Functional Reactive Programming, restated

Guerric Chupin
University of Nottingham
Nottingham, United Kingdom
Guerric.Chupin@nottingham.ac.uk

Henrik Nilsson
University of Nottingham
Nottingham, United Kingdom
Henrik.Nilsson@nottingham.ac.uk

ABSTRACT

Functional Reactive Programming is an approach to declarative programming of reactive systems by describing interactions between time-varying values. FRP implementations are often realised as an *embedding* in a functional host language, making for very expressive reactive programming frameworks. However, this expressiveness comes at a cost: current embedded FRP implementations incur substantial performance overheads, in particular for values that (notionally) vary continuously. The basic idea of FRP is closely related to synchronous data-flow and continuous system simulation languages. In contrast to FRP, these handle values that vary continuously efficiently, but are less expressive. This paper seeks to bridge this gap by proposing a novel approach to embedded FRP-implementation that uses the fundamental implementation approach of synchronous dataflow and simulation languages for efficient handling of continuously varying values, while retaining the expressiveness normally associated with FRP, as well as paying attention to values that only change relatively infrequently. These ideas are applicable beyond FRP, for example for implementing flexible embedded simulation languages. We evaluate our approach on a range of benchmarks, including an existing full-fledged video game where using our new FRP implementation as a drop-in replacement for the old one gave a three-fold performance improvement.

ACM Reference Format:
Guerric Chupin and Henrik Nilsson. 2019. Functional Reactive Programming, restated. In *Principles and Practice of Programming Languages 2019 (PPDP '19), October 7–9, 2019, Porto, Portugal.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3354166.3354172

ACKNOWLEDGMENTS

The authors would like to thank Jérôme Mahuet, author of the Yampa Flappy bird that we used to evaluate our library, as well as Olivier Nicole and anonymous reviewers for helpful feedback on this paper.

1 INTRODUCTION

Reactive programming is everywhere: from embedded systems, via video games, to distributed web-applications and modern mobile applications. Functional Reactive Programming (FRP) [11, 37] is a principled, declarative approach to programming reactive systems, focusing on describing interactions between time-varying values rather than reacting to individual events, and commonly supporting both continuous[1] and discrete notions of time. As such, FRP addresses some of the inherent difficulties in programming reactive systems, and, in various concrete incarnations, it has had a considerable uptake [1, 4, 13, 32] as well as inspired related approaches such as ReactiveX [29] and (the original) Elm [9]. Further, being based on time-varying values, FRP is not intrinsically limited to reactive applications, but can be a useful way to structure time-aware programs more generally, such as simulations [35].

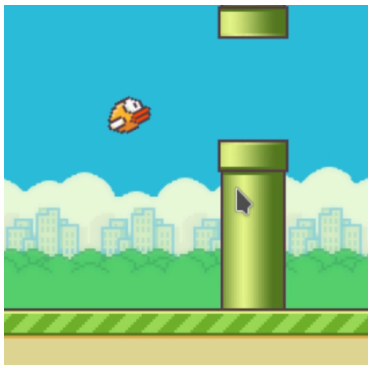


Figure 1: A version of a Haskell Flappy Birds game [19], modified to run using our Scalable FRP library

FRP is often realised as an Embedded Domain-Specific Language (EDSL) in a functional host language like Haskell. Yampa is a prominent representative of this approach. It is structured using arrows [17] and has been used for various kind of applications such as games [8], musical applications [14, 22] or robot simulators [16].

However, current embedded FRP implementations incur substantial performance overhead, in particular for continuously varying values. In fact, most recent FRP endeavours has focused mainly on discrete time [1, 9, 36]. This is in contrast to synchronous dataflow languages: while they might not conceptually provide a continuous notion of time, implementations support values that change "all the time" very efficiently [3]. Similarly, continuous system simulation relies on efficient realisation of continuously varying values [5, 15]. On the other hand, embedded FRP implementations tend to make the flexibility of the functional host language available also at the reactive level, allowing for higher-order programming and description of systems that evolve structurally over time, a level of flexibility usually not offered by synchronous dataflow languages in order to allow for efficient implementation offering static performance guarantees.

[1] Conceptually continuous: Implementations discretize time, but the aim is to still allow programmers to mostly think in terms of continuous time.

PPDP '19, October 7–9, 2019, Porto, Portugal

ACM ISBN 978-1-4503-7249-7/19/10…$15.00
https://doi.org/10.1145/3354166.3354172

In the case of arrowized implementations, like Yampa, the overall performance relies heavily on the efficiency of the "wiring" of the networks, the way the communication between different parts of the network is realised. As we highlight in the following, the overhead can be significant, and worse, at least in the case of Yampa, it grows with the system size, leading to poor scalability.

This paper shows how the performance of arrowized FRP implementations can be improved manifold by making use of a stateful implementation, in many ways similar to that of synchronous dataflow languages and code for continuous system simulation, to mostly eliminate the wiring overhead, while retaining the flexibility normally associated with FRP. Additionally, and unlike Yampa, we maintain a strict separation, enforced at the type level, between continuously varying values and those that change, or exist, only at discrete points in time. This allows for more precise descriptions of reactive networks. We refer to this new implementation approach as *Scalable FRP* (SFRP). It makes use of extensions to the Haskell type system as implemented by the Glasgow Haskell Compiler (GHC), such as Generalized Algebraic Data Types (GADT) [28] and data type promotion [39]. While the setting of this work is arrowized FRP, our techniques are applicable beyond FRP, e.g. for embedding of flexible simulation languages.

We evaluated SFRP on a range of benchmarks designed to systematically test various performance aspects. Performance over Yampa improved consistently, in some specific cases even asymptotically. Additionally, both to check the maturity of our new FRP implementation, and to get an indication of what performance gains one might expect in real applications, we used the new implementation as a mostly drop-in replacement for Yampa in an existing game, *Flappy Birds* [19], see figure 1. In this case, we got a three-fold performance improvement over the original Yampa-based version, which already had more than adequate performance. Our Scalable FRP implementation does not yet cover all of Yampa; for example, collection-based switching [23] and general feedback remain future work (section 7). Nevertheless, the library is evidently already a performant alternative for many real FRP applications, particularly when continuously varying values feature prominently.

The rest of the paper is organized as follows. Section 2 reviews the basics of FRP and related notions, while section 3 looks specifically at how Yampa is implemented and its performance shortcomings. Section 4 then presents the implementation of Scalable FRP, showing in particular how the wiring overhead of Yampa can be eliminated. Section 5 evaluates our work both qualitatively and quantitatively with a particular emphasis on performance. Finally, we consider related work in section 6, focusing in particular on how the work here differs from other FRP systems with effectful implementations, and give conclusions in section 7.

2 TECHNICAL BACKGROUND

This section gives a rapid overview of the concepts used in Functional Reactive Programming (FRP) with a focus on Yampa.

2.1 Signals and signal functions

Yampa is centred around two abstractions: *signals* and *signal functions*. A signal represents a time-varying value: conceptually, it is a function from time to values:

$$Signal\ \alpha \approx Time \to \alpha$$

A signal function is a function from signal to signal:

$$SF\ \alpha\ \beta \approx Signal\ \alpha \to Signal\ \beta$$

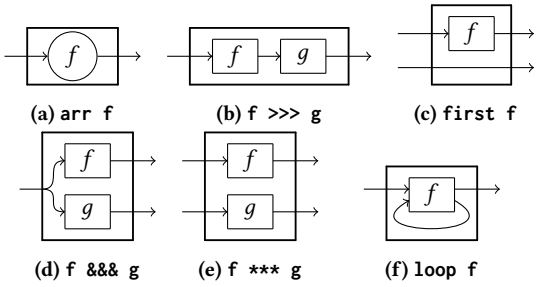
Signal functions are first-class citizens while signals are not.

Signal functions are an example of *arrows* [17], an abstract interface unifying "function-like" types. Programming in Yampa is done by composing primitive signal functions using arrow combinators. Some combinators are, for instance, the *arr* signal function that lifts an ordinary function to a signal functions, serial compositions $\ggg$, two versions of parallel composition $***$ and *&&&*, the latter often called "fan-out" and a feedback combinator *loop*, that instantly feedbacks one output of a signal function to itself:

$$\begin{aligned}
&arr &&:: (a \to b) \to SF\ a\ b\\
&(\ggg) &&:: SF\ a\ b \to SF\ b\ c \to SF\ a\ c\\
&first &&:: SF\ a\ b \to SF\ (a,c)\ (b,c)\\
&(\&\&\&) &&:: SF\ a\ b \to SF\ a\ c \to SF\ a\ (b,c)\\
&(***) &&:: SF\ a\ b \to SF\ c\ d \to SF\ (a,c)\ (b,d)\\
&loop &&:: SF\ (a,c)\ (b,c) \to SF\ a\ b
\end{aligned}$$

The main combinators are illustrated on figure 2 in the form of "boxes and arrows" diagrams, a style Yampa is reminiscent of.

Figure 2: Main arrow combinators



It quickly becomes tedious to program solely using combinators. Fortunately, arrows, such as Yampa signal functions, can be constructed using Paterson's arrow notation [27], also called *proc*-notation. It allows intermediate signals to be named and explicitly give them as arguments to signal functions. Just like the monadic *do*-notation is desugared using the monad combinators, *proc*-notation is desugared using the arrow combinators.

Let us look at an example. Consider the problem of calculating the trajectory of a ball in free fall. Naming the vertical position of the ball y and its velocity v, such a system is described by the following equations:

$$\begin{aligned}
v(t) &= v_0 + \int_0^t -g\ \mathrm{d}\tau\\
y(t) &= y_0 + \int_0^t v(\tau)\ \mathrm{d}\tau
\end{aligned}$$

Yampa provides an *integral* combinator that computes the integral of a signal. The equations above are thus readily translated into the following:

```
freeFall :: (Double, Double) → SF a (Double, Double)
freeFall (y_0, v_0) = proc _ → do
  let g = 9.81
  v ← arr (v_0+) ⋘ integral ⪦ −g
  y ← arr (y_0+) ⋘ integral ⪦ v
  returnA ⪦ (y, v)
```

The **proc** keyword introduces names for the input signal of the signal function, like λ in lambda-abstractions. Then a list of statements instantiate subordinate signal functions, with inputs to the signal functions to the right of ⪦, and their outputs to the left of ←. The last statement's output is the output of the defined signal function. Here we use *returnA*, which is a synonym for the identity arrow.

2.2 Discrete behaviour

Yampa supports dynamic changes in the network structure through a *switching* combinator:

```
switch :: SF a (b, Event c) → (c → SF a b) → SF a b
```

The *Event* type is isomorphic to an option type and is used to model discrete-time signals:

```
data Event c = NoEvent | Event c
```

The output of a switch is the b output from the first signal function, until an event is produced, after which the output is the output from the second signal function, computed from the value carried by the event.

Using *switch*, we can use our *freeFall* signal function to model a bouncing ball. A bouncing ball is a free-falling ball until it reaches the ground (at coordinate $y = 0$). When it reaches the ground, its position remains the same, but its velocity is inverted (assuming bouncing is lossless):

```
bouncing :: (Double, Double) → SF a (Double, Double)
bouncing (y_0, v_0) = switch bounceAux bouncing
  where
    bounceAux = proc _ → do
      (y, v) ← freeFall (y_0, v_0) ⪦ ()
      let bounce = if y > 0 then NoEvent else Event (y, −v)
      returnA ⪦ ((y, v), bounce)
```

3 THE YAMPA IMPLEMENTATION

In this section, we describe a typical Yampa-like FRP implementation as a continuation-based embedding. While the current Yampa implementation [21] is slightly more complicated than the version presented here, it has the same problems we highlight here.

3.1 Implementation principles

Although Yampa is meant for programming hybrid continuous-time and discrete-time systems, it is, of course, fundamentally discrete in its implementation, emulating hybrid features, such as integration, by approximations as we will see.

We define signal functions as:

```
type DTime = Double

data SF a b =
  SF {sfTF :: DTime → a → (b, SF a b)}
```

A signal function computes, from the time passed since its previous invocation (or time 0) and an input, the output and the signal function to execute at the next step. This representation makes it very easy to implement the various combinators discussed in section 2. For example the "fan-out" operator &&&:

```
(&&&) :: SF a b → SF a c → SF a (b, c)
SF tf_1 &&& SF tf_2 = SF {sfTF = tf_3}
  where tf_3 dt a = ((b, c), tf_1' &&& tf_2')
          where (b, tf_1') = tf_1 dt a
                (c, tf_2') = tf_2 dt a
```

Synchrony is maintained by passing the same time difference to both subordinate signal functions of fan-out.

The continuation allows local state to be maintained, as illustrated by the implementation of *integral* using the rectangle rule approximation:

```
integral :: SF Double Double
integral = SF {sfTF = tf 0}
  where tf i dt v = (ni, SF (tf ni))
          where ni = i + dt * v
```

It also allows for dynamic structural changes, exemplified by *switch*:

```
switch :: SF a (b, Event c) → (c → SF a b) → SF a b
switch (SF tf) mknext = SF {sfTF = stf}
  where stf dt a =
    case evt of
      NoEvent → (b, switch sf' mknext)
      Event c → sfTf (mknext c) dt a
    where ((b, evt), sf') = tf dt a
```

Finally, one can make use of Haskell's laziness to implement the *loop* combinator, by simply using the output of the application of the transition function as its input:

```
loop :: SF (a, c) (b, c) → SF a b
loop (SF tf) = SF {sfTF = ltf}
  where ltf dt a = (b, loop tf')
          where ((b, c), tf') = tf dt (a, c)
```

3.2 Performance shortcomings

A natural assumption when writing arrow code is that the "wiring" is essentially free. We refer to wiring, or routing, as the process of guiding a signal from the output of the signal function producing it to the inputs of the consuming signal functions. After all, the way signals flow between signal functions is mostly static, with switch being the prominent exception.

However as can be see from the implementation of &&& above, routing is not free: we have to perform a tuple allocation at each step to pack its result, a tuple that other combinators, like ***, then have to unpack later.

This looks like a small price to pay, especially when allocations are cheap like in most functional language implementations. However this wiring code is often a large part of an FRP network, particularly when using the *proc*-notation. Consider this block:

```
proc x → do
  y ← sf ⪦ x
```

$z \leftarrow sg \prec y$
$returnA \prec (x, z)$

It is desugared by GHC into a slightly more complicated version of the following:

$arr\ (\lambda x \rightarrow (x, x)) \ggg first\ sf \ggg$
$arr\ (\lambda x \rightarrow (x, x)) \ggg first\ (arr\ (\lambda(x, y) \rightarrow y) \ggg sg) \ggg$
$arr\ (\lambda(z, (x, y)) \rightarrow (x, z)) \ggg returnA$

Most of the combinators in the above code handle routing, and all of them allocate and deallocate tuples. In fact, as shown in section 5, the cost associated with routing grows quadratically with the number of lines in a *proc*-notation block.

Surely we can do better? Indeed, the intuition that the wiring is, for the most part, static, is correct. For static routing, there are well-known techniques used in e.g. the implementation of synchronous dataflow languages [3] and continuous system simulation [5] that essentially eliminate the routing overhead by representing signals with shared imperative variables. All that is required is that the writing and reading of each variable, for each time step, is carefully coordinated, which in the setting of (first-order) synchronous dataflow languages and simulation of structurally static continuous systems can be achieved through static scheduling of the computations; i.e., generation of a sequence of imperative assignments in a suitable order.

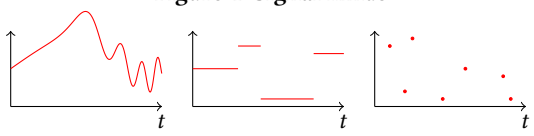
These observations lead us to consider a similarly imperative implementation of FRP networks, where reading and writing of references is mostly scheduled statically, but where the defining flexibility of FRP, such as a dynamic system structure, is retained.

The fact that there is no distinction between tuples used for routing and tuples that are just data in the standard arrows setting becomes problematic when attempting to implement incremental evaluation for avoiding recomputing parts of a network that have not changed, thus improving overall scalability. The reason is that if one component of a tuple changes, then the entire tuple must be considered to have changed, meaning that it often becomes impossible to pinpoint how changes flow through a network. This problem is eliminated if producers and consumers are connected directly through a dedicated channel, such as a shared variable. We have so far not systematically pursued an incremental implementation, but the sketched imperative implementation provides a good setting for such an approach in the future, and, as discussed in the following, the implementation of *arr* is optimized taking change into account.

3.3 Signal kinds

So far, we have considered signals to be (notionally) continuously varying over time. In a hybrid systems setting, there are, as we have seen, other kinds of signals, such as signals defined only at discrete points in time. A discrete-time signal can, of course, be represented by a continuous time signal, and this is, as we have also seen, the approach taken in Yampa. However, there are both conceptual and practical advantages of making a more profound distinction between different kinds of signal. For example, as discrete-time signals can be expected to change relatively infrequently compared to continuous-time signals, exploiting this through a different implementation strategy for such signals can lead to efficiency gains.

Figure 4: Signal kinds



(a) Continuous signals **(b) Step signals** **(c) Event signals**

Indeed, many FRP-implementations make a fundamental distinction between continuous-time signals, commonly called *behaviours*, and discrete-time signals, called *events* [10, 11].

Following the distinctions introduced in [2, 30], we consider three kinds of signals: continuously varying signals, e.g. the position of a physical object; discrete time signals, e.g. information about a collision or a mouse click in a game; and signals that are defined at all points in time, but only changes at discrete points in time. Enforcing the distinction between these signal kinds allows for a more precise and meaningful description of FRP networks and for optimised implementation strategies.

4 SCALABLE FRP IMPLEMENTATION

This section presents an implementation of the above ideas. We first explain how different kinds of signals are distinguished. Then, drawing from compilation techniques for synchronous dataflow languages [3], we show how to compile networks of signal functions to an imperative representation that eliminates most of the routing costs while still allowing for a dynamic system structure. We call this implementation *Scalable FRP*, or SFRP for short.

4.1 Signal representation

We wish to distinguish between three kinds of signals, illustrated on figure 4:

- Continuous signals (C): (notionally) continuously varying time signals (but may exhibit discontinuities). Typically representing a physical quantity such as time, position, etc.
- Step signals (S): piecewise constant continuous-time signals. For instance input from a discrete controller.
- Event signals (E): signals that are present only at discrete points in time.

We capture the distinctions by a type *SD*, for *signal descriptor*:

data $SD\ a$ **where**
$C :: a \rightarrow SD\ a$
$S :: a \rightarrow SD\ a$
$E :: a \rightarrow SD\ a$

SD is not for use at the value level, but at the type level to describe the kind of signals a signal function operates on. In GHC, it is possible to lift a data constructor to the type level [39]. To avoid ambiguities, a lifted data constructor can be prefixed by a quote. Thus, $'C\ Double$ is a descriptor of continuous signals carrying doubles. Much as data constructors are lifted to the type level, type constructors are lifted to the kind (type-of-type) level. The kind of $'C\ Double$ is $SD\ *$, where $*$ is the kind of Haskell types.

For grouping signals we use pairs instead of the vectors used in Sculthorpe and Nilsson [30]. A descriptor denoting a signal that carries no value is also needed to describe the input to signal functions that ignore their input, such as our free falling ball examples:

$P :: SD\ a \to SD\ b \to SD\ (a, b)$
$N :: SD\ Void$

Pairs of signals are now distinct from signals of pairs. For instance $'P\ ('C\ Double)\ ('E\ Bool)$ is a pair of a continuous signal of *Double* and of an event signal carrying a *Bool*. *Void* is the empty type meaning *SD Void* is a signal carrying no values.

These descriptors can now be used in GADT declarations as constraints on the type of data-constructors. For instance, we define the type of values associated to a signal descriptor as:

data *SDVal a* **where**

$$\begin{aligned}
&CVal :: a && \to SDVal\ ('C\ a)\\
&SVal :: (Bool, a) && \to SDVal\ ('S\ a)\\
&EVal :: Maybe\ a && \to SDVal\ ('E\ a)\\
&PVal :: SDVal\ a \to SDVal\ b && \to SDVal\ ('P\ a\ b)\\
&NVal :: && SDVal\ 'N
\end{aligned}$$

As a minor optimization, we represent *SVal* as a tuple of a Boolean and a value, the Boolean indicating whether the signal has changed since the previous step. This allows to avoid some computation when a signal changes rarely, like a step signal is expected to.

For the compilation of the network, we need references to information about the value carried by a signal at a point in time. We thus define a reference type associated to a signal descriptor:

data *SDRef a* **where**

$$\begin{aligned}
&CRef :: IORef\ a && \to SDRef\ ('C\ a)\\
&SRef :: IORef\ (Bool, a) && \to SDRef\ ('S\ a)\\
&ERef :: IORef\ (Maybe\ a) && \to SDRef\ ('E\ a)\\
&PRef :: SDRef\ a \to SDRef\ b && \to SDRef\ ('P\ a\ b)\\
&NRef :: && SDRef\ 'N
\end{aligned}$$

Haskell's *IORef*s are used as the actual reference. Note how the references associated to a pair of signals and the reference associated to a signal of pairs are different: the former, a pair of imperative references; the latter, a reference to a pair.

In the following, for readability, we use $\{.,.\}$ to denote a pair of signal descriptors, $\{\ \}$ to denote a signal carrying no value, and we omit the quote of lifted constructors. Thus $'P\ ('C\ Double)\ ('E\ Bool)$ will be written $\{C\ Double, E\ Bool\}$. We also introduce $readSDRef :: SDRef\ i \to IO\ (SDVal\ i)$ for reading and $writeSDRef :: SDRef\ i \to SDVal\ i \to IO\ ()$ for writing an *SDVal* to/from an *SDRef*.

4.2 Signal function representation

Next, the signal function type is refined to use signal descriptors:

data $SF\ (i :: SD\ p)\ (o :: SD\ q)\ \ldots$

As our goal is to compile signal functions to an imperative representation, we need to represent them in a way that is easy to inspect; i.e., as a deep embedding or abstract syntax tree, except that parts we do not intend to compile can be represented shallowly. We thus introduce constructors for arrow-like combinators:

$$\begin{aligned}
&(:\ggg:) :: SF\ a\ b \to SF\ b\ c \to SF\ a\ c\\
&First\ \ :: SF\ a\ b \to SF\ \{a, d\}\ \{b, d\}\\
&(:\&\&\&:) :: SF\ a\ b \to SF\ a\ c \to SF\ a\ \{b, c\}\\
&(:***:)\ :: SF\ a\ c \to SF\ b\ d \to SF\ \{a, b\}\ \{c, d\}
\end{aligned}$$

Similarly we introduce constructors for a range of primitive signal functions, for example:

$$\begin{aligned}
&Switch\ \ :: SF\ a\ \{E\ c, b\} \to (c \to SF\ a\ b) \to SF\ a\ b\\
&Integral :: SF\ (C\ Double)\ (C\ Double)
\end{aligned}$$

In the following, we will give further examples, including what we term "routers", as we neither can, nor desire to (section 3.2), rely on lifting using *arr* for grouping and ungrouping signals. The set of constructors we provide is complete in the sense that, just as with basic arrows, any network topology can be realised, except that we leave handling feedback as future work (see section 7). In terms of functionality, the set of constructors is also fairly complete in the sense that it covers what is available in Yampa which has proved to suffice in practice, with some exceptions left as future work.

4.3 Interactions between signals of different kinds

Yampa, as discussed in section 3.3, does not make a strict distinction between different kinds of signals. However, enforcing type-level separation between signal kinds is useful for a number of reasons.

As an example of the utility of this separation, consider Yampa's timed delay combinator $delay :: Time \to a \to SF\ a\ a$. This combinator allows any signal to be delayed a specific amount of time. In terms of implementation, it is essentially a buffer for signal samples. However, as Yampa makes no assumptions as to the regularity of sampling, it may happen that individual samples either needs to be discarded or duplicated in order to make up for variations in the sampling frequency. This is fine for signals that conceptually are continuous. However, it is a disaster for signals carrying events, as this can mean that events get lost or are duplicated. Therefore Yampa provides a separate delay combinator for events that works by scheduling a single output event a fixed time into the future. However, because Yampa does not fundamentally differentiate between continuous-time signals and events, there is nothing that stops a user from using the continuous-time delay also for events. In the setting of SFRP this distinction can be enforced, allowing us to provide versions of timed delay with an appropriate implementation for each kind of signal. For this distinction to be meaningful, it must come with restrictions on how to go from one kind of signal to the other.

First, consider *arr*. Clearly, applying a pure function to a signal cannot change its kind. The type of *arr* thus becomes:

$arr :: (a \to b) \to SF\ (k\ a)\ (k\ b)$

Because a and b are types, k must have kind $* \to SD\ *$, thus one of C, S or E, but not a partially applied P. This means that events cannot be created out of thin air using *arr* as the function inside *arr* will only be applied if an event is present. As to S-kinded signals, an optimisation becomes possible. S signals, being expected to change rarely, carry a Boolean flag indicating if they have changed. Hence when we use *arr* on such a signal, we can save work by not recomputing the output if the input didn't change.

As there is now a distinction between pairs of signals and signals of pairs, we need a new combinator *arr2* to apply a pure function to a pair of signals:

$arr2 :: (a \to b \to c) \to SF\ \{k\ a, k\ b\}\ (k\ c)$

While *arr* could be expressed in terms of *arr2*, we prefer not to as that would lose some optimisation opportunities.

We also need primitive operations to mediate between signals of *different kinds* for specific purposes. Here are some examples:

- Tagging the event with the value of the signal at the time it occurs:

  $Tag :: (a \to b \to c) \to SF\ \{C\ a, E\ b\}\ (E\ c)$

- Edge detection for continuous signals:

  $Edge :: s \to (a \to s \to (s, Maybe\ b)) \to SF\ (C\ a)\ (E\ b)$

- Change detection for step-signals.

  $Jump :: s \to (a \to s \to (s, Maybe\ b)) \to SF\ (S\ a)\ (E\ b)$

- Accumulation:

  $Accum :: s \to (a \to s \to (s, b)) \to SF\ (E\ a)\ (S\ b)$

- Removal of events from an event signal:

  $FilterE :: (a \to Maybe\ b) \to SF\ (E\ a)\ (E\ b)$

- Event merging:

  $MergeE :: (a \to a \to a) \to SF\ \{E\ a, E\ a\}\ (E\ a)$

Events act as mediators between step and continuous signals, akin to what is being done in hybrid simulation languages [2]. The combinators we obtain often have equivalences in Yampa.

4.4 Routers

Suppose we wish to compose two signal functions of type:

$sf :: SF\ k\ \{\{E\ a, C\ b\}, S\ c\}$
$sg :: SF\ \ \{C\ b, \{E\ a, S\ c\}\}\ k'$

Surely *sf* and *sg* should be composable, provided we can rearrange the output signal of *sf* to match the way the signals are paired in *sg*. In Yampa, this would be done by interleaving an *arr* rearranging the signals:

$sf \ggg arr\ (\lambda((ea, cb), sc) \to (cb, (ea, sc))) \ggg sg$

However, in our case, the arrow combinators we have introduced at this point are not able to express this. Indeed, with the limitations we introduced on *arr* and *arr2*, it is not possible to use these to perform that task. Even if it had been, doing so would have lost us all the benefits of using our new representation, since we would have to read the inputs and write the outputs at each iteration and pay the cost of wiring again.

Instead, we need a way to describe transformations between signal "shapes": signal functions that only act on the way single signals are paired, but not on the signals themselves. This way, it will be possible to route signals by simply reorganizing the references representing the signal, which can be done once.

Notice that signal descriptors have the shape of binary trees: *P* being the nodes and *N*, *C*, *S* and *E* being the leaves. This means that the set of transformation we need is the set of transformation to match the shape of any binary tree into any other, provided all the non-*N* leaves of the destination tree are present in the original tree. We will show in section 4.6 that a sufficient set of transformations to do so are left- and right-rotations, to modify the shape of the tree without changing the order of the leaves; duplication and deletion of a tree, to remove unused leaves, or add extra ones; and swapping two children of a node, to change the order of the leaves in the tree. In addition, we also need combinators to compose routers together or to apply them to a specific subtree on a node. This points to a type defined in a way similar to *SF*:

data *Router i o* **where**
  IdRout :: *Router a a*
    -- Tree rotations
  LeftRot :: *Router* $\{a, \{b, c\}\}$ $\{\{a, b\}, c\}$
  RightRot :: *Router* $\{\{a, b\}, c\}$ $\{a, \{b, c\}\}$
    -- Subtree deletions
  DelRout :: *Router i N*
  FstRout :: *Router* $\{a, b\}$ *a*
  SndRout :: *Router* $\{a, b\}$ *b*
    -- Leaves duplications
  DupRout :: *Router a* $\{a, a\}$
    -- Swapping leaves
  SwapRout :: *Router* $\{a, b\}$ $\{b, a\}$
    -- Combining routers
  AppLeft :: *Router a b* $\to$ *Router* $\{a, c\}$ $\{b, c\}$
  AppRight :: *Router c d* $\to$ *Router* $\{a, c\}$ $\{a, d\}$
  CompRout :: *Router a b* $\to$ *Router b c* $\to$ *Router a c*

From a *Router*, it is possible to define a function *route* that will rearrange the shape of references accordingly. In fact, the types are precise enough that there is only one possible implementation for the function in each cases:

```
route :: Router i o → SDRef i → SDRef o
route IdRout ir = ir
route DelRout _ = NRef
route DupRout ir = PRef ir ir
route LeftRot (PRef a (PRef b c)) = PRef (PRef a b) c
...
```

A router can then be lifted into a signal function with a dedicated constructor:

$Router :: Router\ i\ o \to SF\ i\ o$

The composition of *sf* and *sg* above can now be written as:

$sf \ggg Router\ (CompRout\ (AppLeft\ SwapRout)\ RightRot) \ggg sg$

We do not expect users to use routers directly, except maybe for simple ones; much like we do not observe in Yampa code uses of complex calls to *arr* in the code user write. However it is crucial to implement an efficient desugaring for the *proc*-notation.

4.5 Compilation

We now describe a translation from a description of a network to an executable machine. In the process, we will see how the same features of GHC's type system we have used to enable more precise specification of FRP networks are useful to make the compilation easier.

This section is organized as to guide the reader from an implementation of a simple, static FRP to implementations of growing complexity as we progress to support more complicated structures.

The simplest idea is to have a compilation function of the following type:

```
compile :: SF i o → SDRef i → IO (SDRef o, IO ())
```

The *compile* function takes an input reference, and produces both an output reference and a *stepping action*. When executed, the stepping action advances the state of the network by one step and updates the output reference accordingly. Executing it many times advances the network by that many steps. One can quickly see that this representation has the advantages we were looking for in terms of routing. When we compile a router, we can simply use the *route* function to transform the input reference into the output reference, while the updating action does nothing. Using $pure :: a \to IO\ a$, that lifts a pure value into an effectful context, not performing side effects, we have:

```
compile (Router rr) ir = pure (route rr ir, pure ())
```

Most combinators can also be neatly expressed this way, for instance serial composition:

```
compile (sf :>>>: sg) ir = do
  (orf, stprSF) ← compile sf ir
  (or, stprSG) ← compile sg orf
  pure (or, stprSF ≫ stprSG)
```

$\gg$ being the sequence operator for two *IO* actions. Notice how the output reference of *sf* can be reused as the input reference of *sg*, as one would expect.

Similarly, the parallel composition also benefits from the more precise representation:

```
compile (sf :***: sg) (PRef ir_1 ir_2) = do
  (or_1, stpr_1) ← compile sf ir_1
  (or_2, stpr_2) ← compile sg ir_2
  pure (PRef or_1 or_2, stpr_1 ≫ stpr_2)
```

The dispatching of the input and output references to both subsignal functions can be done at the compilation step, and there is no need to repeat the process at each iteration.

4.5.1 The difficult case of switching. While it is possible to implement switching using a function like above, it would lead to space- and time-leaks in lots of cases. In essence, this could be done by having extra levels of indirection. The switch's stepping action would store the current signal function stepping action and output reference in an imperative reference. It would execute that stepping action at each step and then update its output reference based on the signal function's output reference. Upon switching, it would simply overwrite the imperative reference with the new output reference and the new stepping action. This leads to the following implementation:

```
compile (Switch sf next) ir = do
  (PRef or (ERef evtRef), stprSF) ← compile sf ir
  let swStpr stprRef outRef = do
    stprSF
    evt ← readIORef evtRef
    case evt of
      Nothing → pure ()
      Just c → do
        (newOr, newStpr) ← compile (next c) ir
        newStpr
        writeIORef stprRef newStpr
        writeIORef outRef newOr
  outRef ← newIORef or
  rec stprRef ← newIORef (swStpr stprRef outRef)
  let switchStpr = do
    stpr ← readIORef stprRef
    () ← stpr
    or ← readIORef outRef
    readSDRef or >>= writeSDRef switchOr
  pure (switchOr, switchStpr)
```

While relatively simple, this approach has a subtle problem. As we saw in section 2, a common pattern in Yampa is to have switches switching back on themselves. In that case, the outermost switch would have to traverse an additional indirection to access its output value, which is stored in the output reference of the innermost switch, and the same traversal to access its stepping action.

There are two problems that we need to fix: how to handle the fact that the stepping action of a switch changes during execution, and how to handle the change of output reference.

Handling changing output references. We considered two fixes. The first one is to introduce a notion of variable references. Instead of using *IORef*s directly in *SDRef*, we could use a type like so:

```
data Ref a = FixRef (IORef a)
           | VarRef (IORef Bool) (IORef (Ref a))
```

A reference is either a fixed reference or a variable reference, pointing on another reference. In the latter case, we also have a reference on a Boolean flag that, when *True*, indicates that the variable reference will always point to the same *Ref*: it has become non-variable. This means that it has become an unnecessary indirection and can be "skipped": another variable reference pointing to it can simply point at the next one.

We preferred to follow another route that did not require to change the type of references. Indeed the problem can be solved by simply passing both output and input references to the compilation function. Then *switch* can pass the same output reference to the first and then the next signal function. Since it is not practical to do so in general, as it neutralizes the benefits of making routing explicit, we opted for a compromise solution of optionally passing the output to the compilation function that will optionally produce the output reference. The following type would suffice:

```
SDRef i → Maybe (SDRef o) → IO (Maybe (SDRef o), IO ())
```

However, we would prefer to encode the relation that when one of the *Maybe* (*SDRef o*) is *Nothing*, then the other must be *Just*. This constraint can be encoded in GHC's type system, with the same tools that we used to constrain signals. We introduce a tagged-optional type that witnesses at the type-level which constructor it is made of:

```
data Opt (b :: Bool) a where
  None ::       Opt False a
  Some :: a → Opt True a
```

Type families [6, 7] are type-level functions that are treated as synonyms by the GHC's type-checker. We can then define a type-level *Not* family:

```
type family Not (b :: Bool) :: Bool where
  Not False = True
  Not True  = False
```

and use it to encode the desired invariant:

```
compile :: ∀b.SDRef i
        → Opt b (SDRef o)
        → IO (Opt (Not b) (SDRef o), IO ())
```

Since we quantify on the presence or absence of an output reference, *compile* must work in both cases. Consider the implementation of the compilation of a router with this type. In the case where no output reference is passed, then we can simply perform the routing of references like we did previously. However, in the case where an output reference is given, it will be necessary to synchronize both references at runtime:

```
compile (Router rr) ir None =
  pure (Some (route rr ir), pure ())
compile (Router rr) ir (Some or) =
  pure (None, readSDRef (route rr ir) >>= writeSDRef or)
```

One could argue that a simpler solution would have been to have two *compile* functions: one where the output reference is provided and one where it is not. However, having a single function is advantageous as in many cases the code for the *compile* function is identical in both cases. The present approach avoids duplicating this code. For instance, serial composition can be compiled without knowing if it has been passed an output reference:

```
compile (sf :>>>: sg) ir or = do
  (Some orf, stprF) ← compile sf ir None
  (nor, stprG) ← compile sg orf or
  pure (nor, stprF >> stprG)
```

Note how we can reuse the output reference of the first signal function as the input to the second in this new setting.

Handling changing stepping actions. We propose solving the problem of the variable stepping action by extending the type of actions to be able to return a new stepper to execute at the next time step:

```
data Stepper = FStpr (IO ())
             | VStpr (IO Stepper)
```

We define a few functions to execute steppers, also returning the stepper to execute at the next step:

```
execStepper :: Stepper → IO Stepper
execStepper (FStpr stpr)  = stpr >> pure (FStpr stpr)
execStepper (VStpr vstpr) = vstpr
```

and to sequence two steppers:

```
seqStepper :: Stepper → Stepper → Stepper
seqStepper (FStpr s_1) (FStpr s_2) = FStpr (s_1 >> s_2)
seqStepper (VStpr vs_1) (FStpr s_2) =
  VStpr $ do
  s_1 ← vs_1
  ()  ← s_2
  pure (seqStepper s_1 (FStpr s_2))
...
```

Note how, in the case of a variable stepper, the recursive call to *seqStepper* allows the simplification of the stepper as parts of the network evolve over time. Suppose we sequence *VStpr vs_1* and *FStpr s_2*, like above, and that the execution of *vs1* yields a fixed stepper, *FStpr s_1′*. Then the variable stepper resulting from the sequencing will also yield a fixed stepper, *FStpr* $(s\_1' \gg s\_2)$.

With this new machinery, we can implement switching in a much simpler way:

```
compile (Switch sf mknext) ir or = do
  (Some (PRef (ERef evtRef) orf), stprSF) ← compile sf ir None
  let switchStpr stpr =
    VStpr $ do
    stpr ← execStepper stpr
    evt ← readIORef evtRef
    case evt of
      Nothing → pure (VStpr (switchStpr stpr))
      Just c → do
        (None, stprNext) ← compile (mknext c) ir (Some orf)
        stprNext ← execStepper stprNext
        pure stprNext
  case or of
    None → pure (Some orf, switchStpr stprSF)
    Some or → do
      let updateRef = readSDRef orf >>= writeSDRef or
          stpr = seqStepper (switchStpr stprSF) (FStpr updateRef)
      pure (None, stpr)
```

On the absence of loop. As we saw in 3.1, the implementation of feedback in Yampa exploits laziness to dynamically schedule the computation of values such that, if there is no cycle in the feedback, a result can be computed. Implementing such a combinator in a setting like SFRP is a lot more complicated because the order of reads and writes to imperative references is fixed during the compilation of the network which means we cannot make use of the dynamic scheduling provided by lazy evaluation.

For now, a *loop* combinator in SFRP remains future work. So far, we have considered two approaches. The first is to analyse the dependency graph and statically schedule the reads and writes in a correct order, like is done in most synchronous language implementations. While this would require major changes to the library, it is more principled and flexible solution. The other approach, in the vein of [38] is to only implement delayed feedback, meaning that the input to the feedback is the output of the previous step. This would not require any change to the implementation. However, we find it unsatisfactory to tie the notion of delay and feedback together. First, while there has to be decoupling somewhere in a feedback loop to make the feedback well-defined, this decoupling need not necessarily be along the designated back edge. Second,

there are many possible ways to decouple (various kinds of delays, integrals), and having to introduce one loop combinator for each kind of decoupling would be unfortunate.

Further optimizations. Our implementation makes some additional optimizations to the compilation. For instance, we have a *Done* stepper, that denotes a stepper that does nothing. It enables further simplification by *seqStepper*, which is useful for code that uses routers a lot, since these will simply disappear rather than remaining in the form of *pure* () actions otherwise. We can also perform some optimizations by retaining the information that a signal function is a router in the compiled form, to simplify parallel composition for example. Thus the type of the output of the compilation is either a compilation function or a router.

4.6 Custom *proc*-notation implementation

The *SF* type is not a standard arrow which means that we cannot make use of the *proc*-notation as supported by GHC. The Rebindable Syntax extension [34] is not enough to get around that problem. This extension allows to use GHC's desugarers with custom arrow combinators, rather than the standard ones. However, our *arr* combinator is too restrictive to be compatible with the way GHC operates, since it is used for routing, while we would need to use the routing combinator introduced in section 4.4.

Fortunately, GHC supports extensions to its syntax through quasi-quoting [20]. Thus, we can define our own *proc*-notation that we can desugar by our own means.

Our notation is as close as possible to GHC's, in an attempt to make porting existing Yampa programs (and users) easier. In our setting, the falling ball of example of section 2 is written like so:

```
freeFall :: (Double, Double)
         → SF a [sd|{C Double, C Double}|]
freeFall (y_0, v_0) =
  [sf|proc i → do
    v ← arr (v_0+) ⋘ integral ⪦ {|− 9.81|}
    y ← arr (y_0+) ⋘ integral ⪦ v
    returnA ⪦ {y, v}|]
```

The syntax $[sf|\cdot|]$ introduces a quasi-quote, instructing GHC to translate the string between the bracket using a quasi-quoter named *sf*. Inside the quasi-quote, most things can be read as normal *proc*-notation code. Since we insist on distinguishing signal kinds, pairs of signals are denoted using $\{\cdot,\cdot\}$. When introducing a signal name, it is possible to also indicate its kind with the syntax $x :: C$, here indicating that x is of kind C. It can be used to improve clarity or to give information to GHC's typechecker. Although we have never found a case where annotations where necessary for GHC to infer a signal's kind, it can at least help to generate better error messages. Using Haskell expressions as input to signal functions is supported, but they must be enclosed in $\{|\cdot|\}$. They may refer to signals but all signals being referred to must be of the same single kind, since we do not wish to mix signals of different kinds implicitly. Pattern matching on Haskell constructors is not supported, however. There is also an *sd* quasi-quoter for typesetting signal descriptors in a more convenient way.

Let's go back to the example we looked at in section 3. The following:

```
proc x → do
  y ← sf ⪦ x
  z ← sg ⪦ y
  returnA ⪦ (x, z)
```

is desugared by GHC to:

```
arr (λx → (x, x)) ⋙ first sf ⋙
arr (λx → (x, x)) ⋙ first (arr (λ(x, y) → y) ⋙ sg) ⋙
arr (λ(z, (x, y)) → (x, z)) ⋙ returnA
```

The general form for desugaring a statement is:

```
arr (λx → (x, x)) ⋙ first (<glue> ⋙ sf)
```

First, the inputs are duplicated, as one must conserve the current inputs so that later statements can use them. For instance, x is used by both *sf* and *returnA*. One of the duplicated input is then fed into *sf*. However some "glue" must filter and rearrange the inputs so that *sf* can use them.

Our idea is to use a similar scheme for desugaring, but using routers instead of *arr* to maintain the efficiency of our setting. Readily, the duplication can be done by using the *Router Dup* signal function. However, the problem of computing the glue remains. As we mentioned when we introduced routers in section 4.4, routers are transformations of binary trees. Hence the glue is simply the transformation between the tree representing all the inputs at this point in the network and the tree of inputs expected by the signal function. We present an algorithm to compute this transformation.

The first step consists in computing the transformation that deletes unused leaves from the initial tree, and duplicates the leaves that appear more than once. That is done simply using the duplicating and deleting routers. For instance, when desugaring the statement where *sg* appears in the above, the current set of inputs is $\{x, y\}$, but since only y is necessary, we use *Router SndRout* to obtain the right number of leaves in the tree.

After this transformation, the tree has the correct leaves in the correct number, however it may not be in the right shape. Since it is a well-known problem to match the shape of two binary trees when the leaves are in the same order (typically, in a binary search tree) using only tree-rotations [31], we will first sort the leaves of the tree, in the order the signal function expects them in, and then shape it correctly.

Computing the sorting transformation is fairly complicated. Indeed, we do not know of a general way to compute the transformation that exchanges two leaves of a tree. We know however that it is simple in one case: when the tree is "list shaped", i.e. when for every node, the left branch points to a leaf, and when the leaves we wish to swap are consecutive, the transformation is simple: a right-rotation, followed by swapping on the node created on the left, followed by a right transformation. This is illustrated on figure 6. This makes computing the transformation to sort this tree feasible by bubble-sort. The way we match the shape of the tree is then as follows: we first compute a transformation from the current tree to a list-shaped tree, then the transformation to bubble sort the list-shaped tree and finally the transformation to the shape the signal function excepts.

The transformations obtained from each step are then composed together in one router that makes the glue.

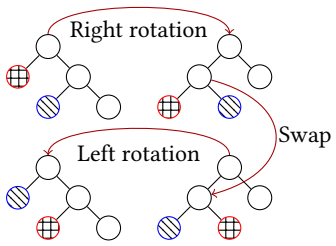


Figure 6: Swapping two leaves in a list-shaped tree

We do not expect this algorithm to be optimal in any way. In practice, however, one should recall that this transformation runs at compile-time inside GHC, where we can expect programs to be fairly small. In our cases, for networks of less than around 100 lines (more than what we expect to encounter), we have not noted significantly different compilation times from the ones of the equivalent Yampa programs. In terms of the optimality of the router being generated, we observe that it produces fairly large routers. However a simple simplifying pass, doing obvious optimizations, like deleting composition of identity routers, is already enough to obtain a much cleaner result.

5 EVALUATION

In this section, we evaluate our implementation both in terms of ease of use compared to Yampa and in terms of performance, to verify that our scalability objective is attained. Our implementation can be found on Gitlab[2].

5.1 Ease of writing

In order to evaluate how different SFRP and Yampa are from a user perspective, we modified a version of the game Flappy bird written using Yampa [19] to use SFRP[3] (see a screenshot on figure 1).

For a large part of the code, the port was as straightforward as the port of the falling ball example from section 4.6, mostly a matter of surrounding the code written in *proc*-notation by quasi-quote brackets and modify the few parts where the syntax is different. The only difficult point was handling inputs to the network. The original game used, as is common, a record of several fields, essentially containing information coming from the Graphical User Interface (GUI) such as the mouse position, mouse clicks, etc. In our setting, this record is a collection of heterogeneous signals: the mouse position is a continuous signal while the mouse clicks are events. However the only collection of signals SFRP supports is nested-pairs. The choice was then between getting rid of the input record in favour of nested pairs of signals, which is inelegant and tedious to write or read; or slightly "lying" and keeping the input record as a continuous signal (even if its components are not all continuous signals) and writing accessor signal functions that convert each field into the relevant signal with the right kind. We decided to go for the latter solution.

Flappy bird, being a light game, did not directly benefit from an increase in performance from SFRP, since the GUI was, by far, the bottleneck. Turning off the GUI, we were able to measure that, although the Yampa version was already running at a respectable 30 000 iterations per second, the SFRP version was running at 90 000 iterations per second, thus a three-fold improvement. This particular measurement was done on a PC with a 4-core Intel i3-7100T @ 3.40 GHz with 8 GB of memory.

5.2 Precise performance measurements

To test the relative performance of SFRP and Yampa in a more meaningful and systematic way, we auto-generated networks of various size and characteristics. These networks, in their Yampa and SFRP versions, were then benchmarked using the Criterion library[4].

5.2.1 Random network generation. The networks were generated in *proc*-notation. Doing so is much easier than to generate networks in "combinator form" and allows to measure the hidden impact of wiring over the network in a clearer way.

The generated network has one input and one output, both continuous signals of doubles as are all intermediate signals in the network. Using continuous signals means that their representation is not optimized in any way, unlike step signals, meaning that any improvement over Yampa can be attributed to the difference in routing. We make sure that every signal is used at least once in the computation of the output, as to not have parts of the network removed. The network is made of three basic blocks: integrals, with one input and one output; sums and negations of two input signals in one output and switches, with one input and one output and a subnetwork.

The network is generated with a target size. Each block counts for 1 except for switches, that count for the size of their subnetwork.

We control the proportion of switches as well as the average number of iterations at which a switch should occur. We make sure that not all switches switch in the same iteration by adding a bit of random noise to the number of iteration a switch should switch after (if n is that number, the noise is chosen between $\pm \min\left(10, \frac{n}{2}\right)$). When a switch is generated, a subnetwork is generated with a size randomly chosen between 1 and the target size of the network it is in. The switch switches back on itself when an event occur. This is both in order to model a very common pattern in this kind of programs and to prevent switches from disappearing entirely from the network.

Once a network has been generated, the time taken for it to run 100 000 iterations is benchmarked, in SFRP and Yampa form. We used a PC with a 8-core Intel Xeon W-2123 @ 3.60 GHz and 32 GB of memory[5]. Results are presented in figure 7. The graphs show the average time a network of a given size has taken to execute 100 000 iterations for each library. The average is taken over 1000 trials. Each trial attempts to average out measurement artifacts, such as the computer's clock precision, potentially by running the program to benchmark several times [24]. We plot on the same graph the ratio of the speedup of SFRP network over the

[2] https://gitlab.com/chupin/scalable-frp
[3] The SFRP version can be found at https://gitlab.com/chupin/flappy-haskell/tree/scalable
[4] http://hackage.haskell.org/package/criterion
[5] This amount of memory was not necessary to run the benchmarks, but to compile them, as it seems GHC is fairly memory hungry when it encounters programs with large literals, as is the case with our auto-generated networks.

Yampa network, as the ratio of the two average running times. The general observation is that SFRP is significantly faster in all cases, except in one, rarely encountered in practice, where it is on par with Yampa.

5.2.2 Performance of static networks. The static benchmarks show that networks are consistently faster when using SFRP, by an order of magnitude for networks of size larger than 10.

Observing the benchmarks when there are no switches in the network confirms the fact that wiring is a major cost in Yampa compared to SFRP. We observe that Yampa's running time is quadratic in the size of the network, while SFRP's is linear. This can be explained by the following observation: as the network grows, the number of combinators introduced for the purpose of wiring grows linearly, but the complexity of these routers also grows linearly with the size of the network as each of them has to route a linearly growing number of signals. This observation was confirmed when we focus on the cost of switch for SFRP as we will show that it exhibits the same behaviour when it has to switch on large networks.

5.2.3 Evaluation of performance with the number of switches. Figure 7b and 7c show two of the benchmarks for networks that were generated with respectively 10% and 25% of each line being a switch and then run by triggering each switch every 50 iterations on average. These benchmarks, by their very nature cannot be used to precisely predict a trend. Indeed, we only control the proportion of switches in a network but leave the size of the subnetwork to be picked randomly. However, we consistently see that, as the number of switches increases, the gain from using SFRP shrinks. Not only because SFRP becomes slower, but also because Yampa becomes faster.

The reason for this gain is quite subtle, and has to do with the fact that, when a network contains a switch, its structure is simpler with regard to routing. This is eminently beneficial to Yampa but not particularly to SFRP. Consider the case of a network of size n, without switches. To produce its output, it must wire n signals together. Consider a network of the same size, but made of two switch blocks, each of size $\frac{n}{2}$. Since each switch is made to take one input and produce one output, to produce their output, both subnetworks must wire $\frac{n}{2}$ signals, and then the two output signals must be wired together. Since the cost of routing is quadratic in the size of the network, it is more efficient to do the latter that the former, as witnessed by the benchmark results.

This is also why Yampa exhibits somewhat better behaviour in practice than in the first benchmarks, since most networks are made in a modular fashion, from small networks linked together, the overall cost of routing is reduced. However, the performance advantage of using SFRP is always significant. It can only increase as the network grows in size, remaining constant at worse.

5.2.4 Evaluation of the cost of switching. To get a more meaningful idea of the cost of switch, we slightly modified our network generator. This time, we generated purely switchless networks that were then enclosed in a switch, switching at a certain rate. By that we mean that, if the generated network is n, the network we benchmark is of the form $n' = switch\ (r \&\&\& n)\ (\lambda\_ \to n')$, where r is the signal function generating the event for the switch to occur. Some results are shown in figure 9.

We observe that when a switch occurs at every iteration (figure 9c), SFRP is as slow as Yampa and that it presents the same quadratic behaviour in the size of the network that we observed for Yampa. This confirms that routing is the cause of the quadratic behaviour: in this case, SFRP compiles a new network at every iteration, having to route references at every iteration, much like Yampa has to route signals at every iterations. Fortunately, networks written this way are rarely seen in practice, and often only with small subnetworks.

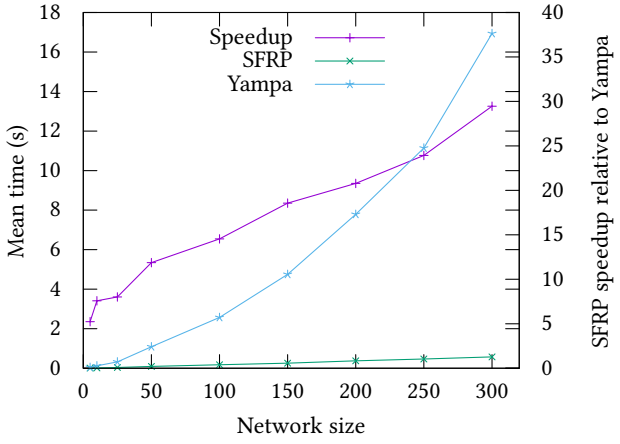
6 RELATED WORK

Push-pull FRP. Elliott, in his 2009 paper [12], revisits FRP. He provides a modernised monadic and applicative interface, but the main motivation is to resolve the tension between supporting continuously changing values efficiently, which calls for a pull-based implementation, and events, which calls for a push-based implementation. The answer is to combine both, and the paper achieves this through an elegant derivation from a denotational semantics. However, the basic behaviours[6] are represented by piece-wise constant behaviours (like our step signals) with overlaid *known* functions, accounting for the behaviour between the points of discrete changes. For example, the time behaviour is in essence a single step with an overlaid identity function. This means that the basic behaviours cannot account for the case where the behaviour is not predictable until the next discrete change, such as is the case with integration (in general). A separate interface is provided for this kind of behaviour, but the paper does not give many details and it is thus difficult to comment on the efficiency. It is also unclear to what extent feedback is supported as the paper does not provide the required primitives, neither for behaviours nor for events. As to the implementation, it relies on an "unambiguous choice" operator that works by spawning threads, using *unsafePerformIO* to provide a pure interface, and then picking the first available result. The specific setup is such that this indeed is safe and does not violate referential transparency. The implementation is thus very different from that of SFRP, and its ultimate efficiency hinges on how well the runtime can handle lots of light-weight threads and how efficiently the sophisticated machinery required to keep everything pure from a user perspective can be compiled. The paper does not provide any performance evaluation. In contrast, the code that ultimately is executed when SFRP programs are run is in essence just conventional, single-threaded imperative code.

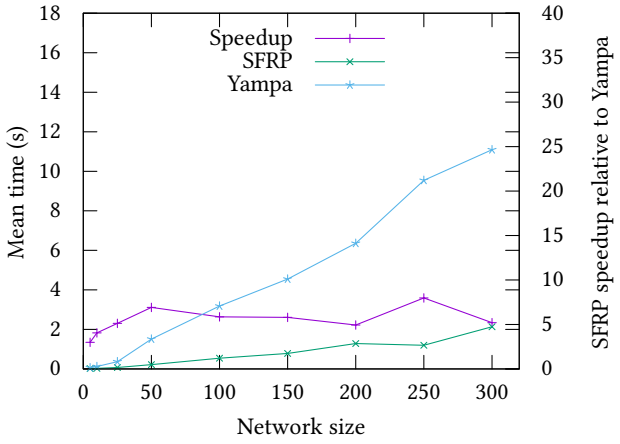
Stream-based FRP. Patai [25, 26] proposes a representation of reactive programs as infinite streams of values, where streams are constructed using stream generators in a way that make them suitable to be represented as an stepping action in the *IO* monad, the elements of the stream being produced by repeatedly performing the action. The implementation retains a high degree of dynamicity, as streams can be higher-order and recursively defined, to account for feedback for instance. By nature, there is no notion of continuous time in this implementation. Behind the scenes, the implementation is imperative. A stream network is constructed, where streams

[6] In some implementations, continuous-time signals are called *behaviours*. We keep this terminology if it is used in the work being discussed.

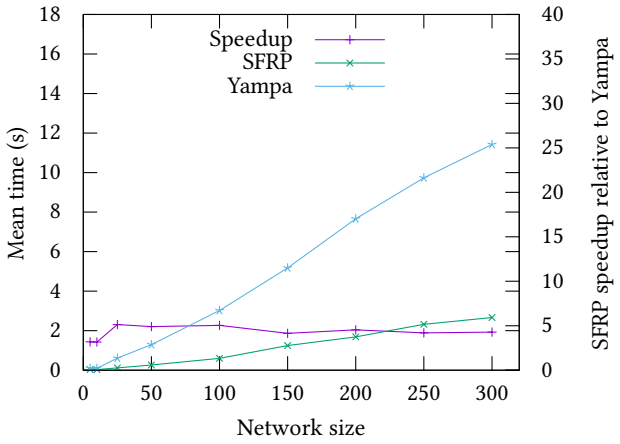
Figure 7: Benchmark results for arbitrarily generated networks. Average runtime and speedup over 100 000 iterations



(a) No switches

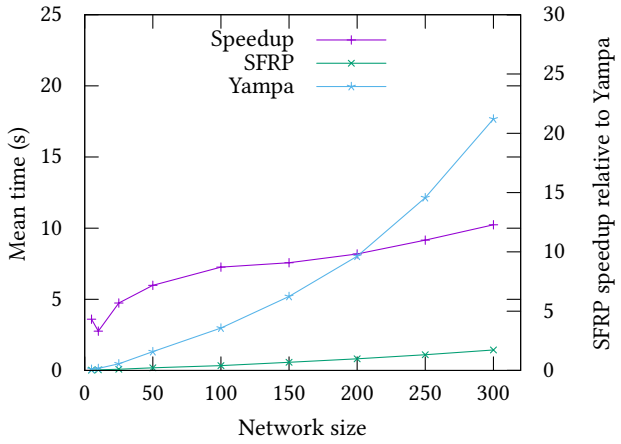


(b) Networks where 10% of nodes are switches, switching every 50 iterations

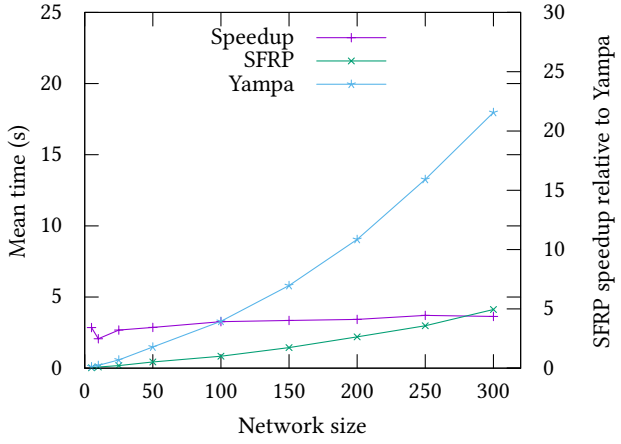


(c) Networks where 25% of nodes are switches, switching every 50 iterations

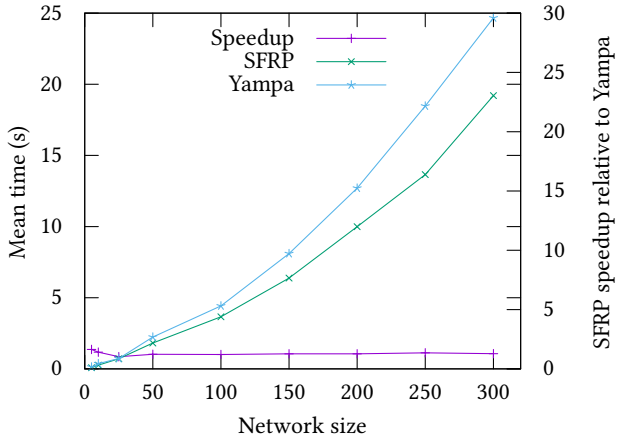
Figure 9: Benchmark results for networks with a single switch enclosing an otherwise switchless network. Average runtime and speedup over 100 000 iterations



(a) Switching every 50 iterations



(b) Switching every 10 iterations



(c) Switching every iteration

may depend on others in arbitrary ways. Carefully coordinated sampling and update is then carried out, with each stream being represented by an imperative variable that indicates that the stream either is being ready to be sampled, or that it has been sampled, along with its current value, to ensure the results of computations are shared and break cycles. The traversal of this graph at each time step thus amounts to dynamic scheduling of the computations, unlike in Scalable FRP (SFRP) where sequential code for reading and writing the imperative signal variables in an appropriate order is constructed once and for all.

Ultrametric FRP. Krishnaswami and Benton [18] propose a denotational semantics for reactive programs represented as programs over streams, like in Patai's work. Also like Patai's work, time is discrete. The authors propose a language abiding to this semantics and an efficient translation to a low-level imperative dataflow graph that preserves the high-level semantics. The dataflow graph is a network of imperative references containing thunks over streams. The evaluation of these thunks is dynamically scheduled as the values of the head of each stream is required. Upon evaluation, each references is updated to point to a new thunk to the tail of the stream. The flexibility of the stream representation means that dynamic higher-order recursive FRP programs can be expressed in this setting. However, unlike Patai's work, the well-foundedness of recursively defined streams as well as the causality is guaranteed.

FRPNow! Van der Ploeg and Claessen [36] propose a new monadic FRP interface, close to the original FRP interface, that by careful construction rules out space leaks and allows monadic *IO* actions to be performed directly from FRP code. They give a denotational semantics and derive an implementation from this, proving that the stated freeness from leaks indeed hold. While the interface provides both behaviours and events, the value of a behaviour can only change at discrete points in time, by switching, and the only assumption regarding time is that there is a total order between time points. Behaviours in this system are thus akin to our step signals: there is no direct support for continuous signals in the SFRP sense. Feedback is limited to behaviours. The implementation is effectful, using Haskell's *IO* monad. Imperative variables are used to share the latest version of events and behaviours, with *unsafePerformIO* used to ensure that those variables are properly shared even though the provided interface is pure, not mentioning the *IO* context. *IO* actions generating primitive events are spawned using threads, and the resulting events are batched into so called rounds to give the illusion of *IO* actions not taking any time. The system keeps track of what computations are ready to run in response to primitive or derived events, ensuring they are run as soon as possible but at most once. Also, computations of which the results are no longer needed and that have no observable side effects are also removed. The scheduling of computations is thus done dynamically, and the overhead for frequently changing signals is consequently substantial. Again, we note that this is different from SFRP where the scheduling is done statically.

Causal commutative arrows. Causal Commutative Arrows (CCAs) [38] are a particular class of arrows that can be put into causal commutative normal form, meaning either a pure function or a single loop containing one pure function and one initial state value. Use of causal commutative arrows has been shown to lead to great improvements in performance over the continuation-based representation presented in section 3, in part by using an imperative representation. In this representation, the actions modifications to the internal state are scheduled statically, however it does not tackle the problem of wiring in the way we do, which remains fully dynamic. Note that, although feedback is supported, it always come with an implicit delay. Unlike SFRP or Yampa, CCAs do not support dynamic change to the network structure.

Reactive Banana. Reactive Banana [1] is a first-class FRP library, oriented around programming an event network between signals, which is then compiled into a stepping action. Although Reactive Banana supports notionally continuous behaviours, the implementation is completely event driven, meaning the network only steps forward when an event occurs. Reactive Banana networks can be dynamic in a way similar to other implementations, by having events carrying new behaviours to switch to.

Netwire and Wires. Netwire, and its successor Wires [32, 33], are libraries very similar to Yampa, where signal functions support monadic effects and errors. To the best of our knowledge, no particular optimization effort is made compared to Yampa. However, given the similarities between these libraries, we believe that the techniques explored in this paper could be applicable to Netwire and Wires.

7 FUTURE WORK AND CONCLUSIONS

In this paper, we showed how the performance of arrowized FRP programs could be greatly improved by making use of an imperative representation, similar to how synchronous dataflow languages are compiled, and helped by a precise type-level description of a network. We presented quantitative evidence of this improvement. We also showed that our library is mature enough to, in many cases, be used as a (mostly) drop-in replacement for Yampa.

The current implementation of SFRP is however lacking in some respects compared to Yampa. In particular, SFRP does not yet support collection-based switches and "freezing" of running signal functions [23].

Also, the arrow *loop* combinator for feedback is not yet supported. The problem here is that *loop* calls for an instantaneous feedback edge which makes static scheduling more difficult. In principle this can be solved by analysing the dataflow graph: there must be some decoupling somewhere in a feedback loop to make it well-defined, but the decoupling need not necessarily be along the back edge itself. Alternatively, if that is too complicated, we could resort to a pragmatical solution where feedback is always decoupled e.g. though a delay element. This solution is actually quite common, even if it fuses distinct concepts (delay and a back edge). It is also worth noting that many of the systems considered in the section on related work also do not support instantaneous feedback edges for various reasons.

Finally, to further improve the scalability, it would be interesting to explore systematic incremental evaluation. As discussed briefly in section 3.2, the SFRP implementation does have an appropriate structure thanks to direct communication between producers and consumers through shared variables.

REFERENCES

[1] Heinrich Apfelmus. Reactive-Banana: Library for Functional Reactive Programming (FRP). https://hackage.haskell.org/package/reactive-banana. (????).

[2] Albert Benveniste, Timothy Bourke, Benoit Caillaud, Bruno Pagano, and Marc Pouzet. 2017. A Type-Based Analysis of Causality Loops in Hybrid Systems Modelers. *Journal of Nonlinear Analysis Hybrid Systems* (2017).

[3] Dariusz Biernacki, Jean-Louis Colaco, Marc Pouzet, and Grégoire Hamon. 2008. Clock-Directed Modular Code Generation for Synchronous Data-flow Languages. In *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. Tucson, Arizona, 10.

[4] Daniel Bünzli. React, Functional Reactive Programming for OCaml. https://erratique.ch/talks/react-ocamlum-2010.pdf. (????). Accessed: 2019-05-09.

[5] Francois E. Cellier and Ernesto Kofman. 2006. *Continuous System Simulation*. Springer-Verlag, Berlin, Heidelberg.

[6] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. 2005. Associated Type Synonyms. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP '05)*. ACM, New York, NY, USA, 241–253. DOI: http://dx.doi.org/10.1145/1086365.1086397

[7] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. 2005. Associated Types with Class. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*. ACM, New York, NY, USA, 1–13. DOI: http://dx.doi.org/10.1145/1040305.1040306

[8] Antony Courtney, Henrik Nilsson, and John Peterson. 2003. The Yampa Arcade. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell (Haskell '03)*. ACM, Uppsala, Sweden, 7–18. DOI: http://dx.doi.org/10.1145/871895.871897

[9] Evan Czaplicki. 2012. *Elm: Concurrent FRP for Functional GUIs*. Undergraduate Thesis. Harward University.

[10] Conal Elliott. 1996. *A Brief Introduction to ActiveVRML*. Technical Report MSR-TR-96-05. Microsoft Research.

[11] Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *International Conference on Functional Programming*.

[12] Conal M. Elliott. 2009. Push-Pull Functional Reactive Programming. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell - Haskell '09*. ACM Press, Edinburgh, Scotland, 25. DOI: http://dx.doi.org/10.1145/1596638.1596643

[13] FRP. Functional reactive programming. http://en.wikipedia.org/wiki/Functional_reactive_programming. (????). Accessed: 2019-05-09.

[14] George Giorgidze and Henrik Nilsson. 2008. Switched-On Yampa. In *Practical Aspects of Declarative Languages*, Paul Hudak and David S. Warren (Eds.). Vol. 4902. Springer Berlin Heidelberg, Berlin, Heidelberg, 282–298. DOI: http://dx.doi.org/10.1007/978-3-540-77442-6_19

[15] George Giorgidze and Henrik Nilsson. 2011. Embedding a Functional Hybrid Modelling Language in Haskell. In *Implementation and Application of Functional Languages: 20th International Symposium, IFL 2008, Revised Selected Papers (Lecture Notes in Computer Science)*, Sven-Bodo Scholz and Olaf Chitil (Eds.), Vol. 5836. Springer-Verlag, 138–155. DOI: http://dx.doi.org/978-3-642-24451-3

[16] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. 2003. Arrows, Robots, and Functional Reactive Programming. In *Summer School on Advanced Functional Programming 2002, Oxford University (Lecture Notes in Computer Science)*, Vol. 2638. Springer-Verlag, 159–187.

[17] John Hughes. 2000. Generalising Monads to Arrows. *Science of computer programming* 37, 1-3 (2000), 67–111.

[18] Neelakantan R. Krishnaswami and Nick Benton. 2011. Ultrametric Semantics of Reactive Programs. In *2011 IEEE 26th Annual Symposium on Logic in Computer Science*. IEEE, Toronto, ON, Canada, 257–266. DOI: http://dx.doi.org/10.1109/LICS.2011.38

[19] Jérôme Mahuet. 2015. Flappy Haskell. https://github.com/Rydgel/flappy-haskell. (2015).

[20] Geoffrey Mainland. 2007. Why It's Nice to Be Quoted: Quasiquoting for Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop - Haskell '07*. ACM Press, Freiburg, Germany, 73. DOI: http://dx.doi.org/10.1145/1291201.1291211

[21] Henrik Nilsson. 2005. Dynamic Optimization for Functional Reactive Programming Using Generalized Algebraic Data Types. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP '05)*. ACM, New York, NY, USA, 54–65. DOI: http://dx.doi.org/10.1145/1086365.1086374

[22] Henrik Nilsson and Guerric Chupin. 2017. Funky Grooves: Declarative Programming of Full-Fledged Musical Applications. In *19th International Symposium on Practical Aspects of Declarative Languages (PADL 2017) (Lecture Notes in Computer Science)*, Yuliya Lierler and Walid Taha (Eds.), Vol. 10137. Springer, Paris, 163–172. DOI: http://dx.doi.org/10.1007/978-3-319-51676-9_11

[23] Henrik Nilsson, Antony Courtney, and John Peterson. 2002. Functional Reactive Programming, Continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*. ACM, Pittsburgh, Pennsylvania, USA, 51–64.

[24] Brian O'Sullivan. 2009. Criterion, a New Benchmarking Library for Haskell. http://www.serpentine.com/blog/2009/09/29/criterion-a-new-benchmarking-library-for-haskell/. (Sept. 2009).

[25] Gergely Patai. Eventless Reactivity from Scratch. (????), 15.

[26] Gergely Patai. 2011. Efficient and Compositional Higher-Order Streams. In *Functional and Constraint Logic Programming*, Julio Mariño (Ed.). Vol. 6559. Springer Berlin Heidelberg, Berlin, Heidelberg, 137–154. DOI: http://dx.doi.org/10.1007/978-3-642-20775-4_8

[27] Ross Paterson. 2001. A New Notation for Arrows. In *International Conference on Functional Programming*. ACM Press, Firenze, Italy, 229–240. http://www.soi.city.ac.uk/ross/papers/notation.html

[28] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple Unification-Based Type Inference for GADTs. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming (ICFP '06)*. ACM, New York, NY, USA, 50–61. DOI: http://dx.doi.org/10.1145/1159803.1159811

[29] ReactiveX. Reactive extensions. http://en.wikipedia.org/wiki/Reactive_extensions. (????). Accessed: 2019-05-09.

[30] Neil Sculthorpe and Henrik Nilsson. 2011. Keeping Calm in the Face of Change: Towards Optimisation of FRP by Reasoning about Change. *Journal of Higher-Order and Symbolic Computation* 23, 2 (2011), 227–271. DOI: http://dx.doi.org/10.1007/s10990-011-9068-x

[31] D D Sleator, R E Tarjan, and W P Thurston. 1986. Rotation Distance, Triangulations, and Hyperbolic Geometry. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing (STOC '86)*. ACM, New York, NY, USA, 122–135. DOI: http://dx.doi.org/10.1145/12130.12143

[32] Ertugrul Söylemez. Netwire: Library for Functional Reactive Programming (FRP). http://hackage.haskell.org/package/netwire. (????). Accessed: 2019-05-09.

[33] Ertugrul Söylemez. Wires: Functional Reactive Programming Library. //hackage.haskell.org/package/wires. (????). Accessed: 2019-05-09.

[34] The GHC Team. *Glasgow Haskell Compiler User's Guide*. https://downloads.haskell.org/~ghc/8.6.5/docs/html/users_guide/ Accessed: 2019-07-01.

[35] Jonathan Thaler, Thorsten Altenkirch, and Peer-Olaf Siebers. 2018. Pure Functional Epidemics: An Agent-Based Approach. In *IFL 2018: The 30th symposium on Implementation and Application of Functional Languages*. ACM, 1–12.

[36] Atze van der Ploeg and Koen Claessen. 2015. Practical Principled FRP: Forget the Past, Change the Future, FRPNow!. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, New York, NY, USA, 302–314. DOI: http://dx.doi.org/10.1145/2784731.2784752

[37] Zhanyong Wan and Paul Hudak. 2000. Functional Reactive Programming from First Principles. *SIGPLAN Not.* 35, 5 (May 2000), 242–252. DOI: http://dx.doi.org/10.1145/358438.349331

[38] Jeremy Yallop and Hai Liu. 2016. Causal Commutative Arrows Revisited. In *Proceedings of the 9th International Symposium on Haskell (Haskell 2016)*. ACM, New York, NY, USA, 21–32. DOI: http://dx.doi.org/10.1145/2976002.2976019

[39] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a Promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation - TLDI '12*. ACM Press, Philadelphia, Pennsylvania, USA, 53. DOI: http://dx.doi.org/10.1145/2103786.2103795