\ j)))) \\ & \text{vanishing of loop} \\ = \ & loop\ (arr\ assoc^{-1} \ggg f \ggg \\ & \qquad first\ (second\ (second\ (init\ i))) \ggg \\ & \qquad second\ (second\ (init\ j)) \ggg arr\ assoc) \\ & \text{Lemma B.1} \\ = \ & loop\ (arr\ assoc^{-1} \ggg f \ggg arr\ shuffle^{-1} \ggg \\ & \qquad second\ (second\ (init\ (i,j))) \ggg \\ & \qquad arr\ shuffle \ggg arr\ assoc) \\ & \text{shuffle}^{-1} \cdot \text{shuffle} = \text{id} \\ = \ & loop\ (arr\ (shuffle^{-1}\ \cdot assoc^{-1}) \ggg arr\ shuffle \ggg f \ggg \end{aligned}$$

$$
\begin{aligned}
& \quad arr\ shuffle^{-1} \ggg second\ (second\ (init\ (i,j))) \ggg \\
& \quad arr\ shuffle \ggg arr\ assoc) \\
& \text{shuffle}^{-1} \cdot \text{assoc}^{-1} = \text{id} \times \text{transpose} \\
= \ & loop\ (arr\ (id \times transpose) \ggg arr\ shuffle \ggg f \ggg \\
& \quad arr\ shuffle^{-1} \ggg second\ (second\ (init\ (i,j))) \ggg \\
& \quad arr\ shuffle \ggg arr\ assoc) \\
& \text{sliding} \\
= \ & loop\ (arr\ shuffle \ggg f \ggg arr\ shuffle^{-1} \ggg \\
& \quad second\ (second\ (init\ (i,j))) \ggg arr\ shuffle \ggg \\
& \quad arr\ assoc \ggg arr\ (id \times transpose)) \\
& \text{shuffle}^{-1} = (\text{id} \times \text{transpose}) \cdot \text{assoc} \\
= \ & loop\ (arr\ shuffle \ggg f \ggg arr\ shuffle^{-1} \ggg \\
& \quad second\ (second\ (init\ (i,j))) \ggg \\
& \quad arr\ shuffle \ggg arr\ shuffle^{-1}) \\
& \text{shuffle} \cdot \text{shuffle}^{-1} = \text{id} \\
= \ & loop\ (arr\ shuffle \ggg f \ggg arr\ shuffle^{-1} \ggg \\
& \quad second\ (second\ (init\ (i,j)))) \\
& \text{definition of loopB} \\
= \ & loopB\ (i,j)(arr\ shuffle \ggg f \ggg arr\ shuffle^{-1})
\end{aligned}
$$

Lemma B.1

$$
\begin{aligned}
& first\ (second\ (second\ (init\ i))) \ggg \\
& second\ (second\ (init\ j)) \\
= \ & arr\ shuffle^{-1} \ggg second(second(init(i,j))) \ggg \\
& arr\ shuffle
\end{aligned}
$$

Proof: We first show

$$first\ (second\ (second\ (init\ i)))$$

$$= \quad arr\ shuffle^{-1} \ggg second(second(first(init\ i))) \ggg arr\ shuffle$$

This can be done by equational reasoning from both sides. From lhs:

$$
\begin{array}{ll}
 & first\ (second\ (second\ (init\ i))) \\
 & \text{definition of second} \\
= & first(arr\ swap \ggg first(arr\ swap \ggg first(init\ i) \ggg \\
 & \qquad arr\ swap) \ggg arr\ swap) \\
 & \text{functor and extension} \\
= & arr(swap \times id) \ggg \\
 & first(first(arr\ swap \ggg first(init\ i) \ggg arr\ swap)) \ggg \\
 & arr(swap \times id) \\
 & \text{association} \\
= & arr(swap \times id) \ggg arr\ assoc \ggg \\
 & first(arr\ swap \ggg first(init\ i) \ggg arr\ swap) \ggg \\
 & arr\ assoc^{-1} \ggg arr(swap \times id) \\
 & \text{functor and extension} \\
= & arr(assoc \cdot (swap \times id)) \ggg arr(swap \times id) \ggg \\
 & first(first(init\ i)) \ggg \\
 & arr(swap \times id) \ggg arr((swap \times id) \cdot assoc^{-1}) \\
 & \text{association} \\
= & arr((swap \times id) \cdot assoc \cdot (swap \times id)) \ggg \\
 & arr\ assoc \ggg first(init\ i) \ggg arr\ assoc^{-1} \ggg \\
 & arr((swap \times id) \cdot assoc^{-1} \cdot (swap \times id)) \\
 & \text{composition} \\
= & arr(assoc \cdot (swap \times id) \cdot assoc \cdot (swap \times id)) \ggg \\
 & first(init\ i) \ggg \\
 & arr((swap \times id) \cdot assoc^{-1} \cdot (swap \times id) \ggg assoc^{-1})
\end{array}
$$

Lemma B.2

$$
\begin{aligned}
&= arr(assoc \cdot (swap \times id) \cdot assoc \cdot (swap \times id)) \ggg \\
&\quad arr(id \times (swap \cdot assoc^{-1} \cdot transpose \cdot assoc^{-1})) \ggg \\
&\quad first(init\ i) \ggg \\
&\quad arr(id \times (assoc \cdot transpose \cdot assoc \cdot swap)) \ggg \\
&\quad arr((swap \times id) \cdot assoc^{-1} \cdot (swap \times id) \ggg assoc^{-1})
\end{aligned}
$$

composition and normalization

$$
\begin{aligned}
&= arr(\lambda((a,(c,d)),(b,e)).(d,(e,((c,b),a)))) \ggg \\
&\quad first(init\ i) \ggg \\
&\quad arr(\lambda(d,(e,((c,b),a))).((a,(c,d)),(b,e)))
\end{aligned}
$$

and from *lhs*:

$$
\begin{aligned}
&\quad arr\ shuffle^{-1} \ggg \\
&\quad second(second(first(init\ i))) \ggg \\
&\quad arr\ shuffle
\end{aligned}
$$

definition of second

$$
\begin{aligned}
&= arr\ shuffle^{-1} \ggg arr\ swap \ggg \\
&\quad first(arr\ swap \ggg first(first(init\ i)) \ggg arr\ swap) \ggg \\
&\quad arr\ swap \ggg arr\ shuffle
\end{aligned}
$$

functor and extension

$$
\begin{aligned}
&= arr(swap \cdot shuffle^{-1}) \ggg arr(swap \times id) \ggg \\
&\quad first(first(first(init\ i))) \ggg \\
&\quad arr(swap \times id) \ggg arr(shuffle \cdot swap)
\end{aligned}
$$

association

$$
\begin{aligned}
&= arr((swap \times id) \cdot swap \cdot shuffle^{-1}) \ggg arr\ assoc \\
&\quad \ggg arr\ assoc \ggg first(init\ i) \ggg arr\ assoc^{-1} \ggg \\
&\quad arr\ assoc^{-1} \ggg arr(shuffle \cdot swap \cdot (swap \times id))
\end{aligned}
$$

composition

$$
= arr(assoc \cdot assoc \cdot (swap \times id) \cdot swap \cdot shuffle^{-1}) \ggg
$$

$first(init\ i) \ggg$
$arr(shuffle \cdot swap \cdot (swap \times id) \cdot assoc^{-1} \cdot assoc^{-1})$
normalization
$= arr(\lambda((a,(c,d)),(b,e)).(d,(e,((c,b),a)))) \ggg$
$first(init\ i) \ggg$
$arr(\lambda(d,(e,((c,b),a))).((a,(c,d)),(b,e)))$

Hence $lhs = rhs$. Using similar technique, we can also prove (details omitted to save space)

$second\ (second\ (init\ j))$
$= arr\ shuffle^{-1} \ggg second(second(second(init\ j))) \ggg$
$arr\ shuffle$

Therefore we have

$first(second(second(init\ i))) \ggg second(second(init\ j))$
substitution
$= arr\ shuffle^{-1} \ggg second(second(first(init\ i))) \ggg$
$arr\ shuffle \ggg arr\ shuffle^{-1} \ggg$
$second(second(second(init\ i))) \ggg arr\ shuffle$
shuffle $\cdot$ shuffle^{-1} = id
$= arr\ shuffle^{-1} \ggg second(second(first(init\ i))) \ggg$
$second(second(second(init\ i))) \ggg arr\ shuffle$
functor and product
$= arr\ shuffle^{-1} \ggg second(second(init(i,j))) \ggg$
$arr\ shuffle$

□

Lemma B.2 $\forall g, g^{-1}, g \cdot g^{-1} = id$, *we have*

$$first\ f = arr\ (id \times g) \ggg first f \ggg arr(id \times g^{-1})$$

Proof:

$$\begin{aligned}
& arr\ (id \times g) \ggg first f \ggg arr(id \times g^{-1}) \\
& \quad \text{exchange} \\
= \ & first f \ggg arr\ (id \times g) \ggg arr(id \times g^{-1}) \\
& \quad \text{composition} \\
= \ & first f \ggg arr\ ((id \times g^{-1}) \cdot (id \times g)) \\
& \quad \text{normalization} \\
= \ & first f \ggg arr\ id \\
& \quad \text{right identity} \\
= \ & first f
\end{aligned}$$

□