This Is Driving Me Loopy: Efficient Loops in Arrowized Functional Reactive Programs

Finnbar Keating
f.keating@warwick.ac.uk
University of Warwick
Coventry, United Kingdom

Michael B. Gale
mbg@github.com
GitHub
Oxford, United Kingdom

Abstract
Arrowized Functional Reactive Programming (AFRP) is one approach to writing reactive programs declaratively, based on the arrows abstraction in Haskell. While AFRP elegantly expresses the relationships between inputs and outputs of a reactive system, naïve implementations suffer from poor performance. In particular, the \texttt{loop} combinator depends on lazy semantics: this inflicts the overheads of lazy evaluation and simultaneously prevents existing optimisation techniques from being applied to it.

We present a novel program transformation which utilises the Arrow and ArrowLoop laws to transform typical uses of \texttt{loop} into restricted forms that have an execution order that is known at compile-time and therefore can be executed strictly. We evaluate the performance gained from our transformations and prove that the transformations are correct.

CCS Concepts: • Software and its engineering → Functional languages; Data flow languages.

Keywords: Functional Reactive Programming, reactive programming, stream programming, arrows, program transformation

ACM Reference Format:

1 Introduction
Arrowised Functional Reactive Programming (AFRP) [8] is a paradigm for writing reactive programs [24], which was popularised by the Haskell library Yampa. In AFRP, reactive programs are built using signal functions: functions which produce streams of outputs from streams of inputs. Execution of a program is broken up into time steps, in each of which signal functions get an input and produce a corresponding output. This means that the program effectively reacts to its inputs over time by producing outputs at the same rate. Signal functions can be combined with the arrow combinators [9] to form larger programs.

As an example of AFRP in action, consider a reactive summing program which, at every time step, retrieves an input and adds it to a running total, which is also the output. This is implemented in Fig. 1 as a Yampa program and visualised as a box-and-wire diagram.

The overall program, which is itself a signal function, is built up from smaller signal functions. We have \texttt{arr sum}, which sums two inputs and returns the sum as both outputs; and \texttt{pre 0}, which returns 0 at the first time step and then the previous input at future time steps. Their inputs and outputs are routed as shown in the diagram using the \texttt{∗∗∗} and \texttt{≫} combinators. We finally enclose all this in \texttt{loop}, which connects the second output of its internal signal function to its second input.

\texttt{loop} seems to introduce a dependency cycle here, in which the second input of \texttt{arr sum} needs the second output of \texttt{arr sum} to be computed. Fortunately, \texttt{pre} can generate its output at a given time step only using its previous input, meaning that we get the output of \texttt{pre} before needing to compute its input. Therefore the above program works by retrieving the previous output of the program “stored” in \texttt{pre}, applying \texttt{arr sum} to that and the current input of the program to compute the new cumulative total, returning that as the output of the program and “storing” it in the \texttt{pre} for use at the next time step.

The above example shows how stateful programs are written in AFRP: \texttt{loop} is used in tandem with \texttt{pre} in order to use outputs from previous time steps as inputs at a subsequent time step. However, within this pattern lies a performance issue that has not yet been addressed by existing literature:
in order to implement \textit{pre} as shown, \textit{loop}'s definition depends on lazy semantics. In pseudo-Haskell, we can define a simplified semantics of \textit{loop} as follows, where \texttt{evalSF} takes a term of the form \texttt{loop} \texttt{f}, \texttt{f} is some signal function, and \texttt{a} is an input at the current time step. The definition is recursive and we can see that the second input \texttt{c} depends on the result of the recursive call:

\begin{verbatim}
  evalSF :: SF a b -> a -> b
  evalSF (loop f) a =
    let (b, c) = evalSF f (a, c) in b
\end{verbatim}

When evaluating \textit{loop} \texttt{f}, the order in which signal functions in \texttt{f} are evaluated to determine \texttt{c} is not always the same or obvious as the signal functions are not always run from left to right: in the example in Fig. 1 we have to first run \texttt{pre} \texttt{0} to get its output, which corresponds to the \texttt{c} that is the second input to \texttt{arr sum}. The execution order is determined at runtime by lazily evaluating \texttt{f}, which means that evaluating \textit{loop} \texttt{f} suffers from the overheads needed for lazy evaluation.

This lazy evaluation does not happen just once at the start of the program, however. When Yampa evaluates a signal function for a single input, it also returns a possibly different signal function to run at the next time step. This is necessary for signal functions like \texttt{pre} \texttt{v} which need access to previous state: when we run \texttt{pre} \texttt{v} with input \texttt{a}, we get the output \texttt{v} and a new signal function \texttt{pre} \texttt{a}, which embeds the new state within the next signal function [17]. This approach to reactive programs that are essentially rewriting themselves at runtime might require us to re-evaluate the order of operations in \texttt{f} that needs to be performed at each time step.

Another issue with dependency resolution through lazy evaluation is that some well-typed \textit{loop} \texttt{f} contain dependency cycles so cannot be run. Consider the program in Fig. 1 but with the \texttt{pre} term omitted: this contains a dependency cycle as \texttt{arr sum} needs its inputs to compute its outputs, but the second output of \texttt{arr sum} depends on its own second input. The presence of a dependency cycle is only noticed when trying to evaluate the \textit{loop}, causing a runtime error.

We address the problems caused by evaluating \textit{loop} lazily by introducing a program transformation which transforms \texttt{loop} \texttt{f} with no dependency cycles into alternative forms with known execution order, which can be executed strictly. More concretely, our contributions are as follows:

\begin{itemize}
  \item We provide a program transformation for a subset of Yampa (Section 3) which transforms \texttt{loop} \texttt{f} without dependency cycles to use variants of \texttt{loop} with known execution orders. We accomplish this by applying the arrow laws [9] as well as novel rules to rewrite \texttt{loop} \texttt{f} so that uses of \texttt{pre} happen before everything else.
  \item We prove that our transformation works on programs without dependency cycles in them, and that this transformation does not affect program meaning (Section 4).
  \item We present a Haskell implementation, \texttt{Severn}, of this transformation (Section 5), which runs the resulting program strictly. We compare the performance of programs written in Severn to their equivalent Yampa programs (Section 6).
\end{itemize}

\section{Yampa and Arrow Laws}

For the benefit of readers unfamiliar with arrows and Yampa, we briefly introduce the arrow constructors and laws that we use throughout this paper.

Our work builds on Yampa, an AFRP implementation built around \textit{signal functions} (SFs). We consider a minimal set of arrow operators\footnote{https://hackage.haskell.org/package/base/docs/Control-Arrow.html} which are enough to define many useful Yampa programs: this minimal set is presented in Fig. 2. We also make use of \texttt{first} \texttt{f} and \texttt{second} \texttt{g} throughout as synonyms for \texttt{f \_\_\_ id} and \texttt{id \_\_\_ f} respectively.

We briefly describe each operator in turn: \texttt{arr} \texttt{f} allows pure functions to be turned into SFs where \texttt{f} transforms an input into an output. SFs can then be composed sequentially with \texttt{\_\_\_\_} and in parallel with \texttt{\_\_\_\_}. These operators give us the ability to run SFs consisting of pure functions and compose them into larger programs.

\texttt{pre} \texttt{v} introduces the effect of state by mirroring its input stream as output, delayed by one timestep. For example, the inputs 1, 2, 3 passed to \texttt{pre} \texttt{v} give us the outputs 1, 2, 3.

Finally, \textit{loop} \texttt{f} provides a way to introduce feedback into our arrow programs by directly connecting the second output of \texttt{f} to its own second input. This is where Yampa requires lazy evaluation, as we cannot run \texttt{f} strictly without its second output. In this work, we focus on its interactions with the \texttt{pre} operator: since \texttt{pre} can generate an output at a given timestep without its corresponding input, it can be used to generate the second output of \texttt{f} without needing the second input. We saw this in Fig. 1, where the second input of the \texttt{loop} is the second output of \texttt{loop} from the previous time step due to \texttt{pre}.

These operators are enough to define common Yampa programs. We discuss additional operators, such as \texttt{switch}, in Section 7.3.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure2.png}
\caption{The minimal set of arrow operators used in this paper, presented as box-and-wire diagrams.}
\end{figure}

\footnote{https://hackage.haskell.org/package/base/docs/Control-Arrow.html}
\footnote{We use \texttt{pre} following Yampa’s terminology, but Liu et. al [15] introduce it as the \texttt{init} operator, and it has also appeared elsewhere as \texttt{delay} or \texttt{iPre}.}
Since SFs are instances of Haskell’s Arrow and ArrowLoop type classes, SFs must obey their laws. These laws define required equivalences between programs, which we use throughout our work to prove that each step of our program transformation preserves program meaning. Throughout this paper we introduce necessary laws as they are needed, beginning with two keys below. Interested readers can consult Hughes [9, Section 7] for the full set.

\[
\text{first } f \gg\gg \text{first } g = \text{first } (f \gg\gg g) \quad (\text{arrow functor})
\]

\[
(f \gg\gg g) \gg h = f \gg\gg (g \gg h) \quad (\text{associativity})
\]

### 2.1 Commutative Causal Arrows (CCAs)

Liu et al define CCAs, which extend arrows with two additional laws that hold for Yampa [14, 15]:

\[
\text{first } f \gg\gg \text{second } g = \text{second } g \gg\gg \text{first } f \quad (\text{commutativity})
\]

\[
\text{pre } i \equiv \text{pre } j = \text{pre } (i, j) \quad (\text{product rule})
\]

With these additional laws, whole AFRP programs can be transformed into one of two forms: a single arrow, or a loop of the form \(\text{LoopD } f i = \text{loop } (f \gg\gg \text{second } (\text{pre } i))\) [30]. This transformation is performed using the ArrowLoop laws, which merge composed and nested loops together into a single loop using routing functions.

At first glance, this seems to solve the problem we are addressing in this paper: \(\text{LoopD } f i\) can be executed strictly by first executing \(\text{pre } i\) and then executing \(f\). However, the routing functions used by the transformation still rely on lazy evaluation, and thus the \(\text{LoopD}\) created by the CCA transformation cannot be executed strictly.

### 3 Transforming loop Into Strict Variants

We claim that the lazy semantics required to evaluate \(\text{loop } f\) is a cause of performance issues due to the involved overheads. Our goal is therefore to determine the execution order of \(f\) within \(\text{loop } f\) at compile-time.

We achieve this by finding decoupled [27] parts of \(f\): those which can produce outputs at time step \(t\) without any of their inputs at \(t\), like \(\text{pre}\). We define restricted forms of \(\text{loop}\) where those decoupled parts are separated out from the rest of \(f\), with the aim of running those parts first. An example of this, and the main restricted form of \(\text{loop}\) we consider, is Yallop and Liu’s \(\text{LoopD } f’ i\) construct [30]. We define this as follows alongside an interpreter \(\text{runLoopD}\) which maps \(\text{LoopD } f’ i\) to a corresponding signal function.

```haskell
data LoopD a b = LoopD (SF (a,c) (b,c)) c
runLoopD :: LoopD a b -> SF a b
runLoopD (LoopD f i) = loop (f >>> second (pre i))
```

A \(\text{loop } f\) can be expressed as \(\text{LoopD}\) only if \(f\) contains a \(\text{pre}\) just before its second output. Since \(\text{pre}\) can produce an output at a given time step without the input at that time step, we know the execution order of \(\text{LoopD } f’ i\): evaluate the final \(\text{pre } i\) to produce the second output, use it as second input, and run the rest of \(f’\) with both inputs. No lazy evaluation is required.

The question is then how to transform arbitrary \(\text{loop } f\) into equivalents that can be expressed as \(\text{LoopD } f’ i\). Informally, given a \(\text{loop } f\), our aim is to move a single \(\text{pre } i\) within \(f\) to appear just before \(f’\)’s second output while preserving the semantics of that loop.

In the rest of this section we present the necessary transformation for \(\text{LoopD}\) and other restricted forms of \(\text{loop}\) with known execution orders. We do this in four parts, as follows:

1. In Section 3.1 we apply ArrowLoop’s sliding law to transform some \(\text{loop } f\) into \(\text{LoopD } f’ i\) by moving \(\text{pre } i\) within \(f\) to be just before the second output. We also discuss a variety of transformations that may need to be applied in order to allow sliding, and introduce CCA composition form to make sliding easier to apply.

2. Sometimes there are multiple looped values in a \(\text{loop}\): a transformed \(\text{loop } f\) will be of the form \(\text{LoopD } f' (i, j)\). For this we slide a single \(\text{pre } (i, j)\) to be before the second output of \(f\). However, \(\text{pre } (i, j)\) can be expressed in a few different ways, such as \(\text{pre } i \equiv \text{pre } j\). To make sure that we are able to work with these equivalent statements of \(\text{pre } (i, j)\), we use CCA’s product rule and a new split rule which finds nearby \(\text{pre}\) to combine them into a single \(\text{pre } (i, j)\). (Section 3.2)

3. There are some loops where the \(\text{pre}\) is “trapped” between two non-\(\text{pre}\) arrows that we cannot slide and which therefore cannot be transformed by the above two steps. Fortunately, such loops also have a trivial execution order, for which we define another restricted form of \(\text{loop}\) called \(\text{LoopM}\) in Section 3.3.

4. We then look at the case where multiple \(\text{loops}\) are present in a program, e.g. \(\text{loop } (\text{loop } f)\), in Section 3.4. The inner \(\text{loop}\) is transformed using the above transformations, and then \(\text{LoopD}\) and \(\text{LoopM}\) are modified to allow nesting.

5. Finally, we combine these transformations into an algorithm in Section 3.5.

We justify that these steps cover all possible cases of a \(\text{loop}\) with no dependency cycles in Section 4.

### 3.1 Sliding

We start by looking at how to move a single \(\text{pre}\) to the rightmost position of a \(\text{loop}\) body. Examples of loops which can
We discuss the consequences of this decision in AFRP systems where signal functions can be effectful in Section 7.1.

We refer to transforming a program of the form on the left to the form on the right as **right sliding**, since we move the body of the loop to the right, causing \( k \) to fall off the end and reappear on the left side. We call the reverse direction **left sliding**. Sliding gives us the rules needed to justify the transformations in Fig. 3 and Fig. 4. The first example is solved through right sliding, moving \( h \) from the right of the loop to the left. In the second example we can left slide twice, moving \( g \) and then \( \text{pre} v \) from the left of the loop to the right.

### 3.1.1 Distributivity of Composition

This presentation of sliding may not be applicable if programs are written in subtly different, but equivalent ways. An equivalent way to write Fig. 3 is \( \text{loop} ( f \circ ( g \circ \text{pre} v \circ h) ) \). However, applying right sliding here moves all of \( g \circ \text{pre} v \circ h \) over to the left side, preventing us from getting \( \text{pre} v \) into the desired position. We solve this by noting that \( \circ \) distributes over \( \circ \circ \) for CCAs, proved as follows:

\[
(f \circ h) \circ (g \circ i) = \begin{cases} \text{first } f \circ \text{second } g = f \circ g & \text{if } \text{first } f \circ \text{second } g \end{cases}
\]

\[
\text{first } f \circ \text{second } g \circ \text{second } i = \begin{cases} \text{by Arrow Functor law} & \text{if } \text{we ignore brackets as } \circ \text{ is associative} \\
\text{first } f \circ \text{first } h \circ \text{second } g \circ \text{second } i = \begin{cases} \text{by } \text{CCA's commutativity law} & \text{if } \text{first } f \circ \text{second } g \circ \text{first } h \circ \text{second } i = \begin{cases} \text{first } f \circ \text{second } g = f \circ g & \text{if } f \circ g \end{cases}
\]

With this distributive law, we can rewrite \( \text{id} = (g \circ \text{pre} v \circ h) \) to \( (\text{id} \circ g) \circ (\text{id} \circ \text{pre} v) \circ (\text{id} \circ h) \). This is the same as the original definition of Fig. 3, allowing us to apply right sliding to get the \( \text{pre} v \) into the correct position.

#### 3.1.2 Sliding next to non-\( \text{id} \)

Another obstacle that can arise is when we have a term in parallel with the one we are trying to slide, as in the first diagram in Fig. 5. We are unable to apply left sliding here since it requires \( \text{id} \circ k \) at the start of the loop, but we have \( y \circ g \) instead of \( \text{id} \circ g \). We require a more general pair of program equivalences:

\[
\text{loop} ( f \circ (g \circ k)) = \begin{cases} \text{right sliding} & \text{loop} ((\text{id} \circ k) \circ f \circ (g \circ \text{id})) \\
\text{left sliding} & \text{loop} ((g \circ \text{id}) \circ f \circ (\text{id} \circ k))
\]

We prove the first of these below. The second is proved symmetrically.

\[
\text{loop} ( f \circ (g \circ k)) = \begin{cases} \text{identity of } \circ \circ & \text{loop} (f \circ ((g \circ \text{id}) \circ (\text{id} \circ k))) \\
\text{distributive law} & \text{loop} (f \circ ((g \circ \text{id}) \circ (\text{id} \circ k)))
\]\n
With these, we can apply our new left sliding rule to reach \( \text{loop} ((y \circ \text{id}) \circ (x \circ \text{pre} v) \circ f \circ (\text{id} \circ g)) \) as shown in the second diagram in Fig. 5, which makes progress towards getting \( \text{pre} v \) into the expected position. Unfortunately, we are now stuck – if we keep applying left sliding, all we do is keep sliding \( \text{id} \), which does not help us move the \( \text{pre} v \).

#### 3.1.3 Pushing non-\( \text{id} \) terms through \( \text{id} \)

To avoid the problem of having \( \text{id} \) block non-\( \text{id} \) terms which we want to slide, we need rules to remove the offending \( \text{id} \). We note that since \( \text{id} \) terms do not change program meaning, we can move them around and remove them as is needed. We therefore define some new rules to “push” a non-\( \text{id} \) term to take the place of an \( \text{id} \), thus allowing it to be used by other rules. Our aim in this section is to take programs such as \( (y \circ \text{id}) \circ (x \circ \text{pre} v) \), shown in the second diagram.
We can now use this to finish transforming Fig. 5:

\[
\text{loop } ((y \gg g) \gg (x \gg v) \gg f) =
\begin{align*}
&\{ \text{left sliding} \} \\
&\text{loop } ((y \gg id) \gg (x \gg v) \gg f \gg (id \gg g)) =
\begin{align*}
&\{ \text{left push} \} \\
&\text{loop } ((y \gg v) \gg (x \gg id) \gg f \gg (id \gg g)) =
\begin{align*}
&\{ \text{left sliding} \} \\
&\text{loop } ((y \gg v) \gg (x \gg id) \gg f) \\
&\gg (id \gg g) \gg (id \gg v)
\end{align*}
\end{align*}
\]

With this set of new rules, we are now able to transform loops which previously could not have left sliding applied to them. We also utilise equivalent right fill and right push laws to move non-id terms to the right for transforming loops to have right sliding applied to them. We omit these definitions as they are symmetric to those for left fill and left push.

3.1.4 CCA Composition Form. The issue of needing our program to be of a certain form in order to apply a rule is not unique to sliding. We also define CCA composition form, which forces loops to have \( \gg \) at the top level only, in order to restrict the shape that a loop can take and thus make it easier to apply our rules.

We require that \( pre \) cannot contain a tuple value. This is because when we apply rules such as sliding, we need to have the \( \gg \) to know that we can split the term in two: for example, if we had \( \text{loop } (pre (i, j) \gg f) \), we could not apply left sliding. Any \( pre (i, j) \) can instead be written as \( pre i \gg pre j \) by CCA’s product rule.

We now formally state the definition of CCA composition form.

Definition 3.1. An AFRP program is in CCA composition form if it can be parsed by the following grammar, where \( L \) is the start symbol, \( F \) is any pure function, and \( V \) is any non-tuple value.

\[
\begin{align*}
L & \coloneqq \text{loop } C \\
C & \coloneqq C \gg C \quad \text{Composition} \\
\mathcal{P} & \coloneqq \mathcal{P} \gg \mathcal{P} \quad \text{Parallel composition} \\
| & \text{arr } F \quad \text{Lifted pure function } F \\
| & \text{pre } V \quad \text{Pre with non-tuple argument } V \\
& \text{id} \quad \text{Identity} \\
& L \quad \text{Internal loop}
\end{align*}
\]
We have shown that using a combination of our new gener-
ated sliding rule, distributive law and push, we can move
a single pre within loop f to be before the second of output
of f. However, we sometimes work with multiple pre rather
than a single one, e.g. in loop (g \gg second (pre i \gg pre j)).
This loop still has a clear execution order: run the two pre to
generate the second outputs of the loop body, which means
we get the second inputs and can run g. We cannot currently
transform it into LoopD however, since LoopD relies on there
being a single pre.

To represent this as a LoopD, we need to merge the two occurrences of pre together into single use of pre using the
previously discussed CCA product rule: pre i \gg pre j =
pre (i, j). This means that whenever we encounter two uses of
pre in parallel, we can merge them and treat them as one.
With this, we can transform our example to loop (g \gg
second (pre (i, j))), which is equivalent to LoopD g (i, j).

3.2 Combining Smaller pre into Larger Ones

We have shown that using a combination of our new gener-
alised sliding rule, distributive law and push, we can move
a single pre within loop f to be before the second of output
of f. However, we sometimes work with multiple pre rather
than a single one, e.g. in loop (g \gg second (pre i \gg pre j)).
This loop still has a clear execution order: run the two pre to
generate the second outputs of the loop body, which means
we get the second inputs and can run g. We cannot currently
transform it into LoopD however, since LoopD relies on there
being a single pre.

To represent this as a LoopD, we need to merge the two occurrences of pre together into single use of pre using the
previously discussed CCA product rule: pre i \gg pre j =
pre (i, j). This means that whenever we encounter two uses of
pre in parallel, we can merge them and treat them as one.
With this, we can transform our example to loop (g \gg
second (pre (i, j))), which is equivalent to LoopD g (i, j).

3.2.1 The split Rule. The CCA product rule lets us combine pre which are in parallel, but the uses of pre we need
to merge may not always be in parallel. Figure 7 shows an
example with two uses of pre which cannot be solely solved
by the product rule and sliding: the two halves of the second
output each have a pre on them, but those uses of pre are not
parallel to each other and the product rule therefore cannot
be applied.

We therefore need a way of rearranging expressions such as
f \gg (g \gg pre j) \gg (pre i \gg h) to correctly group uses
of pre together and merge them with the product rule. To do
this, we define split, which attempts to split an input f into
(f_l, f_d, f_r) where f = f_l \gg f_d \gg f_r and f_d is a decoupled
term containing no \gg. We define this operation through a
collection of rules, shown in Fig. 8.

SPLIT-PRE dictates that if we have a pre v at the end of
the composition, then we already have a trivial split with
f_d = pre v. SPLIT-\gg-R specifies that if we have two parallel
paths given by some g \gg h = (g_1 \gg h_1) \gg \cdots \gg (g_n \gg h_n)
and we are able to split the two paths g and h, then we can
split the two paths in parallel by aligning the g_d and h_d we
get from the subordinate calls to split.

In any other case, we have not found a pre nor a \gg r
where we can find a pre in each of l and r and thus combine
them with CCA’s product rule. SPLIT-NONPRE dictates that
in this case, we can skip over this term as it will not lead to
us finding the required pre. This covers arr f and id.

We present these rules in use with a derivation that cor-
correctly splits our earlier example of f \gg (g \gg pre j) \gg
(pre i \gg h) in Fig. 9. Running split is easy for a given
f = f_1 \gg \cdots \gg f_n: find the rule matching f' \gg f_n,
or in the case of f_n = a \gg b try each of the SPLIT-\gg rules
in turn. We prove that running split always produces a valid
split if it exists in Section 4.

3.2.2 Using split to Find LoopM. The split rule now lets
us find a pre that can be slid into position. Given loop (g \gg
second f), we apply split to f to transform it into loop (g \gg
second (f_i \gg pre v \gg f_r)). We can then right slide f_r
to get LoopM (second f_r \gg g \gg second f_i) v.

This only looks at the right side of the loop however: we
also need to slide anything from the left side over to the right
side so that it is considered by split. This is necessary for
programs in which the pre we are looking for is on the oppo-
site side, such as loop (second (pre v) \gg x). We therefore
left slide as much as we can before applying split.

3.3 LoopM

While the sliding and split rules are enough to transform
most loop f into LoopD f’ i, there is one class of counterex-
amples for which this is not enough. In Fig. 10a, we are
unable to slide a pre into position because neither f nor g
can be slid, meaning that we cannot transform the loop into
LoopD. We need to be able to transform this example how-
ever, as it can be executed by getting the outputs from the
pre, then running g, and finally running f.

To transform loop f where f is split in two halves by a
pre, we introduce a new restricted form of loop called LoopM.
This is defined as follows alongside an interpreter runLoopM
which maps LoopM f i g to a corresponding signal function.

```haskell
data LoopM a b = LoopM (SF (a,c) d) d (SF d (b, c))
runLoopM :: LoopM a b -> SF a b
runLoopM (LoopM f i g) = loop (f >> pre i >> g)
```

Checking whether we can transform loop x into LoopM only
requires application of split. If we can split x to get (f, pre i, g),
then x = f \gg pre i \gg g and therefore loop x = LoopM f i g.
This can be applied to both of our examples in Fig. 10 to get
LoopM f (i, j) g and LoopM (f \gg second h) (x, y) (first i \gg
g) respectively.
We can now transform a single loop \( f \) we can transform compositions of loops or nested loops, with the aim of being able to transform programs consisting of any number of loops.

For composed loops we note that the transformation of a given loop \( f \) relies on nothing except \( f \). This means that we can transform compositions of loops such as \( \text{loop } f \gg \text{loop } g \) by transforming each individual loop, giving us e.g. something of the form \( \text{LoopD } f \gg \text{loop } g \).

Issues arise however when we introduce nested loops such as \( \text{loop } f \). The inner loop \( f \) could contain the \text{pre} that is needed for the outer \text{loop} to be transformed. An example of this is presented in Fig. 11, where the \text{pre} \( i \) in the inner loop is needed by the outer \text{loop} \( f \). We therefore need a way to extract such a \text{pre} from an inner loop \( f \).

In the rest of the section, we look at extracting \text{pre} from nested loops in two cases: one where the inner \text{loop} can be transformed into \text{LoopD}, and another when it can be transformed into \text{LoopM}.

### 3.4 Multiple Loops

We can now transform a single \text{loop } f \) into its equivalent \text{LoopD} or \text{LoopM}. We now consider programs with multiple composed \text{loops} or nested \text{loops}, with the aim of being able to transform programs consisting of any number of loops.

For composed \text{loops} we note that the transformation of a given \text{loop } f \) relies on nothing except \( f \). This means that we can transform compositions of \text{loops} such as \( \text{loop } f \gg \text{loop } g \) by transforming each individual \text{loop}, giving us e.g. something of the form \( \text{LoopD } f \gg \text{loop } g \).

Issues arise however when we introduce nested \text{loops} such as \( \text{loop } f \). The inner \text{loop } f \) could contain the \text{pre} that is needed for the outer \text{loop} to be transformed. An example of this is presented in Fig. 11, where the \text{pre} \( i \) in the inner \text{loop} is needed by the outer \text{loop}. We therefore need a way to extract such a \text{pre} from an inner \text{loop} \( f \).

In the rest of the section, we look at extracting \text{pre} from nested \text{loops} in two cases: one where the inner \text{loop} can be transformed into \text{LoopD}, and another when it can be transformed into \text{LoopM}.

#### 3.4.1 Extracting \text{pre} from \text{LoopD}

When the inner \text{loop} is a \text{LoopD}, we need to get the \text{pre} out of that inner \text{loop} if we want to use it outside of that \text{loop}. We have already seen this in Fig. 11.

To achieve this, we turn again to the \text{ArrowLoop} laws, which given a \text{loop } f \) provide a way to extract unused \text{pre} from \( f \). The two laws we need are stated below.

\[
\text{loop } ((x \gg y) \gg z) \gg y \\
= x \gg \text{loop } ((y \gg z) \gg y) \\
= \text{loop } (x \gg (y \gg z)) \gg y
\]

Right tightening takes a loop with \( (y \gg z) \) as its last term and moves \( y \) outside of the \text{loop}. This preserves program meaning as \( y \) still receives the same inputs and produces the same outputs; it does not need to be within the \text{loop} to still be connected to the first output of \( x \). This is shown in use in Fig. 11, where we use right tightening to move the

![Figure 11](image_url)
pre i outside of the loop while keeping it connected to the first output of g, meaning that we can use it to transform the outer loop. Left tightening is similar, but works with the front of the loop instead.

In most cases there is only one direction in which part of a loop can be tightened. Consider an arbitrary loop ((f₁ ** f₂) ≫ f₃) where any fᵢ can be id. If f₃ is not id, then only f₁ can be moved outside of the loop via left tightening as f₅ “blocks” f₁ from being moved this way, and only f₁ can be moved outside of the loop via right tightening through a similar argument. In these cases, we apply left and right tightening to move f₁ and f₄ out of the loop once we have transformed it into LoopD.

In the case where f₅ = id, we end up with loop ((f₁ ≫ f₂) ≫ (f₃ ≫ f₄)). f₁ and f₄ could be tightened out of the loop in either direction, but we do not know which way we need to tighten them to e.g. get the pre needed for an outer loop. The trick is to consider the more general case of loop (f ≫ g), in which we only need to run f to get the output of the loop: g will never be run as its result is never needed. Thus we avoid having to decide which way to tighten by removing the loop, f₂ and f₅ to get f₁ ≫ f₅.

We can extract a pre from an inner LoopD by therefore either applying left and right tightening to move as much from inside the LoopD out as possible, or remove the LoopD entirely using loop (f ≫ g). In either case, this allows pre that were part of the inner loop to be used when transforming the outer loop.

3.4.2 Extracting pre from LoopM. When we have an inner LoopM, things are simpler than for LoopD. The trick is that LoopM itself is decoupled like pre, as it can produce all of its outputs without any of its inputs by using its internal pre. To run a program like loop (f ≫ second (LoopM g (i, j) h)), we can first get the output of the inner LoopM by running its internal pre, then run h to get the second output of the outer loop, and then finally run f and g.

Since our aim with these restricted forms of loop is to fix the location of a decoupled part so that we know the execution order of the loop at compile time, we allow LoopM to take the place of a pre when transforming our loops. This is implemented via a minor update to the split operation:

\[
\text{split-loopM} \quad x \Rightarrow \text{LoopM} f d g \Rightarrow (x, \text{LoopM} f d g, \text{id})
\]

With this we can now use LoopM wherever we were aiming for a pre, allowing our earlier example to be expressed as LoopD f (LoopM g (pre (i, j) h)). Note that the definitions of LoopD and LoopM have changed slightly since they previously relied on specifically containing a pre, but can now contain arbitrary SFs consisting of pre, LoopM and ≫ ≫. The updated definitions can be seen in the code for Decoupled in Section 5.1.

3.5 Transformation Algorithm

We now combine the rules we have described for transforming different cases of loops into an algorithm that we can run. The overall process that we present inspects loops from innermost to outermost, transforming each one to a LoopD or LoopM until every loop is transformed. After transforming the program into CCA composition form (Section 3.1.4), perform the following for each loop from the innermost to the outermost:

1. Attempt to apply loop (f₁ ≫ f₂) = f₁ (Section 3.4.1) to remove the loop altogether. In CCA composition form, this is equivalent to checking whether each term aᵢ in loop (a₁ ≫ a₂ ≫ ⋯ ≫ aₙ) consists of aᵢ ≫ aᵢ.
2. If that does not work, attempt to transform the loop f to LoopM f’ d g’ by using split to find f = f’ ≫ d ≫ g’ for decoupled d (Section 3.3).
3. If that does not work, attempt to transform the loop f into LoopD f’ i in three stages:
   a. Slide left as much as possible, using left fill as needed, to get a program of the form loop (g ≫ second x) (Section 3.1 and Section 3.2.2).
   b. Apply split to x to get f = g ≫ second (x₁ ≫ x₂ ≫ ⋯ ≫ xᵢ). Right slide xᵢ to get xᵢ in the right position, giving us LoopD (second xᵢ ≫ g ≫ second x₁) x₁ (Section 3.2).
   c. If this works, apply left and right tightening to extract any more pre or LoopM that could be used in outer loop from that LoopD (Section 3.4.1).

4 Completeness

We now prove that our transformation works on all loop without dependency cycles while preserving program meaning. In Section 3, we proved that program meaning is preserved by each individual operation through existing laws for arrows and CCAs. Since our transformation solely uses these operations, it preserves program meaning. We now prove that it is complete for any loop with no dependency cycles.

We first need to formalise what a dependency cycle is, in order to reason about them. Some output o depends on an input i if there is some path through the program from i to o: meaning that in order to get o, we need to know i. Decoupled functions like pre break this dependency as they are able to produce an output at a given time step without the input at that time step; the dependencies we therefore consider are represented by paths which do not go through a decoupled signal function. We define this as a direct dependency:

Definition 4.1. A direct dependency exists from input i to output o of some signal function f if there is a path from i to o through f that does not flow through a pre or LoopM (equivalently, a decoupled signal function).

A dependency cycle arises if there is a direct dependency from i and o and one from o to i. In loop f, dependency cycles...
are created via loop $f$’s backedge from each part of its second output to each part of its second input. We define a direct dependency cycle within a loop as follows.

**Definition 4.2.** A direct dependency cycle within loop $f$ exists if there is a direct dependency through $f$ from a component of the second input of $f$ to the same component of the second output of $f$.

Let us build an intuition for how paths, and thus direct dependencies, are built from each of the arrow constructors:

- **$\text{arr}$**: Since we know nothing about $f$, we assume that there is a path from every input to every output, and therefore every output of $\text{arr} f$ directly depends on every input.
- **$\text{pre}$**: If we have a $\text{pre}$, then there is a path from every input to every output which trivially goes through a decoupled signal function, so there is no direct dependency between $\text{pre}$’s inputs and outputs. The same holds for $\text{LoopM}$.
- **$f \gg g$**: This sequentially composes the paths through $f$ and $g$: if there is a path through $f$ from $i$ to $m$, and a path through $g$ from $m$ to $o$, then there is a path through $f \gg g$ from $i$ to $o$.
- **$f \triangleleft g$**: This composes two paths in parallel that do not interact. Therefore it will have a $\text{pre}$ on every path if $f$ and $g$ each have a $\text{pre}$ on every path between their inputs and outputs.

We now present some auxiliary lemmas used within our main proof. We first define three forms that a loop can take which allow us to perform case analysis in our other proofs:

**Lemma 4.3.** Any loop $f$ can be represented as one of the following cases:

**Case 1** $\text{loop}(a \triangleleft b)$

**Case 2** $\text{loop}(f \gg g)$, where either:

- **Case 2a** $c = r_1$, or
- **Case 2b** $c = r_1 \triangleleft c' \triangleleft r_2$

where $r_1$ is $\text{arr} x$, $\text{pre} v$, $\text{LoopD} f' d$ or $\text{LoopM} f' d' g'$.

**Proof.** Express $f$ in CCA composition form: $f = f_1 \gg f_2 \gg \ldots \gg f_n$. If every $f_i$ is $x_i \triangleleft y_i$, then $f$ is in the form denoted by Case 1, with $a = x_1 \gg \ldots \gg x_n$ and $b = y_1 \gg \ldots \gg y_n$.

If exactly one $f_i$ is not $x_i \triangleleft y_i$, then $f$ is in the form denoted by Case 2a, with $a = x_1 \gg \ldots \gg x_{i-1}, b = y_1 \gg \ldots \gg y_{i-1}, c = f_i, d = x_{i+1} \gg \ldots \gg x_n$ and $e = y_{i+1} \gg \ldots \gg y_n$.

Otherwise, denote the first and last non-$\triangleleft$ terms in $f$ as $f_1$ and $f_j$ respectively. $f$ is of the form in Case 2b, with $a = x_1 \gg \ldots \gg x_{i-1}, b = y_1 \gg \ldots \gg y_{i-1}, c = f_i \gg \ldots \gg f_j, d = x_{i+1} \gg \ldots \gg x_n$ and $e = y_{i+1} \gg \ldots \gg y_n$. □

We now prove that our split operation (Section 3.2.1) will transform every $f$ with no direct dependencies between its inputs and outputs into $f_1 \gg f_2 \gg f_3$ where $f_3$ is decoupled.

We assume that every output of any $\text{LoopD}$ within $f$ directly depends on every input of it: this is necessary as it is possible to construct examples where a $\text{LoopD}$ can have a decoupled signal function “hidden” inside it (discussed in Section 3.4.1). We show later that our transformation does not generate $\text{LoopD}$ like that.

**Lemma 4.4.** Given $f$ in CCA composition form for which:

1. $f$ contains no $\text{loop}$, and
2. For every $\text{LoopD}$ within $f$, all of its outputs directly depend on its inputs,

there are no direct dependencies between inputs and outputs of $f$ if and only if we can apply split to $f$.

**Proof.** We consider the two directions of the equivalence in turn. The $\Leftarrow$ direction is simple: all paths from the inputs of $f$ to the outputs of $f$ must go through $f_2$ by definition of composition. $f_2$ is decoupled so every path goes through a decoupled signal function, and thus there cannot be any direct dependencies.

We now turn to the $\Rightarrow$ direction, which we prove by induction on the size of $f$. We define the size of $f$ as follows:

- $f \triangleleft g$: All of $f$’s outputs directly depend on its inputs.
- $f = \text{arr} x$: All of $\text{arr}$’s outputs directly depend on its inputs.

We consider cases that $f$ and $g$ each have size equal to the sum of the sizes of $f$ and $g$, and all other terms have a size of 1.

We start with our base case: a program of size 1, meaning $f$ is a direct dependency cycle in $f$, by definition and the second condition of our lemma respectively. Therefore, it must be $\text{pre} v$ or $\text{LoopM}$. In either of these cases, split applies to all inputs and we are done.

We now prove our lemma for $f$ of size $k + 1$, assuming that it holds for $f'$ of size $k$ and smaller. We consider cases of $g$ in $f = f' \gg g$:

- **$g = \text{arr} x$**: All of $\text{arr}$’s outputs directly depend on its inputs.
- **$g = \text{pre} v$ or $\text{LoopM}$**: $f$ matches the form needed by split.

If $z$ is $\text{arr}$ or $\text{LoopD}$, then there is a path from every input of $z$ to every output. Consider cases of $w$.

First, if $w$ has no direct dependencies, we can apply split to it by the induction hypothesis and thus split can be applied to skip past $g$ and $z$.

Otherwise, if $w$ has a direct dependency from $w_i$ to $w_o$, there must be no direct dependencies in $g$ for there to be
no direct dependencies in \( f \). This is because if we had a direct dependency from \( g_i \) to \( g_k \) in \( g \), we would have a direct dependency in \( f \): from \( w_i \) to \( w_k \), then through \( z \) to \( g_i \), and finally to \( g_k \).

This means that each of \( x_2 \gg ... \gg x_{k+1} \) and \( y_i \gg ... \gg y_{l+1} \) have no direct dependencies by definition of \( \gg \). We can therefore split each of \( x \) and \( y \) separately by the induction hypothesis and thus can apply SPLIT-\( \gg \)-R.

Therefore, we have proven the property holds for the base and inductive cases, so it holds by induction.

Finally, now show that the transformation is complete.

**Theorem 4.5.** Using our transformation, we can transform a loop \( f \) which either has no direct dependency cycles in it or has \( f = a \gg b \).

**Proof.** By induction on loop nesting. Consider a loop \( f \) which contains no loop, LoopD or LoopM. By Lemma 4.3, we can express \( f \) in one of three forms, which we show can be transformed in turn.

**Case 1.** \([f = a \gg b.]\) Apply step 1 of the transformation to get loop \( f = a \).

**Case 2b.** \([f = (a \gg b) \gg r_1 \gg c \gg r_2 \gg (d \gg e)]\), where \( r_1 \) and \( r_2 \) are arr or pre.] Step 1 does not apply, so we move onto step 2. If step 2 succeeds in splitting \( f \), we finish with a LoopM.

If we cannot split \( f \), then we must have that \( r_1 \) and \( r_2 \) are arr. This is because if either is pre, we are done since \( f \) has a decoupled signal function \( r_l \) on every path from the inputs of \( f \) to its outputs, meaning that it has no direct dependencies and thus can be split according to Lemma 4.4.

We know for the same reason that \( c \) has at least one direct dependency from \( c' \) to \( c'' \), as otherwise we could split \( f \): every path in \( f \) goes through \( c' \), and if \( c' \) has no direct dependencies then every path through \( c' \) has a decoupled signal function on it.

It follows that \( c = arr x \gg c' \gg arr y \) has a direct dependency from all of its inputs to all of its outputs: from each input of \( arr x \) to \( c'_i \), to \( c''_i \), and finally to each output of \( arr y \).

Therefore, if we get to step 3 we are working with \((a \gg b) \gg c \gg (d \gg e)\). We apply step 3a to get \((a \gg id) \gg c \gg (d \gg (e \gg b))\).

We require that it is possible to split \( e \gg b \) in the absence of dependency cycles for step 3b to be applicable. Assume for contradiction that \( e \gg b \) cannot be split, meaning that there exists a direct dependency within \( e \gg b \) from the \( j \)th input to the \( i \)th output. We then have a dependency cycle as follows: from the \( i \)th part of the second input of the loop, through \( c \) to the \( j \)th input of \( e \gg b \) to its \( i \)th output and thus completing the cycle. Therefore, there must be no direct dependencies in \( e \gg b \) and thus it can be split by Lemma 4.4.

We therefore get a LoopD of the form LoopD \( ((a \gg b) \gg c \gg (d \gg (e \gg b))) y \). We apply tightening in step 3c to get \( a \gg LoopD ((id \gg b) \gg c \gg (id \gg e)) z \gg d \). Note that there is a direct dependency from every input to every output of this LoopD, as each input only goes through \( c \) to get to the output, and we showed earlier that \( c \) has a direct dependency from every input to every output.

**Case 2a.** \([f = (a \gg b) \gg r_1 \gg (d \gg e)]\), where \( r_1 \) is arr or pre.] Case 2a is proved similarly to Case 2b. Step 1 does not apply, so we move onto step 2. If step 2 succeeds in splitting \( f \), we finish with a LoopM.

If we cannot split \( f \), then \( r_1 \) must be arr since if it were pre then \( f \) would have no direct dependencies as all paths go through that pre, meaning that \( f \) in that case can be split by Lemma 4.4.

We are therefore working with \( f = (a \gg b) \gg arr x \gg (d \gg e) \). Apply step 3a of the transformation to get \((a \gg id) \gg arr x \gg (d \gg (e \gg b))\). We know that every output of \( arr x \) directly depends on every input, meaning that every input of \( e \gg b \) directly depends on every part of the second input of the loop, by the same logic as in Case 2b. We are guaranteed to be able to apply split in step 3b by the Lemma 4.4.

We end up with a LoopD of the form LoopD \( ((a \gg b) \gg arr x \gg (d \gg (e \gg b))) y \). We apply tightening in step 3c to get \( a \gg LoopD ((id \gg b) \gg arr x \gg (id \gg e)) y \gg d \). Note that each output of this LoopD directly depends on each input, as each path only goes through \( arr x \).

This concludes the base case. We also note that in every case where we create a LoopD, all of its outputs depend on its inputs. This means that we can use Lemma 4.4 in the inductive step as all LoopD considered will have this condition hold.

We now turn to the inductive step: proving the transformation works on loop \( f \) when all loop within \( f \) without dependency cycles can be transformed. We first apply our transformation to the inner loop. Then, the proof above also proves the inductive case, but with some minor modifications. We can now have \( r_1 \) being LoopD or LoopM. Any LoopD can be treated identically to arr, as every LoopD created by this transformation has identical dependencies to an arr. Any LoopM can be treated identically to pre for the same reason. Therefore, the inductive step holds, and the proof holds.

**5 Implementation**

In this section we describe the Haskell implementation of our transformation, Severn\(^1\). We start with a minimal AFRP implementation, then implement the transformation on top, and finally test that the implementation is correct.

\(^1\)Available from https://github.com/finnbar/severn/tree/v1.0.0.0 and in the provided artefact [11].
5.1 Signal Descriptors and CCA Composition Form

We represent signals as in Chupin and Nilsson’s SFRP [2] by using signal descriptors. These are defined by the Desc GADT, which is then lifted to the kind level via DataKinds.

```haskell
data Desc x where
  V :: a -> Desc a
  P :: Desc a -> Desc b -> Desc (a, b)
```

This allows us to define signals which produce values of some type, and pairs of signals: P (V Int) (V Int) describes a pair of signals each containing Ints. Values produced by a signal with descriptor d are represented by the GADT Val d, used throughout the implementation.

Our arrow constructors are parameterised by these descriptors. This again mirrors SFRP, but rather than defining a single GADT with all of the arrow constructors, we enforce CCA composition form (Section 3.1.4) through the definition of multiple GADTs:

```haskell
type CFSF :: Desc s -> Desc s' -> Type
data CFSF x y where
  (>>>:) :: NoLoop a b -> NoLoop b c -> CFSF a c
  Single :: NoComp a b -> CFSF a b

data NoComp x y where
  LoopD :: CFSF (P a c) (P b d) -> Decoupled d c
  NoComp a b
  Arr :: (Val a -> Val b) -> NoComp a b
  Loop :: CFSF (P a c) (P b c) -> NoComp a b
  (<<<: :<<<) :: NoComp a b -> NoComp c d
  NoComp (P a c) (P b d)
  Id :: NoComp (V a) (V a)
  Dec :: Decoupled a b -> NoComp a b

data Decoupled x y where
  LoopM :: CFSF (P a c) d -> Decoupled d e
  CFSF e (P b c) -> Decoupled a b
  Pre :: Val (V a) -> Decoupled (V a) (V a)
  BothDec :: Decoupled a b -> Decoupled a' b'
  Decoupled (P a a') (P b b')
```

By having CFSF (read composed form signal function) only introduce \(\Rightarrow\), and NoComp introduce the remaining combinators, \(\Rightarrow\) can only be added at the top level so that programs must be written in composition form. We also separate out decoupled terms into their own GADT, allowing us to enforce through the type system that a term is decoupled as shown in LoopD.

We provide smart constructors for each of the traditional arrow combinators that produce an equivalent CFSF using the laws discussed in Section 3, to avoid programmers having to directly use the above constructors to write their programs. This means that a programmer can write a CFSF in Severn in the same way that they would an SF in Yampa. We also implement a small optimisation pass which merges consecutive Arr together using the arr f \(\Rightarrow\) arr g = arr (g ∘ f) law, which is also an optimisation applied by Yampa [16].

5.2 The Transformation Algorithm

We now outline our implementation of our transformation on CFSFs. We focus on transformLoop, which transforms a given Loop using the steps outlined in Section 3.5; the transformation itself traverses the input CFSF by calling transformLoop on each Loop from innermost to outermost.

Each of the three cases outlined in our transformation are defined as a CFSF \(a \ b \Rightarrow\) Maybe (CFSF \(a \ b\)) function, since a given case is not applicable to every CFSF. transformLoop is therefore defined as trying out each case using the alternative operator |>

The implementation of each case utilises the rules defined in Section 3, which are each implemented as Haskell functions. As an example, we present a slightly simplified implementation of leftSlide below:

```haskell
data LoopBox a b where
  LB :: CFSF (P a c) (P b c) -> LoopBox a b

leftSlide :: LoopBox a b -> Maybe (LoopBox a b)
leftSlide (LB cfsf) =
  case headTail cfsf of
    Left _ -> Nothing -- Cannot slide if no >>>:
    Right (HT s ss) -> case s of
      s1 :***: s2 -> Just $ LB $ (Single s1 <<< id) >>> ss >> (id <<< Single s2)
      _ -> Nothing -- Cannot slide non-<<<: term.
```

We apply a few tricks here which are common throughout the implementation. We use auxiliary GADTs when we cannot determine the exact type of the output from the type of the input CFSF: here we require LoopBox since we cannot guarantee that sliding will lead to the same type c. We can also use this to guarantee that some part of the output is decoupled, which we do in our implementation of split.

Since CFSF allows arbitrary bracketing of \(\Rightarrow\), we cannot use pattern matching to get the first or last element of a given composition. We therefore provide headTail x to do this, which returns Left x if there are no \(\Rightarrow\), or the head and tail of the chain of \(\Rightarrow\) otherwise. The rest of the implementation follows from the definition of the rule: use our auxiliary functions to match the form of the rule, and if we can, return the result of applying it.

5.3 Running CFSFs

Once the transformation has been applied, we are left with a CFSF containing no Loop. Severn provides runCFSF :: CFSF a b -> a -> (b, CFSF a b) to run these transformed CFSF, taking an input value and producing the output at that time step along with the next CFSF to run. Since any CFSF applied to runCFSF no longer contains Loop, we define it strictly and thus avoid all of the overheads of lazy evaluation. The decoupled parts of LoopD and LoopM are run with

```haskell
runDec :: Decoupled a b
       -> (Val b, Val a -> Decoupled a b)
```
which produces the output without using any input, and the program to run at the next time step once it gets an input. Thus running LoopD \ f \ d consists of getting the output from d, running f and then using d’s input to get the next CFSF.

5.4 Testing via Arbitrary Program Generation
As well as our proof that the transformation is correct (Section 4), we also test arbitrary Severn programs without dependency cycles against their Yampa equivalents to make sure that the implementation is correct. We build test programs with a pair of mutually recursive generators using the Hedgehog library\(^5\) which generate a Yampa SF and its equivalent Severn CFSF. One test generates decoupled programs, and one generates non-decoupled programs.

Programs are generated inductively: start with the smallest decoupled program pre v and the smallest non-decoupled program arr f, and then build larger programs by combining them. We use rules similar to those used by Sculthorpe and Nilsson [27] for their arrow combinator types indexed by decoupledness, with rules such as f \ \gg \ g being decoupled if one of f or g is. To generate a program of a given size and decoupledness, we generate two smaller programs and combine them using one of those rules.

These arbitrary decoupled and non-decoupled programs are then used to build loops for testing. We use the same techniques as in Case 2b of Theorem 4.5 for building generic loops without dependency cycles: start with loop ((a \ \gg \ b) \ \gg \ c \ \gg \ (d \ \gg \ e)), generating LoopM by generating a decoupled c and generating LoopD by generating a decoupled e \ \gg \ b.

We are able to test that if we have an SF and its equivalent CFSF, both programs produce the same results after transforming the CFSF using our transformation. Our implementation passes the tests for programs of arbitrary size.

6 Performance
To show the impact of our work on performance, we have two sets of benchmarks: on fixed networks to identify the improvements for specifics constructs, and on randomised networks. We use the Criterion library\(^6\) to benchmark each program 100,000 times.

We first benchmarked four programs in order to test individual uses of loop. Fig. 12 shows the benchmarks, their definitions and average speedups compared to Yampa. Severn gives a speedup of between 1.5x and 1.7x for the first three benchmarks. The nested program gave a lower speedup of 1.1x, which we expect is due to the CFSF being allocated by Severn being larger in that case.

We also varied the number of arr in arrs, in order to test whether the speedup varied based on the amount of work done within the loop, but found no clear change. We expect this is due to the optimisations implemented by Yampa and in our optimisation pass (Section 5.1): the composition law allows composed arr to be merged into existing arr, so that the enlarged program is effectively the same but with larger pure functions to execute.

To avoid the effects of these optimisations we constructed a benchmark based on LoopM with pres and arrs interleaved. By doing this, the composition law could not be applied. The results for this are shown in Fig. 13. We achieved speedups of between 1x and 2x for programs with 150 or fewer primitives, but performance improved significantly with 200 primitives.

For our randomised tests, we take a similar approach as was done for SFRP [2]. We varied two parameters: the size of the generated program, and the number of loops within that program as a proportion of the size. We do not include the time it takes our transformation to run. We found that our speedups were always greater than 1x, and averaged 2.5x\(^7\).

Since we achieve speedups in all benchmarks, we conclude that our transformation provides an effective improvement for loops in AFRP. Further improvements may possible in the future by using IORef’s to allocate pres rather than returning an entirely new CFSF, by using stream fusion [3], or with the Pipes library\(^8\) to get more performance out of runCFSF.

---

\(^5\)https://hackage.haskell.org/package/hedgehog
\(^6\)https://hackage.haskell.org/package/criterion
\(^7\)The full results can be found alongside the artefact [11].
\(^8\)https://hackage.haskell.org/package/pipes
7 Limitations and Future Work
While our transformation works on a large subset of Yampa programs, we make a few assumptions that may not hold for all AFRP programs. We discuss the consequences here and how our work could be extended in future work.

7.1 Effects and Monadic Signal Functions (MSFs)
We require that our arrows satisfy the laws needed for CCAs in order to apply the distributive law, which underpins most of operations we have defined. We also assume that there are no side-effects in the sliding law. These two points mean that our transformation is not applicable to effectful programs in general: it changes the execution order so that that a program can be run strictly, which may not be the same execution order as before. In the following example, lazy semantics will run \( f \) first since \( \text{pre } x \) produces its input immediately:

\[
\text{loop } (g \gg \text{first } (\text{pre } x \gg f) \gg i \gg \text{second } (\text{pre } v))
\]

However, our transformation will turn this into a LoopD which, when run strictly, runs \( g \) first. This does not pose an issue in Yampa as \( \text{pre} \) is the only “effectful” operation, and its effects are entirely local to itself. However, it is easy to add effects into such a system: Perez et al. embed monads into AFRP with Monadic Signal Functions (MSFs) [20]. Modifying our transformation to preserve execution order, and therefore support MSFs, is future work.

It is important to note however that the reordering does not change the meaning of programs when our effects are commutative: we eventually run every part of the program. Piponi [23] shows a number of monads whose effects are commutative, meaning their computations can be reordered without issues. MSFs built with these commutative monads, such as Reader and Writer with a commutative monoid, could therefore be safely transformed by our technique.

7.2 \textit{arr} is a Black Box
We know nothing about \( f \) within \textit{arr} \( f \) and must assume that all outputs of \( f \) depend on all of its inputs. However, \textit{arr} is the only constructor we have for routing data and it permits programmers to write routing functions like \( \text{swap } = \text{arr } (\lambda (x, y). (y, x)) \) where that assumption is not required. Dealing with this would introduce complex dependencies between the inputs and outputs of an \textit{arr}, but would allow additional ways to transform programs: for example, \( \text{first } (\text{pre } x) \gg \text{swap } = \text{swap } \gg \text{second } (\text{pre } x) \). This poses a particular problem when working with \textit{proc} notation [19], which introduces many additional \textit{arr} during desugaring.

If we differentiated between \textit{arr} for applying pure functions and \textit{arr} for routing, we could modify our transformation to take these into account. Joseph’s \textit{generalised arrows} [10] introduces a variety of additional combinators such as \textit{ga_assoc} and \textit{ga_swap} with which routing can be implemented without using \textit{arr} explicitly. SFRP [2] uses \textit{routers} for arbitrary rearrangements of inputs into outputs.

7.3 Switching and Choice
Members of the \textit{ArrowChoice} class allow for conditional execution of arrows. The key operator is \( f +++ g \), which runs \( f \) if given a \textit{Left} value and \( g \) otherwise. Since \( f +++ g \) depends on its input to decide which of \( f \) and \( g \) to run, we can never decouple it. It can therefore be treated in the same way as \textit{arr} and thus should be easy to add.

Switching is harder to add: \textit{switch } \( f \ c \) uses a continuation \( c \) to change the arrow being run (\( f \)) to a different one at runtime. This means that \textit{switch} can change the structure of a program in a way that is unknown at compile-time. SFRP implements switching by running its transformation again once a switch occurs, but this can slow down the program temporarily as the entire transformation procedure is rerun. Winograd-Cort and Hudak transform some uses of \textit{switch} into +++, which avoids these issues for them [29].

7.4 Well-typed loop
We proved that our transformation only fails if we are unable to run a program anyway (Section 4). Therefore, if our transformation succeeds, a \textit{loop} is well-formed. A type system which guarantees that a \textit{loop} contains no dependency cycles would be helpful to avoid running the transformation on a \textit{loop} with dependency cycles. This also avoids the issue of a \textit{switch } \( f \ c \) generating a \textit{loop} that cannot be transformed, thus generating a runtime error.

A type system for checking for no dependency cycles in \textit{loop} could build on some of the existing work by Sculthorpe and Nilsson [27] who label the decoupledness of signal functions at the type level, and Bahr [1] who introduces a modal type system that detects space leaks.

8 Related Work

\textit{FRP Applications.} FRP sees significant use in a variety of domains where performance is important. It has been used in many embedded settings: the Juniper language for Arduino microcontrollers [7], the Hailstorm language for IoT [25] and the Emfrp language for embedded systems [26] are three examples which use a variant of FRP designed for restricted memory use, but could move to AFRP if it became performant enough. The original introduction to AFRP [8] discussed robots as its basis, which also tend to consist of programs run on embedded systems.

There has also been some research into making FRP safer in these contexts. Perez and Goodloe [22] incorporate fault tolerance into FRP, which could also be useful in domains like robotics. Copilot [21] allows users to write runtime verification systems in the style of FRP.

\textit{AFRP Optimisation.} Beyond CCAs (Section 2.1), Scalable FRP (SFRP) [2] is another optimisation which transforms AFRP programs into IO operations on mutable memory cells to reduce the cost of routing data between signal functions.
Notably, SFRP does not currently support the \( \text{loop } f \) combinator at all since it needs to know the order in which the component signal functions of \( f \) are executed to order its IO operations. Our transformation would allow the \( \text{loop} \) combinator to be added.

Other projects have taken a similar approach to SFRP: Ultrametric FRP [12] implements FRP as an imperatively updated dataflow graph, and Patai’s work on higher-order streams [18] translates FRP to an IO stepper action which runs with each new sample.

A common optimisation in the FRP world that may be applicable to AFRP is deciding whether a value needs to be computed at a given time step. Elliott [4] discusses how values that only sometimes change should only be recomputed when new values are pushed, but also that the results of FRP code should only be recomputed when they are pulled by whatever is utilising its results. Sculthorpe and Nilsson [28] define some temporal logic properties of FRP networks that could also be used to reason about change and thus whether a value needs recomputing.

All of the above optimisations could be combined with our work, and would likely produce improved speedups compared to those we presented in Section 6.

**Synchronous Programming.** Much of the work we have discussed aims to bring FRP’s efficiency and safety closer to that of *synchronous dataflow languages* such as Lustre [6] which also permit writing reactive programs. While less expressive than FRP, they are simpler to efficiently implement [13]. They deal with dataflow cycles (\( \text{loop} \) in AFRP) via a syntactic check for a \text{delay} operator present in every cycle, which is similar to what we have done here. Digital circuits are similar: Ghica et al. [5] introduce a theory for rewriting dataflow categories with a \text{delay} operator, which is then used to talk about digital circuits that could also be applied to AFRP.

9 Conclusions

We showed that \( \text{loops} \) in AFRP without dependency cycles can be transformed into more restrictive \( \text{LoopD} \) and \( \text{LoopM} \) forms which can be evaluated strictly, thus avoiding the overheads of lazy evaluation. This offers performance benefits and allows for easier compilation of bespoke AFRP-style languages in the future since such a language will no longer need lazy evaluation.

We proved that our transformation preserves program meaning, both theoretically using the Arrow and ArrowLoop laws, but also practically through a Haskell implementation whose tests ensure that programs before and after transformation behave equivalently. We also provided a proof that this transformation works on every \( \text{loop} \) expressible in our subset of AFRP that does not contain a dependency cycle.

While our implementation is a subset of Yampa, we believe it is large enough to support most useful programs. We also laid out how we could extend our transformation to be able to support even more programs in future. Finally, our benchmark shows that our implementation, Severn, provided a modest speedup for a variety of AFRP programs, and outlined potential further improvements through implementation of Yampa’s optimisations.

**Data Availability Statement**

The implementation of Severn which is tested in Section 5.4 and benchmarked in Section 6 is available as an artefact on the ACM Digital Library [11]. It is also available on GitHub\(^9\). The full data from our benchmarks in Section 6 is also provided within the artefact.

**Acknowledgments**

We would like to thank the anonymous reviewers for their comments, including for a previous version of this paper submitted to ICFP. We also thank Alex Dixon for proofreading and providing helpful comments on an earlier version of this manuscript. The first author is funded via EPSRC grant #2436228.

**References**


---

\(^9\)https://github.com/finnbar/severn/tree/v1.0.0.0


Received 2023-06-01; accepted 2023-07-04