Causal commutative arrows revisited

Hai Liu
Intel Labs
hai.liu@intel.com

Jeremy Yallop
University of Cambridge
jeremy.yallop@cl.cam.ac.uk

Abstract
Causal commutative arrows (CCA) extend arrows with additional constructs and laws that make them suitable for modelling domains such as functional reactive programming, differential equations and synchronous dataflow. Earlier work has revealed that a syntactic transformation of CCA computations into normal form can result in significant performance improvements, sometimes increasing the speed of programs by orders of magnitude. In this work we reformulate the normalization as a type class instance and derive optimized observation functions via a specialization to stream transformers to demonstrate that the same dramatic improvements can be achieved without leaving the language.

Categories and Subject Descriptors CR-number [subcategory]: third-level
General Terms term1, term2
Keywords keyword1, keyword2

Introduction
Arrows (Hughes 2000) provide a high-level interface to computation, allowing programs to be expressed abstractly rather than concretely, using reusable combinators in place of special-purpose control flow code. Here is a program written using arrows:

\[
\text{exp} = \text{proc} () \rightarrow \text{do}
\text{rec let} \ e = 1 + i
\text{i} \leftarrow \text{integral} \prec e
\text{returnA} \prec e
\]

which corresponds to the following recursive definition of the exponential function

\[
e(t) = 1 + \int_0^t e(t) dt
\]

Paterson’s arrow notation (Paterson 2001), used in the definition of \text{exp}, makes the data flow pleasingly clear: the \text{integral} function forms the shaft of an arrow that turns \text{e} at the nck into \text{i} at the head. The name \text{e} appears twice more, once above the arrow as the successor of \text{i}, and once below as the result of the whole computation. (The definition of \text{integral} itself appears later in this paper, on page 4.) The notation need not be taken as primitive; there is a desugaring into a set of combinators \text{arr}, \ggg, \text{first}, \text{loop}, and \text{init} which construct terms of an overloaded type \text{arr}. Most of the code listings in this paper uses these combinators, which are more convenient for defining instances, in place of the notation; we refer the reader to Paterson (2001) for the details of the desugaring.

Unfortunately, speed does not always follow succinctness. Although arrows in poetry are a byword for swiftness, arrows in programs can introduce significant overhead. Continuing with the example above, in order to run \text{exp}, we must instantiate the abstract arrow with a concrete implementation, such as the causal stream transformer \text{SF} that forms the basis of the Yampa domain-specific language for functional reactive programming (Liu et al. 2009):

\[
\text{newtype} \text{SF} \ a \ b = \text{SF} \{ \text{unSF} \ : \ a \rightarrow (b, \text{SF} \ a \ b) \}
\]

(The accompanying instances for \text{SF}, which define the arrow operators, appear on page 6.)

Instantiating \text{exp} with \text{SF} brings an unpleasant surprise: the program runs orders of magnitude slower than an equivalent program that does not use arrows. The programmer is faced with the familiar need to choose between a high level of abstraction and acceptable performance. Liu et al. (2009) describe the problem in more detail, and also propose a remedy: the laws which arrow implementations must obey can be used to rewrite arrow computations into a normal form which eliminates the overhead of the arrow abstraction. Their design is realized as a Template Haskell library (Sheard and Jones 2002), which transforms the syntax of programs during compilation to rewrite CCA computations into normal form.

The solution described by Liu et al. achieves significant performance improvements, but the use of Template Haskell introduces a number of drawbacks. Perhaps most significantly, the Template Haskell implementation of normalization is untyped: there is no check that the types of the unnormalized and normalized terms are the same. Although the normalizer code operates on untyped syntax, its output is passed to the type checker, so there is no danger of actually running ill-typed code. Nevertheless, the fact that the normalizer is not guaranteed to preserve typing means that errors may be discovered significantly later. An further drawback is that the normalizer can only operate on computations whose structure is fully known during compilation, when Template Haskell operates.

In this paper we address the first of these drawbacks and suggest a path to addressing the second. The reader familiar with recent Template Haskell developments might at this point expect us to propose switching the existing normalizer to using typed quotations and splices. Instead, we present a simpler approach, eschewing syntactic transformations altogether and defining normalization as an operation on values, implemented as a type class instance.

Contributions
Section 2 reviews Causal Commutative Arrows (CCA), their definition as a set of Haskell type classes, the accompanying laws, and
Causal commutative arrows revisited

The contributions of the remainder of this paper are as follows:

- We derive a new implementation of CCA normalization, realised as a set of type class instances for a data type which represents the CCA normal form (Section 3)
- We derive an optimized version of the “observation” function that interprets normalized CCA computations using other arrow instances (Section 4).
- We present a second implementation of the normalizing instances for CCA based on mutable state, and use it to improve the performance of the Euterpea library (Section 5).
- We demonstrate via a series of micro- and macro-benchmarks that the performance of our normalizing instances compares favourably with the Template Haskell implementation used in the original CCA work (Section 6).

Finally, Section 7 contextualizes our work in the existing literature.

Background

For readers that may not be familiar with arrows or CCA, we first begin with a review of some background knowledge of arrows, before introducing CCA and their normalization.

Arrows

Arrows (Hughes 2000) are a generalization of monads that relax the linearity constraint, while retaining a disciplined style of composition. Like monads in Haskell, the type of computation captured by arrows is expressed through the Arrow type class, shown in Figure 1 together with diagrams describing its three combinators. The combinator arr lifts a function from type a to type b to a “pure” arrow computation from a to b, of type arr a b where arr is the arrow type. The combinator ∘∘∘ composes two arrow computations by connecting the output of the first to the input of the second, and represents a sequential composition. Lastly, in order to allow “branching” and “merging” of inputs and outputs, the Arrow class provides the first combinator, based on which all other parallel combinators can be defined. Intuitively, first f is analogous to applying arrow computation f to the first of a pair of inputs to obtain the first output, while connecting the second input directly to the second output. The dual of first, called second can be defined as follows:

\[
\text{second} :: \text{Arrow arr} \Rightarrow \text{arr} a b \rightarrow \text{arr} (c, a) (c, b)
\]

Parallel composition ∘∘∘ of two arrows can then be defined as a sequence of first and second:

\[
(\circ \circ \circ) :: \text{Arrow arr} \Rightarrow \text{arr} a b \rightarrow \text{arr} c d \rightarrow \text{arr} (c, a) (b, d)
\]

Together, these combinators form an interface to first-order computations, i.e. computations which do not dynamically construct or change their compositional structure during the course of their execution.

Like monads, all arrows are governed by a set of algebraic laws, which are shown in Figure 2a. Lindley et al. (2010) further showed that the nine arrow laws can be reduced to eight. It is worth noting that all arrow laws respect the order of sequential composition of “impure” arrows, while a number of them (exchange, unit and association) allow “pure” arrows to be moved around without affecting the computation.

Arrows can be extended to have more operations, governed by additional laws. Paterson (2001) defines the ArrowLoop class (Figure 1) with an operator loop, which corresponds to the rec syntax in the arrow notation. Intuitively, the second output of the arrow inside loop is immediately connected back to its second input, and thus becomes a form of recursion at value level, as opposed to recursively defined arrows (an example of which is given in Section 5.3). Figure 2b gives the set of laws for ArrowLoop.

Causal commutative arrows

Based on looping arrows, Liu et al. (2009) introduces another extension called causal commutative arrows (CCA) with an init combinator in the ArrowInit class (Figure 1), and two additional laws to place further constraints on the computation (Figure 2c). In the context of synchronous circuits, ArrowInit is almost identical to the ArrowCircuit class first introduced by Paterson (2001), with init being equivalent to delay that supplies its argument as its initial output, and copies from its input to the rest of outputs. For the purpose of this paper, we will continue using the name ArrowInit to make a few distinctions: the categorization of CCA defines two additional laws for ArrowInit instances while ArrowCircuit did not, and the fact that CCA goes beyond what is conventionally considered as a circuit (Liu and Hudak 2010).

More specifically, the commutativity law of ArrowInit states that the order in a parallel arrow composition (∘∘∘) does not matter: side effects are still allowed, but they cannot interfere with each other. The product law further restricts that the effect introduced by init is not only polymorphic in the value it carries, but also commutes with product.

Causal commutative normal form

The five operations from the Arrow, ArrowLoop, and ArrowInit classes (Figure 1) can already be used to construct a wide variety of computations. However, the laws that accompany the operations (Figure 2) make many of these computations equivalent. One way to determine whether two computations are equivalent is to put them into a normal form. The set of laws for CCA indeed forms its axiomatic semantics with which such equivalence can be formally reasoned about. It turns out that all CCCAs can be syntactically translated into either a pure arrow, or a single loop containing one pure arrow and one initial state value, in the following form (Liu et al. 2009, 2011), called causal commutative normal form (CCNF):

![Figure 1: The Arrow, ArrowLoop and ArrowInit classes](image)
The five CCA operations are each used exactly once in CCNF, each representing a different component or composition, leaving no room for further reductions. We save the discussion on the normalization details, and instead refer our readers to Liu et al. (2011) for the actual proof. CCNF can be expressed as a Haskell function:

\[
\text{loop} (\text{arr} f \gg \text{second} (\text{init} i))
\]

Examining the type of the initial state value \( i \) and transition function \( f \) reveals that they closely resemble a form of automata called Mealy machines (G. H. Mealy 1955) that are often used to describe the operational semantics of dataflow programming. Informally, a Mealy machine maps each state \( s \) from a given set \( S \) to a function that produces for every input \( x \) a pair of \((y, s')\), consisting of the output \( y \) and the next state \( s' \). In the above \( \text{loop}D \) form, the value \( i \) becomes our initial state \( s_0 \), and the uncurried form of \( f \) corresponds to the transition function. In this sense, CCNF can be seen as making the connection between the axiomatics semantics of CCA to Mealy machines, an operational semantics for dataflow.

In fact, the data type \( \text{SF} \) for causal stream transformers we describe in Section 1 is exactly a Mealy machine if we look at the type of \( \text{unSF} \) that projects type \( \text{SF} \) \( a \rightarrow b \) to its definition:

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

If we take \( \text{SF} \ a \ b \) as the type of a state, then \( \text{unSF} \) becomes the transition function of a Mealy machine. A natural implication is that \( \text{SF} \ a \ b \) is but one implementation of CCA, or in other words, \( \text{SF} \ a \ b \) can be made an instance of the \( \text{ArrowInit} \) class, for which we will discuss later in Section 4.

**Example: the exp arrow**

To illustrate how CCA and CCNF work in practice, we revisit the \( \text{exp} \) arrow presented in Section 1 in more detail. Figure 3 shows three forms of the Haskell definition for both \( \text{exp} \) and \( \text{integral} \): first in arrow notation, then desugared to arrow combinators, and lastly in CCNF.

Like \( \text{exp} \), the \( \text{integral} \) function is defined as a looping arrow where the incoming derivative \( v \) is integrated to become both the output and the next state value \( i \), which has an initial value of 0. Because \( \text{exp} \) itself contains a recursion, and it is defined in terms of \( \text{integral} \), there are two level nesting of loops. This fact is made more evident in the desugared form if we substitute \( \text{integral} \) into the \( \text{exp} \). However, after being normalized to CCNF, the two loops would collapse into just one, represented through the use of \( \text{loop}D \).

**Normalization and optimization**

It is easy to see how normalizing CCA computations can improve their efficiency. While a CCA computation such as \( \text{exp} \) may involve many uses of the arrow operators, its normal form is guaranteed to have precisely one call to \( \text{loop} \), one call to \( \text{init} \), and so on. If the implementations of these operators are computationally expensive (as is the case for the stream transformer (Section 4)) then reducing the number of times they are used is likely to improve performance.

However, programming with normal forms directly is awkward. For instance, the definition of \( \text{exp} \) in terms of \( \text{integral} \) is math-
Causal commutative arrows revisited

Template Haskell

How might we normalize CCA programs? Normalization is a syntactic property, and so it is natural to consider syntactic means. Earlier work on causal commutative arrows (Liu et al. 2009, 2011) used Template Haskell (Sheard and Jones 2002) to rewrite CCA programs during compilation. Template Haskell’s support for syntactic transformations makes it straightforward to implement a reliable CCA normalizer using the arrow laws of Figure 2a, suitably oriented.

However, the drawbacks of using Template Haskell are also significant enough to make it worthwhile investigating alternative approaches. First, in the current Template Haskell design the representation of expressions is untyped— that is, the type of the representation of an expression does not vary with the type of the expression. (There is work ongoing to incorporate support for typed expressions, but those come with additional restrictions which make it difficult or impossible to express the normalization procedure.) This lack of type checking does not introduce unsoundness in the technical sense, since terms generated by Template Haskell are subsequently type checked, but it can delay the detection of errors, and even allow some errors in the code transformer to remain undetected indefinitely. Second, writing the normalization procedure using Template Haskell involves functions that operate on the normalized program rather than as part of the program, leading to a lack of integration between the normalizing program and the normalized program; besides the fact that their types are unrelated, the two programs also cannot easily share values. Lifting values to the reprensetation layers has many restrictions. One a trick to avoid lifting is to inline an entire definition into the representation layer, but doing so would destroy sharing, which leads to inefficient code being generated.

Normalization by construction

An alternative approach to express transformations is to take advantage of the flexibility of type classes. In place of instance definitions that perform computation we can give definitions that simply construct computations in normal form. The technique involves three ingredients:

The first ingredient is a data type that represents exactly those terms of some type class (Monoid, Applicative, Arrow, etc.) that are in normal form.

The second ingredient is an observation function that turns normalized terms back into polymorphic computations that can be used at a concrete instance.

The final ingredient is an instance for the data type that defines the methods of the class as constructors of terms in normal form.

Readers familiar with normalization by evaluation (NBE) may notice a correspondence between these three ingredients and the model, interpretation in the model, and reification function that form the core of NBE.

First ingredient: a data type CCNF for normal forms

The following data type represents the CCA normal form described in Section 2.3:

```
data CCNF a b where
  Arr :: (a → b) → CCNF a b
  LoopD :: c → ((a, c) → (b, c)) → CCNF a b
```

That is, a normalized CCA computation is either a pure function \( f \), represented as \( Arr f \), or a term of the form \( loop (arr f \gg second (init i)) \), represented as \( LoopD i f \).

The definition of \( CCNF \) uses GADT syntax, but it is not a true GADT, since the type parameters do not vary in the return types of the constructors. However, it is an existential definition: the type variable \( c \) that represents the type of the hidden state in \( LoopD \) does not appear in the parameters.

Second ingredient: an observation function for CCNF

The semantics of the \( CCNF \) data type — that is, the interpretation of a \( CCNF \) value as an \( ArrowInit \) instance — is given by the following function:

```
observe :: ArrowInit arr ⇒ CCNF a b a
observe (Arr f) = f
observe (LoopD i f) = loop (arr f \gg second (init i))
```

That is, given an \( ArrowInit \) instance for some type constructor \( arr, observe \) turns a value of type \( CCNF A b A \) into an arrow computation in \( arr \). A pure function \( Arr f \) is interpreted by the \( arr \) method of \( arr \). A value \( LoopD i f \) is interpreted as a call to \( loopD i \) in \( arr \). For clarity the definition of \( loopD \) is inlined in \( observe \).

Final ingredient: an ArrowInit instance for CCNF

Figures 4–6 define instances of \( Arrow, ArrowLoop \) and \( ArrowInit \) for \( CCNF \).

The definition of these instances is closely related to the CCA laws of Figure 2. It is of course the case that each instance for \( CCNF \) is only valid if it satisfies the laws associated with the class (although this property is assumed rather than enforced). But the relationship between the laws and the definitions is closer here, since the instance definitions may be derived directly from the laws.

Before embarking on the derivation we must first establish an appropriate interpretation of the equality symbol in the equations of Figure 2. There are two sets of instances involved in the derivation namely, the \( CCNF \) instances that we wish to derive, and the arrow instances which we will use to interpret the normal forms using \( observe \). The derivation of the first set of instances depends on the laws for the second set, and so the appropriate notion of equality

\[
\text{exp in arrow notation}
\]

\[
\text{exp = proc } () \rightarrow \text{ do }
\]

\[
\text{rec let } e = 1 + i
\]

\[
i \leftarrow \text{ integral } \leftarrow e
\]

\[
\text{returnA } \leftarrow e
\]

\[
\text{integral = proc } v \rightarrow \text{ do }
\]

\[
\text{rec } i \leftarrow \text{ init } 0 \leftarrow i + dt \ast v
\]

\[
\text{returnA } \leftarrow i
\]

\[
\text{exp desugared}
\]

\[
\text{exp = loop}
\]

\[
(\text{second } (\text{integral } \gg \text{arr } (+1)) \gg \\
\text{arr snd } \gg \text{arr } (\lambda x \rightarrow (x, x))
\]

\[
\text{integral = loop}
\]

\[
(\text{arr } (\lambda (v, i) \rightarrow i + dt \ast v) \gg \\
\text{init } 0 \gg \text{arr } (\lambda x \rightarrow (x, x))
\]

\[
\text{exp normalized}
\]

\[
\text{exp = loopD } 0 (\lambda (x, y) \rightarrow \text{ let } i = y + 1
\]

\[
\text{ in } (i, y + dt \ast i))
\]

Figure 3: From arrow notation to CCA normal form
is a semantic one, namely equality under observation, where \( f \) and \( g \) are considered equivalent if \( \text{observe } f \) is equivalent to \( \text{observe } g \). In other words, we can replace \( \text{Arr} \) and \( \text{LoopD} \) with the corresponding right hand sides (from the definition of \( \text{observe} \)) in the instance definitions, and then use the arrow laws (Figure 2) to relate the right hand and left hand sides of the methods in the definitions in Figures 4–6.

Figures 7–9 show parts of the derivations for the \( \text{Arr} \), \( \text{ArrowLoop} \) and \( \text{ArrowInit} \) methods for \( \text{CCNF} \). The full derivations follow a similar pattern of equational reasoning about the observed normalized terms.

Figure 7 derives part of the definition of \( \gg \) for \( \text{CCNF} \) (Figure 4), namely the second case:
\[
\text{Arr } f \gg \text{LoopD } i \ j \ g = \text{LoopD } i \ (g \ . \ f \times \text{id})
\]

As described above, the derivation is based on the behaviour of normal forms under observation, and so we begin by replacing \( \text{Arr} \) with \( \text{loop} \) and \( \text{LoopD} \) with \( \text{loopD} \). The remainder of the derivation is a straightforward application of the left tightening, extension and composition laws (Figure 2).

Figure 8 derives part of the definition of \( \text{loop} \) for \( \text{CCNF} \), namely the first case:
\[
\text{loop } (\text{Arr } f) = \text{Arr } (\text{trace } f)
\]

This time the derivation is even simpler; under observation the left and right sides of the definition become exactly the left and right sides of the extension law of Figure 2.

Finally, Figure 9 shows the derivation of the definition of \( \text{init} \) for \( \text{CCNF} \):
\[
\text{init } i = \text{LoopD } i \ \text{swap}
\]

This last derivation is a little longer, due mostly to the administrative shuffling involved in converting \( \text{second} \) to \( \text{first} \) and eliminating the resulting \( \text{arr swap} \) terms.

**Normalization summary** We have seen the derivation of the normalizing instances. Before moving on to consider further optimizations, let us briefly review their use in programming with arrows.

In order to normalize a computation such as \( \exp \) that is polymorphic in the \( \text{ArrowInit} \) instance, nothing in the definition of the computation needs to change; the author of \( \exp \) can entirely ignore the issue of normalization.

In order to call (i.e. run) \( \exp \), the caller must instantiate the \( \text{ArrowInit} \) constraint. Instantiation is typically implicit, since the type of the context in which \( \exp \) is used is sufficient to select the appropriate instance. However, in order to normalize \( \exp \) before running it the caller must instantiate the constraint twice, first with \( \text{CCNF} \) (by calling \( \text{observe} \)) to obtain a normalized version of \( \exp \), and then with another \( \text{ArrowInit} \) instance, such as \( \text{SF} \).

The original program (such as \( \exp \)) might use the arrow operations many times. However, the definitions of \( \text{CCNF} \) and \( \text{observe} \) guarantee that the \( \text{SF} \) definitions of \( \text{init} \), \( \text{loop} \) and \( \text{second} \), \( \text{arr} \) and \( \gg \) will be applied at most once each. Interposing the \( \text{CCNF} \) instance in this way makes it possible to reduce the number of uses of the arrow operations when running any \( \text{ArrowInit} \) computation.
Optimizing observation

Section 3 showed how to improve the performance of CCA programs by taking advantage of a universal property: every CCA computation can be normalized into a form where each of the five operations occurs exactly once. In this section we move from the general to the specific, and show that much more significant improvements are available if we take advantage of what we know about the context in which a normalized term is used. (The actual improvements resulting from normalization and the changes in this section are quantified in Section 6.)

More specifically, we will derive an optimized version of the polymorphic observe function from Section 3 that uses three opportunities for specialization:

First, we instantiate the ArrowInit constraint in observe to a particular arrow instance (namely SF), replacing the calls to the polymorphic arrow operators with calls to the SF implementations of those operators. This instantiation gives us an observation function which is specialized for the SF arrow.

Second, we make use of the normal form to merge the SF arrow combinators together. Since the observed computation is always in normal form we know, for example, that there is always exactly one use of loop, which is always applied to a term of the same shape. We use this knowledge to derive more efficient versions of the SF arrow operations that are specialized to their arguments.

Finally, we fuse together observe with the context in which it is used. More specifically, noting that observe is typically used in conjunction with an interpretation of SF as stream transformers, we fuse together the optimized observation function with the observation function for streams, which turns an SF value into a transformer on streams. In fact, we then go further still, and build an observation function that is optimized for accessing individual stream elements. In other words, we can build a function of the following type:

\[
\text{ArrowInit} \text{arr} \Rightarrow \text{arr a b} \rightarrow \text{Int} \rightarrow [a] \rightarrow b
\]

that normalizes a CCA computation, and observes particular elements that result from instantiating it as a stream transformer.

The SF Arrow instances Figures 10–12 define the Arrow, ArrowLoop and ArrowInit instances for the SF type introduced in Section 1.

Signal functions are described in considerable detail in the literature (?), and so we will summarize their behaviour only briefly here.

An SF (signal function) value is a function which, when applied to a value, returns a pair of a new value and a new signal function to be used as the continuation. The arr operator (Figure 10) constructs a pure signal function, where the new signal function returned at each application is unchanged. The \(\Rightarrow\) operator composes two signal functions \(f\) and \(g\), threading the argument \(x\) first through \(f\) and then through \(g\), and composing the continuations.

The first operator builds a new signal function from an existing signal function \(f\), and threads some additional state \(z\) alongside the value which is passed to \(f\). The loop operator (Figure 11) connects the second output of its argument arrow \(sf\) as the second input to the same arrow, forming a value-level loop for \(sf\), as well as all its continuations. The init operator (Figure 12) outputs the initial value, while passing the current input to its own continuation as the next value to output, essentially forming an internal state captured in a closure.

One point of note is that all these functions — even arr — are fundamentally recursive, which makes computations built by composing them consequently challenging for a compiler to optimize.

From unoptimized to optimized observation Although optimizing the observation function is difficult for the compiler, we can achieve significant performance improvements by reasoning about it ourselves. Figure 13 shows the path from the unoptimized observation function for CCNF to an optimized version, following the threefold derivation outlined above.

The first step is to instantiate the ArrowInit-constrained variable arr in the type of observe with SF. It is sufficient to give observe a more specific type, but for clarity we also explicitly suffix the class methods — loopSF for loop, arrSF for arr; and so on. At this stage we also perform some minor additional simplifications, expanding the call to second into the primitive computations first, arr and \(\Rightarrow\), and combining the resulting adjacent calls to arr using the composition law.

From this point onwards we will confine our attention to the case for LoopD in the definition of observeSF, since the case for Arr is too simple to expect significant performance improvements.

Next, we name the subexpressions in the definition of observe using a where clause (ensuring that functions remain fully applied in each case). Naming subexpressions makes it easier to specialize applications to known arguments in the next step, and additionally eases the subsequent rewriting of recursive definitions. Here is an example, starting from the following definition, which appears in the definition of observeSF after subexpressions are named:

\[
\text{first_init} = \text{firstSF} \ (\text{initSF} \ i)
\]

Substituting the definition of initSF results in the following pair of definitions

\[
\text{first_init} = \text{firstSF} \ (\text{SF} \ (f \ i)) \ f \ i = (i, \text{SF} \ (f \ x))
\]

Next, substituting the definition of firstSF produces the following set of definitions:

\[
\text{first_init} = \text{SF} \ (g \ (\text{SF} \ (f \ i))) \ f \ i = (i, \text{SF} \ (f \ x)) \ g \ f \ (x, z) = \text{let} \ (y, f') = \text{unSF} \ f \ x \ in \ ((y, z), \text{SF} \ (g \ f'))
\]
Unoptimized observe

\[
\text{observe} \; :: \; \text{ArrowInit arr} \Rightarrow \text{CCNF a b} \rightarrow \text{arr a b}
\]
\[
\text{observe} (\text{LoopD i f}) = \text{loop} (\text{arr f} \gg second (\text{init i}))
\]

Instantiating with SF (with second expanded)

\[
\text{observeSF} :: \text{CCNF a b} \rightarrow \text{SF a b}
\]
\[
\text{observeSF} (\text{Arr f}) = \text{arrSF f}
\]
\[
\text{observeSF} (\text{LoopD i f}) = 
\begin{cases} 
\text{loopSF} (\text{arrSF (swap . f)} \gg SF \text{firstSF} (\text{initSF i})) \\
\text{arrSF swap}
\end{cases}
\]

Naming subexpressions (LoopD case only)

\[
\text{observeSF} (\text{LoopD i f}) = \text{loopcomp2 i f}
\]

where

\[
\begin{align*}
\text{arrSF swap} &= \text{arrSF (swap . f)} \\
\text{arrSF swap} &= \text{arrSF swap}
\end{align*}
\]

first_init i = first_SF (SF (h1 i))

where

\[
\begin{align*}
h1 i x &= (i, SF (h1 x)) \\
\text{...}
\end{align*}
\]

The optimized observeSF

\[
\text{observeSF} (\text{LoopD i f}) = \text{loopD i f}
\]

where

\[
\begin{align*}
\text{loopD i f} &= SF (\lambda x \rightarrow \text{let } (y, i') = f (x, i) \\
&\text{in } (y, \text{loopD i' f}))
\end{align*}
\]

Merging in runSF

\[
\text{runCCNF} :: \text{CCNF a b} \rightarrow [a] \rightarrow [b]
\]
\[
\text{runCCNF} (\text{LoopD i f}) = g i f
\]

where

\[
\begin{align*}
g i f (x : xs) &= \\
&\text{let } (a, b) = f (x, i) \\
&\text{in } a : g b f xs
\end{align*}
\]

Merging in !!

\[
\text{nthCCNF} :: \text{Int} \rightarrow \text{CCNF} () a \rightarrow a
\]
\[
\text{nthCCNF} (\text{LoopD i f}) = \text{aux n i}
\]

where

\[
\begin{align*}
\text{aux n i} &= \text{if } n \equiv 0 \text{ then } x \text{ else } \text{aux} (n - 1) i' \\
\text{where } (x, i') &= f ((), i)
\end{align*}
\]

Handling mutable states

Up to this point, our treatment of state has been purely functional: the init operator extends a pure computation with internal state, and the transition function in a CCNF maps one state to another in a purely functional manner. Threading state through computations in this way is reminiscent of the state monad, which is defined in Haskell as follows:

\[
\begin{align*}
type State s a &= s \rightarrow (a, s) \\
\text{instance Monad (State s) where}
\end{align*}
\]
The Monad instance of `State s` ensures that all operations on the internal state of type `s` are sequentially ordered: monadic composition passes the state along in a linear manner, guaranteeing a deterministic result.

For some programs the encoding of mutable state via threading is unacceptably inefficient. For such cases the `ST` monad (Launchbury and Peyton Jones 1994) offers an interface to genuinely mutable state. Figure 14 shows the `ST` monad and its related operations in Haskell. Conceptually, we view the type `ST s a` as follows:

```haskell
type ST s a = State s a
```

where the type `s` is phantom – i.e. used only for type safety, not as actual type of any data in the program. In the actual `ST` library, the type `ST` is abstract, so that users cannot directly access values of type `State s`; instead they must access the hidden state via the primitive operations on the `STRef` type in Figure 14.

`ST` comes with a number of useful guarantees. First, since `ST s` is a legitimate instance of the `Monad` class, the primitive operations on `STRef` values are guaranteed to be sequenced. Further, the type of the observation function, `runST`, ensures that the type variable `s`, which is used to index the `STRef` values used in a computation, cannot “escape” into the surrounding context. Since the types of references (and hence the references themselves) can not be accessed outside the call to `runST`, mutations to `STRef` values constitute a benign effect: computations in `ST` are indistinguishable from pure terms.

**Implement CCA with ST monad**

Giving the similarity between CCA and the state monad, one would wonder if there is also a corresponding implementation that allows mutable states as an abstraction over state variables.

The idea here is that instead of passing immutable state as values, we have an `ST` action that initializes the mutable state, which becomes the first argument to `LoopDST`. The type variable `e` here can be any mutable type allowed in an `ST` monad, for instance, `STRef`. The state transition function will then takes the mutable object of type `e`, and an input of type `a` to compute the output of type `b`, all in an `ST` monad threaded by the same phantom variable `s` as used by the initialization. Note that there is no state being returned as a result, because the transition function can directly mutate it in-place.

```haskell
data CCNFST s a b where
  ArrST : (a → b) → CCNFST s a b
  LoopDST : ST s e → (e → a → ST s b) → CCNFST s a b

instance Arrow (CCNFST s) where
  arr = ArrST
  ArrST f g = ArrST (g . f)
  ArrST f g = LoopDST i g = LoopDST i h
  where h i = g i . f
  LoopDST i f g = LoopDST i h
  where h i = fmap g . i
  LoopDST i f g = LoopDST i k h
  where k = hM2 (.) i j
        h (i, j) x = f i x ≫ g j
  first (ArrST f) = ′ArrST (first f)
  first (LoopDST i f) = ′LoopDST i g
  where g i (x, y) = hM (., y) (f i x)

instance ArrowLoop (CCNFST s) where
  loop (ArrST f) = ′ArrST (trace f)
  loop (LoopDST i f) = ′LoopDST i h
  where h i x = do
    rec (y, j) ← f i (x, j)
    return y
  ArrowInit (CCNFST s) where
    init i = ′LoopDST (newSTRef i) f
    where f i x = do
      y ← readSTRef i
      writeSTRef i x
      return y
```

Figure 15: An `ST` monad based CCA implementation

It becomes clear if we look at the definition of `ArrowInit` instance for `CCNFST` in Figure 15, where the `init` arrow uses `newSTRef i` as the action to initialize a mutable reference of the `STRef` type, which is then passed to the transition function `f` that can read and write to it. Other instance declarations in Figure 15 are mostly straightforward, where the sequential composition of two `LoopDST` actions is just composition of two initialization actions, and two transition actions. The `ArrowLoop` instance of `loopST` make use of recursive monad (hence the `rec` keyword) to “tie-the-knot” between the two input and the two output values of this arrow, because `ST` monad is a valid instance of `MonadFix`, where value-level recursion is implemented by `fixST` (Figure 15).

Any generic CCA can be instantiated to type `CCNFST`, and all we need is a way to run them. We give the following definition of the `nth` element in the output stream of an `CCNFST s` arrow taking no input:

```haskell
data ST s a
instance Monad (ST s)
  runST :: (forall s . ST s a) → a
  fixST :: (a → ST s a) → a

data STRef s a
newSTRef :: a → ST s (STRef s a)
readSTRef :: STRef s a → ST s a
writeSTRef :: STRef s a → a → ST s a

Figure 14: ST monad and mutable references
```
As with runST, nthST uses existential type to enclose the phantom type variable s, and the helper function nthST takes care of the actual unfolding. It should be noted that all initialization of the mutable state happens only once outside of the actual iteration function next because new state values needs to be passed around: they are mutated in place.

**Proving CCA laws for CCNFST**

Implementing CCA using ST monad may have given us the access to simulate traveling wave and its reflection. We omit the definitions of these helper functions here, and instead refer our readers to the textbook. After unfolding both sides into the observe function, we are left to prove that the monadic sequencing of s₁ and s₂, or f and g actually commutes. At this point we require the following property for STRef objects in our implementation of nthST.) The fact that the only effectful operation in the CCNFST arrow is STRef completes this proof.

**Product law proof sketch** To prove the product law holds for the CCNFST arrow, we again have to resort to extensionality. It comes down to proving using a pair of two distinct STRefs is equivalent to using a one STRef of a pair, which is a reasonable assumption that we can make about mutable references for all practical purposes. We omit the proof detail here.

**Application: sound synthesis circuits**

A popular application of arrows is in the domain of audio processing and sound synthesis. Both Yampa (Giorgidze and Nilsson 2008) and Euterpea (Hudak et al. 2015) are arrow based DSLs that have been successfully applied to modeling sound generating circuits. In digital audio, sound waves are usually produced at a preset signal rate, for instance, the standard frequency is 44100Hz. Hence circuits for sound synthesis often fits well into a synchronous dataflow model, where the unit of time corresponds to the inverse of signal rate. Like electronic circuits, circuits for sound synthesizers have feedback loops. Besides unit delays, they often have to delay signals on the wire for a certain time interval, which conceptually is equivalent to piping a discretized audio data stream through a buffered queue of a given size that is greater than 1. This is what is commonly known as a delay line. We can extend the ArrowInit class to provide this new operation:

```haskell
class ArrowInit arr ⇒ BufferedCircuit arr where
  initLine :: Int → a → arr a a
  delayLine :: (Num a, BufferedCircuit arr) ⇒ Time → arr a a
  delayLine t = initLine (floor (t / sr)) 0
```

The delayLine function takes a time interval, and returns an arrow of the BufferedCircuit class that carries internally a buffer of a size calculated from the standard audio signal rate sr, and initial value 0. The first argument to initLine specifies the size of this buffer, and the second argument is the initial value to initialize buffer. Conceptually a delay line of size n is equivalent to n unit delays, or we can state it as:

```
initLine n i = foldr1 (≫) (replicate n (init i))
```

However, the above definition does not make an efficient implementation, and this is where our ST monad based CCA implementation comes in handy: because a size n buffer can be directly implemented as a size n mutable vector as follows, where Vector stands for the mutable vector module from the Haskell vector package:

```
instance BufferedCircuit (CCNFST s) where
  initLine size i = LoopDST newBuf updateBuf
  where
    newBuf = do
      b ← Vector.newSize
      Vector.set b i
      r ← newSTRef 0
      return (b, r)
    updateBuf (b, r) x = do
      i ← readSTRef r
      x' ← Vector.unsafeRead b i
      Vector.unsafeWrite b i x
      let i' = if i + 1 ≥ size then 0 else i + 1
      writeSTRef r i'
      return x'
```

The internal state to initLine is a tuple (b, r) where b is a mutable vector that acts as a circular buffer, and r is a STRef storing the position to read the next buffered value. We increment the position value stored in r by 1 whenever a new input comes in. Because this position wraps around and is guaranteed to be always in the range of [0, size], direct use of the non-bounds checking unsafeRead and unsafeWrite operation would still be safe.

Since initLine is implemented as a CCNFST, we automatically gain the ability to optimize all buffered circuits by normalizing them, because all CCNFST arrows are valid CCAs by construction. As a comparison, the existing Euterpea implementation also uses mutable arrays under the hood, but has to rely on unsafePerformIO to operate them, which actually triggers a subtle correctness bug when GHC optimization is turned on. For this reason, we consider our implementation of CCNFST as a safe and sound alternative to implementing arrow-based audio and sound processing circuits.

Finally, we give two sample synthesis programs in Figure 16 that we also use to measure the performance in the next section. They are direct ports from Euterpea with little modification.

The flute function simulates the physical model of a slide-flute. It takes a set of parameters that controls various aspects of the output sound wave, and it uses a number of helper functions to process signal source from random white noise, control singal envelope using segmented line and so on, where delayLine is used to simulate traveling wave and its reflection. We omit the definitions of these helper functions here, and instead refer our readers to Cheng and Hudak (2009) for more details.

The shepard function takes a list and returns a list, but has an intriguing structure: it is a recursively defined arrow. It takes a duration in seconds as input, and additively builds up an oscillating wave signal by summing up the signal returned from
In this section, we study the performance characteristics of different Performance measurement and analysis arrow’s structural components, not just its input or output. Benchmarks and measuring methods by measuring the running time of 8 benchmark programs. We compare with the existing Template Haskell based normalization. Of course, these settings are not guaranteed to improve the performance of all arrow programs. Overall benchmarking result

Performance measurement and analysis

In this section, we study the performance characteristics of different CCA interpretations including $SF$, $CCNF$, and $CCNF_{ST}$, which we compare with the existing Template Haskell based normalization by measuring the running time of 8 benchmark programs.

Benchmarks and measuring methods

We use the following benchmarks:

- A micro-benchmark $fib$ that computes the Fibonacci sequence using big integers.

- All the micro-benchmarks discussed in Liu et al. (2009), including $exp$, a $sine$ wave with fixed frequency using Goertzel’s method, an $oscSine$ wave with variable frequency, the 50’s sci-fi sound synthesis program from Giordide and Nilsson (2008), and the robot simulator from Hudak et al. (2003).

- The $flute$ and $shepard$ sound synthesis from Section 5.3. We consider these macro-benchmarks due to their complexity and their reliance on mutable state for efficiency. Since both use $delayLine$, we additionally defined a $BufferedCircuit$ instance for $SF$ and $CCNF$ as well, where the use of $unsafePerformIO$ is unavoidable. The use of $unsafePerformIO$ in Euterpea leads to incorrect behaviour when GHC optimization is enabled; our $ST$-based implementation of $CCNF$ avoids the need for $unsafePerformIO$, eliminating the bug.

Our use of the “micro-” and “macro-benchmark” terminology is by no means scientific, and must be taken in a relative context. Our benchmark programs are written with arrow notation as generic computations parameterized by an arrow type variable. We can therefore reuse the same source for the $SF$, $CCNF$, and $CCNF_{ST}$ versions of these benchmarks, which differ only in the observation functions. For the Template Haskell versions, we first desugar all programs from arrow notation into arrow combinators using a preprocessor from the publicly available CCA package, which is also used to normalize these programs to pairs of initial value and transition function. The normalized programs are then sampled with the same $nthCCNF$ function used for $CCNF$ arrows.

For the benchmarking we use the Criterion benchmarking package to measure the time taken for the $nth$ function to compute $44100 \times 5 = 2,205,000$ samples, i.e. 5 seconds of wave signals. All benchmarks are compiled with GHC 7.10.3 using the flags $-O2 -funfolding-use-limit=512$ on a 64-bit Linux machine with Intel Xeon CPU E5-2680 2.70GHz.

To ensure the best performance for each implementation, we additionally annotate all generic arrow computations with $SPECIALIZE$ and $INLINE$ pragmas, though these are typically not strictly required. The compilation flag $-funfolding-use-limit=512$ flag prompts GHC to inline larger terms than it would by default, including the substantial arrow terms which can result from normalization. Of course, these settings are not guaranteed to improve the performance of all arrow programs.

Overall benchmarking result

Figure 17 shows the benchmarking result for the 8 programs under each of the four implementations ($SF$, $CCNF$, $CCNF_{ST}$, and Template Haskell). We report both the mean kernel time and the relative performance ratio using $\frac{SF}{CCNF}$ as a baseline. Of course, these settings are not guaranteed to improve the performance of all arrow programs.

```
flute :: BufferedCircuit a ⇒ Time ⇒ Double ⇒
  Double ⇒ Double ⇒ Double ⇒ a () Double
flute dur amp fqc press breath =
  proc () → do
  env1 ← envLineSeg [0, 1.1 * press, press, press, 0]
  [0.06, 0.2, dur − 0.16, 0.02] −≺ ()
  env2 ← envLineSeg [0, 1, 1, 0]
  [0.01, dur − 0.02, 0.01] −≺ ()
  envib ← envLineSeg [0, 0, 1, 1]
  [0.5, 0.5, dur − 1] −≺ ()
  flow ← noiseWhite 42 −≺ ()
vib ← osc sineTable 0 ≺ 5
  let emb = breath * flow * env1 + env1 +
      vib * 0.1 * envb
  rec flute = delayLine (1 / fqc) −≺ out
  x ← delayLine (1 / fqc / 2) −≺ emb + flute * 0.4
  out ← filterLowPassBW −≺
    (x − x * x + x + flute * 0.4, 2000)
  returnA = out * amp + env2
shepard :: BufferedCircuit a ⇒ Time ⇒ a () Double
shepard seconds = if seconds ≤ 0
then arr (const 0.0)
else proc _ → do
  f ← envLineSeg [800, 100, 100] [4.0, seconds] −≺ ()
  e ← envLineSeg [0, 1, 0, 0] [2.0, 2.0, seconds] −≺ ()
  s ← osc sineTable 0 ←≺ f
  r ← delayLine 0.5 ≺ shepard (seconds − 0.5) −≺ ()
  returnA = (e * s * 0.1) + r
```

Figure 16: flute and shepard synthesis program

```
To measure the normalization contribution, the same arrow programs are normalized by the \texttt{observe} function to a generic arrow and then specialized to the \textit{SF} type and sampled by \texttt{nthSF}. A percentage is calculated by comparing with the full CCNF implementation. The rest speedups can then be attributed to specializing to the \textit{SF} type and sampled by the optimized \texttt{nthCCNF} function. We show this percentage of performance contribution in Figure 18, which is sorted from left-to-right in an ascending order of the contribution percentage of normalization.

We observe that normalization contributes a bigger percentage to the overall speedup for \texttt{fib}, \texttt{flute} and \texttt{shepard}, where the amount of real computation greatly outweighs the remaining overhead in an normalized \textit{SF} arrow. This is to be expected, and hence the graph is a good indication of what kind of workloads are likely to benefit more from normalization than specialization. It is also not a coincidence that the four benchmarks to the left of the graph, \texttt{exp}, \texttt{sine}, \texttt{oscSine} and \texttt{robot}, graph are also the ones seeing most significant speedups (from 60× to 242×) in Figure 17, where eliminating the final arrow overheads has a greater to their performance.

**Analyzing the performance of CCNF\textsubscript{SF}**

The performance of CCNF\textsubscript{SF} also begs for more explanation. Looking at the time difference between CCNF\textsubscript{SF} and CCNF for oscSine, sci-fi and robot, we notice an intriguing correlation between the kernel time and the number of loops in a program: each loop accounts for about 80ms difference between CCNF\textsubscript{SF} and CCNF. The explanation, however, is rather simple. We translate the \texttt{loop} combinator for CCNF\textsubscript{SF} into a recursive ST monad, which corresponds to a call to \texttt{fixST}. Examining the compiled code in GHC Core for our benchmarks reveals that a recursive data structure \textit{STRep} remains in the program for every \texttt{fixST}, preventing GHC from statically optimizing recursions at the value level.

As we move to more complex programs, however, the situation dramatically changes: the CCNF\textsubscript{ST} implementation becomes twice as fast as CCNF for both flute and shepard. While the Template Haskell version may appear to still lead the performance for flute and only slight lags behind for shepard, this is actually no longer the case as program complexity increases further. Since shepard is a recursively defined arrow, it is straightforward to increase its computational complexity by increasing input size. Figure 19 compares the running time for shepard with the CCNF, CCNF\textsubscript{SF} and Template Haskell implementations. The X-axis shows different input sizes, where every 0.5 second increment corresponds to 8 additional states and 5 additional loops. The Y-axis shows the output rate, i.e., the number of samples produced per second.

Figure 19 shows that as input size increases, the output rate of all implementations reduces in inverse proportion. As the input size increase, CCNF stays around half the speed of CCNF\textsubscript{ST}, while the relative speed of the Template Haskell implementation plummets. From left to right, the Template Haskell version goes from 80% of CCNF\textsubscript{ST} performance to only half. There are clearly overheads with the Template Haskell version that are not present in either CCNF or CCNF\textsubscript{ST}. Our understanding is that the Template Haskell implementation of normalization relies on the ability to expand the entire arrow at compile time. In contrast, both CCNF and CCNF\textsubscript{ST} are able to perform normalization at runtime, and although not all arrow structures are statically optimized away, computations at individual components are shared rather than expanded and duplicated as in the Template Haskell case.

Moreover, the reason that CCNF\textsubscript{ST} performs better than CCNF for real workloads is that implementing mutable states...
through the $ST$ monad has an advantage “at scale”: it avoids building up large number of nested tuples at runtime. Comparing their respective sampling functions $nthST$ and $nthCCNF$, we find that $nthCCNF$ has to pass the new state as a nested tuple at every iteration only for it to be destructed by the transition function again. In contrast, $nthST$ constructs a single nested tuple of mutable references at the initialization stage; there is no need to re-construct a fresh tuple to pass along in every iteration.

**Related work**

**Representing arrow computations as data** The technique of representing arrow computations with a data type in order to perform optimize computations using the laws appears several times in the literature. Hughes (2005) gives a representation of arrow computations that can be used to eliminate the composition of adjacent pure computations, and suggests extending the technique further, but does not measure performance improvements. Nilsson (2005) uses the first four arrow laws (left and right identity, associativity and composition) together with a first-order representation of SF to optimize Yampa, and achieves some modest performance improvements (up to around 2x). Yallop (2010) shows how to use the laws together with a data type for representing normal forms to fully normalize Arrow (but not ArrowLoop or ArrowInit) computations, but does not report any performance improvements.

In each case, the key insight that the normal form enables further optimizations in the observation function seems to be missing; it is this insight that led to the most significant performance improvements in our benchmarks (Figure 18, Section 6.3).

**Generalized arrows** This paper focuses on the optimization of arrow computations, paying relatively little attention to the pure functions which are lifted into computations using the $arr$ operator, although the efficient compilation of these functions is often crucial to performance. Joseph (2014) describes a generalization of the Arrow class which makes it possible to explicitly represent many pure functions in order to support non-standard compilation strategies such as compilation to hardware. It would be interesting to see whether the generalized arrow interface can further improve the results described in this work.

**Deriving implementations of instances and functions** The technique of deriving implementations by equational reasoning, whether of type class instances using the class laws (Section 3), and for functions using the standard equations of the language (Section 4) apparently originates with Hughes (1995), but is now standard, and used to great effect in many places in the functional programming literature. Hinze (2000) is an early and representative example of deriving a general purpose instance (of a monad transformer) by equational reasoning using the laws associated with the class.

**“Free” representations** As Section 3.1 mentions, our normal form representation can be viewed as a “free” representation of arrow computations. Several researchers have investigated transformations involving free representations to optimize (typically monadic) computations, and for related applications. Voigtlander (2008) uses an optimized instance to reassociate computations over free monads to improve their asymptotic complexity from quadratic to linear. Kiselyov and Ishii (2015) uses so-called freer monads (which liberate free monads from the Functor constraint) as a basis for an optimized implementation of extensible effects, and includes an extensive review of previous occurrences of similar constructions in the literature.

**Earlier work on CCA** Finally, we have already devoted considerable space to the previous work on causal commutative arrows and their optimization (Liu et al. 2009, 2011). An early version of the instance-based normalization presented here is given in Liu (2011), but the author did not observe any performance improvements using the technique.

**References**


