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:

$$\begin{array}{l} exp = proc\ () \rightarrow \mathbf{do} \\ \quad \mathbf{rec\ let}\ e = 1 + i \\ \qquad i \leftarrow integral \prec e \\ \qquad returnA \prec e \end{array}$$

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 exp, makes the data flow pleasingly clear: the $integral$ 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 $integral$ 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 arr, $\ggg$, first, $loop$, and $init$ which construct terms of an overloaded type 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 exp, we must instantiate the abstract arrow with a concrete implementation, such as the *causal stream transformer SF* (Liu et al. 2009) that forms the basis of *signal functions* in the Yampa domain-specific language for functional reactive programming (Hudak et al. 2003):

$$\mathbf{newtype}\ SF\ a\ b = SF\ \{unSF :: a \rightarrow (b, SF\ a\ b)\}$$

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

Instantiating 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.

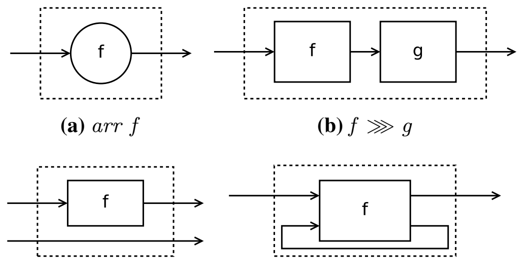
In this paper we address the first of these drawbacks and suggest a path to addressing the second. The reader familiar with recent Template Haskell developments might at this point expect us to propose switching the existing normalizer to using typed quotations and splices. Instead, we present a simpler 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

Haskell'16, September 22-23, 2016, Nara, Japan
ACM. 978-1-4503-4434-0/16/09...
http://dx.doi.org/10.1145/2976002.2976019

$$
\begin{array}{l}
\textbf{class}\ \mathit{Arrow}\ (\mathit{arr} :: * \to * \to *)\ \textbf{where} \\
\quad \mathit{arr} :: (a \to b) \to \mathit{arr}\ a\ b \\
\quad (\ggg) :: \mathit{arr}\ a\ b \to \mathit{arr}\ b\ c \to \mathit{arr}\ a\ c \\
\quad \mathit{first} :: \mathit{arr}\ a\ b \to \mathit{arr}\ (a, c)\ (b, c) \\
\textbf{class}\ \mathit{Arrow}\ \mathit{arr} \Rightarrow \mathit{ArrowLoop}\ \mathit{arr}\ \textbf{where} \\
\quad \mathit{loop} :: \mathit{arr}\ (a, c)\ (b, c) \to \mathit{arr}\ a\ b \\
\textbf{class}\ \mathit{ArrowLoop}\ \mathit{arr} \Rightarrow \mathit{ArrowInit}\ \mathit{arr}\ \textbf{where} \\
\quad \mathit{init} :: a \to \mathit{arr}\ a\ a
\end{array}
$$



(a) $\mathit{arr}\ f$ **(b)** $f \ggg g$

(c) $\mathit{first}\ f$ **(d)** $\mathit{loop}\ f$

Figure 1: The Arrow, $\mathit{ArrowLoop}$ and $\mathit{ArrowInit}$ classes

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

- We derive a new implementation of CCA normalization, 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.

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 composition. 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 combinators. The combinator arr lifts a function from type a to type b to a "pure" arrow computation from a to b, of type $\mathit{arr}\ a\ b$ where arr is the arrow type. The combinator $\ggg$ 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, $\mathit{first}\ f$ 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:

$$
\begin{array}{l}
\mathit{second} \quad :: \mathit{Arrow}\ \mathit{arr} \Rightarrow \mathit{arr}\ a\ b \to \mathit{arr}\ (c, a)\ (c, b) \\
\mathit{second}\ f = \mathit{arr}\ \mathit{swap} \ggg \mathit{first}\ f \ggg \mathit{arr}\ \mathit{swap} \\
\quad \textbf{where}\ \mathit{swap}\ (a, b) = (b, a)
\end{array}
$$

Parallel composition $\star\star\star$ of two arrows can then be defined as a sequence of *first* and second:

$$
\begin{array}{l}
(\star\star\star) :: \mathit{Arrow}\ \mathit{arr} \Rightarrow \mathit{arr}\ a\ b \to \mathit{arr}\ c\ d \to \mathit{arr}\ (a, c)\ (b, d) \\
f \star\star\star g = \mathit{first}\ f \ggg \mathit{second}\ g
\end{array}
$$

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 these 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 $\mathit{ArrowLoop}$ 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 $\mathit{ArrowLoop}$.

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 $\mathit{ArrowInit}$ class (Figure 1), and two additional laws to place further constraints on the computation (Figure 2c). In the context of synchronous circuits, $\mathit{ArrowInit}$ is almost identical to the $\mathit{ArrowCircuit}$ 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 $\mathit{ArrowInit}$ to make a few distinctions: the categorization of CCA defines two additional laws for $\mathit{ArrowInit}$ instances while $\mathit{ArrowCircuit}$ 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 $\mathit{ArrowInit}$ states that the order in a parallel arrow composition ($\star\star\star$) 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, $\mathit{ArrowLoop}$, and $\mathit{ArrowInit}$ classes (Figure 1) can 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 CCAs can be syntactically translated into either a pure arrow, or a single loop containing one*

$$
\begin{array}{rcll}
\mathit{arr}\ \mathit{id} \ggg f & \equiv & f & \text{(left identity)} \\
f \ggg \mathit{arr}\ \mathit{id} & \equiv & f & \text{(right identity)} \\
(f \ggg g) \ggg h & \equiv & f \ggg (g \ggg h) & \text{(associativity)} \\
\mathit{arr}\ (g\ .\ f) & \equiv & \mathit{arr}\ f \ggg \mathit{arr}\ g & \text{(composition)} \\
\mathit{first}\ (\mathit{arr}\ f) & \equiv & \mathit{arr}\ (f \times \mathit{id}) & \text{(extension)} \\
\mathit{first}\ (f \ggg g) & \equiv & \mathit{first}\ f \ggg \mathit{first}\ g & \text{(functor)} \\
\mathit{first}\ f \ggg \mathit{arr}\ (\mathit{id} \times g) & \equiv & \mathit{arr}\ (\mathit{id} \times g) \ggg \mathit{first}\ f & \text{(exchange)} \\
\mathit{first}\ f \ggg \mathit{arr}\ \mathit{fst} & \equiv & \mathit{arr}\ \mathit{fst} \ggg f & \text{(unit)} \\
\mathit{first}\ (\mathit{first}\ f) \ggg \mathit{arr}\ \mathit{assoc} & \equiv & \mathit{arr}\ \mathit{assoc} \ggg \mathit{first}\ f & \text{(association)}
\end{array}
$$

(a) *Arrow* laws

$$
\begin{array}{rcll}
\mathit{loop}\ (\mathit{first}\ h \ggg f) & \equiv & h \ggg \mathit{loop}\ f & \text{(left tightening)} \\
\mathit{loop}\ (f \ggg \mathit{first}\ h) & \equiv & \mathit{loop}\ f \ggg h & \text{(right tightening)} \\
\mathit{loop}\ (f \ggg \mathit{arr}\ (\mathit{id} \times k)) & \equiv & \mathit{loop}\ (\mathit{arr}\ (\mathit{id} \times k) \ggg f) & \text{(sliding)} \\
\mathit{loop}\ (\mathit{loop}\ f) & \equiv & \mathit{loop}\ (\mathit{arr}\ \mathit{assoc}^{-1}\ .\ f\ .\ \mathit{arr}\ \mathit{assoc}) & \text{(vanishing)} \\
\mathit{second}\ (\mathit{loop}\ f) & \equiv & \mathit{loop}\ (\mathit{arr}\ \mathit{assoc}\ .\ \mathit{second}\ f\ .\ \mathit{arr}\ \mathit{assoc}^{-1}) & \text{(superposing)} \\
\mathit{loop}\ (\mathit{arr}\ f) & \equiv & \mathit{arr}\ (\mathit{trace}\ f) & \text{(extension)}
\end{array}
$$

(b) *ArrowLoop* laws

$$
\begin{array}{rcll}
\mathit{first}\ f \ggg \mathit{second}\ g & \equiv & \mathit{second}\ g \ggg \mathit{first}\ f & \text{(commutativity)} \\
\mathit{init}\ i \star\!\star\!\star\ \mathit{init}\ j & \equiv & \mathit{init}\ (i, j) & \text{(product)}
\end{array}
$$

(c) *ArrowInit* laws

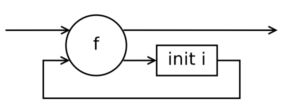
$$
\begin{array}{ll}
\times :: (a \to b) \to (c \to d) \to (a, b) \to (c, d) & \mathit{assoc}^{-1} :: (a, (b, c)) \to ((a, b), c) \\
f \times g = \lambda(x, y) \to (f\ x, g\ y) & \mathit{assoc}^{-1}\ (a, (b, c)) = ((a, b), c) \\
\mathit{assoc} :: ((a, b), c) \to (a, (b, c)) & \mathit{trace} :: ((a, b) \to (c, b)) \to a \to c \\
\mathit{assoc}\ ((a, b), c) = (a, (b, c)) & \mathit{trace}\ f\ a = \mathbf{let}\ (c, b) = f\ (a, b)\ \mathbf{in}\ c
\end{array}
$$

(d) Auxiliary functions

Figure 2: *Arrow*, *ArrowLoop* and *ArrowInit* laws

pure arrow and one initial state value, which is called causal commutative normal form (CCNF) (Liu et al. 2009, 2011):

$$\mathit{loop}\ (\mathit{arr}\ f \ggg \mathit{second}\ (\mathit{init}\ i))$$



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:

$$
\begin{array}{l}
\mathit{loopD} :: \mathit{ArrowInit}\ \mathit{arr} \Rightarrow c \to ((a, c) \to (b, c)) \to \mathit{arr}\ a\ b \\
\mathit{loopD}\ i\ f = \mathit{loop}\ (\mathit{arr}\ f \ggg \mathit{second}\ (\mathit{init}\ i))
\end{array}
$$

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 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 unSF that projects type $\mathit{SF}\ a\ b$ to its definition:

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

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

2.4 Example: the exp Arrow

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

Like exp, the $\mathit{integral}$ 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 exp itself contains a recursion, and it is defined in terms of $\mathit{integral}$, there are two nested levels of loops. This fact is made more evident in the desugared form if we substitute $\mathit{integral}$ into the exp. However, after being normalized to CCNF, the two loops collapse into just one, represented through the use of loopD.

3. Normalization and Optimization

It is easy to see how normalizing CCA computations can improve their efficiency. While a CCA computation such as exp may involve `exp` **in arrow notation**

$$\begin{aligned}
&\mathit{exp} = \mathit{proc}\ () \rightarrow \mathbf{do} \\
&\quad \mathbf{rec\ let}\ e = 1 + i \\
&\qquad\quad i \leftarrow \mathit{integral} \prec e \\
&\quad \mathit{returnA} \prec e \\
&\mathit{integral} = \mathit{proc}\ v \rightarrow \mathbf{do} \\
&\quad \mathbf{rec}\ i \leftarrow \mathit{init}\ 0 \prec i + dt * v \\
&\quad \mathit{returnA} \prec i
\end{aligned}$$

`exp` **desugared**

$$\begin{aligned}
&\mathit{exp} = \mathit{loop} \\
&\quad (\mathit{second}\ (\mathit{integral} \ggg \mathit{arr}\ (+1)) \ggg \\
&\qquad \mathit{arr}\ \mathit{snd} \ggg \mathit{arr}\ (\lambda x \rightarrow (x, x)) \\
&\mathit{integral} = \mathit{loop} \\
&\quad (\mathit{arr}\ (\lambda(v, i) \rightarrow i + dt * v) \ggg \\
&\qquad \mathit{init}\ 0 \ggg \mathit{arr}\ (\lambda x \rightarrow (x, x))
\end{aligned}$$

`exp` **normalized**

$$\begin{aligned}
\mathit{exp} = \mathit{loopD}\ 0\ (\lambda(x, y) \rightarrow\ & \mathbf{let}\ i = y + 1 \\
& \mathbf{in}\ (i, y + dt * i))
\end{aligned}$$

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 $\mathit{integral}$ 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 (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 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:

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

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

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 $\mathit{ArrowInit}$ instance — is given by the following function:

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

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

Final ingredient: an $\mathit{ArrowInit}$ ***instance for*** CCNF Figure 4 defines instances of Arrow, $\mathit{ArrowLoop}$ and $\mathit{ArrowInit}$ 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

$$
\begin{array}{l}
\textbf{instance } \mathit{Arrow}\ \mathit{CCNF}\ \textbf{where} \\
\quad \mathit{arr} = \mathit{Arr} \\
\quad \mathit{Arr}\ f \ggg \mathit{Arr}\ g = \mathit{Arr}\ (g\ .\ f) \\
\quad \mathit{Arr}\ f \ggg \mathit{LoopD}\ i\ g = \mathit{LoopD}\ i\ (g\ .\ f \times \mathit{id}) \\
\quad \mathit{LoopD}\ i\ f \ggg \mathit{Arr}\ g = \mathit{LoopD}\ i\ (g \times \mathit{id}\ .\ f) \\
\quad \mathit{LoopD}\ i\ f \ggg \mathit{LoopD}\ j\ g = \\
\qquad \mathit{LoopD}\ (i,j)\ (\mathit{assoc}'\ (\mathit{juggle}'\ (g \times \mathit{id})\ .\ f \times \mathit{id})) \\
\quad \mathit{first}\ (\mathit{Arr}\ f) = \mathit{Arr}\ (f \times \mathit{id}) \\
\quad \mathit{first}\ (\mathit{LoopD}\ i\ f) = \mathit{LoopD}\ i\ (\mathit{juggle}'\ (\mathit{first}\ f)) \\
\\
\textbf{instance } \mathit{ArrowLoop}\ \mathit{CCNF}\ \textbf{where} \\
\quad \mathit{loop}\ (\mathit{Arr}\ f) = \mathit{Arr}\ (\mathit{trace}\ f) \\
\quad \mathit{loop}\ (\mathit{LoopD}\ i\ f) = \mathit{LoopD}\ i\ (\mathit{trace}\ (\mathit{juggle}'\ f)) \\
\\
\textbf{instance } \mathit{ArrowInit}\ \mathit{CCNF}\ \textbf{where} \\
\quad \mathit{init}\ i = \mathit{LoopD}\ i\ \mathit{swap}
\end{array}
$$

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 $\ggg$ for $CCNF$ (Figure 4), namely the second case:

$$\mathit{Arr}\ f \ggg \mathit{LoopD}\ i\ g = \mathit{LoopD}\ i\ (g\ .\ f \times \mathit{id})$$

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:

$$\mathit{loop}\ (\mathit{Arr}\ f) = \mathit{Arr}\ (\mathit{trace}\ f)$$

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 $CCNF$:

$$\mathit{init}\ i = \mathit{LoopD}\ i\ \mathit{swap}$$

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 $\mathit{Arr}\ f \ggg \mathit{LoopD}\ i\ g = \mathit{LoopD}\ i\ (g\ .\ f \times \mathit{id})$:

$$
\begin{array}{ll}
 & \mathit{arr}\ f \ggg \mathit{loopD}\ i\ g \\
= & \text{(def. } \mathit{loopD}) \\
 & \mathit{arr}\ f \ggg \mathit{loop}\ (\mathit{arr}\ g \ggg \mathit{second}\ (\mathit{init}\ i)) \\
= & \text{(left tightening)} \\
 & \mathit{loop}\ (\mathit{first}\ (\mathit{arr}\ f) \ggg \mathit{arr}\ g \ggg \mathit{second}\ (\mathit{init}\ i)) \\
= & \text{(extension)} \\
 & \mathit{loop}\ (\mathit{arr}\ (f \times \mathit{id}) \ggg \mathit{arr}\ g \ggg \mathit{second}\ (\mathit{init}\ i)) \\
= & \text{(composition)} \\
 & \mathit{loop}\ (\mathit{arr}\ (g\ .\ f \times \mathit{id}) \ggg \mathit{second}\ (\mathit{init}\ i)) \\
= & \text{(def. } \mathit{loopD}) \\
 & \mathit{loopD}\ i\ (g\ .\ f \times \mathit{id})
\end{array}
$$

Derivation of $\mathit{loop}\ (\mathit{Arr}\ f) = \mathit{Arr}\ (\mathit{trace}\ f)$:

$$
\begin{array}{ll}
 & \mathit{loop}\ (\mathit{arr}\ f) \\
= & \text{(extension)} \\
 & \mathit{arr}\ (\mathit{trace}\ f)
\end{array}
$$

Derivation of $\mathit{init}\ i = \mathit{LoopD}\ i\ \mathit{swap}$:

$$
\begin{array}{ll}
 & \mathit{init}\ i \\
= & \text{(right identity)} \\
 & \mathit{init}\ i \ggg \mathit{arr}\ \mathit{id} \\
= & (\mathit{trace}\ \mathit{swap} = \mathit{id}) \\
 & \mathit{init}\ i \ggg \mathit{arr}\ (\mathit{trace}\ \mathit{swap}) \\
= & \text{(extension)} \\
 & \mathit{init}\ i \ggg \mathit{loop}\ (\mathit{arr}\ \mathit{swap}) \\
= & \text{(left tightening)} \\
 & \mathit{loop}\ (\mathit{first}\ (\mathit{init}\ i) \ggg \mathit{arr}\ \mathit{swap}) \\
= & \text{(left identity)} \\
 & \mathit{loop}\ (\mathit{arr}\ \mathit{id} \ggg \mathit{first}\ (\mathit{init}\ i) \ggg \mathit{arr}\ \mathit{swap}) \\
= & (\mathit{swap}\ .\ \mathit{swap} = \mathit{id}) \\
 & \mathit{loop}\ (\mathit{arr}\ (\mathit{swap}\ .\ \mathit{swap}) \ggg \mathit{first}\ (\mathit{init}\ i) \ggg \\
 & \qquad \mathit{arr}\ \mathit{swap}) \\
= & \text{(composition)} \\
 & \mathit{loop}\ (\mathit{arr}\ \mathit{swap} \ggg \mathit{arr}\ \mathit{swap} \ggg \mathit{first}\ (\mathit{init}\ i) \ggg \\
 & \qquad \mathit{arr}\ \mathit{swap}) \\
= & \text{(def. } \mathit{second}) \\
 & \mathit{loop}\ (\mathit{arr}\ \mathit{swap} \ggg \mathit{second}\ (\mathit{init}\ i)) \\
= & \text{(def. } \mathit{loopD}) \\
 & \mathit{loopD}\ i\ \mathit{swap}
\end{array}
$$

Figure 5: Partial derivations of $\ggg$, $loop$ and $init$ 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 $\ggg$ 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

$$
\begin{aligned}
&\textbf{instance } \mathit{Arrow}\ \mathit{SF}\ \textbf{where}\\
&\quad \mathit{arr}\ f = g\ \textbf{where}\ g = \mathit{SF}\ (\lambda x \to (f\ x, g))\\
&\quad f \ggg g = \mathit{SF}\ (h\ f\ g)\\
&\qquad \textbf{where}\ h\ f\ g\ x =\\
&\qquad\qquad \textbf{let}\ (y, f') = \mathit{unSF}\ f\ x\\
&\qquad\qquad\quad\ (z, g') = \mathit{unSF}\ g\ y\\
&\qquad\qquad \textbf{in}\ (z, \mathit{SF}\ (h\ f'\ g'))\\
&\quad \mathit{first}\ f = \mathit{SF}\ (g\ f)\\
&\qquad \textbf{where}\ g\ f\ (x, z) = \textbf{let}\ (y, f') = \mathit{unSF}\ f\ x\\
&\qquad\qquad\qquad\qquad\quad \textbf{in}\ ((y, z), \mathit{SF}\ (g\ f'))\\
&\textbf{instance } \mathit{ArrowLoop}\ \mathit{SF}\ \textbf{where}\\
&\quad \mathit{loop}\ \mathit{sf} = \mathit{SF}\ (g\ \mathit{sf})\\
&\qquad \textbf{where}\ g\ f\ x = (y, \mathit{SF}\ (g\ f'))\\
&\qquad\qquad \textbf{where}\ ((y, z), f') = \mathit{unSF}\ f\ (x, z)\\
&\textbf{instance } \mathit{ArrowInit}\ \mathit{SF}\ \textbf{where}\\
&\quad \mathit{init}\ i = \mathit{SF}\ (f\ i)\ \textbf{where}\ f\ i\ x = (i, \mathit{SF}\ (f\ x))
\end{aligned}
$$

Figure 6: The arrow instances for SF

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. We then go further still, and build an observation function that is optimized for accessing individual stream elements. In effect, we build a function of the following type

$$(\mathit{ArrowInit}\ \mathit{arr} \Rightarrow \mathit{arr}\ a\ b) \to \mathit{Int} \to [a] \to b$$

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

The SF Arrow instances Figure 6 defines the $Arrow$, $ArrowLoop$ and $ArrowInit$ instances for the SF type introduced in Section 1.

The SF transformer can be seen as a simplified definition for signal functions; since these are described in considerable detail in the literature (Hudak et al. 2003; Nilsson 2005; Liu et al. 2009). we summarize their behaviour only briefly here.

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 $\ggg$ operator composes 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 — $loop_{SF}$ for $loop$, arr_{SF} 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 $\ggg$, 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 $observe_{SF}$, 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 $observe_{SF}$ after subexpressions are named:

$$\mathit{first}_{init}\ i = \mathit{first}_{SF}\ (\mathit{init}_{SF}\ i)$$

Substituting the definitions of $first_{SF}$ and $init_{SF}$ results in the following definitions:

$$
\begin{aligned}
&\mathit{first}_{init}\ i = \mathit{SF}\ (g_1\ (\mathit{SF}\ (h_1\ i)))\\
&\quad \textbf{where}\\
&\qquad h_1\ i\ x = (i, \mathit{SF}\ (h_1\ x))\\
&\qquad g_1\ f\ (x, z) = \textbf{let}\ (y, f') = \mathit{unSF}\ f\ x\\
&\qquad\qquad\qquad\quad \textbf{in}\ ((y, z), \mathit{SF}\ (g_1\ f'))
\end{aligned}
$$

Next, inlining the calls to g_1 and h_1 in the first line gives the following:

$$
\begin{aligned}
&\mathit{first}_{init}\ i = \mathit{SF}\ (\lambda (x, z) \to\\
&\qquad\qquad\qquad \textbf{let}\ (y, f') = (i, \mathit{SF}\ (h_1\ x))\\
&\qquad\qquad\qquad \textbf{in}\ ((y, z), \mathit{SF}\ (g_1\ f')))\\
&\quad \textbf{where}\ \ldots
\end{aligned}
$$

Reducing the **let** in the above definition gives the following:

$$
\begin{aligned}
&\mathit{first}_{init}\ i = \mathit{SF}\ (\lambda (x, z) \to ((i, z), \mathit{SF}\ (g_1\ (\mathit{SF}\ (h_1\ x)))))\\
&\quad \textbf{where}\ \ldots
\end{aligned}
$$

a) **Unoptimized** $observe$

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

b) **Instantiating with** SF **(with** $second$ **expanded)**

$$
\begin{aligned}
&observe_{SF} :: CCNF\ a\ b \to SF\ a\ b \\
&observe_{SF}\ (Arr\ f) = arr_{SF}\ f \\
&observe_{SF}\ (LoopD\ i\ f) = \\
&\quad loop_{SF}\ (arr_{SF}\ (swap\ .\ f) \ggg_{SF} first_{SF}\ (init_{SF}\ i) \\
&\qquad \ggg_{SF} arr_{SF}\ swap)
\end{aligned}
$$

c) **Naming subexpressions (**$LoopD$ **case only)**

$$
\begin{aligned}
&observe_{SF}\ (LoopD\ i\ f) = loop_{comp2}\ i\ f \\
&\mathbf{where} \\
&\quad arr_{swapf}\ f = arr_{SF}\ (swap\ .\ f) \\
&\quad arr_{swap} = arr_{SF}\ swap \\
&\quad first_{init}\ i = first_{SF}\ (init_{SF}\ i) \\
&\quad i \ggg_{1} f = arr_{swapf}\ f \ggg_{SF} (first_{init}\ i) \\
&\quad i \ggg_{2} f = i \ggg_{1} f \ggg_{SF} arr_{swap} \\
&\quad loop_{comp2}\ i\ f = loop_{SF}\ (i \ggg_{2} f)
\end{aligned}
$$

d) **Specializing to known arguments (example:** $first_{init}$**)**

$$
\begin{aligned}
&first_{init}\ i = first_{SF}\ (SF\ (h_1\ i)) \\
&\quad \mathbf{where} \\
&\qquad h_1\ i\ x = (i, SF\ (h_1\ x)) \\
&\ \ldots \\
&first_{init}\ i = SF\ (\lambda(x,z) \to ((i,z), first_{init}\ x)) \\
&\ \ldots
\end{aligned}
$$

e) **The optimized** $observe_{SF}$

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

f) **Merging in** run_{SF}

$$
\begin{aligned}
&run_{CCNF} :: CCNF\ a\ b \to [a] \to [b] \\
&run_{CCNF}\ (LoopD\ i\ f) = g\ i\ f \\
&\quad \mathbf{where}\ g\ i\ f\ (x : xs) = \\
&\qquad \mathbf{let}\ (y, i') = f\ (x, i)\ \mathbf{in} \\
&\qquad \mathbf{in}\ y : g\ i'\ f\ xs
\end{aligned}
$$

g) **Merging in** !!

$$
\begin{aligned}
&nth_{CCNF} :: Int \to CCNF\ ()\ a \to a \\
&nth_{CCNF}\ n\ (LoopD\ i\ f) = next\ n\ i \\
&\quad \mathbf{where} \\
&\qquad next\ n\ i = \mathbf{if}\ n \equiv 0\ \mathbf{then}\ x\ \mathbf{else}\ next\ (n-1)\ i' \\
&\qquad\quad \mathbf{where}\ (x, i') = f\ ((), i)
\end{aligned}
$$

Figure 7: From unoptimized to optimized observation

But we saw earlier that $first_{init}\ i$ is equal to $SF\ (g_1\ (SF\ (h_1\ i)))$, and so we can replace $SF\ (g_1\ (SF\ (h_1\ x)))$ with $first_{init}\ x$ to obtain the following simple definition:

$$first_{init}\ i = SF\ (\lambda(x,z) \to ((i,z), first_{init}\ x))$$

Similar reasoning for the other parts of the computation eventually results in the simple implementation of $observe_{SF}$ in Figure 7(e).

$$
\begin{aligned}
&\mathbf{data}\ ST\ s\ a \\
&\mathbf{instance}\ Monad\ (ST\ s) \\
&runST :: (forall\ s\ .\ ST\ s\ a) \to a \\
&fixST\ \ :: (a \to ST\ s\ a) \to a \\
&\mathbf{data}\ STRef\ s\ a \\
&newSTRef\ \ :: a \to ST\ s\ (STRef\ s\ a) \\
&readSTRef\ \ :: STRef\ s\ a \to ST\ s\ a \\
&writeSTRef :: STRef\ s\ a \to a \to ST\ s\ ()
\end{aligned}
$$

Figure 8: ST monad and mutable references

Merging $observe_{SF}$ ***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:

$$
\begin{aligned}
&run_{SF} :: SF\ a\ b \to [a] \to [b] \\
&run_{SF}\ (SF\ f)\ (x : xs) = \mathbf{let}\ (y, f') = f\ x \\
&\qquad\qquad\qquad\qquad\quad \mathbf{in}\ y : g\ f'\ xs
\end{aligned}
$$

Similar reasoning to that used above allows us to fuse the composition $run_{SF}\ .\ observe_{SF}$ (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 run_{CCNF} introduces unnecessary overhead by consuming and constructing input and output lists. Composing the list-indexing function !! with run_{CCNF} 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:

$$
\begin{aligned}
&\mathbf{newtype}\ State\ s\ a = State\ (s \to (a, s)) \\
&\mathbf{instance}\ Monad\ (State\ s)\ \mathbf{where} \ldots
\end{aligned}
$$

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 (Launchbury 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:

$$\mathbf{type}\ ST\ s\ a = State\#\ s \to (a, State\#\ s)$$

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 $STRef$ values in a computation, cannot "escape" into the surrounding context. Since the types of such mutable references (and hence the references themselves) can not be accessed outside the call to $runST$, mutations to $STRef$ values constitute a *benign effect*: computations via $runST$ 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 $CCNF_{ST}$ and the previously seen $CCNF$ data type is in the $LoopD_{ST}$ constructor:

$$LoopD_{ST} :: ST\ s\ c \to (c \to a \to ST\ s\ b) \to CCNF_{ST}\ s\ a\ b$$

The idea here is that instead of passing immutable states as values, we have as the first argument to $LoopD_{ST}$ an ST action that initializes an mutable state. The type variable c here can be any mutable data type allowed in an ST monad, for instance, $STRef$. The state transition function (second argument to $LoopD_{ST}$) 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 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 $CCNF_{ST}$ in Figure 9, 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 9 are mostly straightforward, where the sequential composition of two $LoopD_{ST}$s are 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 second input and the second output values of this arrow. ST monad is a valid instance of $MonadFix$, where value-level recursion is implemented by $fixST$ (Figure 8).

Any generic CCA can be instantiated to type $CCNF_{ST}$; 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 $CCNF_{ST}\ s$ arrow taking no input:

$$
\begin{array}{l}
nth_{ST} :: Int \to (forall\ s\ .\ CCNF_{ST}\ s\ ()\ a) \to a \\
nth_{ST}\ n\ nf = runST\ (nth'_{ST}\ n\ nf) \\
nth'_{ST} :: Int \to CCNF_{ST}\ s\ ()\ a \to ST\ s\ a \\
nth'_{ST}\ n\ (Arr_{ST}\ f) = return\ (f\ ()) \\
nth'_{ST}\ n\ (LoopD_{ST}\ i\ f) = \mathbf{do} \\
\quad g \leftarrow fmap\ f\ i \\
\quad \mathbf{let}\ next\ n = \mathbf{do} \\
\quad\quad x \leftarrow g\ () \\
\quad\quad \mathbf{if}\ n \leqslant 0\ \mathbf{then}\ return\ x\ \mathbf{else}\ next\ (n-1) \\
\quad next\ n
\end{array}
$$

As with $runST$, nth_{ST} uses an existential type to enclose the phantom type variable s, and the helper function nth'_{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.

$$
\begin{array}{l}
\mathbf{data}\ CCNF_{ST}\ s\ a\ b\ \mathbf{where} \\
\quad Arr_{ST}\quad :: (a \to b) \to CCNF_{ST}\ s\ a\ b \\
\quad LoopD_{ST} :: ST\ s\ c \to (c \to a \to ST\ s\ b) \to \\
\qquad\qquad\quad CCNF_{ST}\ s\ a\ b \\
\mathbf{instance}\ Arrow\ (CCNF_{ST}\ s)\ \mathbf{where} \\
\quad arr \qquad\qquad\qquad\quad = Arr_{ST} \\
\quad Arr_{ST}\ f \ggg Arr_{ST}\ g = Arr_{ST}\ (g\ .\ f) \\
\quad Arr_{ST}\ f \ggg LoopD_{ST}\ i\ g = LoopD_{ST}\ i\ h \\
\quad\quad \mathbf{where}\ h\ i = g\ i\ .\ f \\
\quad LoopD_{ST}\ i\ f \ggg Arr_{ST}\ g = LoopD_{ST}\ i\ h \\
\quad\quad \mathbf{where}\ h\ i = fmap\ g\ .\ f\ i \\
\quad LoopD_{ST}\ i\ f \ggg LoopD_{ST}\ j\ g = LoopD_{ST}\ k\ h \\
\quad\quad \mathbf{where}\ k = liftM2\ (,)\ i\ j \\
\qquad\qquad\quad h\ (i,j)\ x = f\ i\ x \ggg\!= g\ j \\
\quad first\ (Arr_{ST}\ f) \qquad = Arr_{ST}\ (first\ f) \\
\quad first\ (LoopD_{ST}\ i\ f) \ \ = LoopD_{ST}\ i\ g \\
\quad\quad \mathbf{where}\ g\ i\ (x,y) = liftM\ (,y)\ (f\ i\ x) \\
\mathbf{instance}\ ArrowLoop\ (CCNF_{ST}\ s)\ \mathbf{where} \\
\quad loop\ (Arr_{ST}\ f) = Arr_{ST}\ (trace\ f) \\
\quad loop\ (LoopD_{ST}\ i\ f) = LoopD_{ST}\ i\ h \\
\quad\quad \mathbf{where}\ h\ i\ x = \mathbf{do} \\
\qquad\qquad\qquad \mathbf{rec}\ (y,j) \leftarrow f\ i\ (x,j) \\
\qquad\qquad\qquad return\ y \\
\mathbf{instance}\ ArrowInit\ (CCNF_{ST}\ s)\ \mathbf{where} \\
\quad init\ i = LoopD_{ST}\ (newSTRef\ i)\ f \\
\quad\quad \mathbf{where}\ f\ i\ x = \mathbf{do} \\
\qquad\qquad\qquad y \leftarrow readSTRef\ i \\
\qquad\qquad\qquad writeSTRef\ i\ x \\
\qquad\qquad\qquad return\ y
\end{array}
$$

Figure 9: An ST monad based CCA implementation

5.2 Proving CCA Laws for $CCNF_{ST}$

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 $CCNF_{ST}$ arrow. We give a sketch of our proofs below.

Commutativity law Proof by case analysis. The cases involving pure arrows in the form of Arr_{ST} are trival. The core of the proof for commutativity of $LoopD_{ST}$ reduces to proving that the following equation holds:

$$
\begin{array}{rl}
& LoopD_{ST}\ (liftM2\ (,)\ s_i\ s_j)\ (\lambda(i,j)\ x \to \\
& \qquad\qquad\qquad\qquad\qquad liftM2\ (,)\ (f\ i\ x)\ (g\ j\ x)) \\
\equiv & LoopD_{ST}\ (liftM2\ (,)\ s_j\ s_i)\ (\lambda(j,i)\ x \to \\
& \qquad\qquad fmap\ swap\ \$\quad liftM2\ (,)\ (g\ j\ x)\ (f\ i\ x))
\end{array}
$$

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 nth_{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 $STRef$ 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 $STRef$ objects in our definitions for $CCNF_{ST}$ instances and nth_{ST}, which ensure that f has no access to j and g has no access to i. The fact that the only effectful operation in the $CCNF_{ST}$ arrow is about $STRef$ completes this proof.

Product law To prove the product law holds for the $CCNF_{ST}$ 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:

```
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:

```
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: [1]

```
instance BufferedCircuit (CCNF_ST s) where
  initLine size i = LoopD_ST 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
        let i′ = if i + 1 ⩾ size then 0 else i + 1
        writeSTRef r i′
        return 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.

```
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 ∗ 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

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 $CCNF_{ST}$, we automatically gain the ability to optimize all buffered circuits by normalizing them, because all $CCNF_{ST}$ 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 $CCNF_{ST}$ 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 functions 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 $CCNF_{ST}$, 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 $sci\text{-}fi$ 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 $CCNF_{ST}$ 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, $CCNF$, $CCNF_{ST}$, 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 $CCNF_{ST}$ and Template Haskell implementations.

On the other hand, $CCNF_{ST}$ is significantly slower than $CCNF$ for micro-benchmarks, likely due to the overhead introduced by the ST monad. For $flute$ and $shepard$, however, $CCNF_{ST}$ significantly outperforms $CCNF$, even though both use mutable buffers to implement delay line. In fact, $CCNF_{ST}$ 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 specializing to the $CCNF$ type and sampled by the optimized nth_{CCNF} 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 $robot$, graph 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 $CCNF_{ST}$

The performance of $CCNF_{ST}$ also begs for more explanation. Looking at the time difference between $CCNF_{ST}$ and $CCNF$ for $oscSine$, $sci\text{-}fi$ 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 $CCNF_{ST}$ and $CCNF$. The explanation is rather simple. We translate the $loop$

Benchmark			Unnormalized	Normalized					
Name	States	Loops	SF	$CCNF$		$CCNF_{ST}$		Template Haskell	
fib	2	1	480	209.8	(2.29×)	222.7	(2.16×)	209.2	(2.30×)
exp	1	2	292	1.204	(242×)	79.44	(3.67×)	1.206	(242×)
$sine$	2	1	229	1.845	(124×)	7.260	(31.5×)	1.570	(146×)
$oscSine$	1	1	216	3.557	(60.6×)	84.72	(2.54×)	3.558	(60.6×)
$sci\text{-}fi$	3	3	859	30.99	(27.7×)	252.7	(3.40×)	31.32	(27.4×)
$robot$	5	4	1162	11.13	(104×)	356.2	(3.26×)	12.02	(96.7×)
flute	16	7	3087	604.4	(5.10×)	285.5	(10.8×)	190.3	(16.2×)
$shepard$	80	30	20490	2741	(7.47×)	1319	(15.5×)	1590	(12.9×)
			time (baseline)	time	speedup	time	speedup	time	speedup

Figure 11: Benchmark kernel time (ms) and speedup

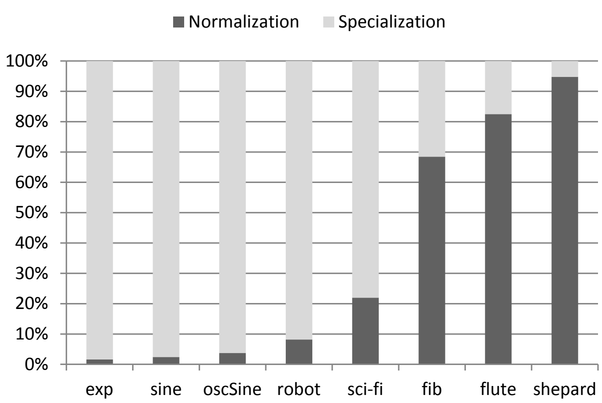


Figure 12: Performance contribution breakdown (pecentage)

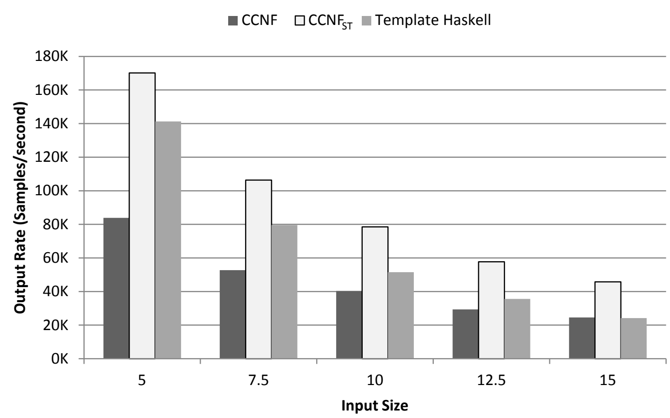


Figure 13: Performance of $shepard$ on different inputs

combinator for $CCNF_{ST}$ into a recursive ST monad, which corresponds to a call to $fixST$. Examining the compiled code in GHC Core for our benchmarks reveals that a recursive data structure $STRep$ remains in the program for every $fixST$, preventing GHC from statically optimizing recursions at the value level.

As we move to more complex programs, however, the situation dramatically changes: the $CCNF_{ST}$ 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 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 nth_{ST} and nth_{CCNF}, we find that in each iteration nth_{CCNF} has to construct a new state as a nested tuple only for it to be destructed by the transition function immediately. In contrast, nth_{ST} 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 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 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. Voigtländer (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

Eric Cheng and Paul Hudak. Audio processing and sound synthesis in Haskell. Technical report, YALEU/DCS/RR-1405, Yale University, 2009.

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

George Giorgidze and Henrik Nilsson. Switched-on Yampa. In *Practical Aspects of Declarative Languages*, pages 282–298. Springer, 2008.

Ralf Hinze. Deriving backtracking monad transformers. In *International Conference on Functional Programming*, ICFP '00, pages 186–197, New York, NY, USA, 2000. ACM.

Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming*, pages 159–187. Springer, 2003.

Paul Hudak, Donya Quick, Mark Santolucito, and Daniel Winograd-Cort. Real-time interactive music in Haskell. In *FARM 2015*, New York, NY, USA, 2015.

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

John Hughes. Programming with arrows. In *Advanced Functional Programming*, volume 3622. Springer Berlin Heidelberg, 2005.

Adam Megacz Joseph. *Generalized Arrows*. PhD thesis, EECS Department, University of California, Berkeley, May 2014.

Oleg Kiselyov and Hiromi Ishii. Freer monads, more extensible effects. Haskell 2015, pages 94–105, New York, NY, USA, 2015. ACM.

John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In *Programming Language Design and Implementation*, PLDI '94, pages 24–35, New York, NY, USA, 1994. ACM.

Sam Lindley, Philip Wadler, and Jeremy Yallop. The Arrow Calculus. *Journal of Functional Programming*, 20(1):51–69, January 2010.

Hai Liu. *The Theory and Practice of Causal Commutative Arrows*. PhD thesis, Yale University, New Haven, CT, USA, 2011. AAI3467550.

Hai Liu and Paul Hudak. An ode to arrows. In *PADL '10: 12th International Symposium on Practical Aspects of Declarative Languages*, 2010.

Hai Liu, Eric Cheng, and Paul Hudak. Causal commutative arrows and their optimization. In *International Conference on Functional Programming*, ICFP '09, pages 35–46, New York, NY, USA, 2009. ACM.

Hai Liu, Eric Cheng, and Paul Hudak. Causal commutative arrows. *Journal of Functional Programming*, 21(4-5):467–496, 2011. ISSN 0956-7968.

Henrik Nilsson. Dynamic optimization for Functional Reactive programming using Generalized Algebraic Data Types. In *ICFP'05*, Tallinn, Estonia, 2005. ACM Press.

Ross Paterson. A new notation for arrows. In *International Conference on Functional Programming*, Firenze, Italy, September 2001.

Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. *SIGPLAN Not.*, 37(12):60–75, December 2002.

Janis Voigtländer. Asymptotic improvement of computations over free monads. In *Mathematics of Program Construction*, Berlin, Heidelberg, 2008. Springer-Verlag.

Jeremy Yallop. *Abstraction for Web Programming*. PhD thesis, University of Edinburgh, 2010.