Chapter 7

Optimisation of Dynamic, Hybrid Signal Function Networks

Neil Sculthorpe¹, Henrik Nilsson²

Category: Research

Abstract: Functional Reactive Programming (FRP) is an approach to reactive programming where systems are structured as networks of functions operating on signals. FRP is based on the synchronous data-flow paradigm and supports both continuous-time and discrete-time signals (hybrid systems). What sets FRP apart from most other languages for similar applications is its support for systems with dynamic structure and for higher-order data-flow constructs. This raises a range of implementation challenges. This paper contributes towards advancing the state of the art of FRP implementation by studying the notion of signal change and change propagation in a setting of hybrid signal function networks with dynamic structure. To sidestep some problems of certain previous FRP implementations that are structured using arrows, we suggest working with a notion of composable, multi-input and multi-output signal functions. A clear conceptual distinction is also made between continuous-time and discrete-time signals. We then show how establishing change-related properties of the signal functions in a network allows such networks to be simplified (static optimisation) and can help reducing the amount of computation needed for executing the networks (dynamic optimisation). Interestingly, distinguishing between continuous-time and discrete-time signals allows us to characterise the change-related properties of signal functions more precisely than what we otherwise would have been able to, which is helpful for optimisation.

¹School of Computer Science, University of Nottingham, United Kingdom; nas@cs.nott.ac.uk
²School of Computer Science, University of Nottingham, United Kingdom; nhn@cs.nott.ac.uk
7.1 INTRODUCTION

Functional Reactive Programming (FRP) grew out of Conal Elliott’s and Paul Hudak’s work on Functional Reactive Animation [13]. The idea of FRP is to allow the full power of modern Functional Programming to be used for implementing reactive systems: systems that interact with their environment in a timely manner. This is achieved by describing such systems in terms of functions mapping signals (time-varying values) to signals and combining such functions into signal processing networks. The nature of the signals depends on the application domain. Examples include input from sensors in robotics applications [25], video streams in the context of graphical user interfaces [11] and games [12, 7], and synthesised sound signals [14].

A number of FRP variants exist. However, the synchronous data-flow principle, and support for both continuous and discrete time (hybrid systems), are common to most of them. There are thus close connections to synchronous data-flow languages like Esterel [4], Lustre [17], and Lucid Synchrone [6, 26], hybrid automata [18], and languages for hybrid modelling and simulation, like Simulink [1]. FRP, however, goes beyond most of these approaches by supporting higher-order data-flow (first-class reactive entities) and highly-dynamic system structure, all tightly integrated with a fully-fledged functional language. The Yampa implementation of FRP [23], a domain-specific embedding in Haskell, is a good example, and the starting point for this paper.

It is well-known how to implement synchronous data-flow networks with static structure efficiently [21, 17]. However, higher-order data-flow and dynamic system structure in combination with support for hybrid systems raise new implementation challenges. Specifically, Yampa, while demonstrably useful for fairly demanding applications [7, 14], has scalability issues.

One such issue is that Yampa implements discrete-time signals as continuous-time signals carrying an option type. Computations on discrete-time signals, which should only be carried out when those signals are defined, thus become computations on continuous-time signals, where computation takes place at every time step. Yampa regains some of the lost efficiency through dynamic optimisation of the signal function network [22], but only up to a point.

Further, Yampa is structured using arrows [19, 24]. Signal functions thus map a single input signal to a single output signal. Multiple inputs and outputs must be encoded through signals carrying tuples of values. The “true signals” (which serve as direct point-to-point communication channels) thus get hidden in an encoding layer, making it difficult to keep track of the signal flow and to exploit knowledge about network interdependencies for optimisation purposes.

This paper is a step towards a more scalable and finally more efficient implementation approach for a Yampa-like reactive language. We first study signal change and how change propagates in a setting of a dynamic network of signal-processing functions and mixed continuous-time and discrete-time signals. To sidestep the second problem outlined above, we adopt a setting of composable, multi-input and multi-output signal functions. While not yet formalised, we have
nevertheless found that this setting helps us to focus on the core issues.

As any digital implementation of a reactive system has to be sampled, one may wonder whether the continuous-time signals really are any different from the discrete-time signals at the implementation level. Interestingly, such a difference does emerge in our analysis of the notion of change, and it turns out to be possible to take advantage of this to characterise the properties of certain signal functions more precisely than what otherwise would have been the case.

Having studied the notion of change, we show how change-related properties of signal functions can be exploited to improve the implementation of a signal function network in two ways. The first is static. Here we provide a number of algebraic identities that can be used to simplify a network before signal processing starts and after each structural change of the network. The other is dynamic. This concerns incremental evaluation of a signal processing network, taking advantage of the fact that an output of a signal function often (but not always!) remains unchanged unless the input changes. Here, the multi-input and multi-output signal function setting is critical: representing multiple signals by a single signal carrying tuples, as in Yampa at present, makes the notion of change far too coarse grained as a change in one field of a tuple implies that the entire tuple has changed.

7.2 SIGNALS AND SIGNAL FUNCTIONS

7.2.1 Fundamentals

Signals are time-varying values. In FRP, they are conceptually modelled as functions from time to value. Signal functions are conceptually functions that operate on signals. We introduce the type constructor \( SF \) for signal functions. Thus, at a first approximation, we have:

\[
\text{Signal } a \approx \text{Time} \rightarrow a \\
\text{SF } a \ b \approx \text{Signal } a \rightarrow \text{Signal } b
\]

We re-iterate that these are just conceptual definitions. In the FRP version presented here, only signal functions are first-class entities. Signals have no independent existence: they only exist indirectly through the signal functions.

To ensure that signal functions are realisable, we require them to be causal (the output must not depend on future inputs). We thus refine the conceptual definition of signal functions by imposing this additional requirement:

\[
\text{SF } a \ b = \{ \text{sf} :: \text{Signal } a \rightarrow \text{Signal } b | \forall t :: \text{Time}, \forall s, s' :: \text{Signal } a, \\
(\forall t' \leq t, s' t' \equiv s t') \Rightarrow (\text{sf} s t \equiv \text{sf} s' t') \}
\]

Some signal functions are such that their output only depends on their input at the current point in time. We refer to these as stateless signal functions:

\[
\text{SF}_{\text{stateless}} a \ b = \{ \text{sf} :: \text{Signal } a \rightarrow \text{Signal } b | \forall t :: \text{Time}, \forall s, s' :: \text{Signal } a, \\
(s t \equiv s' t) \Rightarrow (\text{sf} s t \equiv \text{sf} s' t) \}
\]

Clearly, \( SF_{\text{stateless}} a \ b \subseteq SF a \ b \). Another way of characterising the stateless signal functions is as those that can be defined by applying a function pointwise to an
input signal. This is often referred to as lifting the function. Lifting functions is central to the arrows framework, and is achieved through the combinator \texttt{pure} (also called \texttt{arr}). In our setting, \texttt{pure} could be defined as:

\[
pure :: (a \rightarrow b) \rightarrow SF a b \nnpure f s = f \circ s
\]

The remaining (causal) signal functions, those with output that may depend on past input, are called stateful. In an implementation, stateful signal functions would keep track of any requisite information pertaining to past input by storing it in an internal state. Hence the names stateful and stateless.

Two other subsets of the causal signal functions are of interest to us: constant signal functions and decoupled signal functions. Constant signal functions are those that produce the same output at all points in time, regardless of the input signal. Decoupled signal functions are those with output that does not depend on the current input, only on past inputs:

\[
SF_{\text{constant}} a b = \{ sf :: SF a b | \forall t, t' :: Time, \forall s, s' :: Signal a, \quad sf s t \equiv sf s' t' \}
\]

\[
SF_{\text{decoupled}} a b = \{ sf :: SF a b | \forall t :: Time, \forall s, s' :: Signal a, \quad (\forall t' < t, s t' \equiv s' t') \Rightarrow (sf s t \equiv sf s' t) \}
\]

### 7.2.2 Continuous-Time and Discrete-Time Signals

As previously discussed, FRP supports both continuous-time and discrete-time signals. Discrete-time signals are defined only at discrete points in time. They can be modelled as a signal carrying an option type (e.g. \texttt{Signal (Maybe a)}), with an additional requirement that such a signal only has countably many occurrences (points in time where the signal is not \texttt{Nothing}). This is how discrete-time signals are implemented in Yampa.

However, this particular implementation choice (which was necessitated by a desire to support both continuous-time and discrete-time signals in the arrows framework) is one reason why Yampa does not scale well. Thus, we want to make a clear distinction between continuous-time and discrete-time signals at the conceptual level in the following, so as to ultimately enable an implementation where discrete-time signals truly only “exist” when the conceptual signal has an occurrence. We therefore refine the conceptual definition of signals:

\[
\text{type } \textit{CSignal} a \approx \text{Time } \rightarrow a \quad \text{-- Continuous-time signals}
\]

\[
\text{type } \textit{ESignal} a \approx \text{Time } \rightarrow \text{Maybe } a \quad \text{-- Discrete-time signals}
\]

\[
\text{data } \textit{Signal} a = \textit{C } (\textit{CSignal} a) \quad \text{| } \textit{E } (\textit{ESignal} a)
\]

We are using Haskell-like notation here and in the following. However, we go beyond Haskell in some ways, notably in terms of types. This is not too much of a concern as our main focus is the conceptual level: we are not suggesting that a reactive language should be literally implemented as described. That said,
we have successfully prototyped the relevant parts of our conceptual framework
in the dependently typed language Agda, and also in Haskell by exploiting type
classes [20].
To illustrate, we give a refined conceptual definition of pure:

\[
pure : (a \rightarrow b) \rightarrow SF a b
\]

\[
pure f \langle Cs \rangle = C (f \circ s)
\]

\[
pure f \langle Es \rangle = E (fmap f \circ s)
\]

7.2.3 Higher-Arity Signal Functions
Next, we need to introduce higher-arity signal functions: signal functions having
more than one input and output. To that end, we are going to use what we call
signal vectors. Signal vectors are conceptually products of heterogeneous signals.
However, they do not nest, but are always flat: there are never any signals of
signals. Moreover, it is only a type-level construct, and it is intimately tied to the
notion of signal functions, with no independent meaning of its own outside the
conceptual level, just as signals only exist conceptually. Syntactically, the signal
vector type is constructed as follows:

\[
\text{SigVec} = \langle \rangle \quad \text{-- Empty signal vector}
\]

\[
| \langle C t \rangle \quad \text{-- Singleton, continuous-time, signal vector}
\]

\[
| \langle E t \rangle \quad \text{-- Singleton, discrete-time, signal vector}
\]

\[
| \text{SigVec } \odot/: \text{SigVec} \quad \text{-- Signal-vector concatenation.}
\]

Note that concatenation of signal vectors is associative, and that the empty signal
vector is the unit of concatenation, implying the following type equalities:

\[
\langle as \odot/: bs \rangle \odot/: cs \equiv \langle as \odot/: \langle bs \odot/: cs \rangle \rangle
\]

\[
\langle \rangle \odot/: \langle as \rangle \equiv \langle \rangle \odot/: \langle as \rangle
\]

In our conceptual definitions, we will nevertheless allow ourselves to use signal
vectors as values, just as signals were used conceptually as values in the previous
sections. When needed, we will reuse the above type-level notation for denoting
signal vector values.

Signal functions are refined further to always work on signal vectors.

\[
\text{type } (\text{SigVec as, SigVec bs}) \Rightarrow SF \text{ as bs} = \text{as } \rightarrow \text{bs}
\]

Thus, a type \( SF \text{ as bs} \) implies that the type variables \( as \) and \( bs \) are signal vectors.

We will often use the angle bracket notation directly when writing down signal
function types; for example:

\[
SF \langle \text{C Double, E Int} \rangle \langle E \text{Bool} \rangle
\]

We will also allow ourselves to quantify over the time domain (C or E) of signals,
thus allowing signal functions to be polymorphic in the time domain; for example:

\[
SF \langle \text{td Double} \rangle \langle \text{td Double, td Double} \rangle
\]
To illustrate the ideas above, we give the final conceptual definition of pure:

\[
\text{pure} :: (a \rightarrow b) \rightarrow \text{SF} \langle td\ a \rangle \langle td\ b \rangle \\
\text{pure} \ f \langle C\ s \rangle = \langle C\ (f \circ s) \rangle \\
\text{pure} \ f \langle E\ s \rangle = \langle E\ (\text{fmap}\ f \circ s) \rangle
\]

### 7.3 SIGNAL FUNCTION NETWORKS

Systems are described by composing signal functions into signal function networks. Such networks are directed graphs, where the nodes are signal functions and the edges are signals. We limit ourselves to acyclic networks in the following.

#### 7.3.1 Network Construction

Any acyclic network can be expressed in terms of sequential and parallel composition (and primitive signal functions for duplicating, eliminating, and combining signals). We reuse the names of the arrow combinators to express these compositions (\(\gg\) and \(\ast\ast\ast\) respectively), with the following conceptual definitions:

\[
\gg :: \text{SF as bs} \rightarrow \text{SF bs cs} \rightarrow \text{SF as cs} \\
\text{sf1} \gg \text{sf2} = \text{sf2} \circ \text{sf1} \\
\ast\ast\ast :: \text{SF as cs} \rightarrow \text{SF bs ds} \rightarrow \text{SF (as ++ bs) (cs ++ ds)} \\
\text{sf1} \ast\ast\ast \text{sf2} = \lambda (\text{as} : \text{++ bs}) \rightarrow \text{sf1 as} : \text{++ sf2 bs}
\]

#### 7.3.2 Dynamic Network Structure

A network has a dynamic structure if the structure of the network can change during network execution. This is achieved by using switch constructs that replace signal functions in response to events. A basic switch has type:

\[
\text{switch} :: \text{SF as} (\langle E\ e \rangle : \text{++ bs}) \rightarrow (e \rightarrow \text{SF as bs}) \rightarrow \text{SF as bs}
\]

The behaviour of the switch is to run the initial signal function (the first argument), emitting all but the head (the event) of the output vector as the overall output. When there is an event occurrence in the event signal, the value of that signal is fed into the function (the second argument) to generate a new signal function. The entire switch construct is then removed from the network and replaced with this new signal function. We will use switch as a third primitive combinator.

#### 7.3.3 Network Examples

Figure 7.1 shows an example of a simple static network (one that contains no switches). This network can be expressed as:

\[
\text{sf1} \gg (\text{sf2} \ast\ast\ast \text{sf3}) \gg \text{sf4}
\]

Figure 7.2 shows a simple dynamic network, which can be expressed as:

\[
\text{sfA} \gg (\text{sfB} \ast\ast\ast \text{switch sfC (\lambda \_ \rightarrow \text{sfD})}) \gg \text{sfE}
\]
7.4 CHANGE

A signal function network can be optimised by eliminating parts of the network that will never change. For example, a constant signal function always outputs the same value; the signal function itself is thus not needed, only the value it produces. However, we must first define precisely what we mean by change.

7.4.1 Change Definition

If a continuous-time signal has a continuous co-domain, we say that, at any given point in time, it is changing if it has a nonzero time derivative at that point in time. If a continuous-time signal has a discrete co-domain, then instead we say that, at any given point in time, the signal is changing if there is an abrupt change in the value of the signal at that point in time.

We will consider event (discrete-time) signals to be changing only at the instants of an event occurring. This coincides with the definition of change for continuous-time signals with discrete co-domains.

The above constitute a definition of change for our conceptual model of signals. In practice, however, a signal function network is executed over a discrete sequence of sample times, rather than treating time truly continuously. We will thus adapt the definitions in terms of the type of sample time, isomorphic to the natural numbers:

\[ STime \cong \mathbb{N} \]

Thus, \( STime \) is the set of time points at which a system is sampled. We allow
ourselves to use \( \text{pred } t \) and \( \text{succ } t \) to refer to the preceding and succeeding sample point, respectively (where \( t :: \text{STime} \)).

The discrete-time definition of change is as follows:

- A continuous-time signal (for both continuous and discrete co-domains) is changing iff its value at the current time sample differs from its value at the preceding time sample.
- An event signal is changing iff there is an event occurrence at the current time sample.

\[
\text{unchanging} :: \text{SigVec as } \Rightarrow \text{as } \rightarrow \text{STime } \rightarrow \text{Bool}
\]

\[
\text{unchanging } \langle \rangle t = \text{True}
\]

\[
\text{unchanging } \langle \text{as } \mapsto \text{bs} \rangle t = \text{unchanging as } t \land \text{unchanging bs } t
\]

\[
\text{unchanging } \langle \text{C s} \rangle t = \text{if } t \equiv 0 \text{ then True else s } t \equiv s (\text{pred } t)
\]

\[
\text{unchanging } \langle \text{E s} \rangle t = s t \equiv \text{Nothing}
\]

Notice that for continuous-time signals, the definition of change cannot be applied at the first sample time. In practice, the first sample time is a special case for the execution of signal function networks, when initialisation of signal functions occurs. The details of this are unimportant for our present purposes; we simply define all continuous-time signals to be unchanging at the first sample time.

### 7.4.2 Change Classifications

We now classify our signal functions according to their change properties.

Some signal functions always yield unchanging output. We denote these signal functions with a \( U \) subscript for Unchanging.

These are not equivalent to the constant signal functions. An event source that outputs identical event occurrences at every sample time is constant, but not unchanging because of our definition of change on event signals.

\[
\text{SFU as bs} = \{ \text{sf :: SF as bs } \mid \forall t :: \text{STime}, \forall ss :: \text{as}, \text{unchanging } (\text{sf ss}) t \}
\]

Another group of signal functions are such that their output cannot change unless their input does. We denote these with an \( I \) subscript for Input-dependent. This includes all stateless functions, but, because of our definition of change on event signals, also some stateful signal functions.

\[
\text{SF} I \text{ as bs} = \{ \text{sf :: SF as bs } \mid \forall t :: \text{STime}, \forall ss :: \text{as}, \text{unchanging } ss t \Rightarrow \text{unchanging } (\text{sf ss}) t \}
\]

Finally, the subscript \( V \) for Varying is adopted if there is no particular constraint on when the output can change (i.e. it could vary at any sample time). Note that the unchanging signal functions are a subset of the input-dependent signal functions, which are a subset of the varying signal functions.

\[
\text{SFU as bs} \subseteq \text{SF} I \text{ as bs} \subseteq \text{SFV as bs}
\]

Whenever we refer to a signal function which may be in any change class, we will either omit the subscript or use a variable.
7.4. CHANGE VII–105

7.4.3 Composite Change Classifications

The change class of a composite signal function can be computed from the change classes of the signal functions that it comprises. We annotate the type signatures of the primitive combinators to reflect this:

\[
\text{data ChangeClass} = U | I | V \quad \text{deriving (Eq, Ord)}
\]

\[
(\gg) : SF x as bs \rightarrow SF y bs cs \rightarrow SF (x \gg y) as cs
\]

\[
(\leftrightarrow) : SF x as cs \rightarrow SF y bs ds \rightarrow SF (x \leftrightarrow y) (cs :\leftrightarrow bs) (cs :\leftrightarrow ds)
\]

\[
\text{switch} : SF x as (\langle E e \rangle :\leftrightarrow bs) \rightarrow (e \rightarrow SF y as bs) \rightarrow SF as bs
\]

\[
(x \gg y) : \text{ChangeClass} \rightarrow \text{ChangeClass} \rightarrow \text{ChangeClass}
\]

\[
x \gg U = U
\]

\[
x \gg V = V
\]

\[
x \gg I = x
\]

7.4.4 Example Signal Functions

We now consider some common primitive signal functions, and the change classes they fall into. Conceptual definitions are given for the simpler of these examples.

Two unchanging signal functions are never and constant: never is an event source that never produces an event occurrence; whereas constant emits the same value as output at every time sample.

\[
\text{constant} :: c \rightarrow SF U as \langle C c \rangle
\]

\[
\text{constant } c = \lambda \rightarrow \langle C (\lambda \rightarrow c) \rangle
\]

\[
\text{never} :: SF U as \langle E \rangle
\]

\[
\text{never } = \lambda \rightarrow \langle E (\lambda \rightarrow Nothing) \rangle
\]

The canonical input-dependent signal functions are the stateless ones created by pure.

\[
\text{pure} :: (a \rightarrow b) \rightarrow SF (\langle a \rangle \langle a \rangle)
\]

The edge and hold signal functions mediate between event signals and continuous-time signals: edge emits an event whenever its input changes from False to True; hold emits the value carried by its most recent input event.

\[
\text{edge} :: SF (C \text{Bool}) \langle E (\) \rangle
\]

\[
\text{edge } (C s) = \langle E (\lambda t \rightarrow s \rightarrow s (\text{pred } t)) \rangle \text{ Just } (\) \text{ else } Nothing \rangle
\]

\[
\text{hold} :: a \rightarrow SF (E a) \langle C a \rangle
\]

\[
\text{hold } a (C s) = \langle C \text{ holdAux} \rangle
\]

\[
\text{where } \text{ holdAux } t = \text{ case } s \text{ of}
\]

\[
\text{Nothing} \rightarrow \text{ if } t \equiv 0 \text{ then } a \text{ else } s (\text{pred } t)
\]

\[
\text{Just } b \rightarrow b
\]

Because of the way change was defined for event signals, both edge and hold are classified as input-dependent. The extra precision thus gained, compared with a classification as varying, is very useful for optimisation purposes.
Two varying signal functions are integral (which integrates the input signal over time) and \( i_{\text{Pre}} \) (which introduces a one time-sample-interval delay).

\[
\text{integral} :: \text{Num} \Rightarrow \text{SF}_{V} \langle C \rangle \langle C \rangle \\
i_{\text{Pre}} :: a \rightarrow \text{SF}_{V} \langle C \rangle \langle C \rangle
\]

### 7.5 NETWORK OPTIMISATIONS

We are now in a position to state a number of algebraic identities for compositions of signal functions in terms of the change classes of the involved signal functions. These identities can be used to statically optimise a network. By *optimise*, we mean simplifying a network by eliminating or combining signal functions, such that we reduce the amount of computation required to execute the network. We thereby increase the potential scalability of our networks, though the actual efficiency gain from such optimisations will depend on the details of the implementation used.

Recall that parallel and sequential composition are associative, which can be exploited to maximise the applicability of these identities.

\[
sf_{1} \circ (sf_{2} \circ sf_{3}) \equiv (sf_{1} \circ sf_{2}) \circ sf_{3} \\
sf_{1} \circ (sf_{2} \circsf_{3}) \equiv (sf_{1} \circsf_{2}) \circsf_{3}
\]

Also of use are *identity* and *sfNil*, the units of sequential and parallel composition, respectively.

\[
\text{identity} :: \text{SF}_{I} \text{as as} \\
\text{identity as} = as \\
\text{sfNil} :: \text{SF}_{U} \langle \rangle \langle \rangle \\
\text{sfNil} \langle \rangle = \langle \rangle \\
\text{identity} \circ sf = sf = sf \circ \text{identity} \\
\text{sfNil} \circsf = sf = sf \circsf \text{sfNil}
\]

Two sequential compositions in parallel can be rewritten as two parallel compositions in sequence (and conversely, provided the types match).

\[
(sf_{1} \circ sf_{2}) \circsf (sf_{3} \circsf sf_{4}) \equiv (sf_{1} \circsf sf_{3}) \circsf (sf_{2} \circsf sf_{4})
\]

Unchanging signal functions distribute into switches over sequential composition.

\[
sf_{1U} \circ switch sf_{2} f \equiv switch (sf_{1U} \circ sf_{2}) (\lambda e \rightarrow sf_{1U} \circ sf e)
\]

Signal functions with zero *output* signals can be re-classified as unchanging, as can input-dependent signal functions with zero *input* signals.

\[
\text{SF}_{I} \text{as} \langle \rangle \equiv \text{SF}_{U} \text{as} \langle \rangle \\
\text{SF}_{I} \langle \rangle \text{bs} \equiv \text{SF}_{U} \langle \rangle \text{bs}
\]

Any unchanging signal function (whether primitive or composite) can be eliminated from a network. To aid with this, we define a utility function, *extract*, that
computes the output of any unchanging signal function. A function \( v_{tail} \), which computes the tail of a signal vector, is assumed.

\[
\begin{align*}
&\text{extract} :: \text{SF}_U \text{ as } bs \rightarrow bs \\
&\text{extract (constant } c \text{)} = (C (\lambda_\text{c} \rightarrow c)) \\
&\text{extract never} = (E (\lambda_\text{Nothing} \rightarrow \text{Nothing})) \\
&\text{extract } (\text{sf}1_U \equiv \equiv \text{sf}2_U) = \text{extract sf}1_U : : \equiv \text{extract sf}2_U \\
&\text{extract } (\text{sf}1_U \Rightarrow \Rightarrow \text{sf}2_U) = \text{extract sf}2_U \\
&\text{extract } (\text{sf}1_U \Rightarrow \Rightarrow \text{sf}2_U) = \text{sf}2 (\text{extract sf}1_U) \\
&\text{extract } (\text{switch sf} U f) = v_{tail} (\text{extract sf} U f) \\
&\text{sf} U \\
&\equiv \lambda_\text{Nothing} \rightarrow \text{extract sf} U \\
&\equiv (\text{sf}1 \equiv \equiv \text{sf}2_U \equiv \equiv \text{sf}3) \\
&\equiv (\text{sf}1 \equiv \equiv (\lambda_\text{Nothing} \rightarrow (\text{sf}4 \equiv \equiv \text{sf}3))) \\
&\equiv (\text{sf}1 \equiv \equiv (\lambda (\text{as } \equiv \equiv \text{bs}) \rightarrow \text{sf}4 (\text{as } \equiv \equiv \text{extract sf}2_U : : \equiv \text{bs}))) \\
&\equiv (\text{sf}1 \equiv \equiv (\lambda (\text{as } \equiv \equiv \text{bs}) \rightarrow f (\text{as } \equiv \equiv \text{extract sf}2_U : : \equiv \text{bs}))) \\
&\equiv (\text{sf}4) \\
&\equiv (\lambda (\text{as } \equiv \equiv \text{bs}) \rightarrow \text{sf}4 (\text{as } \equiv \equiv \text{extract sf}2_U : : \equiv \text{bs}))
\end{align*}
\]

### 7.5.1 Optimisation Opportunities

These identities can be used to optimise a network before execution begins (e.g. at compile time). However, for a dynamic signal function network, the structure of the network can change at run-time. New sub-networks are being constructed, and existing networks may shift into an optimisable structure. Thus, there are opportunities for dynamic optimisations. Newly constructed networks can be optimised before they start running. Also, the entire network can be re-optimised whenever a switch occurs, so as to take advantage of any structural changes. However, the latter may be inefficient for large networks, unless we restrict optimisation to the locality of the switch occurrence.

### 7.5.2 Example Optimisation

As an example, consider the following network (where \( f \) is an arbitrary function):

\[
(\text{constant False} \Rightarrow \Rightarrow \text{edge} \Rightarrow \Rightarrow \text{switch (pure id } \equiv \equiv \text{constant 5)} f)
\]

We can add change classifications to the signal functions within this network, allowing us to then reduce the network by applying the identities.

\[
\begin{align*}
&((\text{constant False} \Rightarrow \Rightarrow \text{edge}1)_U \Rightarrow \Rightarrow (\text{switch (pure id } \equiv \equiv \text{constant 5)} f)_U)
\equiv \{ \text{sf} U \equiv \lambda_\text{Nothing} \rightarrow \text{extract sf} U \} \\
&\Longrightarrow (\lambda_\text{Nothing} \rightarrow \text{extract (constant False} \Rightarrow \Rightarrow \text{edge}1)) (\Rightarrow \Rightarrow (\text{switch (pure id } \equiv \equiv \text{constant 5)} f)_U)
\equiv \{ \text{extract (sf}1_U \Rightarrow \Rightarrow \text{sf}2_U) = \text{sf}2_U (\text{extract sf}1_U) \} \\
&\Longrightarrow (\lambda_\text{Nothing} \rightarrow \text{edge} (\text{constant False} (\Rightarrow \Rightarrow (\text{switch (pure id } \equiv \equiv \text{constant 5)} f)))
\equiv \{ \text{extract (constant c) } = \{ C (\lambda_\text{Nothing} \rightarrow c) \} \} \\
&\Longrightarrow (\lambda_\text{Nothing} \rightarrow \text{edge (C (\lambda_\text{Nothing} \rightarrow False)) (\Rightarrow \Rightarrow (\text{switch (pure id } \equiv \equiv \text{constant 5)} f)))
\equiv \{ \text{edge (C s) } = \{ E (\lambda t \rightarrow \text{if st } \land \neg (s (\text{pred t})) \text{ then Just } () \text{ else Nothing}) \} \}
\end{align*}
\]
{(λ₀ → \langle E (λ₀ → \text{if} (λ₀ → \text{False}) \land ¬ ((λ₀ → \text{False}) \text{pred} t)) \text{then} \text{Just} (\)) \text{else} \text{Nothing} \rangle \rangle \text{switch} (\text{pure} f \equiv \text{constant}_U 5 f) t} = 
\{ \text{Simplification} \}
(λ₀ → \langle E (λ₀ → \text{Nothing}) \rangle \rangle \text{switch} (\text{pure} f \equiv \text{constant}_U 5 f) t) = 
\{ \text{never} = λ₀ → \langle E (λ₀ → \text{Nothing}) \rangle \} 
(\text{never}_U \gg \text{switch} (\text{pure} f \equiv \text{constant}_U 5 f))_U = 
\{ sf1U \gg \text{switch} sf2 f \equiv \text{switch} (sf1U \gg sf2) (λ e → sf1U \gg f e) \} 
(\text{switch} (\text{never}_U \gg (\text{pure} f \equiv \text{constant}_U 5))_U (λ e → \text{never}_U \gg f) e)_U = 
\{ sf_U \equiv λ₀ → \text{extract} f_U \} 
λ₀ → \text{switch} (\text{never}_U \gg (\text{pure} f \equiv \text{constant}_U 5)) (λ e → \text{never}_U \gg f e) = 
\{ \text{extract} (\text{switch} sf f) = \text{vtail} (\text{extract} sf f) \} 
λ₀ → \text{vtail} (\text{extract} (\text{never}_U \gg (\text{pure} f \equiv \text{constant}_U 5))_U) = 
\{ \text{extract} (sf1U \gg sf2) = sf2 \text{extract} sf f \} 
λ₀ → \text{vtail} ((\text{pure} f \equiv \text{constant}_U 5) \text{extract never}) = 
\{ \text{extract never} = \langle E (λ₀ → \text{Nothing}) \rangle \} 
λ₀ → \text{vtail} ((\text{pure} f \equiv \text{constant}_U 5) (E (λ₀ → \text{Nothing})) ) = 
\{ sf1 \equiv sf2 = λ (as \equiv bs) → sf1 as \equiv sf2 bs \} 
λ₀ → \text{vtail} ((\lambda (as \equiv bs) → \text{pure} f as \equiv \text{constant} 5 bs) (E (λ₀ → \text{Nothing})) ) = 
\{ as \equiv as \equiv \text{Nothing} (\) \} 
λ₀ → \text{vtail} ((\lambda (as \equiv bs) → \text{pure} f as \equiv \text{constant} 5 bs) (E (λ₀ → \text{Nothing})) : \text{Nothing} (\) \)) = 
\{ \text{Simplification} \}
λ₀ → \text{vtail} (\text{pure} f (E (λ₀ → \text{Nothing})) : \text{Nothing} (\) \) = 
\{ \text{constant} c = λ₀ → (C (λ₀ → c)) \} 
λ₀ → \text{vtail} (\text{pure} f (E s) = E (\text{map} f \circ s) ) = 
\{ \text{vtail} (\text{nothing} (\) \equiv bs) = bs \}
λ₀ → \{ C (λ₀ → 5) \} = 
\{ \text{constant} c = λ₀ → (C (λ₀ → c)) \}
\text{constant} 5

7.6 CHANGE PROPAGATION

In general, as a signal function network grows larger, the size of the quiescent network regions (regions where signals remain unchanging during periods of time), grows in proportion to the overall network size. The present Yampa implementation, while in some cases being able to avoid redundant recomputation of unchanging signals, will nevertheless (redundantly) recompute a large fraction of
these unchanging signals. The amount of wasted work thus grows in proportion
to the size of the network, which is one reason Yampa does not scale well. In
the following, we give a brief description of how the change classifications enable
optimisations based on change propagation that eliminate all redundant computa-
tions of unchanging signals, thus addressing this particular scalability issue.

The intuition is that, at any given time sample, if a signal is known to be
unchanging before we compute its value, then there is no need to perform that
computation. Instead, we use either the value of the signal from the preceding
time sample (for a continuous-time signal), or have no event occurrence (for an
event signal).

\[
\begin{align*}
\text{evalCSignal} & : \text{CSignal} a \to \text{STime} \to a \\
\text{evalCSignal} s t & = \begin{cases} \\
\text{unchanging} \; \langle s \rangle \; t \text{ then } & \text{evalCSignal} s \; (\text{pred} \; t) \text{ else } s \; t \\
\end{cases} \\
\text{evalESignal} & : \text{ESignal} a \to \text{STime} \to \text{Maybe} \; a \\
\text{evalESignal} s t & = \begin{cases} \\
\text{unchanging} \; \langle s \rangle \; t \text{ then } & \text{Nothing} \text{ else } s \; t \\
\end{cases}
\end{align*}
\]

This is of use in a signal function network where, at the implementation level, a
signal is computed by executing a signal function. If, at any given time sample,
we do not need to compute the value of a signal, then we may not need to execute
the signal function that computes it. The set of signal functions where this is
the case is \( SF_I \), with the additional constraint that unchanging input must imply
unchanging internal state. We will denote this set as \( SF_I' \), but note that in many
languages (including Yampa) \( SF_I \) and \( SF_I' \) are equivalent.

\[
\text{unchangingState} : \text{SF as bs} \to \text{as} \to \text{STime} \to \text{Bool}
\]

\[
SF_I' a b = \{ sf : \text{SF as bs} \mid \forall t : \text{STime}, \forall ss : \text{as}, \text{unchanging ss t } \Rightarrow \text{unchanging} \; \langle sf \; ss \rangle \; t \; \wedge \text{unchangingState} sf \; ss \; t \}
\]

Thus, provided we have a way of injecting change information into a signal func-
tion network, we can optimise the run-time execution of the network so as to avoid
the execution of \( I' \) signal functions whenever their input is unchanging.

We will not go into further details about change propagation here.

\section{Related Work}

Incremental evaluation and change propagation have been studied extensively as
optimisation techniques \cite{27, 2}. The key difference between such work and ours
is that the notion of time passing is inherent to our setting. Consequently, we have
varying signal functions with output that can change even when their input does not, and decoupled signal functions which are conceptually connected to signals
from previous time samples.

The synchronous data-flow languages \cite{3, 15, 16} have long modelled reactive
programs as synchronous data-flow networks with an inherent notion of time.
These languages guarantee strong time and space bounds, and consequently have
usually had static, first order structures. However, there has been recent work on
allowing signal functions to be first class entities \cite{8}, though this does not yet have
the same level of dynamism as FRP.
Many of the optimisations we achieve through applying the identities in Section 7.5 are used in the latest version of Yampa [10, 22]. However, because of the limitations of the arrows framework, some optimisations cannot be achieved. In particular, though Yampa tries to encourage the programmer to treat event signals differently to continuous-time signals, they are not inherently different.

Also because of the arrows framework, Yampa uses nested tuples rather than the flat vectors we advocate here. This hinders change propagation by creating incidental dependencies: if one component of a tuple changes, then the tuple as a whole is considered to have changed.

FrTime [9, 5] is another dynamic, hybrid functional reactive programming language that makes a clear distinction between continuous-time signals and event signals. It uses a variety of optimisation techniques, including many of those described in this paper.

FrTime uses dynamic change propagation. When applying a lifted function to a continuous-time signal, the output is only recomputed when the input changes. When applying a signal function to an event signal, the output is only recomputed when an event occurs. This corresponds closely with the change propagation that we propose, though differs slightly in that FrTime performs run-time equality checks on recomputed signals to determine whether they have changed.

FrTime also uses a static optimisation called lowering. Lowering reduces a signal function network by fusing together composite signal functions into single signal functions. This technique is only applied to signal functions that are lifted pure functions. For example, (in our setting) a typical lowering optimisation would look like:

\[ \text{pure } f \gg \text{pure } g = \text{pure } (g \circ f) \]

FrTime’s lowering optimisations are applied statically at compile time, which allows for substantial optimisation of source code, but does not allow dynamic optimisation of the network after structural changes.

Yampa also performs some lowering optimisations, but not to the extent of FrTime. However, Yampa can lower some stateful signal functions as well as stateless ones. Yampa performs its lowering optimisations dynamically, which suffers from additional run-time overhead, but does allow for continued optimisation after structural changes.

We aim to incorporate lowering optimisations with our static optimisations, but have not done so in this paper (though there is some overlap). We advocate an initial optimisation of the network, an initial optimisation of each newly created network, and a (local) re-optimisation after each structural change.

### 7.8 CONCLUSIONS AND FURTHER WORK

We have presented a means of classifying signal functions by their change properties, such that we can use these classifications for static network optimisations and dynamic change propagation. By treating discrete-time and continuous-time...
signals distinctly from each other, we can perform more precise optimisations than would otherwise be the case.

In this paper we have only considered acyclic networks. We are currently investigating how these optimisations interact with cyclic networks, and whether allowing only non-instantaneous feedback cycles would be beneficial. We currently believe that instantaneous cycles would greatly complicate an implementation based on change propagation, for little gain. The fixed-point computation required to compute the instantaneous values of a signal involved in such a cycle takes place completely within a time sample, and thus equally well could be carried out at the purely functional level.

We are in the process of developing a new Yampa prototype implementation. Our goal is to use the optimisations discussed in this paper to achieve an implementation that scales better than the current Yampa implementation.

ACKNOWLEDGEMENTS

We would like to thank Thorsten Altenkirch for his helpful suggestions, in particular in relation to the formulation of signal function properties.

We also thank the anonymous reviewers for their valuable feedback.

REFERENCES


