FRP and Yampa:

- FRP: conceptual framework for programming with time-varying entities.
- Yampa (formerly AFRP): an implementation of FRP embedded in Haskell.
FRP and Yampa:

- FRP: conceptual framework for programming with time-varying entities.
- Yampa (formerly AFRP): an implementation of FRP embedded in Haskell.

Theme of this talk:

Bringing classical FP ideas like first class continuations to the world of hybrid systems and reactive programming to make structurally dynamic systems possible.
Key concept: functions on signals.

Intuition:

\[ \text{Signal } \alpha = \text{Time } \rightarrow \alpha \]
\[ x :: \text{Signal } T1 \]
\[ y :: \text{Signal } T2 \]
\[ f :: \text{Signal } T1 \rightarrow \text{Signal } T2 \]

Additionally: \textit{causality} requirement.
Alternative view:

Functions on signals can encapsulate state.

\[ f(t) \]

\[ state(t) \]

\[ x(t) \rightarrow f(state(t)) \rightarrow y(t) \]

\( state(t) \) summarizes input history \( x(t'), t' \in [0, t] \).

Functions on signals are either:

- **Stateful**: \( y(t) \) depends on \( x(t) \) and \( state(t) \)
- **Stateless**: \( y(t) \) depends only on \( x(t) \)
The Big Picture

Some areas where functions on signals are central:

- Modelling and simulation of physical systems
- Hybrid systems
- Reactive systems
- Embedded systems
- Digital Signal Processing
  
...
Related Languages

Lots of languages designed around the idea of functions on signals, e.g.:

- **Modelling Languages:**
  - Simulink
  - Ptolemy II
- **Synchronous languages:**
  - Esterel
  - Lustre
  - Lucid Synchrone

...
Describing Composite Systems

[Diagram of a circuit with labeled components and symbols, including resistors (R1, R2), inductors (L), capacitors (C), and mathematical operations (summation, inversion, integration). The diagram includes input and output signals (u_R2, i_2, u_L, 1/L, integral, +1 ∑, -1 ∑, u_R1, u_C, u_in, i_1, i).]
What If System Structure Varies?

- What type of structural changes can be expressed?
- What about state?
Support for Structural Changes

Simulink is fairly typical:

- Blocks can be enabled/disabled dynamically.
- State can be preserved or reset.
Support for Structural Changes

Simulink is fairly typical:

- Blocks can be enabled/disabled dynamically.
- State can be preserved or reset.

Number of structural configurations fixed. Blocks cannot be added/deleted dynamically!
Example: Traffic Surveillance
Tailgating detector

Tailgating Detectors:
- $tgd(1,2)$
- $tgd(2,3)$
- ...

Trackers:
- $tr_1$
- $tr_2$
- $tr_3$
- ...

Video:

Highway:
- 1
- 2
- 3
• *Signal Functions* are first class entities.
  Intuition: $SF \alpha \beta = Signal \alpha \rightarrow Signal \beta$
• **Signal Functions** are first class entities. Intuition: $\text{SF } \alpha \beta = \text{Signal } \alpha \rightarrow \text{Signal } \beta$

• Signals are *not* first class entities.
Yampa

- **Signal Functions** are first class entities. Intuition: $SF \alpha \beta = \text{Signal} \alpha \rightarrow \text{Signal} \beta$
- Signals are *not* first class entities.
- **Switchers** “apply” signal functions to signals at some point in time, creating a running signal function instance.
• **Signal Functions** are first class entities. Intuition: $\text{SF } \alpha\beta = \text{Signal } \alpha \rightarrow \text{Signal } \beta$

• Signals are *not* first class entities.

• **Switchers** “apply” signal functions to signals at some point in time, creating a running signal function instance.

• Special combinators to run *collections* of signal functions in parallel.
Static Signal Function Collections

The most basic way to form a SF collection:

\[
\text{parB} :: \text{Functor } \text{col} \Rightarrow \text{col} \ (\text{SF} \ a \ b) \rightarrow \text{SF} \ a \ (\text{col} \ b)
\]

Can’t add or remove SFs from the collection.
Dynamic Signal Function Collections

Idea:

- Switch over *collections* of signal functions.
- On event, “freeze” running signal functions into collection of signal function *continuations*.
- Modify collection as needed and switch back in.

\[
p\text{SwitchB} :: \text{Functor col} \Rightarrow \text{col (SF} \ a \ b) \rightarrow \text{SF} \ (a, \ \text{col} \ b) \ (\text{Event} \ c) \rightarrow (\text{col} \ (\text{SF} \ a \ b) \rightarrow \ c \rightarrow \text{SF} \ a \ (\text{col} \ b)) \rightarrow \text{SF} \ a \ (\text{col} \ b)
\]
Dynamic Signal Function Collections

Idea:

- Switch over *collections* of signal functions.
- On event, “freeze” running signal functions into collection of signal function *continuations*.
- Modify collection as needed and switch back in.

\[
p\text{SwitchB} :: \text{Functor col} \Rightarrow \\
\quad \text{col } (\text{SF } a \text{ } b) \rightarrow \\
\quad \text{SF } (a, \text{ col } b) \text{ (Event } c) \\
\quad \rightarrow (\text{col } (\text{SF } a \text{ } b) \rightarrow c \rightarrow \text{SF } a \text{ (col } b)) \\
\quad \rightarrow \text{SF } a \text{ (col } b)
\]
Dynamic Signal Function Collections

Idea:

- Switch over *collections* of signal functions.
- On event, “freeze” running signal functions into collection of signal function *continuations*.
- Modify collection as needed and switch back in.

```
pSwitchB :: Functor col =>
  col (SF a b) ->
  SF (a, col b) (Event c)
  -> (col (SF a b) -> c -> SF a (col b))
  -> SF a (col b)
```
Dynamic Signal Function Collections

Idea:

- Switch over *collections* of signal functions.
- On event, “freeze” running signal functions into collection of signal function *continuations*.
- Modify collection as needed and switch back in.

\[
pSwitchB :: \text{Functor } \text{col} \Rightarrow
\]
\[
\begin{align*}
\text{col (SF } a b) \\
\rightarrow \text{ SF } (a, \text{ col } b) \text{ (Event } c) \\
\rightarrow \text{ (col (SF } a b) \rightarrow c \rightarrow \text{ SF } a \text{ (col } b)) \\
\rightarrow \text{ SF } a \text{ (col } b)
\end{align*}
\]
Dynamic Signal Function Collections

\[ S_1 \]

\[ S_0 \]

\[ t_e \]
Dynamic Signal Function Collections

\[ S_0 \quad S_1 \quad S_2 \quad S_3 \]

\[ t_e \]
Dynamic Signal Function Collections

\[ S_0, S_1, S_2, S_3 \]

\[ t_e \]
Dynamic Signal Function Collections

\[ S_0, S_1, S_2, S_3 \]

\[ t_e, t_{e2} \]
Dynamic Signal Function Collections
How can flexible communication be achieved?

- Input filtering (+ feedback) is enough.
How can flexible communication be achieved?

- Input filtering (+ feedback) is enough.
- But composing each actual signal function with a filter is awkward and inflexible.
Routing (2)

Idea:

- Generalized \texttt{pSwitch} responsible for routing; obviates need for composition.
Idea:

- Generalized $pSwitch$ responsible for routing; obviates need for composition.
- Desired routing specified by user-supplied routing function.
pSwitch :: Functor col =>
  (forall sf . (a -> col sf -> col (b, sf)))
  -> col (SF b c)
  -> SF (a, col c) (Event d)
  -> (col (SF b c) -> d -> SF a (col c))
  -> SF a (col c)
The Routing Function Type

Universal quantification over the collection members:

\[
\text{Functor } \text{col } \Rightarrow \ \\
(\forall \text{sf} . \ (a \rightarrow \text{col } \text{sf} \rightarrow \text{col } (b, \text{sf})))
\]

Collection members thus \textbf{opaque}:

- Ensures only signal functions from argument can be returned.
- Unfortunately, does not prevent duplication or discarding of signal functions.
type CarTracker = SF (Video, UAVStatus)
                (Car, Event ())

multiCarTracker ::
    SF (Video, UAVStatus, Event CarTracker)
    [(Id, Car)]
multiCarTracker =
    pSwitch route []
    addOrDelCarTrackers
    (\cts' f ->
     multiCarTracker (f cts'))
Related Work (1)

- First-Order Systems: no dynamic collections
  - Esterel [Berry 92], Lustre [Caspi 87], Lucid Synchrone [Caspi 00], SimuLink, RT-FRP [Wan, Taha, Hudak 01]

- Fudgets [Carlsson and Hallgren 93, 98]
  - Continuation capture with extractSP
  - Dynamic Collections with dynListF
  - No synchronous bulk update
Related Work (2)

- Fran [Elliott and Hudak 97, Elliott 99]
  - First class *signals*.
  - But dynamic collections?

- FranTk [Sage 99]
  - Dynamic collections, but only via \(\mathbb{I}\mathbb{O}\) monad.
Obtaining Yampa

These ideas have been implemented in Yampa, yielding a very expressive language for reactive programming.

Yampa 0.9 is available from

http://www.haskell.org/yampa