Functional Reactive Programming, Continued

Henrik Nilsson, Antony Courtney, and John Peterson

Yale University

New Haven, CT, USA

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.

Functional Reactive Programming

FRP and Yampa:

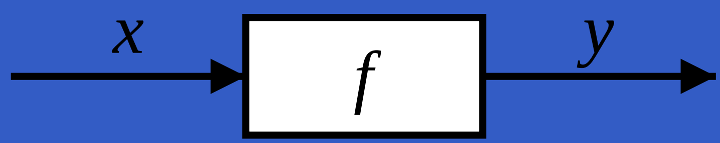
- FRP: conceptual framework for programming with time-varying entities.
- Yampa (formerly AFRP): an implementation of FRP embedded in Haskell.

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.



Intuition:

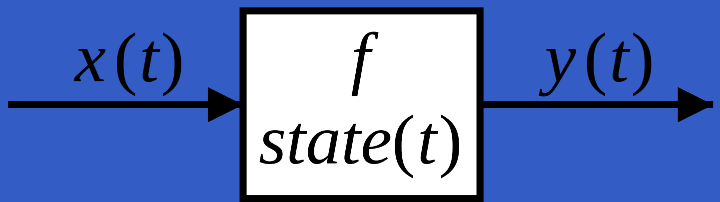
$$\texttt{Signal}\ \alpha = \texttt{Time} \to \alpha$$
$$x :: \texttt{Signal T1}$$
$$y :: \texttt{Signal T2}$$
$$f :: \texttt{Signal T1} \to \texttt{Signal T2}$$

Additionally: ***causality*** requirement.

State

Alternative view:

Functions on signals can encapsulate state.



$state(t)$ 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:

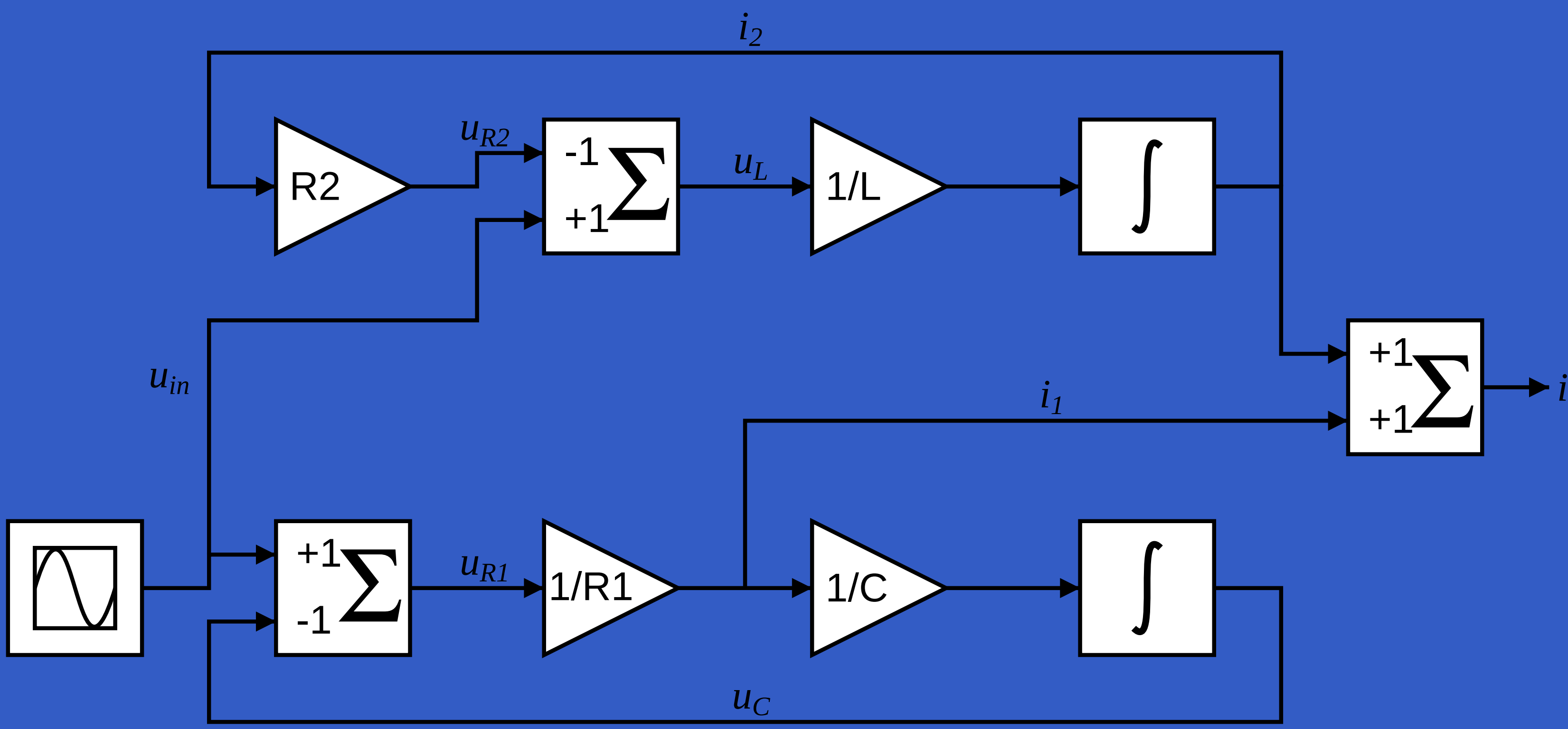
- Modelling and simulation of physical systems
- Hybrid systems
- Reactive systems
- Embedded systems
- Digital Signal Processing
- ...

Related Languages

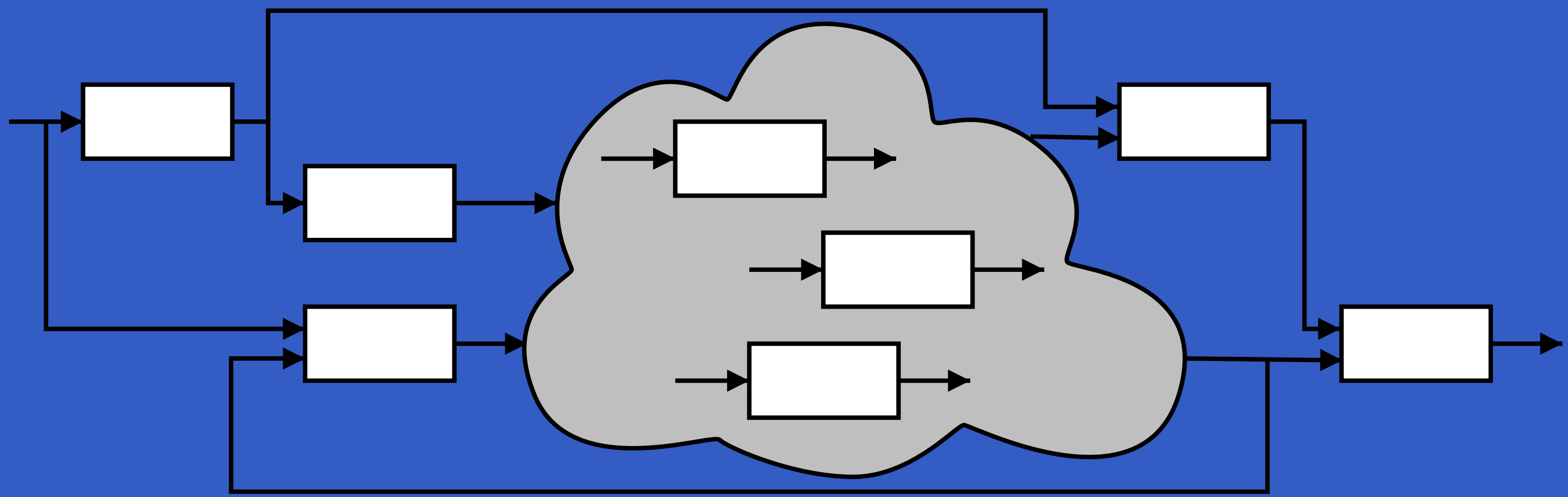
Lots of languages designed around the idea of functions on signals, e.g.:

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

Describing Composite Systems



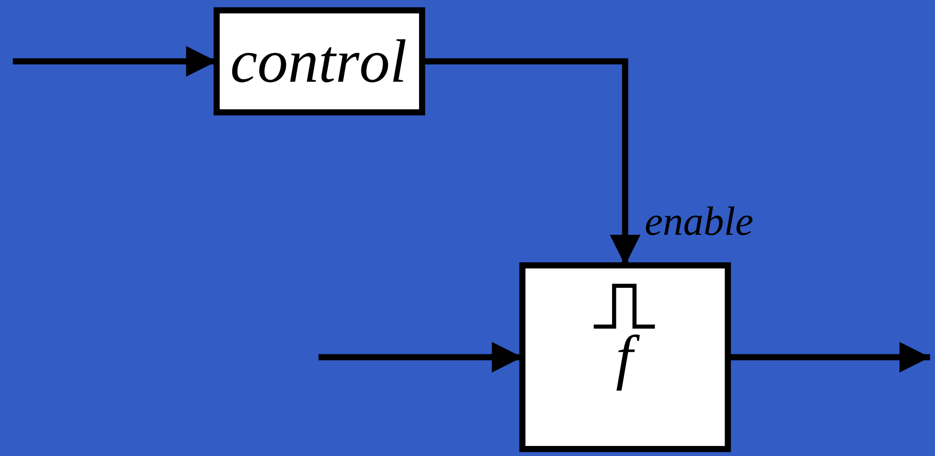
What If System Structure Varies?



- What type of structural changes can be expressed?
- What about state?

Support for Structural Changes

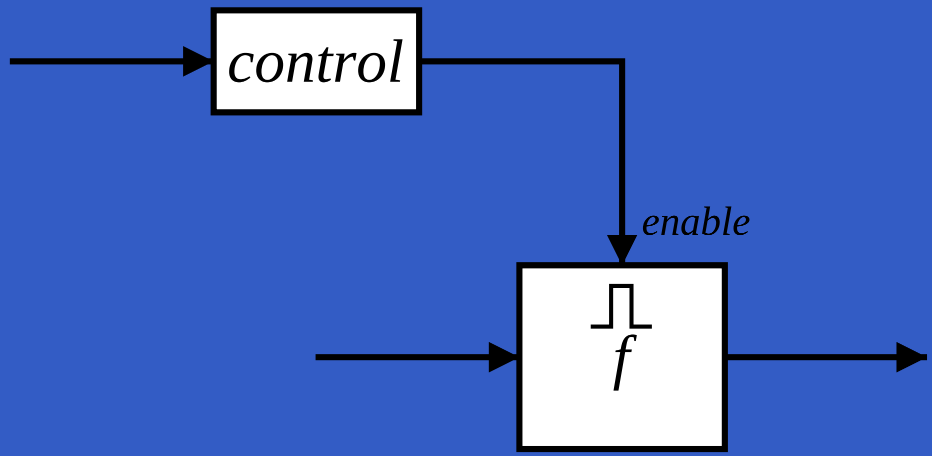
Simulink is fairly typical:



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

Support for Structural Changes

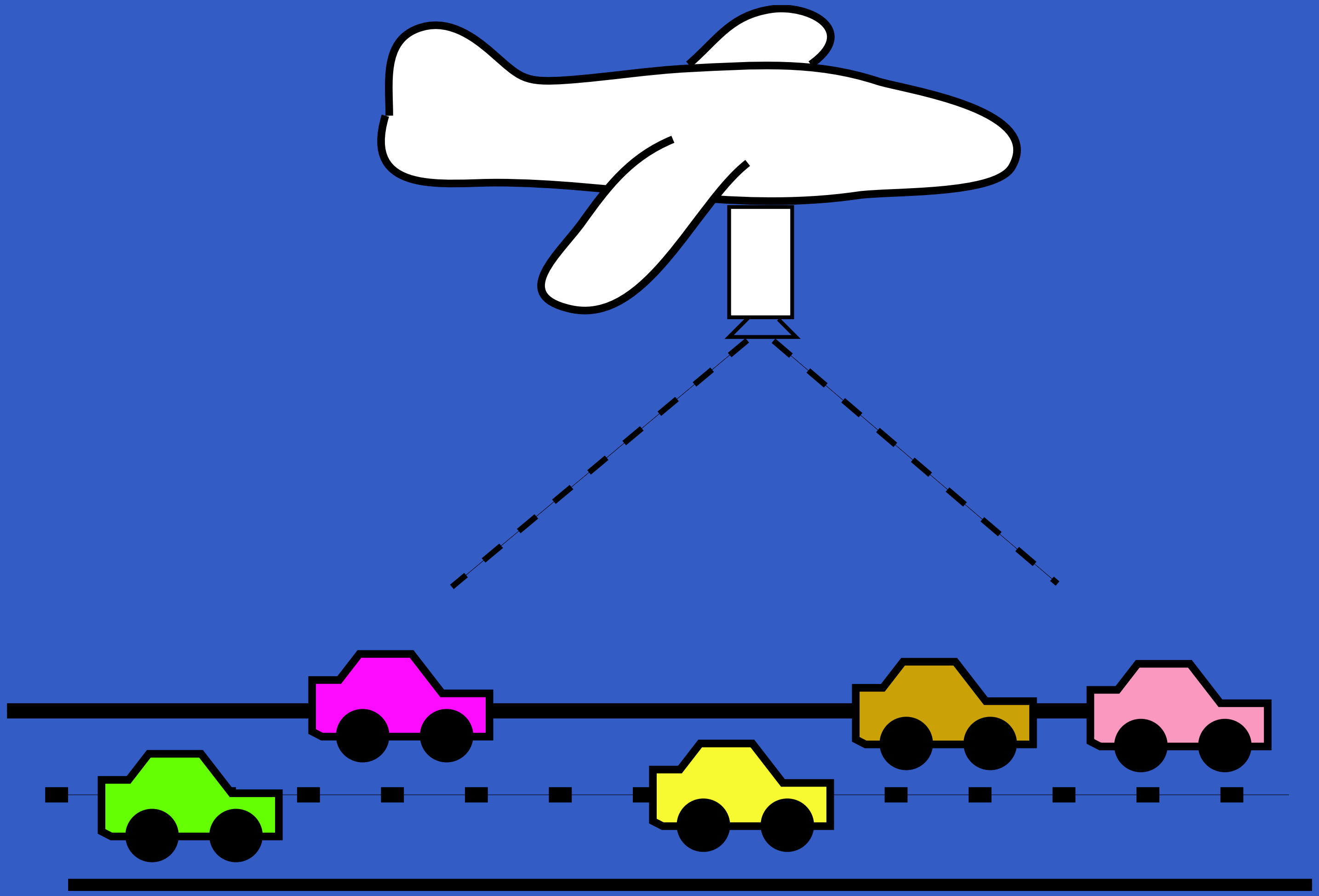
Simulink is fairly typical:



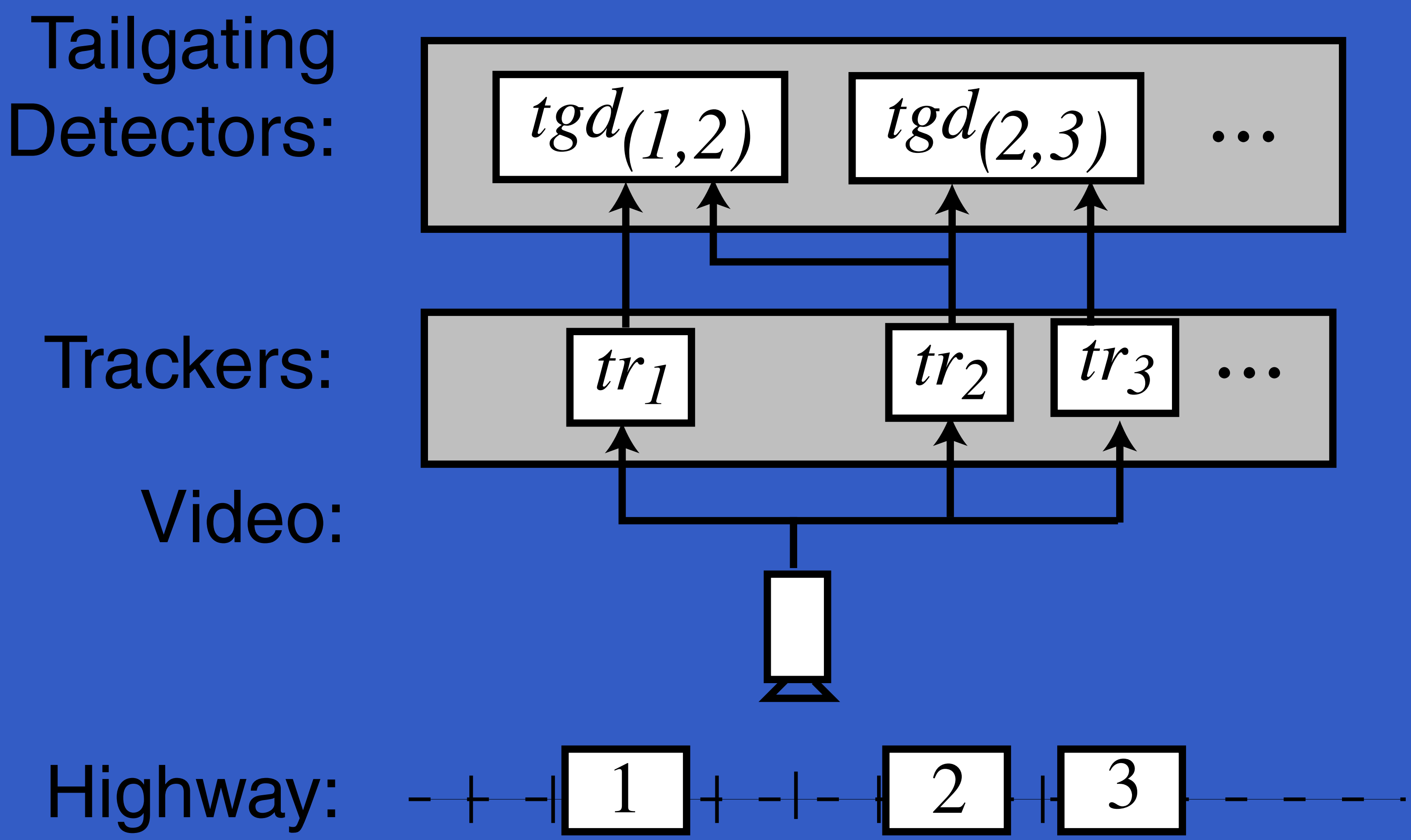
- Blocks can be enabled/disabled dynamically.
- State can be preserved or reset.

Number of structural configurations fixed.
Blocks cannot be added/deleted dynamically!

Example: Traffic Surveillance



Tailgating detector



Yampa

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

Yampa

- ***Signal Functions*** are first class entities.
  Intuition: $\texttt{SF}\ \alpha\ \beta = \texttt{Signal}\ \alpha \rightarrow \texttt{Signal}\ \beta$
- Signals are ***not*** first class entities.

Yampa

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

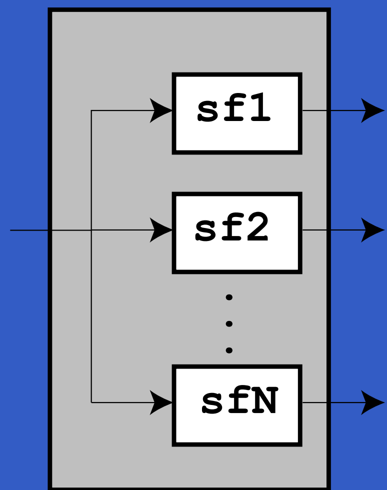
Yampa

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

```
parB :: Functor col =>
        col (SF a b) -> SF a (col b)
```



parB [sf1,sf2,...,sfN]

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.

```
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 :: Functor col =>
    col (SF a b)                                    Initial collection
    -> 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 ::  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)
```

Event source

Dynamic Signal Function Collections

Idea:

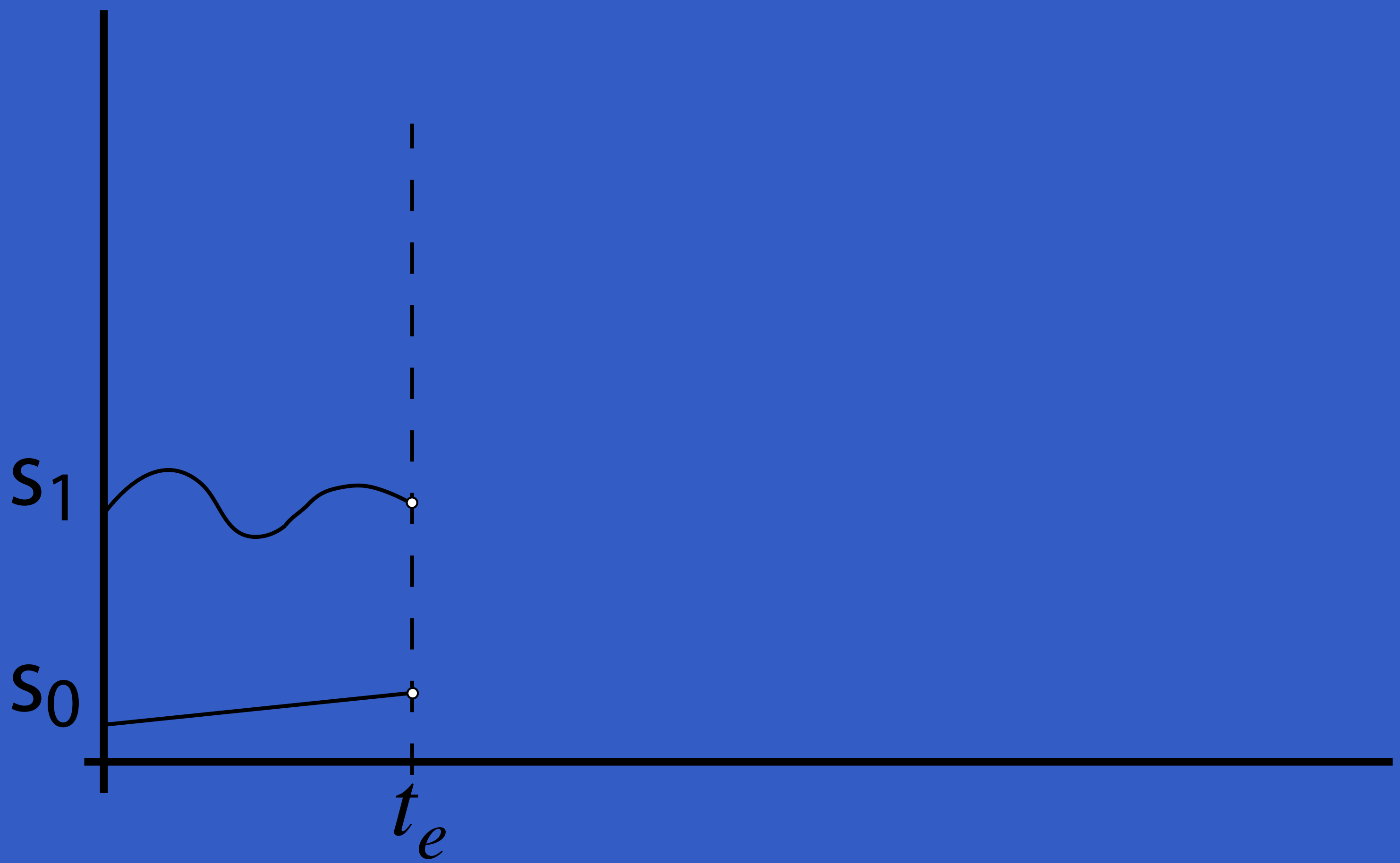
- Switch over ***collections*** of signal functions.
- On event, “freeze” running signal functions into collection of signal function ***continuations***.
- Modify collection as needed and switch back in.

```
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)
```

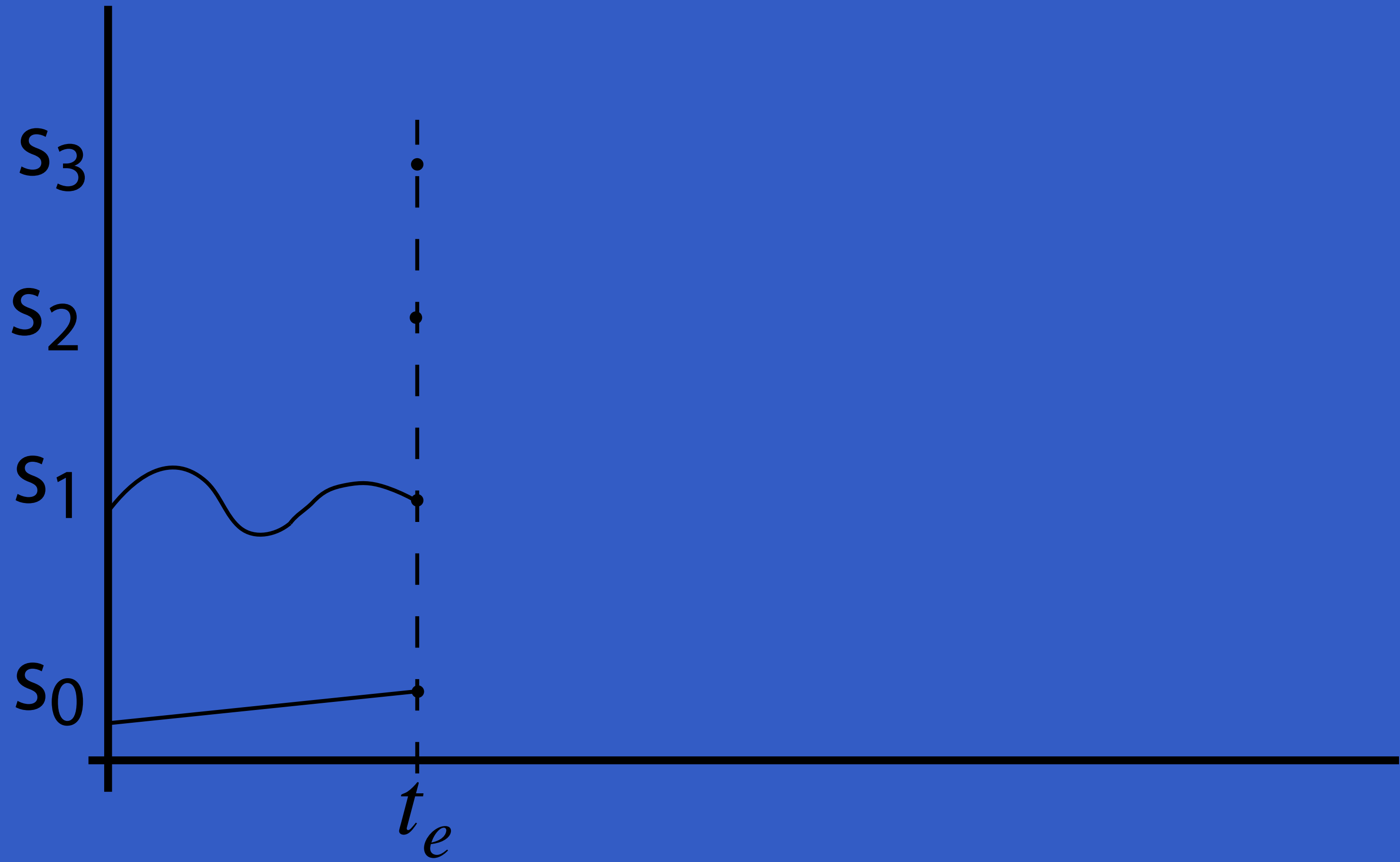
Function yielding SF to switch into

Dynamic Signal Function Collections

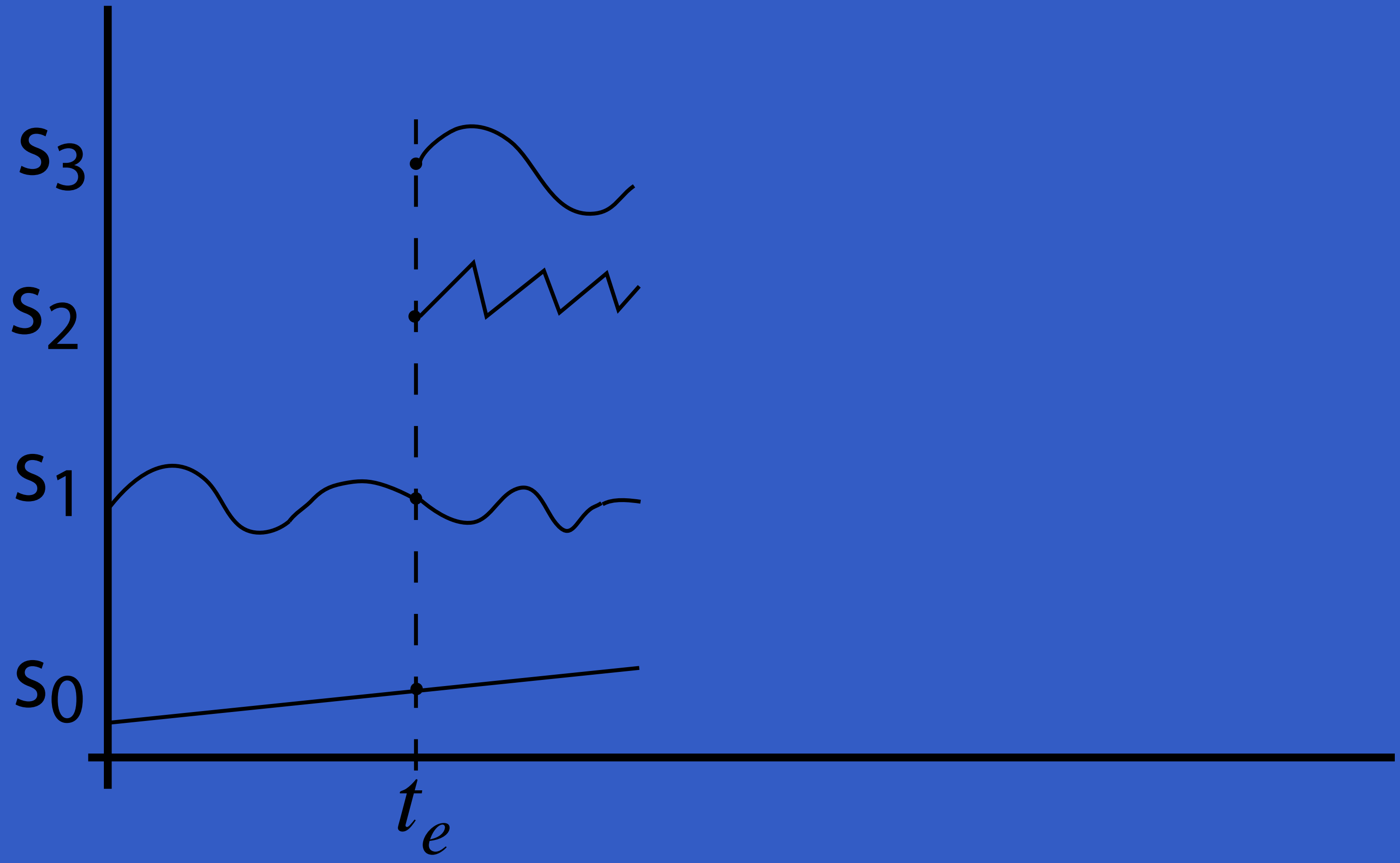
Dynamic Signal Function Collections



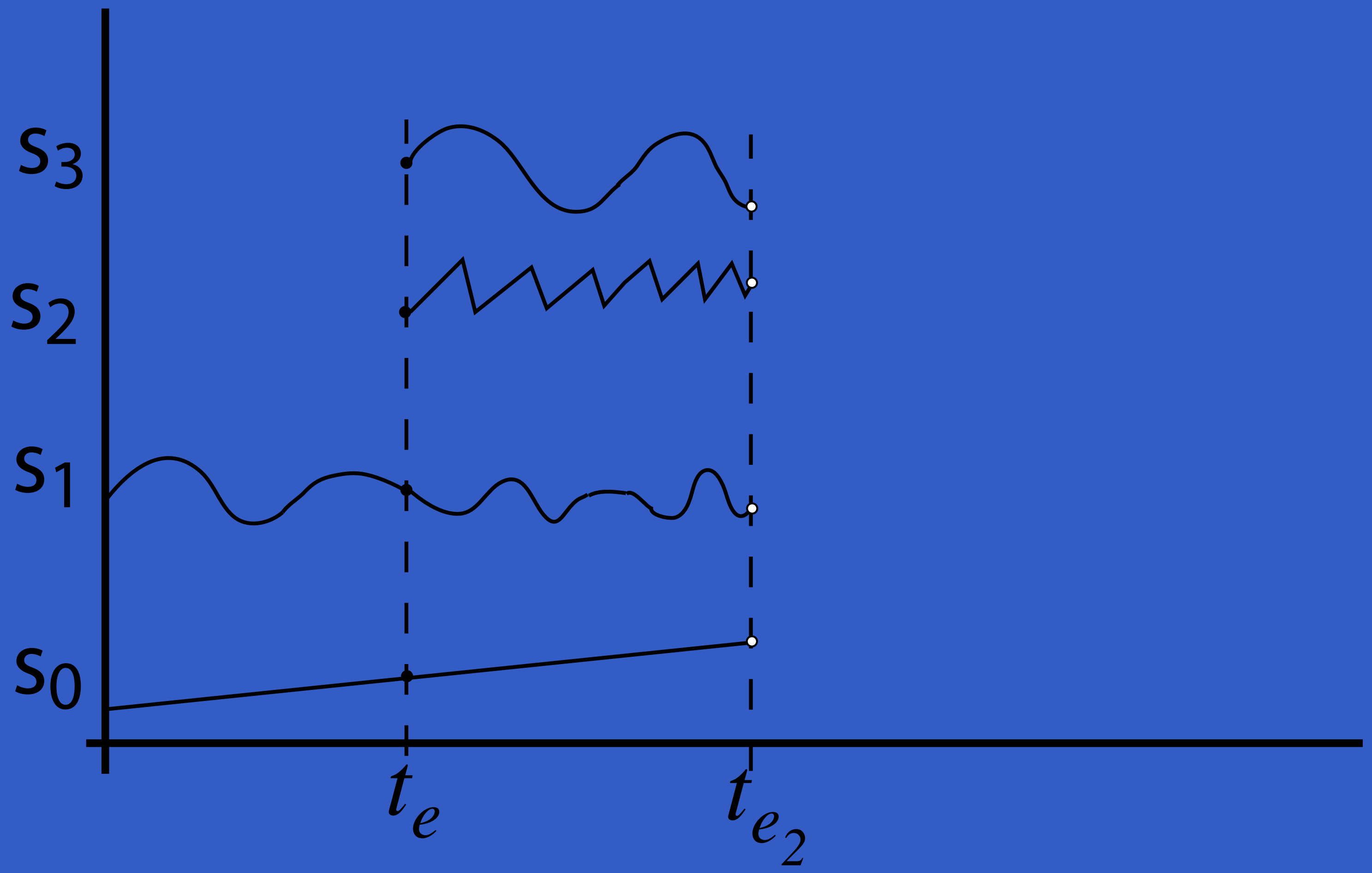
Dynamic Signal Function Collections



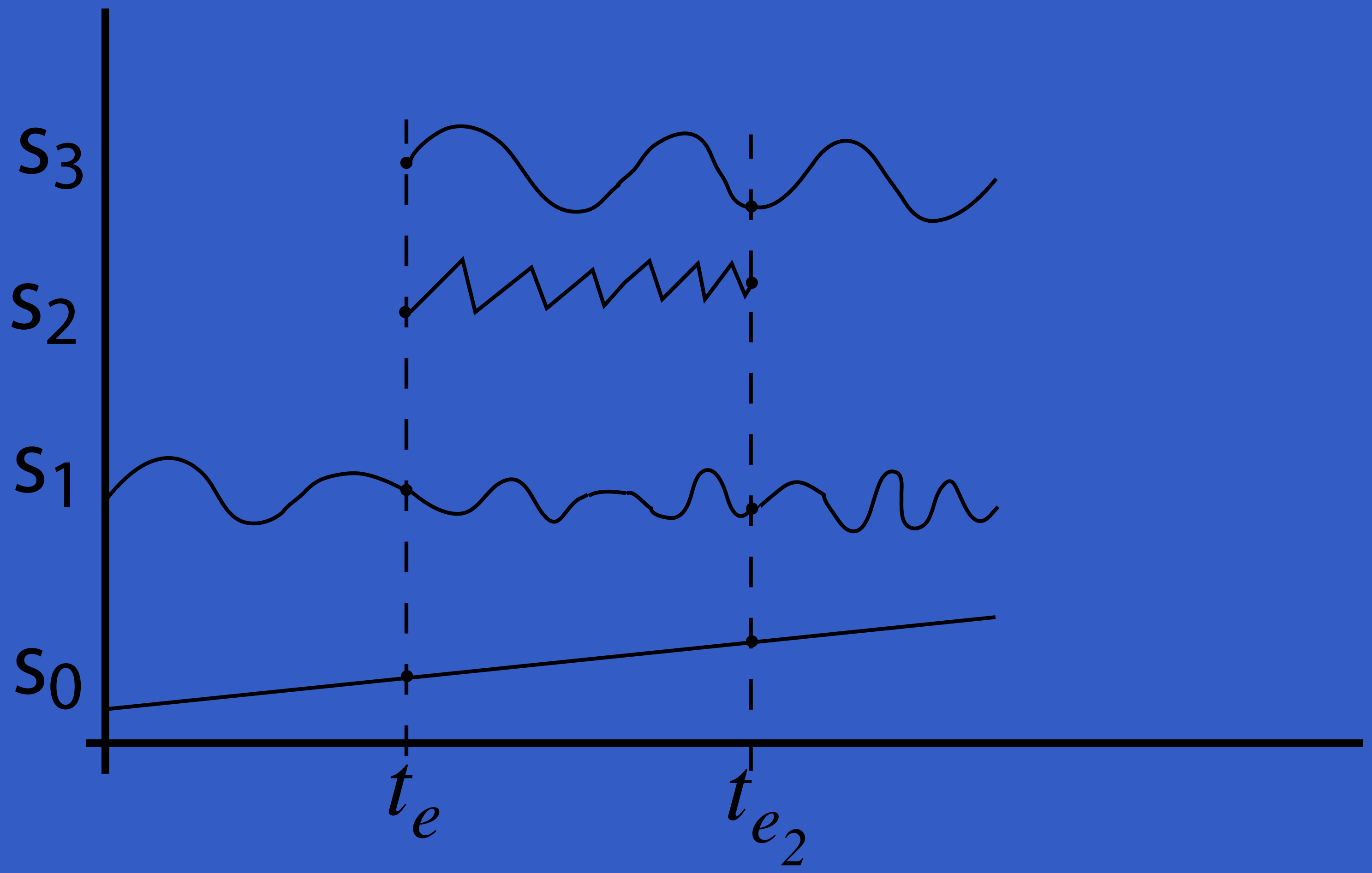
Dynamic Signal Function Collections



Dynamic Signal Function Collections

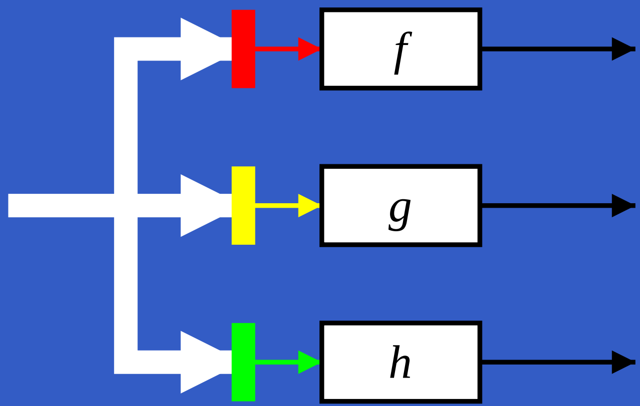


Dynamic Signal Function Collections



Routing (1)

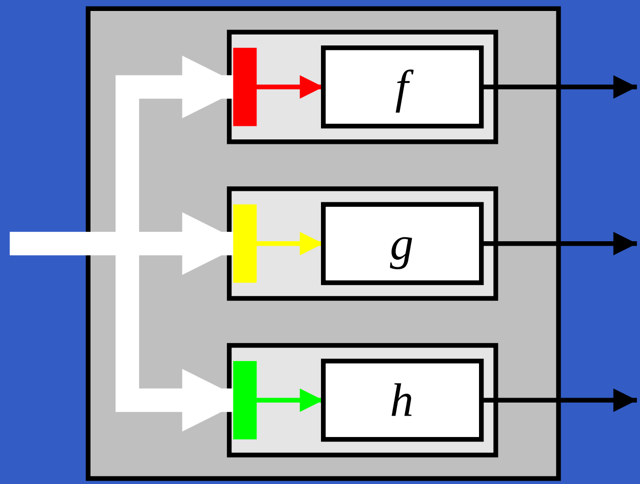
How can flexible communication be achieved?



- Input filtering (+ feedback) is enough.

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)

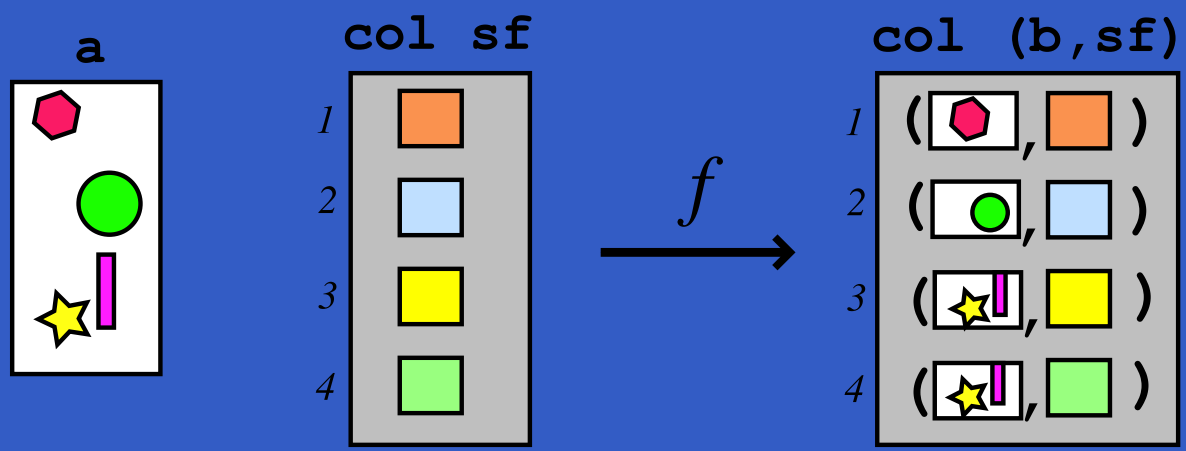
Idea:

- Generalized `pSwitch` responsible for routing; obviates need for composition.

Routing (2)

Idea:

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



pSwitch

```
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:

```
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

```
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 `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`