Functional Reactive Programming

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.

Functional Reactive Programming

Key concept: functions on signals.

\[
\begin{align*}
  &x \rightarrow f \\
  &y
\end{align*}
\]

Intuition:
- Signal \( \alpha = \text{Time} \rightarrow \alpha \)
- \( x :: \text{Signal \, T1} \)
- \( y :: \text{Signal \, T2} \)
- \( f :: \text{Signal \, T1} \rightarrow \text{Signal \, T2} \)

Additionally: causality requirement.

State

Alternative view:
Functions on signals can encapsulate state.

\[
\begin{align*}
  &x(t) \rightarrow \boxed{f_{state(t)}} \\
  &y(t)
\end{align*}
\]

\( state(t) \) 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

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.

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:

Yampa

- **Signal Functions** are first class entities. Intuition: $SF\alpha\beta = Signal\alpha \rightarrow Signal\beta$
- 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:

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

\[
\text{parB} [\text{sf1,sf2,\ldots,sfN}]
\]

Can’t add or remove SFs from the collection.

Dynamic Signal Function Collections

Idea:

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

\[
\text{pSwitchB} :: \text{Functor col} \implies \\
\implies \text{SF a} (\text{col b})
\]

Routing (1)

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** `pSwitch` responsible for routing; obviates need for composition.
- Desired routing specified by user-supplied routing function.

![Diagram](image)

The Routing Function Type

Universal quantification over the collection members:

```haskell
Functor col =>
    (forall sf . (a -> col sf -> col (b,sf)))
```

Collection members thus **opaque**:

- Ensures only signal functions from argument can be returned.
- Unfortunately, does not prevent duplication or discarding of signal functions.

Tailgating Detector: Excerpts

```haskell
type CarTracker = SF (Video, UAVStatus)
    (Car, Event ())

multiCarTracker ::
    SF (Video, UAVStatus, Event CarTracker)
[(Id,Car)]
multiCarTracker =
    pSwitch route []
    addOrDelCarTrackers (\cts’ f ->
    multiCarTracker (f cts’))
```

pSwitch

```haskell
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)
```
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 `IO` 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