Causal Commutative Arrows Revisited

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

Hai Liu
Intel Labs, USA
hai.liu@intel.com

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 D.1.1 [Programming techniques]: Applicative (Functional) Programming

Keywords arrows, stream transformers, optimization, equational reasoning, type classes

1. 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:

\[
\begin{align*}
\text{exp} & \equiv \text{proc } \cdot \rightarrow \text{do} \\
\text{rec let } e & = 1 \rightarrow i \\
& \quad \text{do} \\
& \quad i \leftarrow \text{integral } \leftarrow e \\
& \quad \text{return } A \leftarrow e
\end{align*}
\]

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 \( e \) at the nock into \( i \) at the head. The name \( e \) appears twice more, once above the arrow as the successor of \( i \), 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 \( \text{arr}, \gg\gg, \text{first, loop, and} \text{init} \) 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 \text{causal stream transformer SF} (Liu et al. 2009) that forms the basis of \text{signal functions} in the Yampa domain-specific language for functional reactive programming (Hudak et al. 2003):

\[
\begin{align*}
\text{newtype SF a b} & \equiv \text{SF } \{ \text{unSF} :: a \rightarrow (b, SF a b) \}
\end{align*}
\]

(The accompanying instances for \( SF \), which define the arrow operators, appear on page 6.) Instantiating \( \text{exp} \) with \( SF \) 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 formalized as a more restricted form of arrows, called causal commutative arrows (CCA), and implemented as a Template Haskell library, 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. A 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.

1.1 Contributions
Section 2 reviews Causal Commutative Arrows (CCA), their definition as a set of Haskell type classes, the accompanying laws, and
class Arrow (arr :: * → * → *) where
    arr :: (a → b) → arr a b
    (≫) :: arr a b → arr b c → arr a c
    first :: arr a b → arr (a, c) (b, c)

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

class ArrowLoop arr ⇒ ArrowInit arr where
    init :: a → arr a a

Figure 1: The Arrow, ArrowLoop and ArrowInit classes

their normal form (CCNF). The contributions of the remainder of
this paper are as follows:

- We derive a new implementation of CCA normalization, re-
alised 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 in-
stances 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.

2. 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.

2.1 Arrows

Arrows (Hughes 2000) are a generalization of monads that relax
the linearity constraint, while retaining a disciplined style of com-
position. Like monads in Haskell, the type of computation captured
by arrows is expressed through a type class called Arrow, shown
in Figure 1 together with diagrams describing its three combina-
tors. 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 com-
putations 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 par-
allel 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:

```haskell
second :: Arrow arr ⇒ arr a b → arr (c, a) (b, c)
second f = arr swap ≫ first f ≫ arr swap
where swap (a, b) = (b, a)
```

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

```haskell
(⋆⋆⋆) :: Arrow arr ⇒ arr a b → arr c d → arr (a, c) (b, d)
(f ⋆⋆⋆ g) = first f ≫ second g
```

Together, these combinator forms an interface to first-order com-
putations, i.e. computations which do not dynamically construct or
change their compositional structure during the course of their ex-
cution.

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 these nine arrow laws can be reduced to eight.) It is worth not-
ing 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 af-
factoring the computation.

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

2.2 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 its
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 adds the additional restriction that the effect
introduced by init is not only polymorphic in the value it carries,
but also commutes with product.

2.3 Causal Commutative Normal Form

The five operations from the Arrow, ArrowLoop, and ArrowInit
classes (Figure 1) can be used to construct a wide variety of com-
putations. However, the laws that accompany the operations (Fig-
ure 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 ax-
iomatic semantics with which such equivalence can be formally
reasoned about. It turns out that all CCAs can be syntactically
translated into either a pure arrow, or a single loop containing one
Pure arrow and one initial state value, which is called causal commutative normal form (CCNF) (Liu et al. 2009, 2011): 

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

The five CCA operations are 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:

\[
\text{loopD} :: \text{ArrowInit} \Rightarrow (f, c) 
\]

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 \( \text{loopD} \) 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 axiomatic semantics of CCA to Mealy machines, an operational semantics for dataflow.

In fact, the data type \( SF \) for causal stream transformers we describe in Section 1 is a form of Mealy machine, as witnessed by the type of \( \text{unSF} \) that projects type \( SF a b \) to its definition:

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

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

2.4 Example: the \( \text{exp} \) 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 finally 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 nested levels of \( \text{loops} \). 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 collapse into just one, represented through the use of \( \text{loopD} \).

3. 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


**exp in arrow notation**

\[
\begin{align*}
\text{exp} & = \text{proc} (i) \to \text{do} \\
\text{rec \ let} \ e & = 1 + i \\
 & \quad i \leftarrow \text{integral} \leftarrow e \\
\text{return} & \leftarrow e \\
\text{integral} = \text{proc} \ v \to \text{do} \\
\text{rec} \ i & \leftarrow \text{init} 0 \leftarrow i + dt \ast v \\
\text{return} & \leftarrow i
\end{align*}
\]

**exp desugared**

\[
\begin{align*}
\text{exp} & = \text{loop} \\
& \quad (\text{second} \ (\text{integral} \gg \text{arr} \ (+1))) \gg \\
& \quad \text{arr \ snd} \gg \text{arr} \ (\lambda x \to (x, x)) \\
\text{integral} = \text{loop} \\
& \quad (\text{arr} \ (\lambda (v, i) \to i + dt \ast v) \gg \\
& \quad \text{init} 0 \gg \text{arr} \ (\lambda x \to (x, x))
\end{align*}
\]

**exp normalized**

\[
\begin{align*}
\text{exp} & = \text{loopD} 0 \ (\lambda (z, y) \to \text{let} \ i = y + 1 \\
& \quad \text{in} \ (i, y + dt \ast i))
\end{align*}
\]

**Figure 3:** From arrow notation to CCA normal form

Many uses of the arrow operators, its normal form is guaranteed to have precisely one call to loop, one call to init, and so on. If the implementations of these operators are computationally expensive (as is the case for the stream transformer SF, 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 exp in terms of integral is mathematically familiar, and emphasizes code re-use and modularity. The normalization property, however, is not modular: inserting a normalized term as a subexpression of another normalized term is not generally guaranteed to produce a term in normal form. It is therefore much more convenient to program with the standard set of arrow operations and treat normalization as a separate step.

**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 these 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 representation layers has many restrictions. One 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.

### 3.1 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 (Monad, 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 by constructing 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:

\[
\begin{align*}
\text{data \ CCNF} & \ a \ b \ \text{where} \\
\text{Arr} :: (a \to b) \to \text{CCNF} \ a \ b \\
\text{LoopD} :: c \to ((a, c) \to (b, c)) \to \text{CCNF} \ a \ b
\end{align*}
\]

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

The definition of \( \text{CCNF} \) 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 \( \text{LoopD} \) does not appear in the parameters.

**Second ingredient: an observation function for CCNF** The semantics of the \( \text{CCNF} \) data type — that is, the interpretation of a \( \text{CCNF} \) value as an ArrowInit instance — is given by the following function:

\[
\begin{align*}
\text{observe} :: \text{ArrowInit} \Rightarrow \text{CCNF} \ a \ b \\
\text{observe} \ (\text{Arr} \ f) & = \text{arr} \ f \\
\text{observe} \ (\text{LoopD} \ i \ f) & = \text{loop} \ (\text{arr} \ f \gg \text{second} \ (\text{init} \ i))
\end{align*}
\]

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

**Final ingredient: an ArrowInit instance for CCNF** Figure 4 defines instances of \( \text{Arrow}, \ Arrow\text{Loop} \) and \( \text{ArrowInit} \) for \( \text{CCNF} \).

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 \( \text{CCNF} \) 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
Instance Arrow CCNF where
\[ arr = Arr \]
\[ Arr f \gg Arr g = Arr (g \cdot f) \]
\[ Arr f \gg \text{LoopD} i g = \text{LoopD} i (g \cdot f \times id) \]
\[ \text{LoopD} i f \gg Arr g = \text{LoopD} i (g \cdot id \cdot f) \]
\[ \text{LoopD} i f \gg \text{LoopD} j g = \text{LoopD} (i, j) \cdot (\text{assoc'} (\text{juggle'}) (g \cdot id) \cdot f \cdot id)) \]
\[ \text{first} (Arr f) = Arr (f \cdot id) \]
\[ \text{first} (\text{LoopD} i f) = \text{LoopD} i (\text{juggle'} (\text{first} f)) \]

Instance ArrowLoop CCNF where
\[ \text{loop} (Arr f) = Arr (\text{trace} f) \]
\[ \text{loop} (\text{LoopD} i f) = \text{LoopD} i (\text{trace} (\text{juggle'} f)) \]

Instance ArrowInit CCNF where
\[ \text{init} i = \text{LoopD} i \text{swap} \]

Figure 4: The arrow instances for CCNF

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 is a semantic one, namely equality under observation, where \( f \) and \( g \) are considered equivalent if observe \( f \) is equivalent to observe \( g \). In other words, we can replace Arr and LoopD with the corresponding right hand sides (from the definition of 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 Figure 4.

Figure 5 shows parts of the derivations for the Arrow, ArrowLoop and ArrowInit methods for CCNF. The full derivations follow a similar pattern of equational reasoning about the observed normalized terms.

The top part of Figure 5 derives part of the definition of \( \gg \) for CCNF (Figure 4), namely the second case:
\[ Arr f \gg \text{LoopD} i g = \text{LoopD} i (g \cdot f \times id) \]

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

The middle part of Figure 5 derives part of the definition of loop for CCNF, namely the first case:
\[ \text{loop} (Arr f) = Arr (\text{trace} f) \]

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, the bottom part of Figure 5 shows the derivation of the definition of init for 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 second to first and eliminating the resulting 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.

Derivation of \( Arr f \gg \text{LoopD} i g = \text{LoopD} i (g \cdot f \times id) \):
\[ arr f \gg \text{loopD} i g = \text{loopD} i (g \cdot f \times id) \]
\[ = (\text{def. loopD}) \]
\[ arr f \gg \text{loop (arr g \gg second (init i))} \]
\[ = (\text{left tightening}) \]
\[ \text{loop (first (arr f)) \gg arr g \gg second (init i))} \]
\[ = (\text{extension}) \]
\[ \text{loop (arr (f \cdot id) \gg arr g \gg second (init i))} \]
\[ = (\text{composition}) \]
\[ \text{loop (arr (g \cdot f \times id) \gg second (init i))} \]
\[ = (\text{def. loopD}) \]
\[ \text{loopD} i (g \cdot f \times id) \]

Derivation of \( \text{loop (Arr f) = Arr (trace f)} \):
\[ \text{loop (arr f)} = \text{(extension)} \]
\[ \text{arr (trace f)} \]

Derivation of \( \text{init i = LoopD i swap} \):
\[ \text{init i} = \text{(right identity)} \]
\[ \text{init i \gg arr id} \]
\[ = (\text{trace swap = id}) \]
\[ \text{init i \gg arr (trace swap)} \]
\[ = (\text{extension}) \]
\[ \text{init i \gg loop (arr swap)} \]
\[ = (\text{left tightening}) \]
\[ \text{loop (first (init i)) \gg arr swap} \]
\[ = (\text{left identity}) \]
\[ \text{loop (arr id \gg first (init i)) \gg arr swap} \]
\[ = (\text{swap . swap = id}) \]
\[ \text{loop (arr (swap . swap) \gg first (init i) \gg arr swap)} \]
\[ = (\text{composition}) \]
\[ \text{loop (arr swap \gg arr swap \gg first (init i) \gg arr swap)} \]
\[ = (\text{def. second}) \]
\[ \text{loop (arr swap \gg second (init i))} \]
\[ = (\text{def. loopD}) \]
\[ \text{loopD i swap} \]

Figure 5: Partial derivations of \( \gg \), \text{loop} and \text{init} for CCNF

In order to normalize a computation such as \( exp \) that is polymorphic in the 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 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 CCNF (by calling observe) to obtain a normalized version of \( exp \), and then with another ArrowInit instance, such as \( SF \).

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

4. Optimizing Observation

Section 3 showed how to improve the performance of CCA programs by taking advantage of a universal property: every CCA
instance Arrow SF where
  arr f = g where g = SF (λx → (f x, g))
  h f x =
    let (y, f') = unSF f x
    (z, g') = unSF g y
    in (z, SF (h f' g'))

first f = SF (g f)
where g f (x, z) = let (y, f') = unSF f x
                   in ((y, z), SF (g f'))

instance ArrowLoop SF where
  loop sf = SF (g sf)
where
  g f x = (y, SF (g f'))
    where ((y, z), f') = unSF f (x, z)

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

Figure 6: The arrow instances for SF

An SF transformer is a function which, when applied to a value, returns a pair of a new value and a new transformer to be used as the continuation. The arr operator (Figure 6) constructs a pure transformer, where the new transformer returned as the continuation is just itself. The >> operator compose two transformers f and g by threading the argument x first through f and then through g, and composing the continuations. The first operator builds a new transformer from an existing transformer f, and threads through an unmodified input z alongside the computation of passing input x to f. The loop operator (Figure 6) 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 6) outputs the initial value, while passing the current input to its own continuation as the next value to output, essentially forming an internal state living in a closure.

One point of note is that all these functions — even arr — are fundamentally recursive, which makes computations built by composing them 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. To illustrate the path from the unoptimized observation function for CCNF to an optimized version (Figure 7), we follow 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 >>, and combining the resulting adjacent calls to arr using the composition law (Figure 7(b)).

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 (Figure 7(c)).

Naming subexpressions makes it easier to specialize applications to known arguments in the next step (Figure 7(d)), 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:

\[
\text{first}_\text{init} i = \text{first}_\text{SF} (\text{init}_\text{SF} i)
\]

Substituting the definitions of firstSF and initSF results in the following definitions:

\[
\text{first}_\text{init} i = \text{SF} (g_i (\text{SF} (h_i i)) \text{ where } h_i i x = (i, \text{SF} (h_i x)) g_i f (x, z) = \text{let} (y, f') = \text{unSF} f x \text{ in } ((y, z), \text{SF} (g_i f'))
\]

Next, inlining the calls to \(g_i\) and \(h_i\) in the first line gives the following:

\[
\text{first}_\text{init} i = \text{SF} (\lambda(x, z) \to \text{let} (y, f') = (i, \text{SF} (h_i x)) \text{ in } ((y, z), \text{SF} (g_i f')))\]

where ...

Reducing the let in the above definition gives the following:

\[
\text{first}_\text{init} i = \text{SF} (\lambda(x, z) \to ((i, z), \text{SF} (g_i (\text{SF} (h_i x)))))\]

where ...
a) Unoptimized observe

\[\text{observe :: ArrowInit \( a \rightarrow CCNF \ a \ b \rightarrow arr \ a \ b \)\]

observe (Arr f) = arr f

observe (LoopD i f) = loop (arr f \( \gg \gg \) second (init i))

b) Instantiating with SF (with second expanded)

\[\text{observeSF :: CCNF \ a \ b \rightarrow SF \ a \ b} \]

\[\text{observeSF (Arr f) = arrSF f} \]

\[\text{observeSF (LoopD i f) = loopSF (arrSF (swap . f) \( \gg \gg \) firstSF (initSF i))} \]

\( \gg \gg \) SF (arrSF swap)

c) Naming subexpressions (LoopD case only)

\[\text{observeSF (LoopD i f) = loop_comp2 i f}\]

\[\text{where}\]

\[\text{arrswap f = arrSF (swap . f)} \]

\[\text{arrswap = arrSF swap}\]

\[\text{firstSF i = firstSF i f} \]

\[\text{firstSF i} \gg \gg\]

\[\text{secondSF (initSF i)} \]

\[\text{loop_comp2 i f = loopSF (i \( \gg \gg \) 2 f)} \]

d) Specializing to known arguments (example: firstSF)

\[\text{firstSF i} \equiv\]

\[\text{where}\]

\[\text{h_i \rightarrow i, SF (h_i x)} \]

\[\text{firstSF i} = SF (\lambda x, z \rightarrow ((i, z), firstSF i x)) \]

e) The optimized observeSF

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

\[\text{where}\]

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

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

\[\text{runD' i f)} \]

f) Merging in runSF

\[\text{runSF :: CCNF a b \rightarrow [a] \rightarrow [b]} \]

\[\text{runSF (LoopD i f) = g i f}\]

\[\text{where}\]

\[\text{g i f (x : xs) =}\]

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

g) Merging in !!

\[\text{nthCCNF :: Int \rightarrow CCNF} \ () \ a \rightarrow a\]

\[\text{nthCCNF n (LoopD i f) = next n i}\]

\[\text{where}\]

\[\text{next n i = \text{if } n \equiv 0 \text{ then } x \text{ else } next (n - 1) i'} \]

\[\text{where (x, i') = f ((, i))}\]

Figure 7: From unoptimized to optimized observation

But we saw earlier that firstSF i is equal to SF (g_i (SF (h_i i))),

and so we can replace SF (g_i (SF (h_i i))) with firstSF x to obtain the following simple definition:

\[\text{firstSF i = SF (\lambda (x, i) \rightarrow ((i, z), firstSF i x))}\]

Similar reasoning for the other parts of the computation eventually results in the simple implementation of observeSF in Figure 7(e).

data ST s a

instance Monad (ST s)

runST :: (forall s. ST s a) \rightarrow a

fixST :: (a \rightarrow ST s a) \rightarrow a

data STRef s a

newSTRef :: a \rightarrow ST s (STRef s a)

readSTRef :: STRef s a \rightarrow ST s a

writeSTRef :: STRef s a \rightarrow a \rightarrow ST s ()

Figure 8: ST monad and mutable references

Merging observeSF with observation for SF The reasoning above has given us an observe function that is optimized for the CCA normal form and for the SF instance. However, SF is not typically used directly. Instead, the following function, which serves as a kind of observation function for SF, turns an SF stream transformer into a transformation on concrete streams:

\[\text{runSF :: SF f (x : xs) = let } (y, f') = f x\]

\[\text{in } y: g f' xs\]

Similar reasoning to that used above allows us to fuse the composition runSF . observeSF (Figure 7(f)).

Finally, in cases where we wish to retrieve only a single element when running an arrow with a constant input stream of units, even runCCNF introduces unnecessary overhead by consuming and constructing input and output lists. Composing the list-indexing function !!! with runCCNF results in the following function, which avoids the construction of the intermediate list altogether (Figure 7(g)).

5. Handling Mutable States

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

newtype State s a = State (s \rightarrow (a, s))

instance Monad (State s) where ...

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 in spite of laziness.

For programs where this purely functional encoding of mutable state is unacceptably inefficient the ST monad (Lauchbury and Peyton Jones 1994) offers an interface to genuinely mutable state. Figure 8 shows the ST monad and its related operations in Haskell. Conceptually, we view the type ST s a as follows:

\[\text{type ST s a = State# s \rightarrow (a, State# s)}\]

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 use supported primitive operations where the state remains hidden, including those for the mutable reference STRef type, shown in Figure 8.

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 are guaranteed to be sequenced. Further, the type of the observation function, runST, ensures that the phantom
type variable $s$, which is used to index the $\text{STRef}$ values in a computation, cannot “escape” into the surrounding context. Since the types of such mutable references (and hence the references themselves) cannot be accessed outside the call to $\text{runST}$, mutations to $\text{STRef}$ values constitute a benign effect: computations via $\text{runST}$ are indistinguishable from pure terms.

5.1 Implementing CCA with the ST Monad

The similarity between CCA and the state monad suggests a question: is there a more efficient implementation of CCA based on mutable state? After all, the commutativity and product laws already assert that any side effect on a state internal to a CCA is isolated, and only affects future states of the same arrow when run. The question then becomes whether we can implement mutable states as an ArrowInit instance in Haskell. We give one such implementation in Figure 9, where mutation is suitably handled by the ST monad. The difference between $\text{CCNF}$ and the previously seen $\text{CCNF}$ data type is in the $\text{LoopD}$ constructor:

$$
\text{LoopD}_{\text{ST}} :: ST \ s \ c \to (c \to \text{ST} \ s \ b) \to \text{CCNF}_{\text{ST}} \ s \ a \ b
$$

The idea here is that instead of passing immutable states as values, we have as the first argument to $\text{LoopD}_{\text{ST}}$ an $\text{ST}$ action that initializes an mutable state. The type variable $s$ here can also be any mutable data type allowed in an $\text{ST}$ monad, for instance, $\text{STRef}$. The state transition function (second argument to $\text{LoopD}_{\text{ST}}$) will then take the mutable object of type $c$, and an input of type $a$ to compute the arrow output of type $b$, all in an $\text{ST}$ 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.

We give a definition of ArrowInit instance for $\text{CCNF}_{\text{ST}}$ in Figure 9, where the $\text{init}$ arrow uses $\text{newSTRef}$ $i$ as the action to initialize a mutable reference of the $\text{STRef}$ type, which is then passed to the transition function $f$ that can read and write to it. Other instance declarations in Figure 9 are mostly straightforward, where the sequential composition of two $\text{LoopD}$ arrows is just composition of two initialization actions, and two transition actions. The $\text{ArrowLoop}$ instance of $\text{loopST}$ make use of recursive monad (hence the $\text{recv}$ keyword) to “tie-the-knot” between the second input and the second output values of this arrow. $\text{ST}$ monad is a valid instance of $\text{MonadFix}$, where value-level recursion is implemented by $\text{fixST}$ (Figure 8).

Any generic CCA can be instantiated to type $\text{CCNF}_{\text{ST}}$; and all we need is a way to run them. We give the following definition of sampling the $n^{th}$ element in the output stream of an $\text{CCNF}_{\text{ST}}$ arrow taking no input:

$$
nth_{\text{ST}} :: Int \to (\forall \ s. \ \text{CCNF}_{\text{ST}} \ s \ () \ a) \to a\\n\text{nth}’_{\text{ST}} :: Int \to \text{CCNF}_{\text{ST}} \ s \ () \ a \to \text{ST} \ s \ a\\n\text{nth}’_{\text{ST}} n \ (\text{Arr}_{\text{ST}} f) = \text{return} \ (f \ () )\\n\text{nth}_{\text{ST}} n \ (\text{LoopD}_{\text{ST}} i f) = \text{do}\\ng \leftarrow \text{fmap} \ f \ i\\n\text{let} \ \text{next} \ n = \text{do}\\nx \leftarrow g ()\\n\text{if} \ n \leq 0 \ \text{then} \ \text{return} \ x \ \text{else} \ \text{next} \ (n - 1)\\nnext n
$$

As with $\text{runST}$, $\text{nth}_{\text{ST}}$ uses an existential type to enclose the phantom type variable $s$, and the helper function $\text{nth}’_{\text{ST}}$ takes care of the actual unfolding. All initialization of the mutable state happens only once outside of the actual iteration function next because all state references remain unchanged: it is their values that are mutated in-place.

5.2 Proving CCA Laws for $\text{CCNF}_{\text{ST}}$

Implementing CCA using ST monad may have given us the access to mutable states, but wouldn’t the stringent linearity imposed by monads be too restrictive for CCA? In particular, it is hard to imagine that the commutativity law would hold for the $\text{CCNF}_{\text{ST}}$ arrow. We give a sketch of our proofs below.

Commutativity law Proof by case analysis. The cases involving pure arrows in the form of $\text{Arr}_{\text{ST}}$ are trivial. The core of the proof for commutativity of $\text{LoopD}_{\text{ST}}$ reduces to proving that the following equation holds:

$$
\text{LoopD}_{\text{ST}} (\text{liftM2} (\_ \_ \_ \_ \_ j) s_i) (\lambda (i,j) x \to \text{liftM2} (\_ \_ \_ \_ \_ j) (f \ i \ x) (g \ j \ x))\\\equiv \text{LoopD}_{\text{ST}} (\text{liftM2} (\_ \_ \_ \_ \_ j) s_i) (\lambda (i,j) x \to \text{fmap} \ \text{swap} \$ \ \text{liftM2} (\_ \_ \_ \_ \_ j) (g \ j \ x) (f \ i \ x))
$$

As usual, we interpret the equality extensionally: the equation holds if and only if the two sides are observably equivalent, using an observation function similar to $\text{nth}_{\text{ST}}$. After unfolding both sides into the observe function, we are left to prove that the monadic sequencing of $s_i$ and $s_j$, and of $f \ i \ x$ and $g \ j \ x$ actually commutes. In order to show this we require that effectful operations on distinct $\text{STRef}$ objects do not interfere with each other, allowing us to change the order of $s_i$ and $s_j$, or $f$ and $g$ without affecting the output of the observe function. While this property does not hold in general cases, it does hold in the restricted use of $\text{STRef}$ objects in our definitions for $\text{CCNF}_{\text{ST}}$ instances and $\text{nth}_{\text{ST}}$, which ensure that $f$ has no access to $j$ and $g$ has no access to $i$. The fact that the

---

**Figure 9:** An ST monad based CCA implementation
only effectful operation in the CCNFST arrow is about STRef completes this proof.

**Product law** To prove the product law holds for the CCNFST arrow, we again have to resort to extensionality. Proving the product law amounts to showing the reasonable property that using a pair of two distinct STRef's is equivalent to using one STRef of a pair. We omit the proof detail here.

5.3 Application: Sound Synthesis Circuits

A popular application of arrows is found 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.

Sound waves are usually produced at a preset signal rate for digital audio. For instance, 44100Hz is considered a standard frequency. Hence circuits for sound synthesis often fit well into a synchronous data-flow 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
  sr = 44100    -- signal rate
```

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`, initialized to 0. The first argument to `initLine` specifies the size of this buffer, and the second argument is the initial value for the buffer. Conceptually a delay line of size `n` is equivalent to `n` unit delays, or we can state it as:

```haskell
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 we use `Vector` to refer to the mutable vector module from the Haskell vector package:

```haskell
instance BufferedCircuit (CCNFST s) where
  initLine size i = LOOP s newBuf updateBuf
  where
    newBuf = do
      b ← Vector.new size
      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
```

1 If a strict vector is used (e.g. unboxed vector), we need to ensure that `initLine` is sufficiently lazy to work with `loop`, which can be achieved by composing `initLine` with an extra `init 0`.

```haskell
flute :: BufferedCircuit a ⇒ Time → Double →
    Double → Double → Double → a () Double
flute dar amp fqc press breath =
  proc () → do
    env1 ← envLineSeg [0, 1.1 * press, press, 0] [0.06, 0.2, dar - 0.16, 0.02] ≫ ()
    env2 ← envLineSeg [0, 1, 1] [0.01, dur - 0.02, 0.01] ≫ ()
    envib ← envLineSeg [0, 0, 1, 1] [0.5, 0.5, dur - 1] ≫ ()
    flow ← noise White 42 ≫ ()
    vib ← osc sineTable 0 ≫ 5
    let emb = breath * flow * env1 + env1 +
       vib * 0.1 * envib
    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 ≤ 0.0 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 10: flute and shepard synthesis program

```haskell
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, incremented each time a new input arrives. 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` operations 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 enabled. We therefore consider our implementation of CCNFST as a safe and sound alternative to implementing arrow-based audio and sound processing circuits.

Finally, Figure 10 gives two sample synthesis programs used to measure 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 uses a number of helper functions including a source from random white noise, envelope control using segmented line and so on. The use of `delayLine` here simulates a traveling wave and its reflection. We omit the definitions of these helper func-
tions here, and instead refer our readers to Cheng and Hudak (2009) for additional details.

The second example shepard may look slightly simpler than flute, 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 all signals returned from recursively calling itself with a duration that is 0.5 second less. The use of arrow-level recursion makes a complex structure. Note that this is different from having a feedback loop, because the parameter seconds affects both the setting and composition of the arrow’s structural components, not just its input or output.

6. Performance Measurement and Analysis

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

6.1 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 Giorgidze 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 both SF and CCNF as well. We took extra caution to ensure our implementation is free of the correctness bug affecting Euterpea despite that we had to use unsafePerformIO too. Such details are tricky to get right, fragile and prone to future changes in the compiler.

Our use of the “micro-” and “macro-benchmark” terminology is by no means scientific, and must be taken in a relative context.

All programs are written with arrow notation as generic computations parameterized by an arrow type variable. We can therefore reuse the same source code for the SF, CCNF, and CCNFST versions of these benchmarks; the only difference is 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, and normalize these programs to pairs of initial value and transition function. The normalized programs are then sampled with a similar nth function used for CCNF arrows.

We use the Criterion benchmarking package for Haskell to measure the time taken for the nth function to compute 44100 \times 5 = 2,205,000 samples, which is equivalent to 5 seconds of audio for sound synthesis programs. 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 consistent performance across all implementations, we additionally annotate all generic arrow computations with SPECIALIZE and INLINE pragmas, though these are not strictly required in most cases. The flag -funfolding-use-limit=512 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.

6.2 Overall Benchmarking Result

Figure 11 shows the benchmarking result for the 8 programs under each of the four implementations (SF, CCNF, CCNFST, and Template Haskell). We report both the mean kernel time and the relative performance speedups using SF as a baseline. Also shown are the number of internal states and loop counts in the source of each benchmark, which give rough estimates of the program complexity.

We make a few observations on the data reported in Figure 11. First, for micro-benchmarks, the CCNF implementation is as fast as the original Template Haskell implementation. It appears that our instance-based normalization, combined with GHC’s optimizations, appears able to normalize these CCA computations at compile time as effectively as the Template Haskell implementation. For macro-benchmarks like flute and shepard, CCNF lags behind the CCNFST and Template Haskell implementations.

On the other hand, CCNFST is significantly slower than CCNF for micro-benchmarks, likely due to the overhead introduced by the ST monad. For flute and shepard, however, CCNFST significantly outperforms CCNF, even though both use mutable buffers to implement delay line. In fact, CCNFST even outperforms the Template Haskell version of shepard.

For fib, normalization seems less effective than for the other benchmarks, barely doubling the performance of the SF version. This is a scenario where the arrow overhead (of a simple structure) weighs much less than the real computation (big integer arithmetic), so optimizing away the intermediate structure does not save as much. But the 2\times performance gain is still worthwhile!

6.3 Analyzing the Performance of CCNF

As discussed in Section 3, our implementation for CCNF is systematically derived first from a normalization by interpretation strategy, and then specialized to the observe function. We have already seen the effects of the combined optimization strategy (Figure 11), but it is interesting to investigate how much of the performance improvements comes from normalization, and how much from specialization.

To measure the normalization contribution, the same arrow programs are normalized by the observe function to a generic arrow and then specialized to the SF type and sampled by nthSF. A percentage is calculated by comparing with the full CCNF implementation. The remaining speedups can then be attributed to specialization to the CCNF type and sampled by the optimized nthCCNF function. We show this percentage of performance contribution in Figure 12, 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 fib, flute and shepard, where the amount of real computation greatly outweighs the remaining overhead in an normalized 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, exp, sine, oscSine and robot, are also the ones seeing most significant speedups (from 60\times to 242\times in Figure 11), where eliminating the final arrow overheads gives a greater improvement to their overall performance.

6.4 Analyzing the Performance of CCNFST

The performance of CCNFST also begs for more explanation. Looking at the time difference between CCNFST 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 CCNFST and CCNF. The explanation is rather simple. We translate the loop
As we move to more complex programs, however, the situation dramatically changes: the CCNF 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 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 workload by increasing the input size. Figure 13 compares the running time for shepard with the CCNF, CCNF-ST 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 13 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-ST, while the relative speed of the Template Haskell implementation plummets. From left to right, the Template Haskell version goes from 80% of CCNF-ST performance to only half. Clearly the Template Haskell version contains overheads that are not present in either CCNF or CCNF-ST. Our understanding is that the normalization implemented via Template Haskell has to expand the entire arrow at compile time. In contrast, both CCNF and CCNF-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-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 in each iteration nthCCNF has to construct a new state as a nested tuple only for it to be destructed by the transition function immediately. In contrast, nthST constructs a single nested tuple of mutable references only at the initialization stage.

6.5 Final Remark

It is clear that our technique relies on the actual Haskell compiler to do the heavy lifting. Or put in another way, we have demonstrated a simple and yet effective approach to help GHC better optimize high-level arrow programs without sacrificing usability or modularity. However, as GHC’s optimizer has grown in complexity over the years, performance tuning becomes a challenging task and requires a good knowledge of GHC’s internals just to understand the results. For this reason, we have restrained ourselves from resorting to more obscure and GHC-specific features to pursue further performance gain in the hope that our approach remains generally applicable.

As much as we appreciate GHC’s amazing ability to simplify complex programs, we still find there is room for improvements. For example, examining the optimized GHC Core of CCNF or CCNF-ST versions of flute shows that not all intermediate structures (including boxed numerical values) are eliminated statically. In particular, for the CCNF-ST version, we would very much like to
see the nested tuple of mutable references to be completely inlined into the transformer function. We have experimented with alternatives such as strict and/or unboxed tuples, but have yet to find a satisfying solution. Likewise, GHC is often very effective at breaking the recursive “knot” that is introduced by the `trace` function, but unable to do so as soon as an intermediate data structure is present, as in the case of unfolding an recursive ST monad. We leave further explorations to future work.

7. Related Work

Representing arrow computations as data The technique of representing arrow computations with a data type in order to 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 mention 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 `ArrowUnit`) 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 12, 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), or of functions using the standard equations of the language (Section 4) is standard, and used to great effect in many places in the functional programming literature. Hinze (2000) gives 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) use 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


