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

25th British Colloquium on Theoretical Computer Science
University of Warwick
7th April 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: 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: 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

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.

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

- 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.

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

Example

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

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).

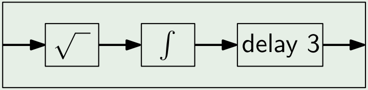
Signal Functions Characteristics

- 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

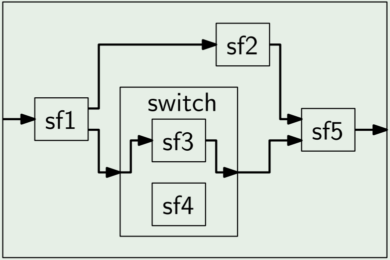
- All signal functions are (temporally) causal: current output does not depend upon future input.
- We identify some subsets of the causal signal functions:
  - Stateless signals functions: current output only depends upon current input (e.g. square root).
  - Stateful signal functions: current output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



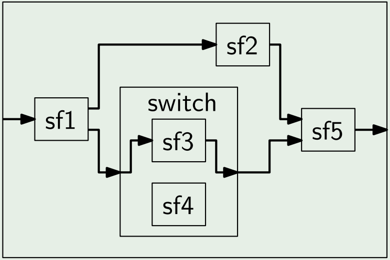
Synchronous Data-Flow Networks

Example



Synchronous Data-Flow Networks

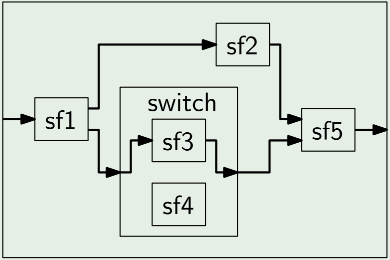
Example



- Similar to the synchronous data-flow languages. (Esterel, Lustre, Lucid Synchrone etc...)

Synchronous Data-Flow Networks

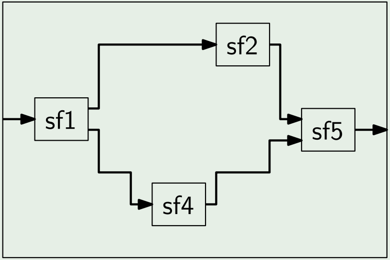
Example



- 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.

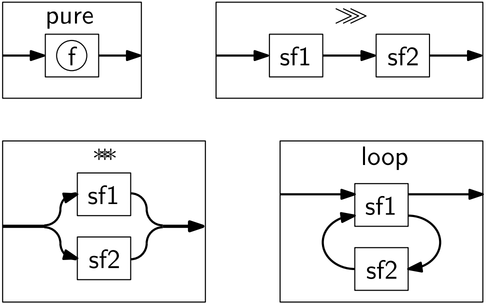
Synchronous Data-Flow Networks

Example



- 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.

Some Primitive Combinators



$$
\begin{aligned}
\text{pure} &: (a \to b) \to \text{SF } a\ b \\
\_ \ggg \_ &: \text{SF } a\ x \to \text{SF } x\ b \to \text{SF } a\ b \\
\_ *** \_ &: \text{SF } a\ x \to \text{SF } b\ y \to \text{SF } (a, b)\ (x, y) \\
\text{loop} &: \text{SF } (a, x)\ (b, y) \to \text{SF } y\ x \to \text{SF } a\ b
\end{aligned}
$$

Dependently-Typed Functional Programming

- The type of the result can depend upon the value of the argument.

Dependently-Typed Functional Programming

- The type of the result can depend upon the value of the argument.
- Little distinction between types and values:
  - data can appear in the types;
  - types can be manipulated as data.

Dependently-Typed Functional Programming

- The type of the result can depend upon the value of the argument.
- Little distinction between types and values:
  - data can appear in the types;
  - types can be manipulated as data.
- Types can encode properties of data:
  - propositions as types;
  - programs as proofs.

Dependently-Typed Functional Programming

- The type of the result can depend upon the value of the argument.
- Little distinction between types and values:
  - data can appear in the types;
  - types can be manipulated as data.
- Types can encode properties of data:
  - propositions as types;
  - programs as proofs.
- We're using Agda (similar to Haskell).

Dependently-Typed Functional Programming

- The type of the result can depend upon the value of the argument.
- Little distinction between types and values:
  - data can appear in the types;
  - types can be manipulated as data.
- Types can encode properties of data:
  - propositions as types;
  - programs as proofs.
- We're using Agda (similar to Haskell).

Example

$$\text{divide} : \mathbb{N} \to (n : \mathbb{N}) \to n > 0 \to \mathbb{N}$$

$$\text{append} : \text{Vector } A\ m \to \text{Vector } A\ n \to \text{Vector } A\ (m+n)$$

$$\text{take} : (m : \mathbb{N}) \to \text{Vector } A\ n \to m \leqslant n \to \text{Vector } A\ m$$

Dependent Types in FRP

- We use dependent types in two ways:

Dependent Types in FRP

- We use dependent types in two ways:
  - A domain-specific dependent type system for FRP.

Dependent Types in FRP

- We use dependent types in two ways:
  - A domain-specific dependent type system for FRP.
  - An implementation (using this type system) embedded in a dependently-typed host language (Agda).
    - Currently just a proof of concept implementation.
    - Not yet useable for practical applications.
    - But Agda accepts it, proving the soundness of the type system.
    - (Agda guarantees totality and termination.)

Dependent Types in FRP

- We use dependent types in two ways:
  - A domain-specific dependent type system for FRP.
  - An implementation (using this type system) embedded in a dependently-typed host language (Agda).
    - Currently just a proof of concept implementation.
    - Not yet useable for practical applications.
    - But Agda accepts it, proving the soundness of the type system.
    - (Agda guarantees totality and termination.)
- The rest of the talk will be about one aspect of the type system: ensuring safe feedback loops.

Decoupling Cycles

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

Decoupling Cycles

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

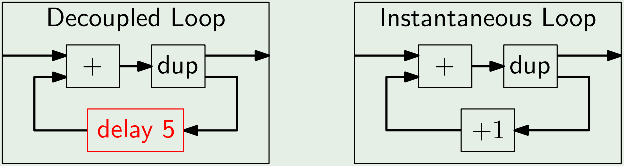
Decoupling Cycles

- Badly defined feedback loops can cause a program to diverge.
- Feedback loops are safe if somewhere in the cycle they are broken by a decoupled signal function.
- Methods of decoupling: delays, constants, some primitives (e.g. integration using the rectangle rule)...

Decoupling Cycles

- Badly defined feedback loops can cause a program to diverge.
- Feedback loops are safe if somewhere in the cycle they are broken by a decoupled signal function.
- Methods of decoupling: delays, constants, some primitives (e.g. integration using the rectangle rule)...

Example



Existing Approaches

- Existing languages either rely on the programmer to correctly define feedback loops...
  - Does not restrict expressiveness.
  - Easy to introduce bugs into programs.

Existing Approaches

- Existing languages either rely on the programmer to correctly define feedback loops...
  - Does not restrict expressiveness.
  - Easy to introduce bugs into programs.
- ...or require explicit use of a specific delay primitive in all recursive (looping) definitions.
  - Can be confirmed as safe by the type checker (conservatively).
  - Limits expressiveness (cannot use dynamic or higher order signal functions for decoupling).

Our Approach: Decoupledness in the Types

- We index the types of signal functions by their decoupledness.

Our Approach: Decoupledness in the Types

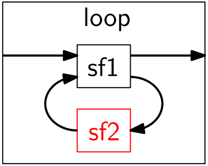
- We index the types of signal functions by their decoupledness.
- The types then enforce that feedback loops are decoupled.

Our Approach: Decoupledness in the Types

- We index the types of signal functions by their decoupledness.
- The types then enforce that feedback loops are decoupled.

$$\begin{aligned} \text{dec} &= \text{true} \\ \text{inst} &= \text{false} \end{aligned}$$

$$\begin{aligned}
\text{pure} &: (a \rightarrow b) \rightarrow \text{SF } a\ b\ \textcolor{red}{\text{inst}} \\
_ \ggg _ &: \text{SF } a\ x\ \textcolor{red}{d_1} \rightarrow \text{SF } x\ b\ \textcolor{red}{d_2} \rightarrow \text{SF } a\ b\ \textcolor{red}{(d_1 \vee d_2)} \\
_ *** _ &: \text{SF } a\ x\ \textcolor{red}{d_1} \rightarrow \text{SF } b\ y\ \textcolor{red}{d_2} \rightarrow \text{SF } (a, b)\ (x, y)\ \textcolor{red}{(d_1 \wedge d_2)} \\
\text{loop} &: \text{SF } (a, x)\ (b, y)\ \textcolor{red}{d} \rightarrow \text{SF } y\ x\ \textcolor{red}{\text{dec}} \rightarrow \text{SF } a\ b\ \textcolor{red}{d}
\end{aligned}$$



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.
- An example is tracking decoupledness to prevent instantaneous feedback loops.
- See our paper for further details: `http://www.cs.nott.ac.uk/~nas/icfp09.pdf`