Safe Functional Reactive Programming through Dependent Types

Neil Sculthorpe and Henrik Nilsson

School of Computer Science
University of Nottingham
United Kingdom
{nas,nhn}@cs.nott.ac.uk

Types '09
Aussois, France
12th May 2009

Reactive Programming

- Reactive Program: one that continually interacts with its environment, interleaving input and output in a timely manner.

Reactive Programming

- Reactive Program: one that continually interacts with its environment, interleaving input and output in a timely manner.
- Examples: MP3 players, robot controllers, video games, aeroplane control systems. . .

Reactive Programming

- Reactive Program: one that continually interacts with its environment, interleaving input and output in a timely manner.
- Examples: MP3 players, robot controllers, video games, aeroplane control systems...
- Contrast with transformational programs, which take all input at the start of execution and produce all output at the end (e.g. a compiler).

Motivation

- Existing reactive programming languages make a trade-off between static safety guarantees and expressiveness.

Motivation

- Existing reactive programming languages make a trade-off between static safety guarantees and expressiveness.
- Most emphasise safety properties (such as the absence of deadlock and run-time errors), which are often crucial in reactive domains.

Motivation

- Existing reactive programming languages make a trade-off between static safety guarantees and expressiveness.
- Most emphasise safety properties (such as the absence of deadlock and run-time errors), which are often crucial in reactive domains.
- Functional Reactive Programming (FRP) differs in that it is very expressive, but lacking in these guarantees.

Motivation

- Existing reactive programming languages make a trade-off between static safety guarantees and expressiveness.
- Most emphasise safety properties (such as the absence of deadlock and run-time errors), which are often crucial in reactive domains.
- Functional Reactive Programming (FRP) differs in that it is very expressive, but lacking in these guarantees.
- This work is about using dependent types to get some of these safety guarantees within FRP (without sacrificing expressiveness).

Outline

1. Motivation
2. Outline
3. Dependent Types in FRP
4. Functional Reactive Programming (FRP)
5. New Type System
6. Safe Feedback Loops
7. Uninitialised Signals
8. Summary

Dependent Types in FRP

- A domain-specific dependent type system for FRP that enforces safety properties.

Dependent Types in FRP

- A domain-specific dependent type system for FRP that enforces safety properties.
- An implementation, using this type system, in Agda.

Dependent Types in FRP

- A domain-specific dependent type system for FRP that enforces safety properties.
- An implementation, using this type system, in Agda.
- Currently just a proof of concept implementation.

Dependent Types in FRP

- A domain-specific dependent type system for FRP that enforces safety properties.
- An implementation, using this type system, in Agda.
- Currently just a proof of concept implementation.
- The implementation serves as a proof of the soundness of the type system. (Agda checks totality and termination.)

Functional Reactive Programming

- A functional approach to reactive programming.

Functional Reactive Programming

- A functional approach to reactive programming.
- Usually a domain specific embedding inside an existing functional language (e.g. Haskell).

Functional Reactive Programming

- A functional approach to reactive programming.
- Usually a domain specific embedding inside an existing functional language (e.g. Haskell).
- Fundamental concept: time varying values called signals.

$$\text{Signal A} \;\approx\; \text{Time} \rightarrow \text{A}$$

Functional Reactive Programming

- A functional approach to reactive programming.
- Usually a domain specific embedding inside an existing functional language (e.g. Haskell).
- Fundamental concept: time varying values called signals.

$$\text{Signal A} \approx \text{Time} \rightarrow \text{A}$$

- We (following the FRP language Yampa) take signal functions as the basic building blocks of our language.

Functional Reactive Programming

- A functional approach to reactive programming.
- Usually a domain specific embedding inside an existing functional language (e.g. Haskell).
- Fundamental concept: time varying values called signals.

$$\text{Signal A} \;\approx\; \text{Time} \rightarrow \text{A}$$

- We (following the FRP language Yampa) take signal functions as the basic building blocks of our language.
- Signal functions are (conceptually) functions mapping signals to signals.

$$\text{SF A B} \;\approx\; \text{Signal A} \rightarrow \text{Signal B}$$

Functional Reactive Programming

- A functional approach to reactive programming.
- Usually a domain specific embedding inside an existing functional language (e.g. Haskell).
- Fundamental concept: time varying values called signals.

$$\text{Signal A} \approx \text{Time} \rightarrow \text{A}$$

- We (following the FRP language Yampa) take signal functions as the basic building blocks of our language.
- Signal functions are (conceptually) functions mapping signals to signals.

$$\text{SF A B} \approx \text{Signal A} \rightarrow \text{Signal B}$$

Example: Robot Controller

$$\text{RobotController} = \text{SF Sensor ControlValue}$$

Signal Functions Characteristics

- All signal functions are (temporally) causal: current output does not depend upon future input.

Signal Functions Characteristics

- All signal functions are (temporally) causal: current output does not depend upon future input.
- We identify some subsets of the causal signal functions:

Signal Functions Characteristics

- All signal functions are (temporally) causal: current output does not depend upon future input.
- We identify some subsets of the causal signal functions:
  - Stateless signals functions: current output only depends upon current input (e.g. square root).

Signal Functions Characteristics

- All signal functions are (temporally) causal: current output does not depend upon future input.
- We identify some subsets of the causal signal functions:
    - Stateless signals functions: current output only depends upon current input (e.g. square root).
    - Stateful signal functions: current output can depend upon past and current input (e.g. integration).

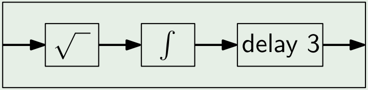
Signal Functions Characteristics

- All signal functions are (temporally) causal: current output does not depend upon future input.
- We identify some subsets of the causal signal functions:
    - Stateless signals functions: current output only depends upon current input (e.g. square root).
    - Stateful signal functions: current output can depend upon past and current input (e.g. integration).
    - Decoupled signal functions: current output only depends upon past inputs (e.g. time delay).

Signal Functions Characteristics

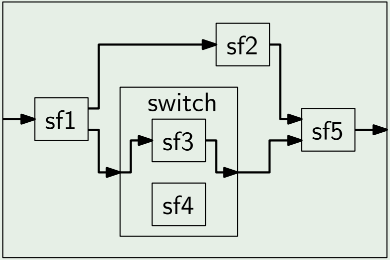
- All signal functions are (temporally) causal: current output does not depend upon future input.
- We identify some subsets of the causal signal functions:
  - Stateless signals functions: current output only depends upon current input (e.g. square root).
  - Stateful signal functions: current output can depend upon past and current input (e.g. integration).
  - Decoupled signal functions: current output only depends upon past inputs (e.g. time delay).
- We compose signal functions to form signal function networks.

Example: Composing Signal Functions



Synchronous Data-Flow Networks

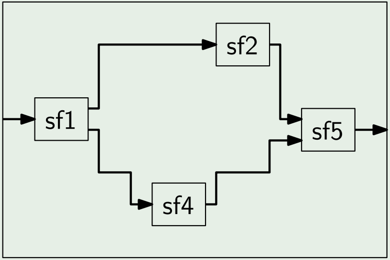
Example: A Signal Function Network



- Similar to the synchronous data-flow languages. (Esterel, Lustre, Lucid Synchrone etc...)
- FRP differs in that it allows dynamic higher-order system structures, but lacks some of their safety guarantees.

Synchronous Data-Flow Networks

Example: A Signal Function Network



- Similar to the synchronous data-flow languages. (Esterel, Lustre, Lucid Synchrone etc...)
- FRP differs in that it allows dynamic higher-order system structures, but lacks some of their safety guarantees.

Hybrid Signals

- FRP is also hybrid: it has both continuous-time and discrete-time signals.

Hybrid Signals

- FRP is also hybrid: it has both continuous-time and discrete-time signals.
- We call discrete-time signals event signals.

Hybrid Signals

- FRP is also hybrid: it has both continuous-time and discrete-time signals.
- We call discrete-time signals event signals.
- Event signals are usually (in FRP) embedded in continuous-time signals using an option type.
  Event A = Signal (Maybe A)

Hybrid Signals

- FRP is also hybrid: it has both continuous-time and discrete-time signals.
- We call discrete-time signals event signals.
- Event signals are usually (in FRP) embedded in continuous-time signals using an option type.
  $\text{Event A} \;=\; \text{Signal (Maybe A)}$
- However, this is insufficiently abstract to be able to exploit their discrete properties, and can lead to conceptual errors on behalf of the programmer.

Hybrid Signals

- FRP is also hybrid: it has both continuous-time and discrete-time signals.
- We call discrete-time signals event signals.
- Event signals are usually (in FRP) embedded in continuous-time signals using an option type.
  Event A = Signal (Maybe A)
- However, this is insufficiently abstract to be able to exploit their discrete properties, and can lead to conceptual errors on behalf of the programmer.
- To address this, we introduce signal vectors: conceptually heterogeneous vectors of signals that allows us to precisely identify signals (and their time domains) in the types.

Signal Descriptors

Descriptor Definitions

data SigDesc : Set **where**
  E : Set → SigDesc
  C : Set → SigDesc

SVDesc : Set
SVDesc = List SigDesc

Example: A Signal Vector Descriptor

svdExample : SVDesc
svdExample = $(C\ \mathbb{R} :: E\ \mathrm{Bool} :: C\ \mathbb{Z} :: [\,])$

Signal Functions

Original SF Type

$$SF : \mathsf{Set} \longrightarrow \mathsf{Set} \longrightarrow \mathsf{Set}$$

Revised SF Type

$$SF : \mathsf{SVDesc} \longrightarrow \mathsf{SVDesc} \longrightarrow \mathsf{Set}$$

Signal Functions

Original SF Type

$$\mathsf{SF} : \mathsf{Set} \longrightarrow \mathsf{Set} \longrightarrow \mathsf{Set}$$

Revised SF Type

$$\mathsf{SF} : \mathsf{SVDesc} \longrightarrow \mathsf{SVDesc} \longrightarrow \mathsf{Set}$$

Example: Some Primitive Signal Functions

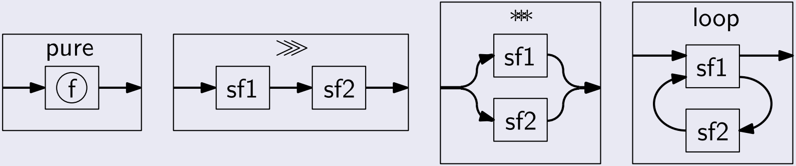
$$\begin{aligned}
\mathsf{now} &: \mathsf{SF}\ []\ [\mathsf{E}\ \mathsf{Unit}] \\
\mathsf{time} &: \mathsf{SF}\ []\ [\mathsf{C}\ \mathsf{Time}] \\
\mathsf{edge} &: \mathsf{SF}\ [\mathsf{C}\ \mathsf{Bool}]\ [\mathsf{E}\ \mathsf{Unit}] \\
\int &: \mathsf{SF}\ [\mathsf{C}\ \mathbb{R}]\ [\mathsf{C}\ \mathbb{R}]
\end{aligned}$$

Constructing Signal Functions

Primitive Combinators

$$
\begin{aligned}
\mathsf{pure} &: (a \rightarrow b) \rightarrow SF\ [C\ a]\ [C\ b] \\
\_\ggg\_ &: SF\ as\ bs \rightarrow SF\ bs\ cs \rightarrow SF\ as\ cs \\
\_{*\!*\!*}\_ &: SF\ as\ cs \rightarrow SF\ bs\ ds \rightarrow SF\ (as +\!\!+ bs)\ (cs +\!\!+ ds) \\
\mathsf{loop} &: SF\ (as +\!\!+ cs)\ (bs +\!\!+ ds) \rightarrow SF\ ds\ cs \rightarrow SF\ as\ bs
\end{aligned}
$$

Graphical Representations

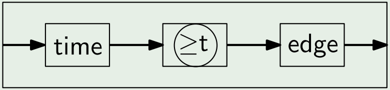


Constructing Signal Functions

Example: The *after* Signal Function

The signal function after t produces an event at time t.

after : Time → SF [] [E Unit]
after t = time ⋙ pure (_⩽_ t) ⋙ edge



Well Defined Feedback Loops

Well Defined Feedback Loops

- Badly defined feedback loops can cause a program to diverge.

Well Defined Feedback Loops

- Badly defined feedback loops can cause a program to diverge.
- Feedback loops are well defined if somewhere in the cycle they are broken by a decoupled signal function.

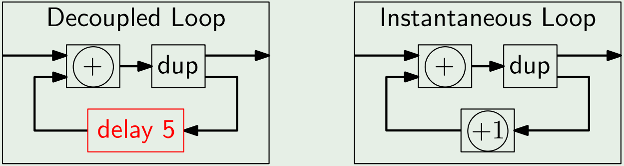
Well Defined Feedback Loops

- Badly defined feedback loops can cause a program to diverge.
- Feedback loops are well defined if somewhere in the cycle they are broken by a decoupled signal function.
- Reminder: a signal function is decoupled if its current output only depends upon its past inputs.

Well Defined Feedback Loops

- Badly defined feedback loops can cause a program to diverge.
- Feedback loops are well defined if somewhere in the cycle they are broken by a decoupled signal function.
- Reminder: a signal function is decoupled if its current output only depends upon its past inputs.
- Methods of decoupling: time delays, constants, some primitives (e.g. integration using the rectangle rule). . .

Examples: Loops



Existing Approaches to Decoupling

Relying on the programmer to correctly define loops.

- Does not restrict expressiveness.
- Easy to introduce bugs into programs.
- Most FRP variants take this approach.

Existing Approaches to Decoupling

Relying on the programmer to correctly define loops.

- Does not restrict expressiveness.
- Easy to introduce bugs into programs.
- Most FRP variants take this approach.

Explicit use of a decoupling primitive in all recursive definitions.

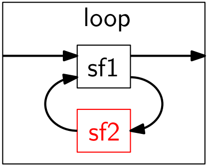
- Can be confirmed as safe by the type checker (conservatively).
- Limits expressiveness (cannot use dynamic or higher order signal functions for decoupling).
- Most synchronous data-flow languages take this approach.

Our Approach: Decoupledness in the Types

We index signal function types with a boolean to denote their decoupledness.

Primitive Combinators Indexed by Decoupledness

$$\begin{array}{lll}
\mathrm{pure} & : & (\mathrm{a} \rightarrow \mathrm{b}) \rightarrow \mathrm{SF}\ [\mathrm{C\ a}]\ [\mathrm{C\ b}]\ \mathrm{false} \\
\_\ggg\_ & : & \mathrm{SF\ as\ bs}\ d_1 \rightarrow \mathrm{SF\ bs\ cs}\ d_2 \rightarrow \mathrm{SF\ as\ cs}\ (d_1 \vee d_2) \\
\_{*}{*}{*}\_ & : & \mathrm{SF\ as\ cs}\ d_1 \rightarrow \mathrm{SF\ bs\ ds}\ d_2 \rightarrow \mathrm{SF}\ (\mathrm{as} +\!\!+ \mathrm{bs})\ (\mathrm{cs} +\!\!+ \mathrm{ds})\ (d_1 \wedge d_2) \\
\mathrm{loop} & : & \mathrm{SF}\ (\mathrm{as} +\!\!+ \mathrm{cs})\ (\mathrm{bs} +\!\!+ \mathrm{ds})\ d \rightarrow \mathrm{SF\ ds\ cs\ true} \rightarrow \mathrm{SF\ as\ bs}\ d
\end{array}$$



Our Approach: Decoupledness in the Types

We index signal function types with a boolean to denote their decoupledness.

Primitive Combinators Indexed by Decoupledness

$\text{pure} : (a \to b) \to \text{SF}\ [\text{C}\ a]\ [\text{C}\ b]\ \text{false}$

$\_ \ggg \_ : \text{SF as bs}\ d_1 \to \text{SF bs cs}\ d_2 \to \text{SF as cs}\ (d_1 \vee d_2)$

$\_ \ast\!\!\ast\!\!\ast \_ : \text{SF as cs}\ d_1 \to \text{SF bs ds}\ d_2 \to \text{SF}\ (\text{as} +\!\!+ \text{bs})\ (\text{cs} +\!\!+ \text{ds})\ (d_1 \wedge d_2)$

$\text{loop} : \text{SF}\ (\text{as} +\!\!+ \text{cs})\ (\text{bs} +\!\!+ \text{ds})\ d \to \text{SF ds cs true} \to \text{SF as bs}\ d$

Examples: Primitive Signal Functions Indexed by Decoupledness

$\text{now} : \text{SF}\ []\ [\text{E Unit}]\ \text{true}$

$\text{time} : \text{SF}\ []\ [\text{C Time}]\ \text{true}$

$\text{edge} : \text{SF}\ [\text{C Bool}]\ [\text{E Unit}]\ \text{false}$

$\int : \text{SF}\ [\text{C}\ \mathbb{R}]\ [\text{C}\ \mathbb{R}]\ ?$

Uninitialised Signals

Uninitialised Signals

- The decoupled signal function pre introduces an infinitesimal time delay in a continuous-time signal.
- But this also means the signal is initially undefined.

Initialisation Primitives

pre : SF [C a] [C a] true

initialise : a → SF [C a] [C a] false

iPre : a → SF [C a] [C a] true

Uninitialised Signals

Boolean Synonyms

$$\begin{aligned}\mathsf{Init} &= \mathsf{Bool}\\ \mathsf{init} &= \mathsf{true}\\ \mathsf{unin} &= \mathsf{false}\end{aligned}$$

Adding Initialisation to Signal Descriptors

```
data SigDesc : Set where
  E :          Set → SigDesc
  C : Init  →  Set → SigDesc
```

Note that event signals are only defined at discrete points in time, so there is no need to initialise them.

Uninitialised Signals

Primitives updated with Initialisation Descriptors

$$\begin{aligned}
&\mathsf{pure} &&: (a \rightarrow b) \rightarrow SF\ [C\ i\ a]\ [C\ i\ b]\ \mathsf{false}\\
&\mathsf{pre} &&: SF\ [C\ \mathsf{init}\ a]\ [C\ \mathsf{unin}\ a]\ \mathsf{true}\\
&\mathsf{initialise} &&: a \rightarrow SF\ [C\ \mathsf{unin}\ a]\ [C\ \mathsf{init}\ a]\ \mathsf{false}\\
&\mathsf{iPre} &&: a \rightarrow SF\ [C\ \mathsf{init}\ a]\ [C\ \mathsf{init}\ a]\ \mathsf{true}
\end{aligned}$$

Summary

- FRP and synchronous data-flow languages make a trade-off between expressiveness and safety.
- Dependent types allow us to have FRP with safety guarantees, while retaining dynamic higher-order data-flow.
- One such safety guarantee is the absence of instantaneous feedback loops.
- Another is that all signals (that require it) are correctly initialised.
- See our paper for further details: `http://www.cs.nott.ac.uk/~nas/icfp09.pdf`