JFP **21** (4 & 5): 467–496, 2011. © Cambridge University Press 2011
doi:10.1017/S0956796811000153 First published online 29 June 2011

Causal commutative arrows

HAI LIU*, ERIC CHENG† and PAUL HUDAK
Department of Computer Science, Yale University, New Haven, CT 06520, USA
(*e-mail:* hai.liu@aya.yale.edu,eric.cheng@aya.yale.edu,paul.hudak@yale.edu)

Abstract

Arrows are a popular form of abstract computation. Being more general than monads, they are more broadly applicable, and, in particular, are a good abstraction for signal processing and dataflow computations. Most notably, arrows form the basis for a domain-specific language called *Yampa*, which has been used in a variety of concrete applications, including animation, robotics, sound synthesis, control systems, and graphical user interfaces. Our primary interest is in better understanding the class of abstract computations captured by Yampa. Unfortunately, arrows are not concrete enough to do this with precision. To remedy this situation, we introduce the concept of *commutative arrows* that capture a noninterference property of concurrent computations. We also add an *init* operator that captures the causal nature of arrow effects, and identify its associated law. To study this class of computations in more detail, we define an extension to arrows called *causal commutative arrows* (CCA), and study its properties. Our key contribution is the identification of a normal form for CCA called *causal commutative normal form* (CCNF). By defining a normalization procedure, we have developed an optimization strategy that yields dramatic improvements in performance over conventional implementations of arrows. We have implemented this technique in Haskell, and conducted benchmarks that validate the effectiveness of our approach. When compiled with the Glasgow Haskell Compiler (GHC), the overall methodology can result in significant speedups.

1 Introduction

Consider the following recursive mathematical definition of the exponential function:

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

In Yampa (Hudak *et al.*, 2003), a domain-specific language (DSL) embedded in Haskell (Peyton Jones *et al.*, 2003), we can write this using arrow syntax (Paterson, 2001) as follows:

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

*Presently at Intel Labs, Intel Corporation.
†Presently at Google Inc.

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

Yampa has been used in a variety of applications, including robotics (Paterson *et al.*, 1999a, 1999b; Hudak *et al.*, 2003), sound synthesis (Giorgidze & Nilsson, 2008; Cheng & Hudak, 2009), animation (Hudak *et al.*, 2003), video games (Courtney *et al.*, 2003; Cheong, 2005), biochemical processes (Hudak *et al.*, 2008), control systems (Oertel, 2006), and graphical user interfaces (Courtney & Elliott, 2001; Courtney, 2004). There are several reasons that we prefer a language design based on arrows over, for example, an approach such as that used in Fran (Elliott & Hudak, 1997). First, arrows are more *direct*—they convey information about input as well as output, whereas Fran's inputs are implicit and global. Second, the use of arrows eliminates a subtle but devastating form of *space leak*, as described in Liu & Hudak (2007). Third, arrows introduce a meta level of computation that aids in reasoning about program correctness, transformation, and optimization.

But in fact, conventional arrows are not strong enough to capture the family of computations that we are interested in—more laws are needed to constrain the computation space. Unfortunately, more constrained forms of computation—such as monads (Moggi, 1991) and applicative functors (McBride & Paterson, 2008) – are not general enough. In addition, there are not enough operators. In particular, we find the need for an abstract *initialization* operator and its associated laws.

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

1. We define a notion of *commutative arrow* by extending the conventional set of arrow laws to include a commutativity law.
2. We define an *ArrowInit*-type class with an *init* operator that captures the essence of causal computation and satisfies a *product law*.
3. We define a restricted language called *CCA*, in which the above ideas are manifest. For such arrows, we establish:
   (a) a *normal form* and
   (b) a *normalization procedure*.

   We achieve this result using only CCA laws, without referring to any concrete semantics or implementation.
4. We define an *optimization technique* for causal commutative arrows that yields substantial improvements in performance over previous attempts to optimize arrow combinators and arrow syntax.
5. Finally, we show how to implement our ideas in Glasgow Haskell Compiler (GHC) to yield speedups, in certain cases, by over two orders of magnitude.

We begin the presentation with a brief overview of arrows in Section 2. The knowledgeable reader may prefer to skip directly to Section 3, where we give the definition and laws for CCA. In Section 4, we show that any CCA program can be transformed into a uniform representation that we call *causal commutative*

https://doi.org/10.1017/S0956796811000153 Published online by Cambridge University Press

normal form (CCNF). We prove that the normalization procedure is sound, based on equational reasoning using only the CCA laws. In Section 5, we discuss further optimizations and, in Section 6, their implementations in GHC. We present some benchmarks showing the effectiveness of our approach in Section 7. We discuss, in Section 8, possible extensions and, in Section 9, related work.

2 An introduction to arrows

Arrows (Hughes, 2000) are a generalization of monads that relax the stringent linearity imposed by monads, while retaining a disciplined style of composition. Arrows have enjoyed a wide range of applications, often as a domain-specific embedded language (DSEL) (Hudak, 1996; Hudak, 1998), including the many Yampa applications cited earlier, as well as parsers and printers (Jansson & Jeuring, 1999), parallel computing (Huang *et al.*, 2007), and so on. Arrows also have a theoretical foundation in category theory, where they are strongly related to (but not precisely the same as) *Freyd categories* (Power & Thielecke, 1999; Atkey, 2008).

2.1 Conventional arrows

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

```
class Arrow a where
  arr   :: (b → c) → a b c
  (>>>) :: a b c → a c d → a b d
  first :: a b c → a (b,d) (c,d)
```

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

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

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

https://doi.org/10.1017/S0956796811000153 Published online by Cambridge University Press

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

Fig. 1. Conventional arrow laws.

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

Fig. 2. Arrow loop laws.

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

$$(\star\star\star) \quad :: (Arrow\ a) \Rightarrow a\ b\ c \to a\ b'\ c' \to a\ (b,b')\ (c,c')$$
$$f \star\star\star g = first\ f \ggg second\ g$$

A mere implementation of the arrow combinators, of course, does not make it an arrow—it must additionally satisfy a set of *arrow laws*, which are shown in Figure 1.

2.2 Looping arrows

To model recursion, we can introduce a *loop* combinator (Paterson, 2001). The exponential example given in the introduction requires recursion, as do many applications in signal processing, for example. In Haskell, this combinator is captured in the *ArrowLoop* class:

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

A valid instance of this class should satisfy the additional laws shown in Figure 2. This class and its associated laws are related to the trace operator in Street *et al.* (1996) and Hasegawa (1997), which was generalized to arrows in Paterson (2001).

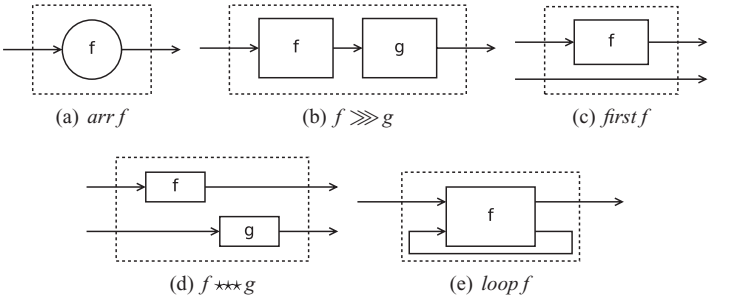
We find that arrows are best viewed pictorially, especially for applications such as signal processing, where domain experts commonly draw signal flow diagrams. Figure 3 shows some of the basic combinators in this manner, including *loop*.

2.3 Arrow syntax

Recall the Yampa definition of the exponential function given earlier:

https://doi.org/10.1017/S0956796811000153 Published online by Cambridge University Press

$$
\begin{aligned}
&\mathit{arr} \quad :: \mathit{Arrow}\ a \Rightarrow (b \to c) \to a\ b\ c\\
&(\ggg) :: \mathit{Arrow}\ a \Rightarrow a\ b\ c \to a\ c\ d \to a\ b\ d\\
&\mathit{first} \quad :: \mathit{Arrow}\ a \Rightarrow a\ b\ c \to a\ (b,d)\ (c,d)\\
&(\star\star\star) :: \mathit{Arrow}\ a \Rightarrow a\ b\ c \to a\ b'\ c' \to a\ (b,b')\ (c,c')\\
&\mathit{loop} \quad :: \mathit{Arrow}\ a \Rightarrow a\ (b,d)\ (c,d) \to a\ b\ c
\end{aligned}
$$



(a) *arr f* (b) $f \ggg g$ (c) *first f*

(d) $f \star\star\star g$ (e) *loop f*

Fig. 3. Commonly used arrow combinators.

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

Fig. 4. Causal commutative arrow laws.

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

This program is written using *arrow syntax*, introduced by Paterson (2001) and adopted by GHC (the predominant Haskell implementation) because it ameliorates the cumbersome nature of writing in the point-free style demanded by arrow combinators. The above program is equivalent to the following sugar-free program:

```
exp = fixA (integral ⋙ arr (+1))
  where fixA f = loop (second f ⋙ arr (dup . snd))
        dup x = (x, x)
```

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

3 Causal commutative arrows

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

First, as mentioned in the introduction, the set of arrow and arrow loop laws is not strong enough to capture stream computations. In particular, the *commutativity law* shown in Figure 4 establishes a noninterference property for concurrent computations – effects are still allowed, but this law guarantees that concurrent effects cannot interfere with each other. We say that an arrow is *commutative* if it

https://doi.org/10.1017/S0956796811000153 Published online by Cambridge University Press

satisfies the conventional laws as well as this critical additional law. Yampa is, in fact, based on commutative arrows.

Second, we note that Yampa has a primitive operator called *iPre* that is used to inject a delay into a computation; indeed, it is the primary effect imposed by the Yampa arrow (Hudak *et al.*, 2003). Similar operators, often called *delay*, also appear in dataflow programming (Wadge & Ashcroft, 1985), stream processing (Stephens, 1997; Thies *et al.*, 2002), and synchronous languages (Caspi *et al.*, 1987; Colaço *et al.*, 2004). In all cases, the operator introduces stateful computation into an otherwise stateless setting.

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

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

Intuitively, the argument to *init* is the initial output; subsequent output is a copy of the input to the arrow. It captures the essence of causal computations, namely that the current output depends only on the current as well as previous inputs. Besides causality, we make no other assumptions about the nature of these values: they may or may not vary with time, and the increment of change may be finite or infinitesimally small.

More importantly, a valid instance of the *ArrowInit* class must satisfy the *product* law shown in Figure 4. This law states that two *init*s paired together are equivalent to one *init* of a pair. Here we use the *** operator instead of its expanded definition *first*... ⋙ *second*... to imply that the product law assumes commutativity.

We will see in a later section that *init* and the product law are critical to our normalization and optimization strategies. But *init* is also important in allowing us to define operators that were previously taken as domain-specific primitives. In particular, consider the *integral* operator used in the exponentiation examples. With *init*, we can define *integral* using the Euler integration method and a fixed global step *dt* as follows:

```
integral :: ArrowInit a ⇒ a Double Double
integral = loop (arr acc ⋙ init 0 ⋙ arr dup) where acc (x, i) = i + dt * x
dup x = (x, x)
```

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

- *SF a b* is an arrow representing functions (transformers) from streams of type *a* to streams of type *b*. It is essentially a recursively defined data type consisting of a function with its continuation, a concept closely related to a form of finite state automaton called a *Mealy Machine* (Mealy, 1955). *Yampa* employs a similar implementation, and the same data type was called *Auto* in Paterson (2001).
- *SF* is declared an instance of type classes *Arrow*, *ArrowLoop*, and *ArrowInit*. For example, *exp* can be instantiated as type *exp* :: *SF* () *Double*. These

https://doi.org/10.1017/S0956796811000153 Published online by Cambridge University Press

```
newtype SF a b = SF {unSF :: a → (b, SF a b)}
instance Arrow SF where
  arr f = SF h          where h x       = (f x,SF h)
  first f = SF (h f)    where h f (x,z) = let (y,f') = unSF f x
                                          in ((y,z), SF (h f'))
  f >>> g = SF (h f g) where h f g x    = let (y,f') = unSF f x
                                              (z,g') = unSF g y
                                          in (z,SF (h f' g'))
instance ArrowLoop SF where
  loop f = SF (h f)     where h f x     = let ((y,z),f') = unSF f (x,z)
                                          in (y,SF (h f'))
instance ArrowInit SF where
  init i = SF (h i)     where h i x     = (i,SF (h x))
run_sf :: SF a b → [a] → [b]
run_sf f = g f        where g f (x:xs) = let (y,f') = unSF f x
                                         in y:g f' xs
```

Fig. 5. Causal stream transformer.

instances obey all of the arrow laws, including the two additional laws that we introduced.

- $run_{sf} :: SF\ a\ b \to [a] \to [b]$ converts an SF arrow into a stream transformer that maps an input stream of type $[a]$ to an output stream of type $[b]$.

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

```
dt  = 0.01          :: Double
inp = () : inp      :: [()]
* Main > run_sf exp inp
[1.0, 1.01, 1.0201, 1.030301, 1.04060401, 1.0510100501, ...
```

We must stress that the SF type is but one instance of a causal commutative arrow, and alternative implementations such as the synchronous circuit type *SeqMap* in Paterson (2001) and the stream-function-type (incidentally also called) SF in Hughes (2004) also qualify as valid instances. The abstract properties such as normal forms that we develop in the next section are applicable to any of these instances, and thus are more broadly applicable than optimization techniques based on a specific semantic model, such as the one considered in Capsi & Pouzet (1998).

4 Normalization of CCA

In most implementations, arrow programs carry a run-time overhead, primarily due to the use of a data structure for arrow instances, as well as the extra tupling forced onto function's arguments and return values. There have been several attempts (Hughes, 2004; Nilsson, 2005) to optimize arrow-based programs using arrow laws, but the result has not been entirely satisfactory. Although conventional arrow and arrow loop laws offer ways to combine pure arrows and collapse nested loops, they are not specific enough to target *effectful* arrows, such as the *init* combinator.

https://doi.org/10.1017/S0956796811000153 Published online by Cambridge University Press

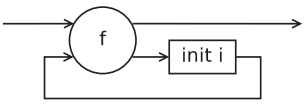
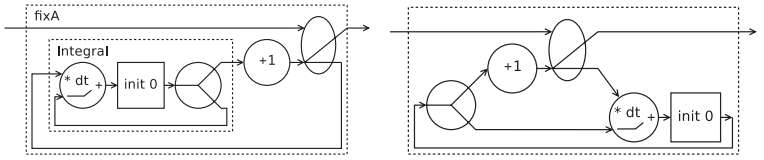


Fig. 6. Diagram for *loopD*.



(a) Original (b) Optimized

Fig. 7. Diagrams for *exp*.

Certain effectful arrows are dynamically optimized in Nilsson (2005), but they are based on somewhat *ad-hoc* laws, and there are no normal forms.

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

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

where f is a pure function and i is an initial state. Note that all the *essential* arrow combinators, arr, $\ggg$, $second$, $loop$, and $init$, are used *exactly once*, and therefore all of the overheads (tupling, etc.) associated with multiple occurrences and compositions of the arrow combinators are completely eliminated. Not surprisingly, the resulting improvement in performance is rather dramatic, as we will see later.

First, we define a combinator called *loopD* that can be viewed as syntactic sugar for the above form:

$$\begin{aligned} &loopD \quad :: ArrowInit\ a \Rightarrow d \to ((b,d) \to (c,d)) \to a\ b\ c \\ &loopD\ i\ f = loop\ (f \ggg second\ (init\ i)) \end{aligned}$$

A pictorial view of *loopD* is given in Figure 6. The second argument to *loopD* is a pure function mapping a tuple of (b,d) to (c,d), where the value of type d is initialized before looping back, and is often regarded as an internal state.

As a concrete example, Figure 7(a) is a diagram of the original *exp* example given earlier. In Figure 7(b), we have in-lined the definition of *integral* and applied the optimization strategy. The result is a single loop, where all pure functions can be combined together to minimize arrow implementation overheads.

To be more precise as to what kind of arrows are subject to normalization, we want to restrict our discussion in this section on well-formed and closed CCA terms.

Definition 4.1 (*CCA*)
A CCA is a valid instance of the *ArrowInit* class, whose terms are definable from only the following combinators: arr, $first$, $\ggg$, $loop$, and $init$. All CCAs must satisfy the two CCA laws given in Figure 4 in addition to the arrow and arrow loop laws.

https://doi.org/10.1017/S0956796811000153 Published online by Cambridge University Press

composition	$arr\ f \ggg arr\ g \mapsto arr\ (g\ .\ f)$
left tightening	$arr\ f \ggg loopD\ i\ g \mapsto loopD\ i\ (g\ .\ (f \times id))$
right tightening	$loopD\ i\ f \ggg arr\ g \mapsto loopD\ i\ ((g \times id)\ .\ f)$
sequencing	$loopD\ i\ f \ggg loopD\ j\ g \mapsto loopD\ (i,j)\ (assoc'\ (juggle'\ (g \times id)\ .\ (f \times id)))$
extension	$first\ (arr\ f) \mapsto arr\ (f \times id)$
superposing	$first\ (loopD\ i\ f) \mapsto loopD\ i\ (juggle'\ (f \times id))$
loop-extension	$loop\ (arr\ f) \mapsto arr\ (trace\ f)$
vanishing	$loop\ (loopD\ i\ f) \mapsto loopD\ i\ (trace\ (juggle'\ f))$

$$f \times g\ (x,y) = (f\ x, g\ y) \qquad swap\ (x,y) = (y,x)$$
$$assoc\ \ ((x,y),z) = (x,(y,z)) \qquad trace\ f\ x = \textbf{let}\ (y,z) = f\ (x,z)\ \textbf{in}\ y$$
$$assoc^{-1}\ (x,(y,z)) = ((x,y),z) \qquad juggle\ ((x,y),z) = ((x,z),y)$$
$$assoc'\ f = assoc\ .\ f\ .\ assoc^{-1} \qquad juggle'\ f = juggle\ .\ f\ .\ juggle$$

Fig. 8. Single-step reduction for CCA.

$$\text{(NORM1)}\ \frac{}{arr\ f \Downarrow arr\ f} \qquad \text{(NORM2)}\ \frac{}{loopD\ i\ f \Downarrow loopD\ i\ f}$$

$$\text{(INIT)}\ \frac{}{init\ i \Downarrow loopD\ i\ swap} \qquad \text{(SEQ)}\ \frac{e_1 \Downarrow e_1' \quad e_2 \Downarrow e_2' \quad e_1' \ggg e_2' \mapsto e}{e_1 \ggg e_2 \Downarrow e}$$

$$\text{(FIRST)}\ \frac{f \Downarrow f' \quad first\ f' \mapsto e}{first\ f \Downarrow e} \qquad \text{(LOOP)}\ \frac{f \Downarrow f' \quad loop\ f' \mapsto e}{loop\ f \Downarrow e}$$

Fig. 9. Normalization of CCA.

Definition 4.1 effectively means:

1. We are only concerned with CCAs written in closed terms, without referencing arrow definitions from the environment (syntactic sugar is still allowed).
2. Such arrows must not be recursively defined, otherwise normalization would fail to terminate.
3. There is no lambda abstraction or application at the arrow level so that we can avoid talking about beta reduction or variable substitution of arrow terms. Note, however, that we do allow all forms of functions, including recursive ones, to be lifted by *arr*.

Definition 4.1 may seem too restrictive as it precludes any form of modularity for CCA since all terms are closed. We believe this is only a necessary step to avoid complications in formalizing CCA properties, and can certainly be relaxed in real-world implementations.

Our intuition behind the normalization process is to extend arrow loop laws to *loopD*, so that we only get the *loopD* form as a result. Formally, we define a single-step reduction $\mapsto$ for CCA as a set of rules in Figure 8, and a normalization procedure in Figure 9. The normalization relation $\Downarrow$ can be seen as a large-step reduction following an innermost strategy, and is indeed a function.

https://doi.org/10.1017/S0956796811000153 Published online by Cambridge University Press

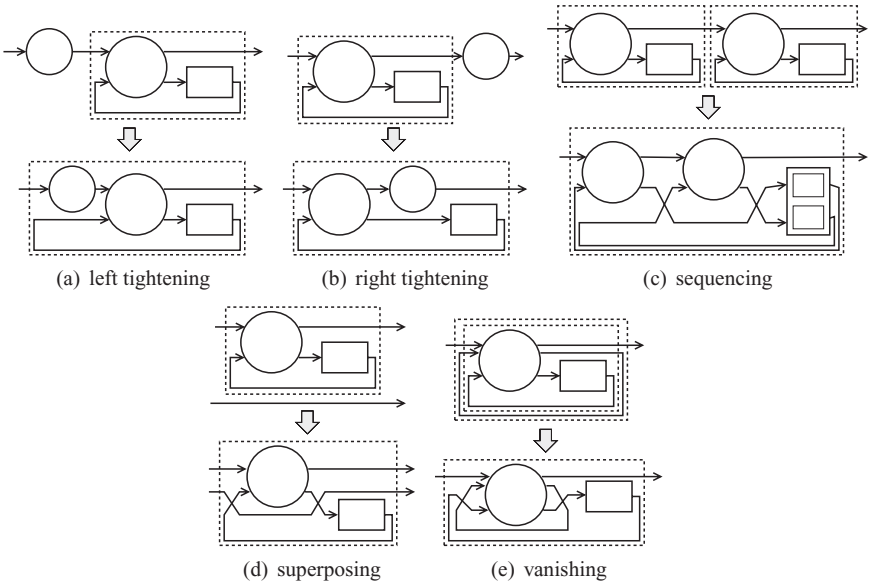


Fig. 10. Illustrations of reduction rules.

Note that some of the reduction rules resemble the arrow laws of the same name. However, there are some subtle but important differences: First, unlike the laws, reduction is directed. Second, the rules are extended to handle *loopD* instead of *loop*.

To see how this works, it is helpful to visualize a few examples of the reduction rules in Figure 8, as shown in Figure 10. We omit the simpler rules that follow directly from the laws, and only show those that involve *loopD*. The diagrams in Figure 10 can be explained as follows:

(a) **left tightening**. Figure 10(a) shows that we can move a pure arrow from a left composition inside a *loopD* arrow. This follows directly from the left tightening law of *loop*.

(b) **right tightening**. Figure 10(b) shows that we can move a pure arrow from a right composition inside a *loopD* arrow so that it fuses with the pure function inside the *loopD*. This follows directly from the left tightening law of *loop* and the commutativity law of CCA.

(c) **sequencing**. Figure 10(c) shows that we can combine two *loopD* arrows into one. In doing so, we have to reroute part of the computation, and make use of product law to fuse two *init* arrows into one. Notice that there is also a reordering of a pure arrow and an *init* arrow, which is due to the commutativity law of CCA.

(d) **superposing**. Figure 10(d) shows a variant of the superposing law for *loopD* using *first* instead of *second*. Instead of an outer parallel composition, the second line simply passes through the *loopD* unchanged with some rerouting.

(f) **vanishing**. Figure 10(e) shows an extension of the vanishing law for loops to handle *loopD*. Since the outer loop only acts on the pure function, it can be

https://doi.org/10.1017/S0956796811000153 Published online by Cambridge University Press

moved inside and composed with the *trace* function due to the loop extension law.

Lemma 4.1 (*Soundness*)
The reduction rules given in Figure 8 are both type and semantics preserving, i.e., if $e \mapsto e'$ then $e = e'$ is syntactically derivable from the set of CCA laws.

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

Lemma 4.2 (*Termination*)
The normalization procedure for CCA given in Figure 9 terminates for all CCAs.

Proof: By structural induction over all possible definitions of a CCA program. The rules NORM1, NORM2, and INIT are the base case, and SEQ, FIRST, and LOOP cover the inductive case, where the sub arrows (such as e_1 and e_2 in $e_1 \ggg e_2$, and f in *first* f and *loop* f) are normalized inductively. It also explains why there are exactly eight reduction rules for $\mapsto$, because the premises of SEQ, FIRST, and LOOP require four, two, and two reduction rules, respectively. □

Theorem 4.1 (*CCNF*)
For any CCA term $e :: ArrowInit\ a \Rightarrow a\ b\ c$ satisfying Definition 4.1, there exists a normal form e_{norm}, called the causal commutative normal form, which is either of the form *arr* f, or *loopD* i f for some i and f, such that $e_{norm} :: ArrowInit\ a \Rightarrow a\ b\ c$, and $e \Downarrow e_{norm}$. In unsugared form, the second form is equivalent to *loop* (*arr* $f \ggg$ *second* (*init* i)).

Proof
Follows from Lemmas 4.1 and 4.2. It is easy to see that when the normalization algorithm (Figure 9) terminates, the result is a CCNF. □

Since we have presented a normal form and proved the soundness of its reduction rules, a natural question to ask is whether the result leads to completeness. Unfortunately, this is not the case. For example, the following two arrows f and g are already in CCNF, and they are both equivalent to *init* $0 \ggg$ *init* 1 with respect to the CCA laws:

$$f = loopD\ (0,1)\ (\lambda(x,(y,z)) \rightarrow (z,(y,x)))$$
$$g = loopD\ (1,0)\ (\lambda(x,(y,z)) \rightarrow (y,(z,x)))$$

With some tricks of tuple reshuffling, we can easily convert f to g or vice versa in a sound way respecting the CCA laws, but still, they are both normal forms, and yet different.

Moreover, due to the introduction of general recursion at value level through the loop combinator, it is hard to reason about equivalence relationship when there is nontermination. For example, the following two arrows should both reduce to

https://doi.org/10.1017/S0956796811000153 Published online by Cambridge University Press

nonterminating traces that are bottom, but this equivalence is not derivable from the laws:

$$p = loop\ (arr\ (\lambda(\_, x) \to (x, x+1)))$$
$$q = loop\ (arr\ (\lambda(\_, x) \to (x, x+2)))$$

After all, the lack of completeness does not necessarily undermine the importance or the usefulness of our technique, as will be shown in the next section.

5 Optimization

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

SF Arrow One observation is that instead of defining *loopD* as syntactic sugar, we can implement it directly for a given arrow instance. For instance, using the *SF* data type shown in Figure 5, *loopD* can be defined as

$$loopD\ i\ f = SF\ (g\ i)\ \textbf{where}\ g\ i\ x = \textbf{let}\ (y, i') = f\ (x, i)\ \textbf{in}\ (y, SF\ (g\ i'))$$

As a more concrete example, a special case of run_{sf} (defined in Section 3) often used in practice is computing the nth element of the output stream when the input is a constant unit stream. This gives the following function that avoids constructing the output stream using the list data structure:

$$nth_{sf} :: Int \to SF\ ()\ b \to b$$
$$nth_{sf}\ n\ (SF\ f) = x\ `seq`\ \textbf{if}\ n \equiv 0\ \textbf{then}\ x\ \textbf{else}\ nth_{sf}\ (n-1)\ f'\ \textbf{where}\ (x, f') = f\ ()$$

Notice the use of *seq* in order to force strict evaluation in each iteration because we want the computation of each iteration to finish before next iteration, instead of being postponed until the very end due to laziness.

CCNF Tuple Given the fact that a CCNF is no more than a tuple of a state and a pure function, we can drop the *SF* data structure altogether by simply using the tuple instead of an arrow written in *loopD* form. Correspondingly, we can define the stream transformer run_{ccnf} and the nth element evaluator nth_{ccnf} for CCNF tuples:

$$run_{ccnf} :: (d, ((b, d) \to (c, d))) \to [b] \to [c]$$
$$run_{ccnf}\ (i, f) = g\ i\ \textbf{where}\ g\ i\ (x : xs) = \textbf{let}\ (y, i') = f\ (x, i)\ \textbf{in}\ y : g\ i'\ xs$$
$$nth_{ccnf} :: Int \to (d, ((), d) \to (c, d)) \to c$$
$$nth_{ccnf}\ n\ (i, f) = aux\ n\ i\ \textbf{where}\ aux\ n\ i = x\ `seq`\ \textbf{if}\ n \equiv 0\ \textbf{then}\ x\ \textbf{else}\ aux\ (n-1)\ i'$$
$$\textbf{where}\ (x, i') = f\ ((), i)$$

Instead of taking an arrow, the above two functions just take a CCNF tuple and use the pure function to update the state in a loop computation. In doing so, we have successfully transformed away all arrow instances, including the data structure used to implement them!

https://doi.org/10.1017/S0956796811000153 Published online by Cambridge University Press

In-lining As we have mentioned in Section 4, the normalization of CCA only acts on fully in-lined arrow terms. For example, in order to normalize the *exp* arrow, we need to in-line *integral*. After the in-lining, we then normalize it to the CCNF below:

$$
\begin{aligned}
&exp = loopD\ exp_i\ exp_f \\
&exp_i = 0 \\
&exp_f = trace\ (juggle\ .\ ((dup\ .\ snd) \times id)\ .\ (swap \times id)\ .\ juggle\ .\ ((((+1) \times id)\ .\\
&\qquad trace\ (juggle\ .\ (dup \times id)\ .\ swap\ .\ ((\lambda(x,i) \rightarrow i + dt * x) \times id)\ .\ juggle))\ \times id)\ .\\
&\qquad juggle\ .\ (swap \times id)\ .\ juggle)
\end{aligned}
$$

The CCNF tuple for the *exp* arrow is just (exp_i, exp_f). Notice the sequence of function compositions in exp_f is the result of our CCA normalization procedure.

To take the in-lining one step further, we can simplify the pure function from a CCNF tuple, such as exp_f given above. The default optimization and in-lining techniques supported by GHC can already do this without any user intervention. We demonstrate this step with the above CCNF for *exp*, though the technique is equally applicable to any CCNF.

For example, to compute the *n*th element of the exponential sequence, we can just apply run_{ccnf} to the CCNF tuple (exp_i, exp_f) like this:

$$
\begin{aligned}
&nth_{exp} :: Int \rightarrow Double \\
&nth_{exp}\ n = nth_{ccnf}\ n\ (exp_i, exp_f)
\end{aligned}
$$

When given proper optimization flags, GHC is able to agressively optimize the above code and fully in-line all functions in the definition of exp_f and nth_{exp}. The following is the equivalent intermediate representation extracted from GHC after optimization:

$$
\begin{aligned}
&nth_{exp}\ n = \textbf{case}\ n\ \textbf{of}\ \{I\#\ m \rightarrow go\ 0.0\ m\} \\
&go :: Double\# \rightarrow Int\# \rightarrow Double \\
&go\ i\ n = \textbf{case}\ n\ \textbf{of} \\
&\qquad\qquad \_ DEFAULT \rightarrow go\ (i + (dt * (i + 1.0)))\ (n - 1) \\
&\qquad\qquad 0 \qquad\qquad\ \rightarrow D\#\ (i + 1.0)
\end{aligned}
$$

As we can see, GHC has successfully in-lined not only the function nth_{ccnf} but also exp_i and exp_f, and transformed everything into a tight loop using only strict and unboxed types (those marked by #). This kind of aggressive optimization essentially results in compiling a CCA program directly to a tight loop that is free of any memory allocation of intermediate data structures.

6 Implementation

6.1 Template Haskell-based implementation

We implement CCA normalizations in Haskell with help from Template Haskell (Sheard & Peyton Jones, 2002), an extension to Haskell that allows type-safe compile-time meta-programming. Our compilation process consists of three steps:

https://doi.org/10.1017/S0956796811000153 Published online by Cambridge University Press

1. The source arrow program is translated to an abstract syntax tree (AST).
2. The AST is then normalized to CCNF.
3. The result is spliced back into the original program before the Haskell compiler finishes the rest of compilation.

The first step requires in-lining of all arrow terms in preparation for the second step that does the actual normalization. So merely grabbing the AST of a single CCA definition is not enough since it may contain references to other definitions. Our solution here is to allow a generic *ArrowInit* instance to be instantiated as an AST directly from within Haskell, so that when we evaluate such a term, we get a full AST. This is performed by the Haskell compiler at the meta level and can help achieving similar effects of in-lining or substitutions. The following code snippet demonstrates this approach:

```
data AExp = Arr ExpQ
          | First AExp
          | AExp :>>> AExp
          | Loop AExp
          | Init ExpQ
          | LoopD ExpQ ExpQ
newtype ASyn b c = AExp AExp
```

The *AExp* data type represents an AST for CCA, and *ASyn b c* employs phantom types so that we can declare *ASyn* to be an instance of the *Arrow*, *ArrowLoop*, and *ArrowInit* type classes. The *ExpQ* type used here is the internal syntactic representation of a Haskell expression provided by Template Haskell.

For example, the *Arrow* instance of *Asyn* can be declared as follows:

```
instance Arrow ASyn where
  arr f = error "use arr' instead"
  AExp f >>> AExp g = AExp (f :>>> g)
  first (AExp f) = AExp (First f)
```

As we can see here, the usual arrow combinators are just syntactic operations over the *AExp* data type. The problem, however, is in defining the *arr* combinator. For instance, consider the following program:

```
f :: Arrow a => a b c
g :: Arrow a => a b' c'
h :: Arrow a => a (b,b') (c,c')
h = first f >>> arr swap >>> first g >>> arr swap
```

We can obtain the AST for *h* by instantiating the generic arrow type *a* to *Asyn*. This step is automatic because any function over type *ASyn u v* can be applied to any generic arrow of type *Arrow a => a u v*. Simply, evaluating *h :: ASyn (b,b') (c,c')* in a Haskell interpreter such as GHCi shall return its AST as something like below:

```
AExp (((First f' :>>> Arr swap) :>>> First g') :>>> Arr swap)
```

https://doi.org/10.1017/S0956796811000153 Published online by Cambridge University Press

where f' and g' stand for the AST for f and g, respectively. The instantiation of f and g is automatic because the concrete arrow types for f and g are inferred to be *ASyn* too. Another way to look at this is that $AExp\ f' :: ASyn\ b\ c$ and $AExp\ g' :: ASyn\ b'\ c'$ are instances of the generic arrow f and g.

The real issue here, however, is that we cannot automatically reify a Haskell function such as *swap* to the meta level, so the above AST for h has a type error, because *Arr swap* will not type check.

Template Haskell can reify certain expressions to the meta level, and this step is called *quotation*. But it cannot do so for all values, and requires explicit quotation using a special syntax [|...|] for certain things, such as references to global definitions.

To work around this problem, we ask the programmer to always state the quotation explicitly, and disallow direct usage of *arr* as indicated in the arrow instance delcaration for *ASyn*. Instead of *arr*, we provide an *arr'* function that additionally takes a quoted expression, for example:

$$arr'\ [|\ \lambda x \to (x, x)\ |]\ (\lambda x \to (x, x))$$

The above would give us the needed AST for the pure function $\lambda x \to (x, x)$ in addition to the function itself, where the [|...|] operation is a special Template Haskell syntax for quoting ("quotable") Haskell expression into an *ExpQ* representation. We provide *arr'* (as well as *init'*, since *init* faces a similar problem) in the *ArrowInit* class defined below:

$$\begin{aligned}&\textbf{class}\ (Arrow\ a, ArrowLoop\ a) \Rightarrow ArrowInit\ a\ \textbf{where}\\ &\quad init :: b \to a\ b\ b\\ &\quad arr' :: ExpQ \to (b \to c) \to a\ b\ c\\ &\quad init' :: ExpQ \to b \to a\ b\ b\\ &\quad loopD :: e \to ((b, e) \to (c, e)) \to a\ b\ c\end{aligned}$$

Since asking the programmer to write in *arr'* gets tedious over time, we provide a modified arrow syntax translator that directly outputs combinator programs written in *arr'* and *init'* instead of *arr* and *init*.

The actual normalization of an *AExp* can be straightforwardly implemented as a traversal using the algorithms given in Figures 8 and 9. We omit the details here by just saying the CCNF of an *AExp* is eventually represented by either the *Arr* or the *LoopD* constructors.

We provide the normalization function as a Template Haskell splice operation:

$$norm :: ArrowInit\ a \Rightarrow a\ b\ c \to ExpQ$$

So if we evaluate \$(*norm e*) for any generic arrow $e :: ArrowInit\ a \Rightarrow a\ b\ c$, what happens in the background is that it will first instantiate e to an *ASyn* arrow which is internally represented by the *AExp* data type, then normalized to a *LoopD* (or *Arr*) form, and finally spliced back as an Haskell expression $loopD\ i\ f$ (or $arr\ f$) for some i and f. The \$(...) operation is a special Template Haskell syntax for splicing. Since GHC has native support for Template Haskell, the entire process happens without any user intervention during either the compilation of a Haskell source program using GHC, or an intepretive session using the interactive Haskell evaluator *GHCi*.

https://doi.org/10.1017/S0956796811000153 Published online by Cambridge University Press

6.2 Technical limiations

The use of Template Haskell allows a very simple implementation of the meta-operations required for CCA normalization, and its integration with GHC gives a seamless user experience. The most significant advantage, however, is that all normalizations are done at compile-time, so that GHC may further optimize the result through strictness analysis, unboxing, in-lining, and various other techniques.

An immediate restriction of this approach is that a full AST has to be made available at compile-time, which is only applicable to a subset of all *ArrowInit* instances. Alternative implementations may lift this restriction by allowing normalization at runtime or even utilizing a JIT compiler, and we leave this to the future work.

Besides the compile-time restriction, one caveat to this approach has to do with a limitation of Template Haskell. For example, suppose we define a constant arrow like this:

$$\begin{aligned} &\mathit{constant} :: \mathit{ArrowInit}\ a \Rightarrow c \to a\ b\ c \\ &\mathit{constant}\ x = \mathbf{proc}\ \_ \to \mathit{returnA} \prec x \end{aligned}$$

When we try to instantiate the above arrow type a to *ASyn*, the compiler will complain that the type for *constant* is wrong, and insist that type c must be a member of the *Lift* class, which is how Template Haskell reifies a Haskell value to the meta level. To see exactly where is the problem, we translate the above from arrow syntax to combinators using arr':

$$\mathit{constant}\ x = arr'\ [|\ \lambda\_ \to x\ |]\ (\lambda\_ \to x)$$

The quotation of the lambda expression $\backslash\_ \to x$ above has a reference to x, a parameter of the function *constant*, and Template Haskell cannot quote it unless it knows how to reify the value of x to the meta level, and hence requires that the type for x is an instance of the *Lift* class.

On the other hand, our goal in the first step of in-lining CCA definitions is not to lift arbitrary Haskell values, but to generate top-level code that could somehow still relate the occurrence of variable x in a quotation to the actual parameter of *constant* symbolically, so that we know how to handle substitution without evaluating the actual value of x. In order to do so in Template Haskell would require quoting the entire definition of *constant* and restricting variable x to be of the *ExpQ* type, the type for quoted expressions. This will prevent us from reusing the same unmodified code without CCA normalization, and hence compromise one of our initial goals of making the implementation nonintrusive.

A more effective solution is to perform in-linings and substitutions at the meta level, for example, using a heavily customized arrow syntax preprocessor, which will no longer rely on the evaluation of Haskell terms, and hence bypass the whole reification issue. Unfortunately, this is beyond what Template Haskell does, and we leave it to future work.

https://doi.org/10.1017/S0956796811000153 Published online by Cambridge University Press

Table 1. *Performance ratio (greater is better)*

Name	*GHC*	*arrowp*	$CCNF_{sf}$	$CCNF_{tuple}$
exp	1.0	3.58	30.84	672.79
sine	1.0	2.81	18.89	442.48
oscSine	1.0	2.91	14.28	29.53
50's sci-fi	1.0	3.15	18.72	21.37
robotSim	1.0	2.84	24.67	34.93

7 Benchmarks

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

1. Compiled with GHC, which has a built-in translator for arrow syntax, and ran nth_{sf} on the resulting arrow. (*GHC*)
2. Translated using Paterson's *arrowp* preprocessor to arrow combinators, compiled with GHC, and ran nth_{sf} on the resulting arrow. (*arrowp*)
3. Normalized to CCNF, compiled with GHC and ran nth_{sf} on the normalized arrow. ($CCNF_{sf}$)
4. Normalized to CCNF, compiled with GHC and ran nth_{ccnf} on the CCNF tuple. ($CCNF_{tuple}$)

The five benchmarks we used are the exponential function given earlier, a sine wave with fixed frequency using Goertzel's method (Goertzel, 1958), a sine wave with variable frequency, "50's sci-fi" sound synthesis program taken from Giorgidge & Nilsson (2008), and a robot simulator taken from Hudak *et al.* (2003). The programs were compiled and run on an Intel Atom N270 1.6GHz machine with GHC version 6.10.4, using compilation options `-O2 -fvia-C -fno-method-sharing -fexcess-precision`. We measured the CPU time used to run a program through 10^6 samples. The results are shown in Table 1, where the numbers represent normalized speedup ratios, and we include the source program for all benchmarks in Appendix A.

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

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

https://doi.org/10.1017/S0956796811000153 Published online by Cambridge University Press

$$\textbf{class}\ \mathit{Arrow}\ a \Rightarrow \mathit{ArrowChoice}\ a\ \textbf{where}$$
$$\mathrm{left} :: a\ b\ c \rightarrow a\ (\mathit{Either}\ b\ d)\ (\mathit{Either}\ c\ d)$$

extension	$\mathit{left}\ (\mathit{arr}\ f) = \mathit{arr}\ (f \oplus \mathit{id})$
functor	$\mathit{left}\ (f \ggg g) = \mathit{left}\ f \ggg \mathit{left}\ g$
exchange	$\mathit{left}\ f \ggg \mathit{arr}\ (\mathit{id} \oplus g) = \mathit{arr}\ (\mathit{id} \oplus g) \ggg \mathit{left}\ f$
unit	$\mathit{arr}\ \mathit{left} \ggg \mathit{left}\ f = f \ggg \mathit{arr}\ \mathit{left}$
association	$\mathrm{left}\ (\mathit{left}\ f) \ggg \mathit{arr}\ \mathit{assocsum} = \mathit{arr}\ \mathit{assocsum} \ggg \mathit{left}\ f$

$$f \oplus g\ (\mathit{Left}\ x) = \mathit{Left}\ (f\ x) \qquad \mathit{assocsum}\ (\mathit{Left}\ (\mathit{Left}\ x)) = \mathit{Left}\ x$$
$$f \oplus g\ (\mathit{Right}\ y) = \mathit{Right}\ (g\ y) \qquad \mathit{assocsum}\ (\mathit{Left}\ (\mathit{Right}\ x)) = \mathit{Right}\ (\mathit{Left}\ x)$$
$$\mathit{assocsum}\ (\mathit{Right}\ x) = \mathit{Right}\ (\mathit{Right}\ x)$$

Fig. 11. *ArrowChoice* class and its laws.

8 Expressiveness and extensions

The CCA language remains highly abstract due to the use of the arrow laws. Even though in Functional Reactive Programming (FRP) or dataflow programming, we tend to think very operationally of *init* as a unit delay, there is no reason to confine CCA only to this application domain. For instance, CCA also forms the basis of a DSL for ordinary differential equations (ODEs), and arrows are indeed a better choice in implementation DSL level let-expressions in comparison to tagged AST, or higher-order abstract syntax (HOAS), because they help maintain the sharing of computation and avoid space leaks (Liu & Hudak, 2010).

Many dataflow and stream programming languages provide conditionals, such as **if-then-else**, as part of the language (Wadge & Ashcroft, 1985; Caspi *et al.*, 1987). Conditionals at the arrow level are captured by the *ArrowChoice* class together with a set of the *ArrowChoice* laws shown in Figure 11. We can easily extend the reduction rules in Figure 8 to handle *ArrowChoice* as follows:

extension	$\mathit{left}\ (\mathit{arr}\ f) \mapsto \mathit{arr}\ (f \oplus \mathit{id})$
superposition	$\mathit{left}\ (\mathit{loopD}\ i\ f) \mapsto \mathit{loopD}\ i\ (\mathit{tag}^{-1}\ .\ (f \oplus \mathit{id})\ .\ \mathit{tag})$

$$\mathit{tag}\ (\mathit{Left}\ \ x, y) = \mathit{Left}\ \ (x, y) \qquad \mathit{tag}^{-1}\ (\mathit{Left}\ \ (x, y)) = (\mathit{Left}\ \ x, y)$$
$$\mathit{tag}\ (\mathit{Right}\ x, y) = \mathit{Right}\ (x, y) \qquad \mathit{tag}^{-1}\ (\mathit{Right}\ (x, y)) = (\mathit{Right}\ x, y)$$

The soundness of the above extension to the reduction rules can be easily proved with respect to the arrow choice laws shown in Figure 11, and we omit the details here.

We also need a new inference rule for the normalization procedure shown in Figure 9, which is given below:

$$(\mathrm{LEFT})\ \frac{f \Downarrow f' \quad \mathit{left}\ f' \mapsto p}{\mathit{left}\ f \Downarrow p}$$

Similarly, termination can be proved for the above extension. It can also be easily shown that the above extensions requires no modification to CCNF, and Theorem 4.1

https://doi.org/10.1017/S0956796811000153 Published online by Cambridge University Press

still holds. In other words, CCA extended with *ArrowChoice* can still be normalized to the same normal form as in the original CCA.

On the other hand, there are also things not representable in CCA. For example, the switch combinator introduced in Yampa is able to dynamically replace a running arrow with a new one depending on an input event, and hence to switch the system behavior completely. With CCA, there is no way to change the compositional structure of the arrow program itself at run time.

It should also be noted that the local state introduced by *init* is one of the minimal side effects one can introduce to arrow programs. The commutativity law for CCA ensures that the effect of one arrow cannot interfere with another when composed together, and it is no longer satisfiable when such ordering becomes important, for example, when arrows are used to model parsers and printers (Jansson & Jeuring, 1999).

9 Related work

9.1 Alternative formalisms

Apart from arrows, other formalisms such as monads, comonads, and applicative functors have been used to model computations over data streams (Bjesse *et al.*, 1998; Uustalu & Vene, 2005; McBride & Paterson, 2008). Central to many of these approaches are the representation of streams and computations about them. However, notably missing are the connections between stream computation and the related laws. For example, Uustalu and Vene (2005) concluded that comonad is a suitable model for dataflow computation, but it lacks the discussion on comonadic laws.

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

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

Lindley *et al.* (2010) give a more explicit explanation of the arrow laws by constructing an arrow calculus and turning the nine arrow laws into five laws for the calculus, and discover a redundancy in the original nine arrow laws. Unfortunately, arrow loop is not included in their formulation.

The *loop* combinator and its arrow loop laws play a key role in CCA normalization because multiple loops can be fused together, and nested loops can be collapsed into just one. This is actually very close to yet another instantiation of the Folk Theorem (Harel, 1980) that all computer programs can be simulated by a single while-loop, if not for the fact that arrows and arrow loop only model a specific subset of but not all computations. We are interested in both the generality and the discipline brought forward by the laws.

On the topic of program optimization under an FRP or arrow setting, Burchett *et al.* (2007) introduce a concept called "lowering" that helps fuse pure functions in FrTime, a strict FRP language embedded in Scheme, but unfortunately it does not handle stateful computation such as the single unit delay; Nilsson (2005) makes use of several arrow laws and generalized algebraic types to optimize Yampa implementation, and, in particular, some stateful computations and event processing; Sculthorpe and Nilsson (2008) consider change propagation as a means to optimize Yampa programs with a dynamic structure.

9.2 Coalgebraic streams and synchronous languages

The coalgebraic property of streams is well known, and most relevant to our work is Caspi and Pouzet's representation of stream and stream functions in a functional language setting (Caspi & Pouzet, 1998), which also uses a primitive similar to the trace operator (and hence the arrow loop combinator) to model recursion. Their compilation technique, however, lacks a systematic approach to optimize nested recursions. We consider our technique more effective and more abstract.

Most synchronous languages, including the one introduced by Caspi and Pouzet (1998), are able to compile stream programs into a form called *single loop code* by performing a causality analysis to break the feedback loop of recursively defined values. Many efforts have been made to generate efficient single loop code (Halbwachs *et al.*, 1991; Amagbegnon *et al.*, 1995), usually by a compilation from a high-level dataflow source language to a target language that is usually imperative and low level, but few express the transformation at the source level to reach a normal form with strong characterization. Our discovery of CCNF is original, and the optimization by normalization approach is targeting a lazy functional language, namely Haskell, and making use of an advanced Haskell compiler to further optimize and produce low-level code.

Traditional compilation techniques for synchronous dataflow also had a modularity problem: they either require in-lining of all definitions in order to properly analyze feedback loops and thus lose modularity, or impose too strong a causality constraint that every feedback loop must cross an explicit delay even for submodules. Recently, a range of solutions aim to address this problem (Lublinerman & Tripakis, 2008; Lublinerman *et al.*, 2009; Pouzet & Raymond, 2009) using techniques centered around the decomposition mechanism first proposed by Raymond (1988). However, such a problem simply ceases to exist when we adopt a lazy functional language as an intermediate or even the target language, for instance, as in our staged compilation for CCA. This is because the ability to represent immediate loopbacks, or recursions at value level, is a coherent feature of lazy languages. Also as a side note, CCA by itself does not preclude modular compilation, even though its current implementation through Template Haskell requires full in-lining of arrow terms and hence is not modular.

Also relevant is the work by Rutten (2006) on high-order functional stream derivatives. We believe that arrows are a more general abstraction than functional stream derivatives, because the latter still exposes the structure of a stream. Moreover,

arrows give rise to a high-level language with richer algebraic properties than the 2-adic calculus considered by Rutten (2006).

9.3 Stream fusion

Stream fusion (Coutts *et al.*, 2007) can help fuse zips, left folds, and nested lists into efficient loops. But on its own, it does not optimize recursively and lazily defined streams effectively.

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

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

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

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

```
data Stream a = forall s . Stream (s → Step a s) s
data Step a s = Yield a s
fib_stream :: Stream Int
fib_stream = Stream next (0,1) where next (a,b) = Yield r (b,r) where r = a + b
f1 :: Int
f1 = nth 5 fib_stream   -- 13
```

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

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

We must stress that writing stepper functions is not always as easy as in trivial examples such as *fib* and *exp*. Most nontrivial stream programs that we are concerned with contain many recursive parts, and expressing them in terms of combinators in

https://doi.org/10.1017/S0956796811000153 Published online by Cambridge University Press

a nonrecursive way can get unwieldy. Moreover, this kind of coding style exposes a lot of operational details which are arguably unnecessary for representing the underlying algorithm. In contrast, arrow syntax relieves the burden of coding in combinator form and allows recursion via the **rec** keyword. It also completely hides the actual implementation of the underlying stream structure and is therefore more abstract.

10 Conclusion

Our key contribution is the discovery of a normal form for core Yampa, or CCA, programs: any CCA program can be transformed into a single loop with just one pure (and strongly normalizing) function and a set of initial states. This discovery has practical implications in implementing not just Yampa, but a broader class of synchronous dataflow languages and stream computations. Any CCA program can be reliably and predictably optimized into an efficient machine-friendly loop. The process can be fully automated, allowing programmers to program at an abstract level while getting performance competitive to programs written in low-level imperative languages.

Acknowledgments

We thank anonymous reviewers for pointers to relevant work. This research was supported in part by NSF grants CCF-0811665 and CNS-0720682, and by a grant from Microsoft Research.

References

Amagbegnon, P., Besnard, L., & Guernic, P. L. (1995) Implementation of the data-flow synchronous language SIGNAL. In *Conference on Programming Language Design and Implementation*. ACM, pp. 163–173.

Atkey, R. (2008) What is a categorical model of arrows? In *Proceedings of the Second Workshop on Mathematically Structured Functional Programming*. vol. 229 pp. 19–37.

Bjesse, P., Claessen, K., Sheeran, M., & Singh, S. (1998) Lava: Hardware design in Haskell. In *Proceedings of International Conference on Functional Programming (ICFP)*. ACM, pp. 174–184.

Burchett, K., Cooper, G. H. & Krishnamurthi, S. (2007) Lowering: A static optimization technique for transparent functional reactivity. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. ACM, pp. 71–80.

Caspi, P., Halbwachs, N., Pilaud, D. & Plaice, J. A. (1987) LUSTRE: A declarative language for programming synchronous systems. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages (POPL)*. ACM, pp. 178–188.

Caspi, P. & Pouzet, M. (1998) A co-iterative characterization of synchronous stream functions. In *Coalgebraic Methods in Computer Science (CMCS)*. Electronic Notes in Theoretical Computer Science. Elsevier. Extended version available as a VERIMAG Tech. Rep. no. 97–07 at `www.lri.fr/~pouzet`. pp. 1–21.

Cheng, E. & Hudak, P. (2009) *Audio Processing and Sound Synthesis in Haskell*. Tech. Rep. YALEU/DCS/RR-1405. Computer Science Department, Yale University, New Heaven, CT. Also see http://haskell.cs.yale.edu.

Cheong, M. H. (2005) *Functional Programming and 3D Games*. M.Phil. thesis, University of New South Wales, Sydney, Australia. Also see http://www.haskell.org/haskellwiki/Frag.

Colaço, J.-L., Girault, A., Hamon, G., & Pouzet, M. (2004) Towards a higher-order synchronous data-flow language. In *Proceedings of the 4th ACM International Conference on Embedded Software (EMSOFT)*. New York: ACM, pp. 230–239.

Courtney, A. (2004) *Modelling User Interfaces in a Functional Language*. PhD thesis, Department of Computer Science, Yale University, New Heaven, CT.

Courtney, A. & Elliott, C. (2001) Genuinely functional user interfaces. In *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*. ACM, pp. 41–69.

Courtney, A., Nilsson, H. & Peterson, J. (2003) The Yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN Haskell Workshop*. Uppsala, Sweden: ACM, pp. 7–18.

Coutts, D., Leshchinskiy, R. & Stewart, D. (2007) Stream fusion: From lists to streams to nothing at all. In *Proceedings of the International Conference on Functional Programming (ICFP)*. ACM. pp. 315–326.

Elliott, C. & Hudak, P. (1997) Functional reactive animation. In *Proceedings of the International Conference on Functional Programming (ICFP)*. ACM, pp. 263–273.

Giorgidze, G. & Nilsson, H. (2008) Switched-on Yampa. In *Proc*eedings, Hudak, P. & Warren, D. S. (eds), Lecture Notes in Computer Science, vol. 4902. San Francisco, CA: Springer, pp. 282–298.

Goertzel, G. (1958) An algorithm for the evaluation of finite trigonometric series. *Am. Math. Mon.* **65**(Jan.), 34–35.

Halbwachs, N., Raymond, P. & Ratel, C. (1991) Generating efficient code from data-flow programs. In *Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming*, Maluszyński, J. & Wirsing, M. (eds). Springer-Verlag, pp. 207–218.

Harel, D. (1980) On folk theorems. *Commun. ACM* **23**(July), 379–389.

Hasegawa, M. (1997) Recursion from cyclic sharing: Traced monoidal categories and models of cyclic lambda calculi. In *Proceedings of the Third International Conference on Typed Lambda Calculi and Applications (TLCA)*. London: Springer-Verlag, pp. 196–213.

Huang, L., Hudak, P. & Peterson, J. (2007) HPorter: Using arrows to compose parallel processes. In *Proceedings of Practical Aspects of Declarative Languages (PADL)*. Springer-Verlag, pp. 275–289.

Hudak, P. (1996) Building domain specific embedded languages. *ACM Computing Surveys*, **28A**, (electronic).

Hudak, P. (1998) Modular domain specific languages and tools. In *Proceedings of Fifth International Conference on Software Reuse*. IEEE Computer Society, pp. 134–142.

Hudak, P., Courtney, A., Nilsson, H. & Peterson, J. (2003) Arrows, robots, and functional reactive programming. In *Summer School on Advanced Functional Programming (AFP 2002), Oxford University*. Lecture Notes in Computer Science, vol. 2638. Springer-Verlag, pp. 159–187.

Hudak, P., Liu, H., Stern, M & Agarwal, A. 2008 (July). *Yampa Meets the Worm.* Tech. Rep. YALEU/DCS/RR-1408. Yale University, New Heaven, CT. Also see http://haskell.cs.yale.edu.

Hughes, J. (2000) Generalising monads to arrows. *Sci. Comput. Program.* **37**(1-3), 67–111.

Hughes, J. (2004) Programming with arrows. In *Advanced Functional Programming*, Vene, V. & Uustalu, T. (eds), Lecture Notes in Computer Science, vol. 3622. Springer, pp. 73–129.

Jansson, P. & Jeuring, J. (1999) Polytypic compact printing and parsing. In *European Symposium on Programming (ESOP)*, Swierstra, S. D. (ed), Lecture Notes in Computer Science, vol. 1576. Springer, pp. 273–287.

Lindley, S., Wadler, P. & Yallop, J. (2010) The arrow calculus (functional pearl). *J. Funct. Program.* **20**(1), 51–69.

Liu, H. & Hudak, P. (2007) Plugging a space leak with an arrow. *Electron. Notes Theor. Comput. Sci.* **193**, 29–45.

Liu, H. & Hudak, P. (2010) An ode to arrows. In *Proceedings of the 12th International Symposium on Practical Aspects of Declarative Languages (PADL), Madrid, Spain, January 18–19, 2010,* Carro, M. & Peña, R. (eds), Lecture Notes in Computer Science, vol. 5937. Springer, pp. 152–166.

Lublinerman, R., Szegedy, C. & Tripakis, S. (2009) Modular code generation from synchronous block diagrams: Modularity vs. code size. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '09. New York: ACM, pp. 78–89.

Lublinerman, R. & Tripakis, S. (2008) Modularity vs. reusability: Code generation from synchronous block diagrams. In *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE '08. New York: ACM, pp. 1504–1509.

McBride, C. & Paterson, R. (2008) Applicative programming with effects. *J. Funct. Program.* **18**(1), 1–13.

Mealy, G. H. (1955) A method for synthesizing sequential circuits. *Bell Syst. Tech. J.* **34**(5), 1045–1079.

Moggi, E. (1991) Notions of computation and monads. *Inf. Comput.* **93**(1), 55–92.

Nilsson, H. (2005) Dynamic optimization for functional reactive programming using generalized algebraic data types. In *Proceedings of International Conference on Functional Programming (ICFP)*. ACM, pp. 54–65.

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

Paterson, R. (2001) A new notation for arrows. In *Proceedings of International Conference on Functional Programming (ICFP)*. ACM, pp. 229–240.

Peterson, J., Hager, G. & Hudak, P. (1999a) A language for declarative robotic programming. In *International Conference on Robotics and Automation*, IEEE pp. 1144–1151.

Peterson, J., Hudak, P. & Elliott, C. (1999b) Lambda in motion: Controlling robots with Haskell. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages (PADL)*. ACM, pp. 91–105.

Peyton Jones, Simon, *et al.* (2003) The Haskell 98 language and libraries: The revised report. *J. Funct. Program.* **13**(1), 0–255.

Pouzet, M. & Raymond, P. (2009) Modular static scheduling of synchronous data-flow networks: an efficient symbolic representation. In *Proceedings of the seventh ACM International Conference on Embedded Software*. EMSOFT '09. New York: ACM, pp. 215–224.

Power, J. & Thielecke, H. (1999) Closed freyd- and kappa-categories. In *Proceedings of 26th International Colloquium on Automata, Languages and Programming (ICALP)*, Springer-Verlag pp. 625–634.

Raymond, P. (1988) *Compilation séparée de programmes lustre*. Master's thesis. IMAG (in French).

Rutten, J. J. M. M. (2006) Algebraic specification and coalgebraic synthesis of mealy automata. *Electron. Notes Theor. Comput. Sci.*, **160**, 305–319.

Sculthorpe, N. & Nilsson, H. (2008) Optimisation of dynamic, hybrid signal function networks. In *Proceedings of the 9th Symposium on Trends in Functional Programming (TFP)*. Intellect, pp. 97–112.

Sheard, T. & Peyton Jones, S. (2002) Template metaprogramming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop*, Chakravarty, M. M. T. (ed). ACM, pp. 1–16.

Stephens, R. (1997) A survey of stream processing. *Acta Inform.* **34**(7), 491–541.

Street, R. H., Joyal, A. & Verity, D. (1996) Traced monoidal categories. *Math. Proc. Camb. Phil. Soc.* **119**(3), 425–446.

Thies, W., Karczmarek, M. & Amarasinghe, S. P. (2002) StreamIt: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction (CC)*. London: Springer-Verlag, 179–196.

Uustalu, T. & Vene, V. (2005) The essence of dataflow programming. In *Central European Functional Programming School (CEFP)*, Horváth, Z. (ed), Lecture Notes in Computer Science, vol. 4164. Springer, pp. 135–167.

Wadge, W. W., & Ashcroft, E. A. (1985) *LUCID, the Dataflow Programming Language*. San Diego, CA: Academic Press Professional, Inc.

Appendix A Benchmark programs

```
sr = 44100 :: Int
dt = 1 / (fromIntegral sr)
exp :: ArrowInit a ⇒ a () Double
exp = proc () → do
  rec let e = 1 + i
      i ← integral ≺ e
  returnA ≺ e
integral :: ArrowInit a ⇒ a Double Double
integral = proc x → do
  rec let i' = i + x * dt
      i ← init 0 ≺ i'
  returnA ≺ i
sine :: ArrowInit a ⇒ Double → a () Double
sine freq = proc _ → do
  rec x ← init i ≺ r
      y ← init 0 ≺ x
      let r = c * x − y
  returnA ≺ r
  where
    omh = 2 * pi / (fromIntegral sr) * freq
    i = sin omh
    c = 2 * cos omh
oscSine :: ArrowInit a ⇒ Double → a Double Double
oscSine f0 = proc cv → do
  let f = f0 * (2 ** cv)
  phi ← integral ≺ 2 * pi * f
  returnA ≺ sin phi
testOsc :: ArrowInit a ⇒ (Double → a Double Double) → a () Double
testOsc f = constant 1 ⋙ f 440
```

https://doi.org/10.1017/S0956796811000153 Published online by Cambridge University Press

```
sciFi :: ArrowInit a ⇒ a () Double
sciFi = proc () → do
  und ← oscSine 3.0−≺ 0
  swp ← integral−≺ −0.25
  audio ← oscSine 440−≺ und * 0.2 + swp + 1
  returnA−≺ audio
robot :: ArrowInit a ⇒ a (Double, Double) Double
robot = proc inp → do
    let vr = snd inp
      vl = fst inp
      vz = vr + vl
    t ← integral−≺ vr − vl
    let t′ = t / 10
    x ← integral−≺ vz * cos t′
    y ← integral−≺ vz * sin t′
    returnA−≺ x / 2 + y / 2
testRobot :: ArrowInit a ⇒ a (Double, Double) Double → a () Double
testRobot bot = proc () → do
    u ← sine 2−≺ ()
    robot−≺ (u, 1 − u)
```

Appendix B Proof for the *sequencing* rule of *loopD*

The sequencing rule from Figure 8 is directly derivable from the following equation:

$$loopD\ i\ f \ggg loopD\ j\ g = loopD\ (i,j)\ (assoc'\ (juggle'\ (g \times id)\ .\ (f \times id)))$$

In order to show the above is true, we first prove three lemmas.

Lemma 10.1
Given a function definition $revjuggle\ (a,(b,c)) = (b,(a,c))$, we show that for all f:

$$second\ (second\ f) = arr\ revjuggle \ggg second\ (second\ f) \ggg arr\ revjuggle$$

Proof
We start from the left-hand side by equational reasoning.

$$\begin{aligned}
&lhs\\
&= second\ (second\ f)\\
&\quad \text{definition of } second\\
&= arr\ swap \ggg first\ (arr\ swap \ggg first\ f \ggg arr\ swap) \ggg arr\ swap\\
&\quad \text{functor of } first, \text{extension of } first\\
&= arr\ swap \ggg arr\ (swap \times id) \ggg first\ (first\ f) \ggg arr\ (swap \times id) \ggg arr\ swap\\
&\quad \text{identity of } arr, id = assoc^{-1}\ .\ assoc, \text{and composition of } arr\\
&= arr\ swap \ggg arr\ (swap \times id) \ggg first\ (first\ f) \ggg arr\ assoc \ggg arr\ assoc^{-1}\\
&\qquad \ggg arr\ (swap \times id) \ggg arr\ swap\\
&\quad \text{association of } first
\end{aligned}$$

https://doi.org/10.1017/S0956796811000153 Published online by Cambridge University Press

$= arr\ swap \ggg arr\ (swap \times id) \ggg arr\ assoc \ggg first\ f \ggg arr\ assoc^{-1}$
$\quad \ggg arr\ (swap \times id) \ggg arr\ swap$
composition of *arr*
$= arr\ (assoc\ .\ (swap \times id)\ .\ swap) \ggg first\ f \ggg arr\ (swap\ .\ swap \times id\ .\ assoc^{-1})$
unfold function definition and beta reduce
$= arr\ (\lambda(a,(b,c)) \to (c,(a,b))) \ggg first\ f \ggg arr\ (\lambda(c,(a,b)) \to (a,(b,c)))$

Then from the right-hand side:

rhs
$= arr\ revjuggle \ggg second\ (second\ f) \ggg arr\ revjuggle$
definition of *second*
$= arr\ revjuggle \ggg arr\ swap \ggg first\ (arr\ swap \ggg first\ f \ggg arr\ swap)$
$\quad \ggg arr\ swap \ggg arr\ revjuggle$
functor of *first*, extension of *first*
$= arr\ revjuggle \ggg arr\ swap \ggg arr\ (swap \times id) \ggg first\ (first\ f)$
$\quad \ggg arr\ (swap \times id) \ggg arr\ swap \ggg arr\ revjuggle$
identity of $arr, id = assoc^{-1}\ .\ assoc$, and composition of *arr*
$= arr\ revjuggle \ggg arr\ swap \ggg arr\ (swap \times id) \ggg first\ (first\ f)$
$\quad \ggg arr\ assoc \ggg arr\ assoc^{-1} \ggg arr\ (swap \times id) \ggg arr\ swap \ggg arr\ revjuggle$
association of *first*
$= arr\ revjuggle \ggg arr\ swap \ggg arr\ (swap \times id) \ggg arr\ assoc \ggg first\ f$
$\quad \ggg arr\ assoc^{-1} \ggg arr\ (swap \times id) \ggg arr\ swap \ggg arr\ revjuggle$
composition of *arr*
$= arr\ (assoc\ .\ swap \times id\ .\ swap\ .\ revjuggle) \ggg first\ f$
$\quad \ggg arr\ (revjuggle\ .\ swap\ .\ swap \times id\ .\ assoc^{-1})$
unfold function definition and beta reduce
$= arr\ (\lambda(a,(b,c)) \to (c,(a,b))) \ggg first\ f \ggg arr\ (\lambda(c,(a,b)) \to (a,(b,c)))$

Therefore, lhs = rhs. □

Lemma 10.2
We show that for all f and g:

$$second\ (first\ f) \ggg arr\ (assoc\ .\ swap) = arr\ (assoc\ .\ swap) \ggg first\ f$$

Proof
We start from the left-hand side by equational reasoning:

lhs
$= second\ (first\ f) \ggg arr\ (assoc\ .\ swap)$
definition of *second*
$= arr\ swap \ggg first\ (first\ f) \ggg arr\ swap \ggg arr\ (assoc\ .\ swap)$
identity of $arr, id = assoc^{-1}\ .\ assoc$, and composition of *arr*
$= arr\ swap \ggg first\ (first\ f) \ggg arr\ assoc \ggg arr\ assoc^{-1} \ggg arr\ swap$
$\quad \ggg arr\ (assoc\ .\ swap)$
association of *first*
$= arr\ swap \ggg arr\ assoc \ggg first\ f \ggg arr\ assoc^{-1} \ggg arr\ swap$

https://doi.org/10.1017/S0956796811000153 Published online by Cambridge University Press

$$
\begin{aligned}
&\ggg arr\ (assoc\ .\ swap)\\
&\quad \text{composition of } arr\\
&= arr\ (assoc\ .\ swap) \ggg first\ f \ggg arr\ (assoc\ .\ swap\ .\ swap\ .\ assoc^{-1})\\
&\quad \text{identity of } arr, id = assoc\ .\ swap\ .\ swap\ .\ assoc^{-1}\\
&= arr\ (assoc\ .\ swap) \ggg first\ f
\end{aligned}
$$

□

Lemma 10.3
We show that for all f:

$$first\ f = arr\ revjuggle \ggg second\ (first\ f) \ggg arr\ revjuggle$$

Proof
We start from the right-hand side by equational reasoning:

$$
\begin{aligned}
&rhs\\
&= arr\ revjuggle \ggg second\ (first\ f) \ggg arr\ revjuggle\\
&\quad \text{definition of } second\\
&= arr\ revjuggle \ggg arr\ swap \gg first\ (first\ f) \ggg arr\ swap \ggg arr\ revjuggle\\
&\quad \text{identity of } arr, id = assoc^{-1}\ .\ assoc, \text{and composition of } arr\\
&= arr\ revjuggle \ggg arr\ swap \gg first\ (first\ f) \ggg arr\ assoc \ggg arr\ assoc^{-1}\\
&\qquad \ggg arr\ swap \ggg arr\ revjuggle\\
&\quad \text{association of } first\\
&= arr\ revjuggle \ggg arr\ swap \gg arr\ assoc \ggg first\ f \ggg arr\ assoc^{-1}\\
&\qquad \ggg arr\ swap \ggg arr\ revjuggle\\
&\quad \text{identity of } arr, id = id \times swap\ .\ id \times swap, \text{and composition of } arr\\
&= arr\ revjuggle \ggg arr\ swap \gg arr\ assoc \ggg arr\ (id \times swap) \ggg arr\ (id \times swap)\\
&\qquad \ggg first\ f \ggg arr\ assoc^{-1}\\
&\quad \text{exchange of } first\\
&= arr\ revjuggle \ggg arr\ swap \gg arr\ assoc \ggg arr\ (id \times swap) \ggg first\ f\\
&\qquad \ggg arr\ (id \times swap) \ggg arr\ assoc^{-1}\\
&\quad \text{composition of } arr\\
&= arr\ (id \times swap\ .\ assoc\ .\ swap\ .\ revjuggle) \ggg first\ f\\
&\qquad \ggg arr\ (revjuggle\ .\ swap\ .\ assoc^{-1}\ .\ id \times swap)\\
&\quad \text{identity of } arr, id = id \times swap\ .\ assoc\ .\ swap\ .\ revjuggle\\
&\quad \text{and } id = revjuggle\ .\ swap\ .\ assoc^{-1}\ .\ id \times swap\\
&= first\ f
\end{aligned}
$$

□

We then show that

$$loopD\ i\ f \ggg loopD\ j\ g = loopD\ (i,j)\ (assoc'\ (juggle'\ (g \times id)\ .\ (f \times id)))$$

holds by equational reasoning, starting from the left-hand side:

$$
\begin{aligned}
&loopD\ i\ f \ggg loopD\ j\ g\\
&\quad \text{definition of } loopD\\
&= loop\ (arr\ f \ggg second\ (init\ i)) \ggg loop\ (arr\ g \ggg second\ (init\ j))\\
&\quad \text{left tightening of } loop\\
&= loop\ (first\ (loop\ (arr\ f \ggg second\ (init\ i))) \ggg (arr\ g \ggg second\ (init\ j)))
\end{aligned}
$$

https://doi.org/10.1017/S0956796811000153 Published online by Cambridge University Press

definition of *second*

$= loop\ (arr\ swap \ggg second\ (loop\ (arr\ f \ggg second\ (init\ i))) \ggg arr\ swap$
$\ggg (arr\ g \ggg second\ (init\ j)))$

superposing of *loop*

$= loop\ (arr\ swap \ggg loop\ (arr\ assoc \ggg second\ (arr\ f \ggg second\ (init\ i))$
$\ggg arr\ assoc^{-1}) \ggg arr\ swap \ggg (arr\ g \ggg second\ (init\ j)))$

left tightening of *loop*

$= loop\ (loop\ (first\ (arr\ swap) \ggg arr\ assoc \ggg second\ (arr\ f \ggg second\ (init\ i))$
$\ggg arr\ assoc^{-1}) \ggg arr\ swap \ggg (arr\ g \ggg second\ (init\ j)))$

extension of *first*

$= loop\ (loop\ (arr\ (swap \times id) \ggg arr\ assoc \ggg second\ (arr\ f \ggg second\ (init\ i))$
$\ggg arr\ assoc^{-1}) \ggg arr\ swap \ggg (arr\ g \ggg second\ (init\ j)))$

associativity of $\ggg$

$= loop\ (loop\ (arr\ (swap \times id) \ggg arr\ assoc \ggg second\ (arr\ f \ggg second\ (init\ i))$
$\ggg arr\ assoc^{-1}) \ggg (arr\ swap \ggg arr\ g \ggg second\ (init\ j)))$

right tightening of *loop*

$= loop\ (loop\ (arr\ (swap \times id) \ggg arr\ assoc \ggg second\ (arr\ f \ggg second\ (init\ i))$
$\ggg arr\ assoc^{-1} \ggg first\ (arr\ swap \ggg arr\ g \ggg second\ (init\ j))))$

vanishing of *loop*

$= loop\ (arr\ assoc^{-1} \ggg arr\ (swap \times id) \ggg arr\ assoc \ggg second\ (arr\ f$
$\ggg second\ (init\ i)) \ggg arr\ assoc^{-1} \ggg first\ (arr\ swap \ggg arr\ g \ggg second\ (init\ j))$
$\ggg arr\ assoc)$

identity of $arr, id = id \times swap\ .\ id \times swap$, composition of *arr*

$= loop\ (arr\ (id \times swap) \ggg arr\ (id \times swap) \ggg arr\ assoc^{-1} \ggg arr\ (swap \times id)$
$\ggg arr\ assoc \ggg second\ (arr\ f \ggg second\ (init\ i)) \ggg arr\ assoc^{-1}$
$\ggg first\ (arr\ swap \ggg arr\ g \ggg second\ (init\ j)) \ggg arr\ assoc)$

sliding of *loop*

$= loop\ (arr\ (id \times swap) \ggg arr\ assoc^{-1} \ggg arr\ (swap \times id) \ggg arr\ assoc$
$\ggg second\ (arr\ f \ggg second\ (init\ i)) \ggg arr\ assoc^{-1} \ggg first\ (arr\ swap$
$\ggg arr\ g \ggg second\ (init\ j)) \ggg arr\ assoc \ggg arr\ (id \times swap))$

composition of $arr, swap\ .\ assoc^{-1} = assoc\ .\ (swap \times id)\ .\ assoc^{-1}\ .\ (id \times swap)$

$= loop\ (arr\ (swap\ .\ assoc^{-1}) \ggg second\ (arr\ f \ggg second\ (init\ i)) \ggg arr\ assoc^{-1}$
$\ggg first\ (arr\ swap \ggg arr\ g \ggg second\ (init\ j)) \ggg arr\ assoc \ggg arr\ (id \times swap))$

definition of *second*

$= loop\ (arr\ (swap\ .\ assoc^{-1}) \ggg second\ (arr\ f \ggg arr\ swap \ggg first\ (init\ i)$
$\ggg arr\ swap) \ggg arr\ assoc^{-1} \ggg first\ (arr\ swap \ggg arr\ g \ggg second\ (init\ j))$
$\ggg arr\ assoc \ggg arr\ (id \times swap))$

definition of *first*

$= loop\ (arr\ (swap\ .\ assoc^{-1}) \ggg second\ (arr\ f \ggg arr\ swap \ggg first\ (init\ i)$
$\ggg arr\ swap) \ggg arr\ assoc^{-1} \ggg arr\ swap \ggg second\ (arr\ swap \ggg arr\ g$
$\ggg second\ (init\ j)) \ggg arr\ swap \ggg arr\ assoc \ggg arr\ (id \times swap))$

functor and extension of *second*

$= loop\ (arr\ (swap\ .\ assoc^{-1}) \ggg second\ (arr\ (swap\ .\ f)) \ggg second\ (first\ (init\ i))$
$\ggg arr\ (id \times swap) \ggg arr\ assoc^{-1} \ggg arr\ swap \ggg arr\ (id \times swap)$

https://doi.org/10.1017/S0956796811000153 Published online by Cambridge University Press

$$\ggg second\ (arr\ g) \ggg second\ (second\ (init\ j)) \ggg arr\ swap \ggg arr\ assoc \ggg arr\ (id \times swap))$$

Lemma 10.1

$$= loop\ (arr\ (swap\ .\ assoc^{-1}) \ggg second\ (arr\ (swap\ .\ f)) \ggg second\ (first\ (init\ i)) \ggg arr\ (id \times swap) \ggg arr\ assoc^{-1} \ggg arr\ swap \ggg arr\ (id \times swap) \ggg second\ (arr\ g) \ggg arr\ revjuggle \ggg second\ (second\ (init\ j)) \ggg arr\ revjuggle \ggg arr\ swap \ggg arr\ assoc \ggg arr\ (id \times swap))$$

composition of $arr, id = (id \times swap)\ .\ assoc\ .\ swap\ .\ revjuggle$, identity of arr

$$= loop\ (arr\ (swap\ .\ assoc^{-1}) \ggg second\ (arr\ (swap\ .\ f)) \ggg second\ (first\ (init\ i)) \ggg arr\ (id \times swap) \ggg arr\ assoc^{-1} \ggg arr\ swap \ggg arr\ (id \times swap) \ggg second\ (arr\ g) \ggg arr\ revjuggle \ggg second\ (second\ (init\ j)))$$

composition of $arr, assoc\ .\ swap = (id \times swap)\ .\ swap\ .\ assoc^{-1}\ .\ (id \times swap)$

$$= loop\ (arr\ (swap\ .\ assoc^{-1}) \ggg second\ (arr\ (swap\ .\ f)) \ggg second\ (first\ (init\ i)) \ggg arr\ (assoc\ .\ swap) \ggg second\ (arr\ g) \ggg arr\ revjuggle \ggg second\ (second\ (init\ j)))$$

Lemma 10.2

$$= loop\ (arr\ (swap\ .\ assoc^{-1}) \ggg second\ (arr\ (swap\ .\ f)) \ggg arr\ (assoc\ .\ swap) \ggg first\ (init\ i) \ggg second\ (arr\ g) \ggg arr\ revjuggle \ggg second\ (second\ (init\ j)))$$

commutativity

$$= loop\ (arr\ (swap\ .\ assoc^{-1}) \ggg second\ (arr\ (swap\ .\ f)) \ggg arr\ (assoc\ .\ swap) \ggg second\ (arr\ g) \ggg first\ (init\ i) \ggg arr\ revjuggle \ggg second\ (second\ (init\ j)))$$

Lemma 10.3

$$= loop\ (arr\ (swap\ .\ assoc^{-1}) \ggg second\ (arr\ (swap\ .\ f)) \ggg arr\ (assoc\ .\ swap) \ggg second\ (arr\ g) \ggg arr\ revjuggle \ggg second\ (first\ (init\ i)) \ggg arr\ revjuggle \ggg arr\ revjuggle \ggg second\ (second\ (init\ j)))$$

composition of $arr, id = revjuggle\ .\ revjuggle$, identity of arr

$$= loop\ (arr\ (swap\ .\ assoc^{-1}) \ggg second\ (arr\ (swap\ .\ f)) \ggg arr\ (assoc\ .\ swap) \ggg second\ (arr\ g) \ggg arr\ revjuggle \ggg second\ (first\ (init\ i)) \ggg second\ (second\ (init\ j)))$$

extension of *second*, composition of *arr*

$$= loop\ (arr\ (revjuggle\ .\ (id \times g)\ .\ assoc\ .\ swap\ .\ id \times (swap\ .\ f)\ .\ swap\ .\ assoc^{-1}) \ggg second\ (first\ (init\ i)) \ggg second\ (second\ (init\ j)))$$

$$assoc'\ (juggle'\ (g \times id)\ .\ (f \times id)) = revjuggle\ .\ (id \times g)\ .\ assoc\ .\ swap\ .\ id \times (swap\ .\ f)\ .\ swap\ .\ assoc^{-1}$$

$$= loop\ (arr\ (assoc'\ (juggle'\ (g \times id)\ .\ (f \times id))) \ggg second\ (first\ (init\ i)) \ggg second\ (second\ (init\ j)))$$

functor of *second*

$$= loop\ (arr\ (assoc'\ (juggle'\ (g \times id)\ .\ (f \times id))) \ggg second\ (first\ (init\ i) \ggg second\ (init\ j)))$$

product of *init*

$$= loop\ (arr\ (assoc'\ (juggle'\ (g \times id)\ .\ (f \times id))) \ggg second\ (init\ (i,j)))$$

□

https://doi.org/10.1017/S0956796811000153 Published online by Cambridge University Press