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

The 14th International Conference on Functional Programming
Edinburgh, Scotland
31st August 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:
  - Static safety guarantees vs Expressiveness

Motivation

- Existing reactive programming languages make a trade-off:
  - Static safety guarantees vs Expressiveness
- Most emphasise safety guarantees:
  - Absence of deadlock, absence of run-time errors, etc...
  - Often crucial in reactive domains.

Motivation

- Existing reactive programming languages make a trade-off:
  - Static safety guarantees vs Expressiveness
- Most emphasise safety guarantees:
  - Absence of deadlock, absence of run-time errors, etc...
  - Often crucial in reactive domains.
- Functional Reactive Programming (FRP):
  - Very expressive.
  - Lacks many safety guarantees.

Motivation

- Existing reactive programming languages make a trade-off:
  - Static safety guarantees vs Expressiveness
- Most emphasise safety guarantees:
  - Absence of deadlock, absence of run-time errors, etc...
  - Often crucial in reactive domains.
- Functional Reactive Programming (FRP):
  - Very expressive.
  - Lacks many safety guarantees.
- This work: using dependent types to get safety guarantees within FRP without sacrificing expressiveness.

Outline

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.
- A proof of the soundness of the type system, in the form of an Agda implementation.

Dependent Types in FRP

- A domain-specific dependent type system for FRP that enforces safety properties.
- A proof of the soundness of the type system, in the form of an Agda implementation.

Agda

- Dependently typed language.
- Similarities with Haskell.
- Totality and termination checks.

Dependent Types in FRP

- A domain-specific dependent type system for FRP that enforces safety properties.
- A proof of the soundness of the type system, in the form of an Agda implementation.
- In development: a Haskell implementation (using GHC language extensions).

Agda

- Dependently typed language.
- Similarities with Haskell.
- Totality and termination checks.

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.

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

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.

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

- 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

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

Signal Functions

- All signal functions are (temporally) causal: current output does not depend upon future input.
- We build FRP programs by composing signal functions to form signal function networks.

Signal Functions

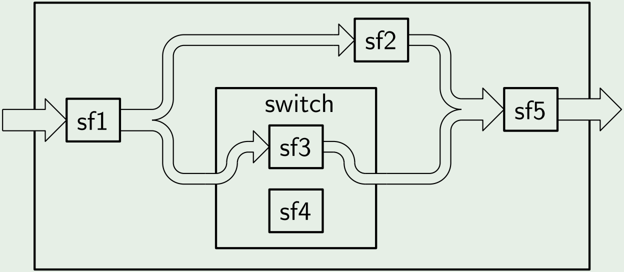
- All signal functions are (temporally) causal: current output does not depend upon future input.
- We build FRP programs by composing signal functions to form signal function networks.

Implementing Signal Functions

- In practise, FRP implementations run signal functions over a discrete sequence of time samples (synchronously).
- This is hidden by the signal function abstraction.

Synchronous Data-Flow Networks

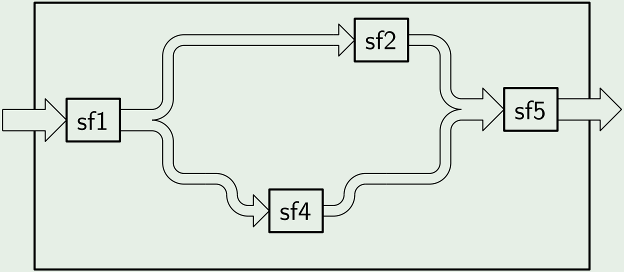
Example: A Signal Function Network



- 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 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 safety guarantees.

Hybrid Signals

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

Hybrid Signals

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

Hybrid Signals

- FRP is also hybrid: continuous-time and discrete-time signals.
- We call discrete-time signals event signals.
- Event signals are usually embedded in continuous-time signals using an option type:

$$\text{Event } A \;=\; \text{Signal } (\text{Maybe } A)$$

Hybrid Signals

- FRP is also hybrid: continuous-time and discrete-time signals.
- We call discrete-time signals event signals.
- Event signals are usually embedded in continuous-time signals using an option type:

$$\text{Event } A \;=\; \text{Signal } (\text{Maybe } A)$$

- Problems:
  - Insufficiently abstract to exploit their discrete properties.
  - Can lead to conceptual errors by the programmer.

Signal Vectors

- Signal Vector: a heterogeneous vector of signals with the time domain explicit in the type.

Signal Vectors

- Signal Vector: a heterogeneous vector of signals with the time domain explicit in the type.
- Signal Descriptor: a type and time domain (C or E).

Signal Vectors

- Signal Vector: a heterogeneous vector of signals with the time domain explicit in the type.
- Signal Descriptor: a type and time domain (C or E).
- Signal Vector Descriptor: a list of signal descriptors.

Signal Vectors

- Signal Vector: a heterogeneous vector of signals with the time domain explicit in the type.
- Signal Descriptor: a type and time domain (C or E).
- Signal Vector Descriptor: a list of signal descriptors.

Example: A Signal Vector Descriptor

$$[\mathsf{C}\ \mathsf{Bool}, \mathsf{E}\ (\mathsf{Tree}\ \mathbb{Z}), \mathsf{C}\ \mathbb{R}]$$

Signal Vectors

- Signal Vector: a heterogeneous vector of signals with the time domain explicit in the type.
- Signal Descriptor: a type and time domain (C or E).
- Signal Vector Descriptor: a list of signal descriptors.

Example: A Signal Vector Descriptor

$$[\mathsf{C}\ \mathsf{Bool}, \mathsf{E}\ (\mathsf{Tree}\ \mathbb{Z}), \mathsf{C}\ \mathbb{R}]$$

Example: Some Primitive Signal Functions

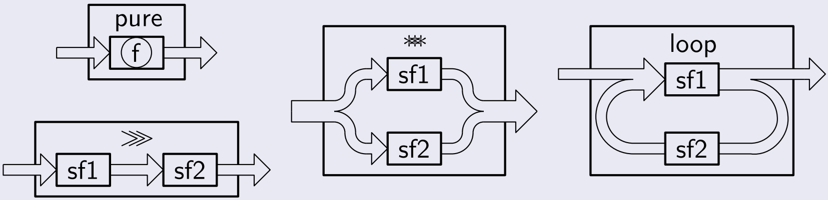
$$\begin{aligned}
\mathsf{now} &: \mathsf{SF}\ [\,]\ [\mathsf{E}\ \mathsf{Unit}] \\
\mathsf{time} &: \mathsf{SF}\ [\,]\ [\mathsf{C}\ \mathsf{Time}] \\
\mathsf{edge} &: \mathsf{SF}\ [\mathsf{C}\ \mathsf{Bool}]\ [\mathsf{E}\ \mathsf{Unit}] \\
\int &: \mathsf{SF}\ [\mathsf{C}\ \mathbb{R}]\ [\mathsf{C}\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

Constructing Signal Functions

Primitive Combinators

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

Graphical Representations

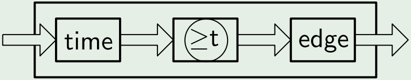


Constructing Signal Functions

Example: The *after* Signal Function

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

$$\text{after} : \text{Time} \rightarrow \text{SF}\ [\,]\ [\text{E Unit}]$$
$$\text{after t} = \text{time} \ggg \text{pure}\ (\geqslant \text{t}) \ggg \text{edge}$$



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.

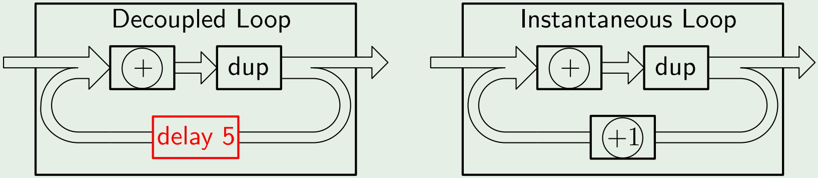
Well Defined Feedback Loops

- 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.
- Decoupled signal function: 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.
- Decoupled signal function: current output only depends upon its past inputs.
- Methods of decoupling: time delays, infinitesimal delays, 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.

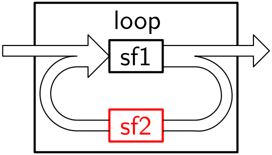
- Can be confirmed as safe by the type checker (conservatively).
- Limits expressiveness (in particular, structural dynamism and higher-order signal functions).
- Most synchronous data-flow languages take this approach.

Our Approach: Decoupledness in the Types

Index signal functions by booleans to denote decoupledness.

Primitive Combinators Indexed by Decoupledness

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



Our Approach: Decoupledness in the Types

Index signal functions by booleans to denote decoupledness.

Primitive Combinators Indexed by Decoupledness

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

Examples: Primitive Signal Functions Indexed by Decoupledness

$$\begin{array}{ll}
\mathsf{now} & : \mathsf{SF}\ [\,]\ [\mathsf{E\ Unit}]\ \color{red}{\mathsf{true}} \\
\mathsf{time} & : \mathsf{SF}\ [\,]\ [\mathsf{C\ Time}]\ \color{red}{\mathsf{true}} \\
\mathsf{edge} & : \mathsf{SF}\ [\mathsf{C\ Bool}]\ [\mathsf{E\ Unit}]\ \color{red}{\mathsf{false}} \\
\int & : \mathsf{SF}\ [\mathsf{C}\ \mathbb{R}]\ [\mathsf{C}\ \mathbb{R}]\ \color{red}{?}
\end{array}$$

Uninitialised Signals

Uninitialised Signals

The Signal Function pre

- Conceptually an infinitesimal time delay.
- Decoupled.
- Initial output is undefined.

Uninitialised Signals

The Signal Function pre

- Conceptually an infinitesimal time delay.
- Decoupled.
- Initial output is undefined.

Initialisation Primitives

$$\begin{array}{ll} \text{pre} & : \text{SF [C a] [C a] true} \\ \text{initialise} & : \text{a} \rightarrow \text{SF [C a] [C a] false} \\ \text{iPre} & : \text{a} \rightarrow \text{SF [C a] [C a] true} \end{array}$$

Uninitialised Signals

Primitives updated with Initialisation Descriptors

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

Boolean Synonyms

$$
\begin{array}{lcl}
\text{init} & = & \text{true} \\
\text{unin} & = & \text{false}
\end{array}
$$

Event signals are only defined at discrete points in time, so there is no need to ensure initialisation.

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.
- Examples:
  - Absence of instantaneous feedback loops.
  - Correct initialisation of signals.
- See the paper for further details.