Chapter 7

Optimisation of Dynamic, Hybrid Signal Function Networks

Neil Sculthorpe[1], Henrik Nilsson[2]
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.

---

[1]School of Computer Science, University of Nottingham, United Kingdom; nas@cs.nott.ac.uk

[2]School 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:

$$Signal\ a \approx Time \to a$$
$$SF\ a\ b \quad \approx Signal\ a \to Signal\ b$$

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:

$$SF\ a\ b = \{\ sf :: Signal\ a \to Signal\ b \mid \forall\ t :: Time, \forall\ s, s' :: Signal\ a,$$
$$(\forall\ t' \leqslant t, s\ t' \equiv s'\ t') \Rightarrow (sf\ s\ t \equiv sf\ s'\ t)\ \}$$

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:

$$SF_{stateless}\ a\ b = \{\ sf :: Signal\ a \to Signal\ b \mid \forall\ t :: Time, \forall\ s, s' :: Signal\ a,$$
$$(s\ t \equiv s'\ t) \Rightarrow (sf\ s\ t \equiv sf\ s'\ t)\ \}$$

Clearly, $SF_{stateless}\ a\ b \subseteq SF\ a\ b$. 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 *pure* (also called *arr*). In our setting, *pure* could be defined as:

$$pure \quad :: (a \to b) \to SF\ a\ b$$
$$pure\ f\ s = f \circ s$$

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:

$$SF_{constant}\ a\ b = \{sf :: SF\ a\ b \mid \forall\ t, t' :: Time, \forall\ s, s' :: Signal\ a,$$
$$sf\ s\ t \equiv sf\ s'\ t'\}$$
$$SF_{decoupled}\ a\ b = \{sf :: SF\ a\ b \mid \forall\ t :: Time, \forall\ s, s' :: Signal\ a,$$
$$(\forall\ t' < t, s\ t' \equiv s'\ t') \Rightarrow (sf\ s\ t \equiv sf\ s'\ t)\}$$

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. *Signal* (*Maybe a*)), with an additional requirement that such a signal only has *countably* many *occurrences* (points in time where the signal is not *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:

type *CSignal a* $\approx$ *Time* $\to$ *a*   -- Continuous-time signals
type *ESignal a* $\approx$ *Time* $\to$ *Maybe a*   -- Discrete-time signals

data *Signal a*   = *C* (*CSignal a*)
             | *E* (*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 $(C\ s) = C\ (f \circ s)$
pure f $(E\ s) = E\ (fmap\ f \circ s)$

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:

SigVec = $\langle\rangle$ -- Empty signal vector
      | $\langle C\ t\rangle$ -- Singleton, continuous-time, signal vector
      | $\langle E\ t\rangle$ -- Singleton, discrete-time, signal vector
      | *SigVec* ⧺ *SigVec* -- 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:

$(as \text{ ⧺ } bs) \text{ ⧺ } cs \equiv as \text{ ⧺ } (bs \text{ ⧺ } cs)$
$as \text{ ⧺ } \langle\rangle \equiv as \equiv \langle\rangle \text{ ⧺ } as$

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.

type $(SigVec\ as, SigVec\ bs) \Rightarrow SF\ as\ bs = as \rightarrow bs$

Thus, a type *SF 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:

$SF\ \langle C\ Double, E\ Int\rangle\ \langle E\ Bool\rangle$

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:

$SF\ \langle td\ Double\rangle\ \langle td\ Double, td\ Double\rangle$

To illustrate the ideas above, we give the final conceptual definition of *pure*:

$$pure \qquad :: (a \rightarrow b) \rightarrow SF \langle td\ a \rangle\ \langle td\ b \rangle$$
$$pure\ f\ \langle C\ s \rangle\ =\ \langle C\ (f \circ s) \rangle$$
$$pure\ f\ \langle E\ s \rangle\ =\ \langle E\ (fmap\ f \circ s) \rangle$$

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 ($\ggg$ and $***$ respectively), with the following conceptual definitions:

$$(\ggg) \qquad :: SF\ as\ bs \rightarrow SF\ bs\ cs \rightarrow SF\ as\ cs$$
$$sf1 \ggg sf2 = sf2 \circ sf1$$

$$(***) \qquad :: SF\ as\ cs \rightarrow SF\ bs\ ds \rightarrow SF\ (as :\!+\!\!: bs)\ (cs :\!+\!\!: ds)$$
$$sf1 *** sf2\ =\ \lambda(as :\!+\!\!: bs) \rightarrow sf1\ as :\!+\!\!: sf2\ bs$$

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:

$$switch :: SF\ as\ (\ \langle E\ e \rangle\ :\!+\!\!: bs) \rightarrow (e \rightarrow SF\ as\ bs) \rightarrow SF\ as\ bs$$

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:

$$sf1 \ggg (sf2 *** sf3) \ggg sf4$$

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

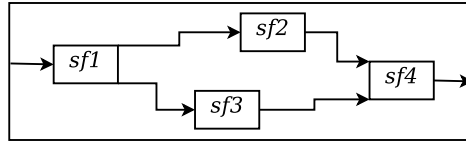$$sfA \ggg (sfB *** switch\ sfC\ (\lambda \_ \rightarrow sfD)) \ggg sfE$$

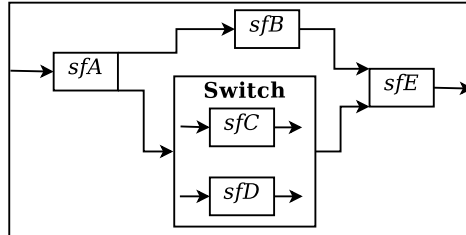**FIGURE 7.1.   An example of a static network.**

FIGURE 7.2.   An example of a dynamic network.

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 *pred t* and *succ t* to refer to the preceding and succeeding sample point, respectively (where $t :: STime$).

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.

$$
\begin{array}{ll}
unchanging & :: SigVec\ as \Rightarrow as \rightarrow STime \rightarrow Bool \\
unchanging\ \langle\rangle & t = True \\
unchanging\ (as \divideontimes bs)\ t = unchanging\ as\ t \wedge unchanging\ bs\ t \\
unchanging\ \langle C\ s\rangle & t = \mathbf{if}\ t \equiv 0\ \mathbf{then}\ True\ \mathbf{else}\ s\ t \equiv s\ (pred\ t) \\
unchanging\ \langle E\ s\rangle & t = s\ t \equiv Nothing
\end{array}
$$

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.

$$SF_U\ as\ bs = \{\ sf :: SF\ as\ bs \mid \forall\ t :: STime, \forall\ ss :: as, unchanging\ (sf\ ss)\ t\ \}$$

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.

$$
\begin{aligned}
SF_I\ as\ bs = \{\ sf :: SF\ as\ bs \mid \forall\ t :: STime, \forall\ ss :: as, \\
unchanging\ ss\ t \Rightarrow unchanging\ (sf\ ss)\ t\ \}
\end{aligned}
$$

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.

$$SF_U\ as\ bs \subseteq SF_I\ as\ bs \subseteq SF_V\ as\ bs$$

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

7.4.3 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:

> **data** *ChangeClass* $= U \mid I \mid V$ **deriving** (Eq, Ord)
>
> $(\ggg)$ $\quad :: SF_x \; as \; bs \to SF_y \; bs \; cs \to SF_{(x \; \ggg \; y)} \; as \; cs$
> $(\ast\!\ast\!\ast)$ $\quad :: SF_x \; as \; cs \to SF_y \; bs \; ds \to SF_{(x \sqcup y)} \; (as \;{:\!\!+\!\!:}\; bs) \; (cs \;{:\!\!+\!\!:}\; ds)$
> *switch* $\quad :: SF_x \; as \; (\; \langle E \; e \rangle \;{:\!\!+\!\!:}\; bs) \to (e \to SF_y \; as \; bs) \to SF_x \; as \; bs$
>
> $(\ggg)$ $\quad :: ChangeClass \to ChangeClass \to ChangeClass$
> $x \;\ggg\; U = U$
> $x \;\ggg\; V = V$
> $x \;\ggg\; I \; = x$

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.

> *constant* $\quad :: c \to SF_U \; as \; \langle C \; c \rangle$
> *constant* $c = \lambda \_ \to \; \langle C \; (\lambda \_ \to c) \rangle$
>
> *never* $\qquad :: SF_U \; as \; \langle E \; e \rangle$
> *never* $\qquad = \lambda \_ \to \; \langle E \; (\lambda \_ \to Nothing) \rangle$

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

> *pure* $:: (a \to b) \to SF_I \; \langle td \; a \rangle \; \langle td \; b \rangle$

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.

> *edge* $\qquad :: SF_I \; \langle C \; Bool \rangle \; \langle E \; () \rangle$
> *edge* $\langle C \; s \rangle \; = \; \langle E \; (\lambda t \to \mathbf{if} \; s \; t \wedge \neg \; (s \; (pred \; t)) \; \mathbf{then} \; Just \; () \; \mathbf{else} \; Nothing) \rangle$
>
> *hold* $\qquad :: a \to SF_I \; \langle E \; a \rangle \; \langle C \; a \rangle$
> *hold* $a \; \langle C \; s \rangle \; = \; \langle C \; holdAux \rangle$
> $\qquad\qquad\qquad$ **where** $holdAux \; t = \mathbf{case} \; s \; t \; \mathbf{of}$
> $\qquad\qquad\qquad\qquad\qquad\qquad Nothing \to \mathbf{if} \; t \equiv 0 \; \mathbf{then} \; a \; \mathbf{else} \; s \; (pred \; t)$
> $\qquad\qquad\qquad\qquad\qquad\qquad Just \; b \quad \to b$

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 *iPre* (which introduces a one time-sample-interval delay).

$$integral :: Num\ a \Rightarrow SF_V\ \langle C\ a \rangle\ \langle C\ a \rangle$$
$$iPre \quad :: a \rightarrow SF_V\ \langle C\ a \rangle\ \langle C\ a \rangle$$

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.

$$sf1 \ggg (sf2 \ggg sf3) \equiv (sf1 \ggg sf2) \ggg sf3$$
$$sf1 \lllll (sf2 \lllll sf3) \quad \equiv (sf1 \lllll sf2) \lllll sf3$$

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

$$identity \quad :: SF_I\ as\ as$$
$$identity\ as = as$$
$$sfNil \quad\quad :: SF_U\ \langle \rangle\ \langle \rangle$$
$$sfNil\ \langle \rangle \quad = \langle \rangle$$
$$identity \ggg sf \equiv sf \equiv sf \ggg identity$$
$$sfNil \quad\quad \lllll sf \equiv sf \equiv sf \lllll sfNil$$

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

$$(sf1 \ggg sf2) \lllll (sf3 \ggg sf4) \equiv (sf1 \lllll sf3) \ggg (sf2 \lllll sf4)$$

Unchanging signal functions distribute into switches over sequential composition.

$$sf1_U \ggg switch\ sf2\ f \equiv switch\ (sf1_U \ggg sf2)\ (\lambda e \rightarrow sf1_U \ggg f\ e)$$

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

$$SF_x\ as\ \langle \rangle \ \equiv SF_U\ as\ \langle \rangle$$
$$SF_I\ \langle \rangle\ bs \ \equiv SF_U\ \langle \rangle\ bs$$

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 *vtail*, which computes the tail of a signal vector, is assumed.

$$
\begin{aligned}
extract &:: SF_U \ as \ bs \rightarrow bs \\
extract \ (constant \ c) &= \langle C \ (\lambda\_ \rightarrow c) \rangle \\
extract \ never &= \langle E \ (\lambda\_ \rightarrow Nothing) \rangle \\
extract \ (sf1_U \ \text{∗∗} \ sf2_U) &= extract \ sf1_U \ \text{⧻} \ extract \ sf2_U \\
extract \ (sf1 \ggg sf2_U) &= extract \ sf2_U \\
extract \ (sf1_U \ggg sf2_I) &= sf2 \ (extract \ sf1_U) \\
extract \ (switch \ sf_U \ f) &= vtail \ (extract \ sf_U)
\end{aligned}
$$

$$
\begin{aligned}
sf_U &\equiv \lambda\_ \rightarrow extract \ sf_U \\
(sf1 \ \text{∗∗} \ sf2_U \ \text{∗∗} \ sf3) &\equiv (sf1 \ \text{∗∗} \ (\lambda\_ \rightarrow \langle\rangle) \ \text{∗∗} \ sf3) \ggg \\
\ggg switch \ sf4 \ f &\quad switch \ (\lambda(as \ \text{⧻} \ bs) \rightarrow sf4 \ (as \ \text{⧻} \ extract \ sf2_U \ \text{⧻} \ bs)) \\
&\quad (\lambda e \rightarrow (\lambda(as \ \text{⧻} \ bs) \rightarrow f \ e \ (as \ \text{⧻} \ extract \ sf2_U \ \text{⧻} \ bs))) \\
(sf1 \ \text{∗∗} \ sf2_U \ \text{∗∗} \ sf3) &\equiv (sf1 \ \text{∗∗} \ (\lambda\_ \rightarrow \langle\rangle) \ \text{∗∗} \ sf3) \ggg \\
\ggg sf4 &\quad (\lambda(as \ \text{⧻} \ bs) \rightarrow sf4 \ (as \ \text{⧻} \ extract \ sf2_U \ \text{⧻} \ bs))
\end{aligned}
$$

7.5.1 Optimisation Oppurtunites

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):

$$(constant \ False) \ggg edge \ggg switch \ (pure \ id \ \text{∗∗} \ constant \ 5) \ f$$

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

$$((constant_U \ False \ggg edge_I)_U \ggg (switch \ (pure_I \ id \ \text{∗∗} \ constant_U \ 5)_I) \ f)_I)_U$$

$$= \quad \{ sf_U \equiv \lambda\_ \rightarrow extract \ sf_U \}$$

$$(\lambda\_ \rightarrow extract \ (constant_U \ False \ggg edge_I)) \ggg (switch \ (pure_I \ id \ \text{∗∗} \ constant_U \ 5)_I f)_I$$

$$= \quad \{ extract \ (sf1_U \ggg sf2_I) = sf2_I \ (extract \ sf1_U) \}$$

$$(\lambda\_ \rightarrow edge \ (extract \ (constant \ False)) \ggg (switch \ (pure_I \ id \ \text{∗∗} \ constant_U \ 5)_I f)_I$$

$$= \quad \{ extract \ (constant \ c) = \langle C \ (\lambda\_ \rightarrow c) \rangle \}$$

$$(\lambda\_ \rightarrow edge \ \langle C \ (\lambda\_ \rightarrow False) \rangle) \ggg (switch \ (pure_I \ id \ \text{∗∗} \ constant_U \ 5)_I f)_I$$

$$= \quad \{ edge \ \langle C \ s \rangle = \langle E \ (\lambda t \rightarrow \textbf{if} \ s \ t \wedge \neg (s \ (pred \ t)) \ \textbf{then} \ Just \ () \ \textbf{else} \ Nothing) \rangle \}$$

$$(\lambda_- \to \langle E \, (\lambda t \to \textbf{if } (\lambda_- \to False) \, t \wedge \neg \, ((\lambda_- \to False) \, (pred \, t)) \textbf{ then } Just \, ()$$
$$\textbf{else } Nothing) \rangle \,) \ggg (switch \, (pure_I \, id \ast\!\ast \, constant_U \, 5)_I \, f)_I$$

= 　　　　$\{ Simplification \}$

$$(\lambda_- \to \langle E \, (\lambda t \to Nothing) \rangle \,) \ggg (switch \, (pure_I \, id \ast\!\ast \, constant_U \, 5)_I \, f)_I$$

= 　　　　$\{ never = \lambda_- \to \langle E \, (\lambda_- \to Nothing) \rangle \,\}$

$$(never_U \ggg (switch \, (pure_I \, id \ast\!\ast \, constant_U \, 5)_I \, f)_I)_U$$

= 　　　　$\{ sf1_U \ggg switch \, sf2 \, f \equiv switch \, (sf1_U \ggg sf2) \, (\lambda e \to sf1_U \ggg f \, e) \}$

$$(switch \, (never_U \ggg (pure_I \, id \ast\!\ast \, constant_U \, 5)_I)_U \, (\lambda e \to never_U \ggg f \, e))_U$$

= 　　　　$\{ sf_U \equiv \lambda_- \to extract \, sf_U \}$

$$\lambda_- \to extract \, (switch \, (never_U \ggg (pure_I \, id \ast\!\ast \, constant_U \, 5)_I)_U \, (\lambda e \to never_U \ggg f \, e))$$

= 　　　　$\{ extract \, (switch \, sf_U \, f) = vtail \, (extract \, sf_U) \}$

$$\lambda_- \to vtail \, (extract \, (never_U \ggg (pure_I \, id \ast\!\ast \, constant_U \, 5)_I))$$

= 　　　　$\{ extract \, (sf1_U \ggg sf2_I) = sf2_I \, (extract \, sf1_U) \}$

$$\lambda_- \to vtail \, ((pure_I \, id \ast\!\ast \, constant_U \, 5) \, (extract \, never))$$

= 　　　　$\{ extract \, never = \langle E \, (\lambda_- \to Nothing) \rangle \,\}$

$$\lambda_- \to vtail \, ((pure_I \, id \ast\!\ast \, constant_U \, 5) \, \langle E \, (\lambda_- \to Nothing) \rangle \,)$$

= 　　　　$\{ sf1 \ast\!\ast \, sf2 = \lambda (as \,:\!\#\!\!:\, bs) \to sf1 \, as \,:\!\#\!\!:\, sf2 \, bs \}$

$$\lambda_- \to vtail \, ((\lambda (as \,:\!\#\!\!:\, bs) \to pure \, id \, as \,:\!\#\!\!:\, constant \, 5 \, bs) \, \langle E \, (\lambda_- \to Nothing) \rangle \,)$$

= 　　　　$\{ as \equiv as \,:\!\#\!\!:\, \langle \rangle \,\}$

$$\lambda_- \to vtail \, ((\lambda (as \,:\!\#\!\!:\, bs) \to pure \, id \, as \,:\!\#\!\!:\, constant \, 5 \, bs) \, (\langle E \, (\lambda_- \to Nothing) \rangle \,:\!\#\!\!:\, \langle \rangle \,))$$

= 　　　　$\{ Simplification \}$

$$\lambda_- \to vtail \, (pure \, id \, \langle E \, (\lambda_- \to Nothing) \rangle \,:\!\#\!\!:\, constant \, 5 \, \langle \rangle \,)$$

= 　　　　$\{ constant \, c = \lambda_- \to \langle C \, (\lambda_- \to c) \rangle \,\}$

$$\lambda_- \to vtail \, (pure \, id \, \langle E \, (\lambda_- \to Nothing) \rangle \,:\!\#\!\!:\, (\lambda_- \to \langle C \, (\lambda_- \to 5) \rangle \,) \, \langle \rangle \,)$$

= 　　　　$\{ Simplification \}$

$$\lambda_- \to vtail \, (pure \, id \, \langle E \, (\lambda_- \to Nothing) \rangle \,:\!\#\!\!:\, \langle C \, (\lambda_- \to 5) \rangle \,)$$

= 　　　　$\{ pure \, f \, \langle E \, s \rangle \, = \langle E \, (fmap \, f \circ s) \rangle \,\}$

$$\lambda_- \to vtail \, (\, \langle E \, (fmap \, id \circ (\lambda_- \to Nothing)) \rangle \,:\!\#\!\!:\, \langle C \, (\lambda_- \to 5) \rangle \,)$$

= 　　　　$\{ vtail \, (\, \langle a \rangle \,:\!\#\!\!:\, bs) = bs \}$

$$\lambda_- \to \langle C \, (\lambda_- \to 5) \rangle$$

= 　　　　$\{ constant \, c = \lambda_- \to \langle C \, (\lambda_- \to c) \rangle \,\}$

$$constant \, 5$$

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 computations 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).

$$evalCSignal \quad :: CSignal\ a \rightarrow STime \rightarrow a$$
$$evalCSignal\ s\ t = \textbf{if}\ unchanging\ \langle s \rangle\ t\ \textbf{then}\ evalCSignal\ s\ (pred\ t)\ \textbf{else}\ s\ t$$

$$evalESignal \quad :: ESignal\ a \rightarrow STime \rightarrow Maybe\ a$$
$$evalESignal\ s\ t = \textbf{if}\ unchanging\ \langle s \rangle\ t\ \textbf{then}\ Nothing\ \textbf{else}\ s\ t$$

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.

$$unchangingState :: SF\ as\ bs \rightarrow as \rightarrow STime \rightarrow Bool$$
$$SF_{I'}\ a\ b = \{\,sf :: SF\ as\ bs \mid \forall\ t :: STime, \forall\ ss :: as,$$
$$unchanging\ ss\ t \Rightarrow unchanging\ (sf\ ss)\ t \wedge unchangingState\ sf\ ss\ t\,\}$$

Thus, provided we have a way of injecting change information into a signal function 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.

7.7 RELATED WORK

Incremental evaluation and change propagation have been studied extensively as optimisation techniques [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 [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 [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:

$$pure\ f \ggg pure\ g = pure\ (g \circ f)$$

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

REFERENCES

[1] *Using Simulink, Version 7.1.* 3 Apple Hill Drive, Natick, MA, 2008. http://www.mathworks.com.

[2] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. In *Principles of Programming Languages (POPL '02)*, pages 247–259. ACM, 2002.

[3] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE, Special issue on embedded systems*, 91(1):64–83, 2003.

[4] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[5] K. Burchett, G. H. Cooper, and S. Krishnamurthi. Lowering: A static optimization technique for transparent functional reactivity. In *Partial Evaluation and Program Manipulation (PEPM '07)*, pages 71–80. ACM, 2007.

[6] P. Caspi and M. Pouzet. Synchronous Kahn networks. In *International Conference on Functional Programming (ICFP '96)*, pages 226–238. ACM, 1996.

[7] M. H. Cheong. Functional programming and 3D games. BEng thesis, University of New South Wales, Sydney, Australia, 2005.

[8] J.-L. Colaço, A. Girault, G. Hamon, and M. Pouzet. Towards a higher-order synchronous data-flow language. In *Embedded Software (EMSOFT '04)*, pages 230–239. ACM, 2004.

[9] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *European Symposium on Programming (ESOP '06)*, pages 294–308. Springer-Verlag, 2006.

[10] A. Courtney. *Modeling User Interfaces in a Functional Language*. PhD thesis, Yale University, 2004.

[11] A. Courtney and C. Elliott. Genuinely functional user interfaces. In *Haskell Workshop (Haskell '01)*, pages 41–69, 2001.

[12] A. Courtney, H. Nilsson, and J. Peterson. The Yampa arcade. In *Haskell Workshop (Haskell '03)*, pages 7–18. ACM, 2003.

[13] C. Elliott and P. Hudak. Functional reactive animation. In *International Conference on Functional Programming (ICFP '97)*, pages 263–273. ACM, 1997.

[14] G. Giorgidze and H. Nilsson. Switched-on Yampa: Declarative programming of modular synthesizers. In *Practical Aspects of Declarative Languages (PADL '08)*, pages 282–298. Springer-Verlag, 2008.

[15] N. Halbwachs. *Synchronous Programming of Reactive Systems*. The Springer International Series in Engineering and Computer Science. Springer-Verlag, 1993.

[16] N. Halbwachs. Synchronous programming of reactive systems, a tutorial and commented bibliography. In *Computer Aided Verification (CAV '98)*, pages 1–16. Springer-Verlag, 1998.

[17] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[18] T. A. Henzinger. The theory of hybrid automata. In *Logics in Computer Science (LICS '96)*, pages 278–292. IEEE Computer Society, 1996.

[19] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, 2000.

[20] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Haskell Workshop (Haskell '04)*, pages 96–107. ACM, 2004.

[21] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1):24–35, 1987.

[22] H. Nilsson. Dynamic optimization for functional reactive programming using generalized algebraic data types. In *International Conference on Functional Programming (ICFP '05)*, pages 54–65. ACM, 2005.

[23] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Haskell Workshop (Haskell '02)*, pages 51–64. ACM, 2002.

[24] R. Paterson. A new notation for arrows. In *International Conference on Functional Programming (ICFP '01)*, pages 229–240. ACM, 2001.

[25] J. Peterson, P. Hudak, and C. Elliott. Lambda in motion: Controlling robots with Haskell. In *Practical Aspects of Declarative Languages (PADL '99)*, pages 91–105. Springer-Verlag, 1999.

[26] M. Pouzet. *Lucid Synchrone, version 3.0: Tutorial and reference manual*, 2006.

[27] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *Principles of Programming Languages (POPL '93)*, pages 502–510. ACM, 1993.