Modeling User Interfaces in a Functional Language

Antony Courtney

Advisor:	Paul Hudak
Committee:	John Peterson
	Zhong Shao
	Conal Elliott
Acknowledgement:	Henrik Nilsson

Thesis

Thesis:

Functional Reactive Programming (FRP) provides a suitable basis for writing rigorous executable specifications of Graphical User Interfaces.

Overview

- Background / Motivation
- Foundations:
  - Yampa – adaptation of FRP to Arrows framework
  - Fruit – GUI model based on Yampa
- Small Example
- Extensions
  - Continuations and Dynamic Collections
- Larger Examples
- Conclusions

Background / Motivation (I)

- **GUI Programming is difficult!**
  [Myers 1993] gives some reasons:
  - Graphics, usability testing, concurrency, …
- GUI builders only help with the superficial challenges (visual layout)
  - still have to write code for interactive behavior
  - programming model is still "spaghetti" of callbacks [Myers 1991]

- Historically: Many programming problems became much easier once the theoretical foundations were understood.
  - parsing before BNF [Padua 2001], relational DB model [Codd 1970], …
- We need:

  A rigorous formal basis for GUI programming.

Related Work (I): Formal Models

Lots of formal approaches to UI specification:

  - Task Models / ConcurTaskTrees (Paterno)
  - Petri Nets / Interactive Cooperative Objects (Palanque)
  - Model-based IDEs: HUMANOID / MASTERMIND (Szekely)

- Emphasis: UI analysis, design, evaluation
  - My primary interest: UI **implementation.**
- Not full programming languages:
  - **Specifications not directly executable.**
  - What *doesn't* get modeled? (input devices? graphics? layout?)
  - Model-based IDEs: Semantics of generated programs?

    [Szekeley 95]: "a lot of the semantics of the model is implicit in the way the tools make use of the attributes being modeled."

Related Work (II) : FP

- Historically: strong connection between functional programming and formal modeling.
- But: functional languages were once considered "weak" for expressing I/O and user interaction.
- The "solution": **monads** / monadic IO [Wadler 1989]

$$putStrLn :: String \rightarrow IO\ ()$$
$$getStrLn :: IO\ String$$

we read: $f :: IO\ a$
as: "f performs some IO action and then returns an a."

- type distinction between pure computations and imperative actions.
- very useful technique for structuring functional programs.

Background: FP and Monads

Q: But what is the ***denotation*** of type (IO a) ?

Answer:

$$[\![IO\ \ a]\!] \;=\; World \;\rightarrow\; (World, a)$$

Q: What are the formal properties of "World"?!

Answer: ???

Monadic IO tells us *where* IO actions occur in our programs, but does nothing whatsoever to deepen our understanding of such actions.

Background / Motivation

- Our goals:
  1. A simple *functional model* of GUIs that:
     - Makes no appeal to imperative programming.
     - Uses only formally tractable types.
     - Expressive enough to describe real GUIs:
       - $\Rightarrow$ model input devices and graphics explicitly.
  2. A concrete *implementation* of this model:

     …so that our specifications are *directly executable.*

Summary of Contributions

- Yampa (Chapters 3-5, [Courtney & Elliott 2001], [Nilsson, Courtney, Peterson 2002]):
  - A purely functional model of reactive systems based on synchronous dataflow.
  - Based on adapting Fran [Elliott & Hudak 1997] and FRP [Wan & Hudak 2001] to Arrows Framework [Hughes 2000].
  - Simple denotational and operational semantics.
- Haven (Chapter 6):
  - A functional model of 2D vector graphics.
- Fruit (Chapters 7, 10, 11, [Courtney & Elliott 2001], [Courtney 2003]):
  - A GUI library defined solely using Yampa and Haven.
- Dynamic Collections (Ch. 8, [Nilsson, Courtney, Peterson 2002]):
  - Continuation-based and parallel switching primitives

Overview

- Background / Motivation
- Foundations:
  - Yampa – adaptation of FRP to Arrows framework
  - Fruit – GUI model based on Yampa
- Small Example
- Extensions
  - Continuations and Dynamic Collections
- Larger Examples
- Conclusions

Yampa

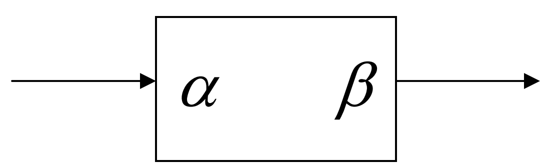
- An implementation of Functional Reactive Programming (FRP) in Haskell, using *Arrows* Framework [Hughes 2000].
- Key Concepts:
  - **Signal:** function from continuous time to value:

$$\mathbf{Signal}\ \alpha \quad = \quad Time \ \rightarrow \ \alpha$$

  - **Signal Function:** function from Signal to Signal:

$$\mathbf{SF}\ \alpha\ \beta \quad = \quad Signal\ \alpha \ \rightarrow \ Signal\ \beta$$

Visually:



Yampa Programming

- Implementation provides:
  - a number of **primitive** SFs
  - (arrow) **combinators** for composing SFs
- Programming consists of:
  - composing SFs into a data flow graph.

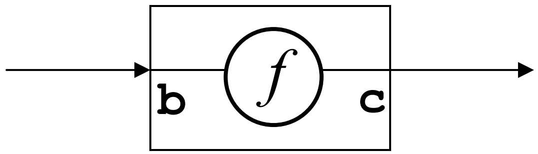
    ...much like composing a digital circuit.

- Implementation approximates continuous-time semantic model with discrete sampling.

Arrow Combinators for SFs

- Lifting (point-wise application):

$$arr :: (a \to b) \to SF\ a\ b$$

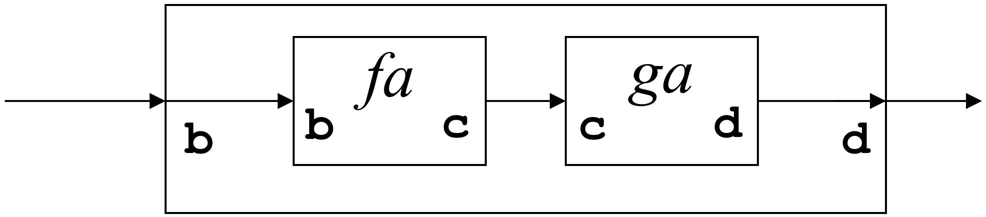


arr f

$$[\![arr\ f]\!] = \lambda s.\lambda t.[\![f]\!](s\ t)$$

- Serial Composition:

$$\ggg :: SF\ b\ c \to SF\ c\ d \to SF\ b\ d$$



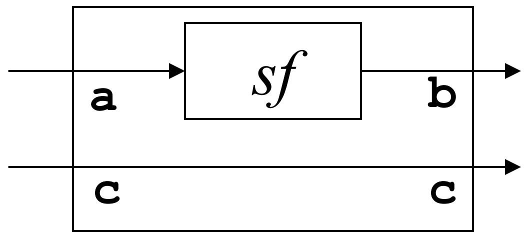
fa **>>>** *ga*

$$[\![fa \ggg ga]\!] = [\![ga]\!] \circ [\![fa]\!]$$

Other Arrow Combinators

- Use *tuples* to group signals:

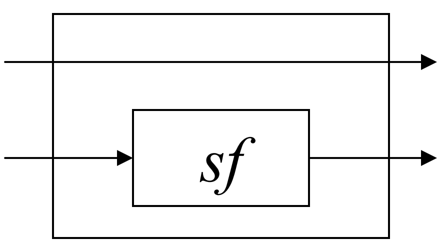
$$\mathit{first} :: SF\ a\ b \rightarrow SF\ (a, c)\ (b, c)$$



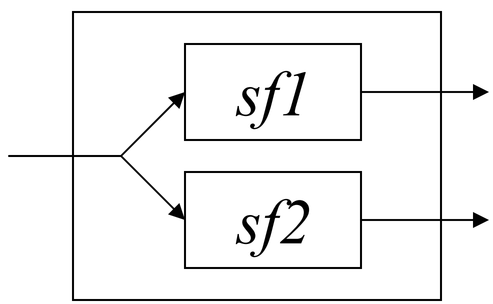
$\mathit{first}\ \mathit{sf}$

$$[\![\mathit{first}\ \mathit{sf}]\!] \;=\; \lambda s.\mathit{pairZ}\ ([\![\mathit{sf}]\!]\ (\mathit{fstZ}\ s))\ \ (\mathit{sndZ}\ s)$$

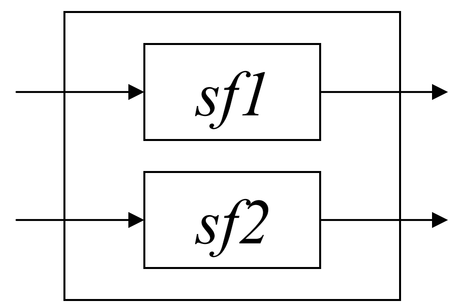
- Other (derived) combinators to form arbitrary digraphs:



$\mathit{second}\ \mathit{sf}$



$\mathit{sf1}\ \&\&\&\ \mathit{sf2}$

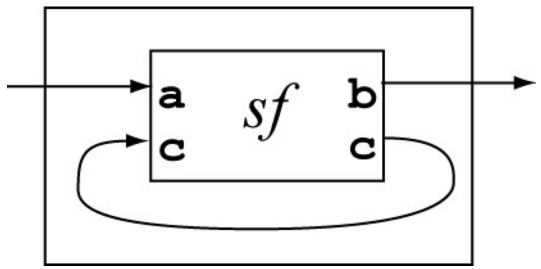


$\mathit{sf1}\ {*}{*}{*}\ \mathit{sf2}$

Feedback

- Can define cyclic graphs with $loop$:

$$loop :: SF\ (a, c)\ (b, c) \rightarrow SF\ a\ b$$



$loop\ sf$

$$[\![loop\ fa]\!] = \lambda s.fstZ(\mathbf{Y}(\lambda r.[\![sf]\!](pairZ\ s\ (sndZ\ r))))$$

- Allows an SF to accumulate local state

  …just like a digital circuit (flip flop).

- Delay needed on feedback signal to avoid a “black hole”.

  …just like a digital circuit.

Discrete Event Sources

- A *discrete event* is a condition that occurs at discrete points in time
  - pressing of a button
  - rising/falling edge of a Boolean signal
- A *possible occurrence* modeled by type:

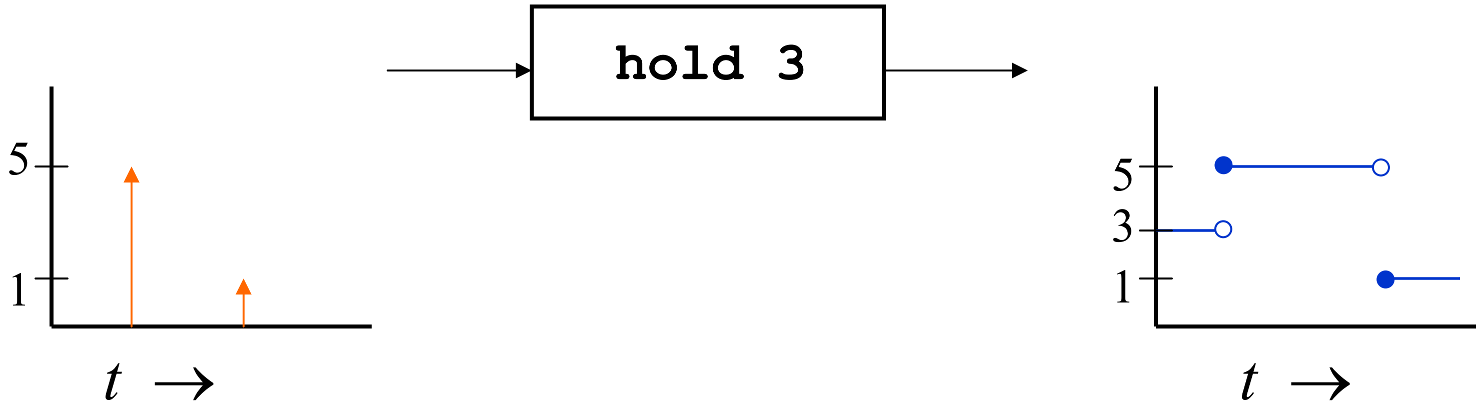
$$\begin{aligned} \textbf{data}\ Event\ a &= EvOcc\ a \\ &\mid\ NoEvent \end{aligned}$$

- Some basic operations (used point-wise):

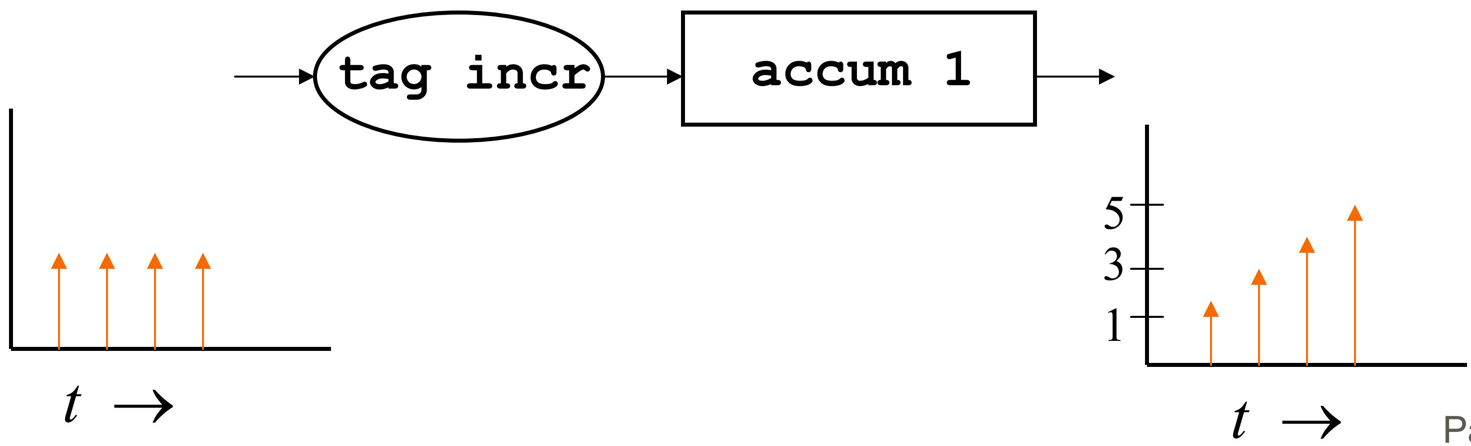
$$\begin{aligned} tag &:: b \rightarrow Event\ a \rightarrow Event\ b \\ mergeE &:: Event\ a \rightarrow Event\ a \rightarrow Event\ a \\ gate &:: Event\ a \rightarrow Bool \rightarrow Event\ a \end{aligned}$$

Event Processors

$$hold :: a \rightarrow SF\ (Event\ a)\ a$$



$$accum :: a \rightarrow SF\ (Event\ (a \rightarrow a))\ (Event\ a)$$

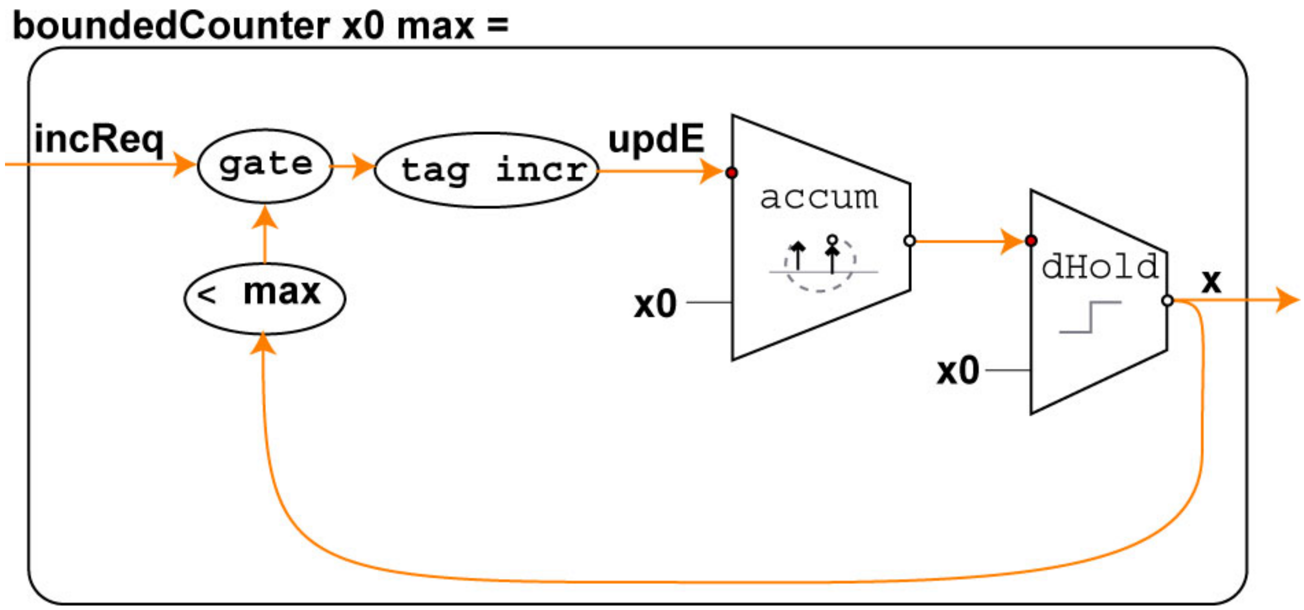


Example: A Bounded Counter

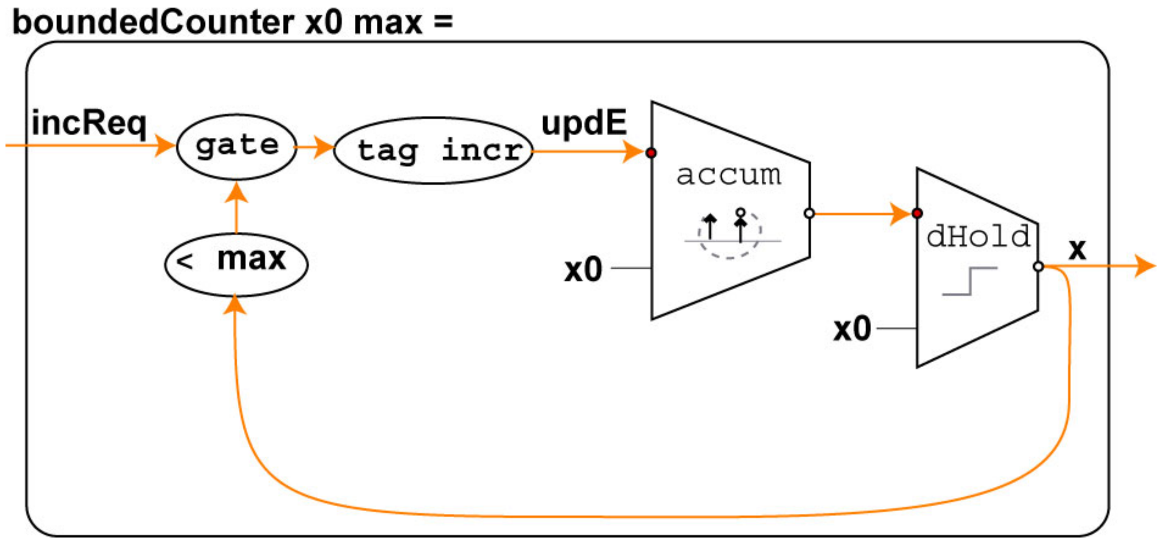
```
bc :: Int -> Int -> SF (Event ()) Int
bc x0 max = …
```

- Initial value: **`x0`**
- Increment on each event until **`max`** reached

Implementation:



Arrows Syntax [Paterson 2001]



$$bc :: Int \rightarrow Int \rightarrow SF\ (Event\ ())\ Int$$

$$bc\ x0\ max\ =$$

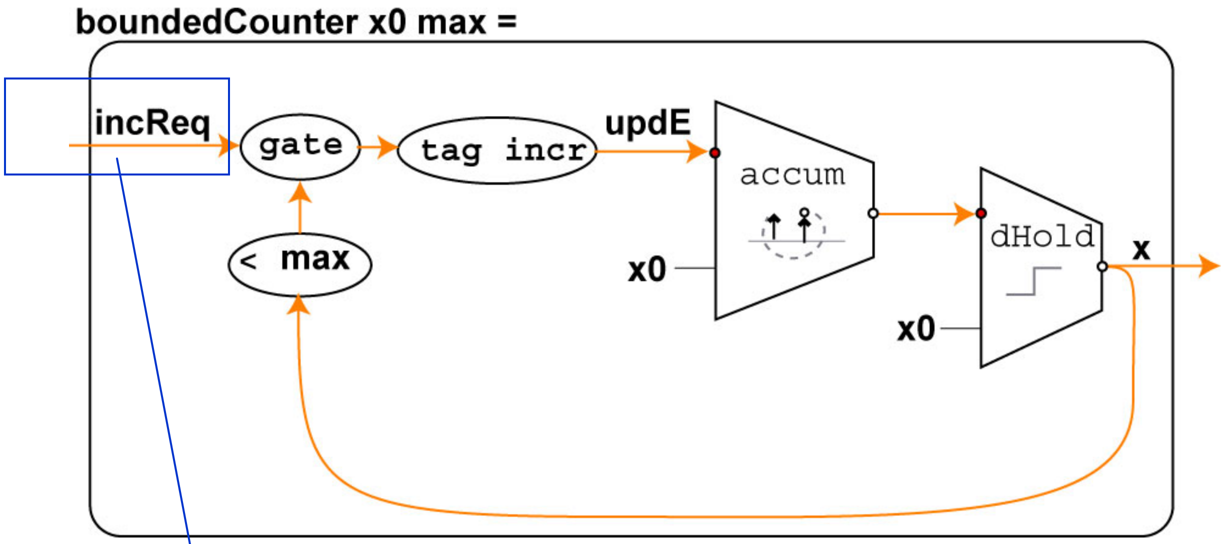
$$\quad \mathbf{proc}\ incReq \rightarrow \mathbf{do}$$

$$\qquad \mathbf{let}\ updE = (incReq\ `gate`\ (x < max))\ `tag`\ incr$$

$$\qquad updE \succ\!\!- dAccumHold\ x0 \rightarrow x$$

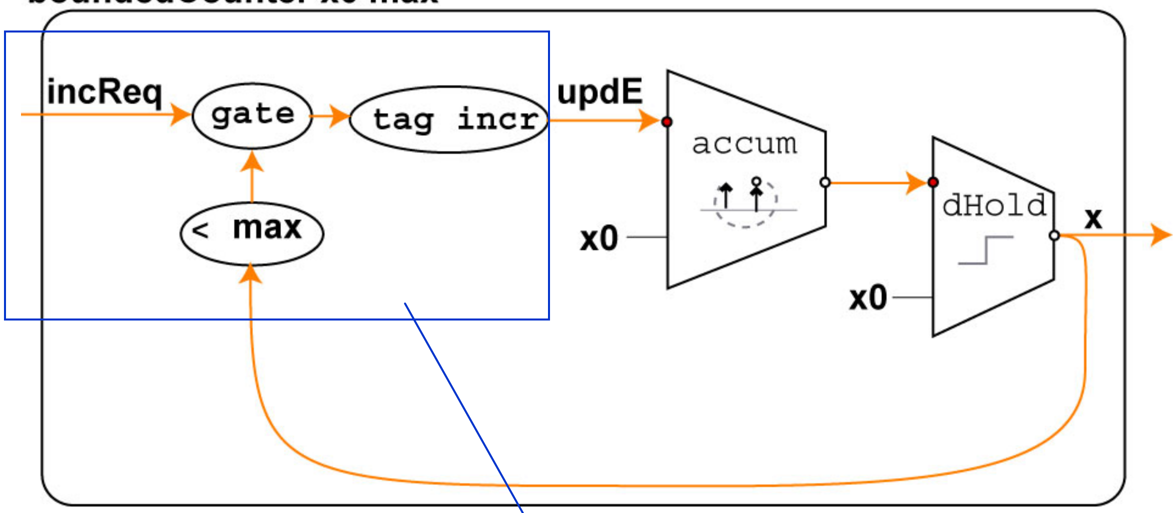
$$\qquad x \qquad \succ\!\!- returnA$$

Arrows Syntax [Paterson 2001]



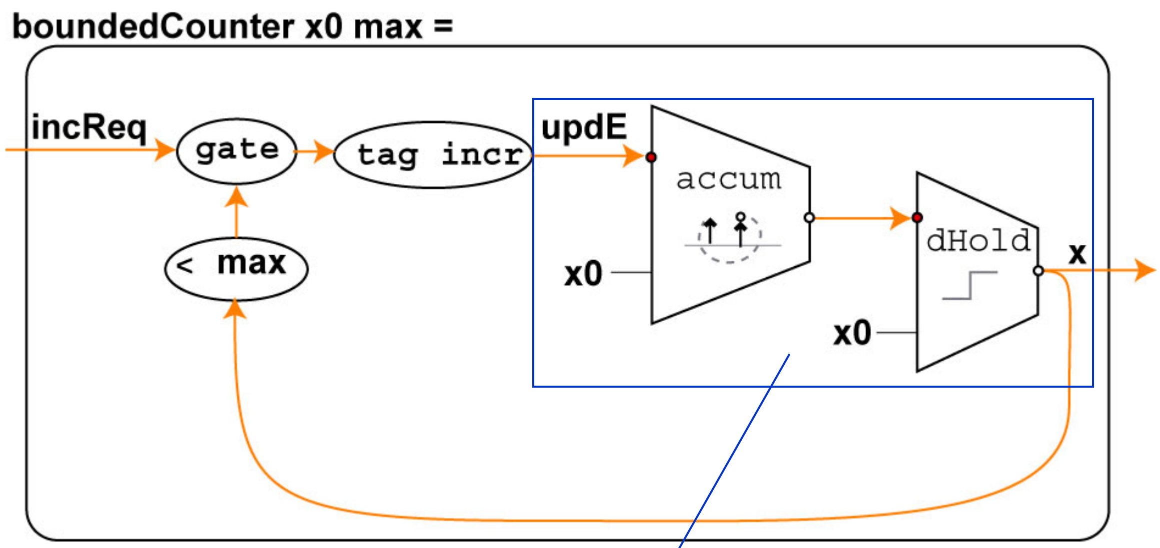
$$
\begin{array}{l}
bc :: Int \rightarrow Int \rightarrow SF\ (Event\ ())\ Int \\
bc\ x0\ max = \\
\quad \mathbf{proc}\ \boxed{incReq} \rightarrow \mathbf{do} \\
\qquad \mathbf{let}\ updE = (incReq\ `gate`\ (x < max))\ `tag`\ incr \\
\qquad updE \succ\!\!- dAccumHold\ x0 \rightarrow x \\
\qquad x \quad \succ\!\!- returnA
\end{array}
$$

Arrows Syntax [Paterson 2001]



$$
\begin{array}{l}
bc :: Int \rightarrow Int \rightarrow SF\ (Event\ ())\ Int \\
bc\ x0\ max\ = \\
\quad \textbf{proc}\ incReq \rightarrow \textbf{do} \\
\qquad \textbf{let}\ updE = \boxed{(incReq\ `gate`\ (x < max))\ `tag`\ incr} \\
\qquad updE \succ\!\!- dAccumHold\ x0 \rightarrow x \\
\qquad x \quad\quad \succ\!\!- returnA
\end{array}
$$

Arrows Syntax [Paterson 2001]



$$bc :: Int \rightarrow Int \rightarrow SF\ (Event\ ())\ Int$$

$$bc\ x0\ max\ =$$

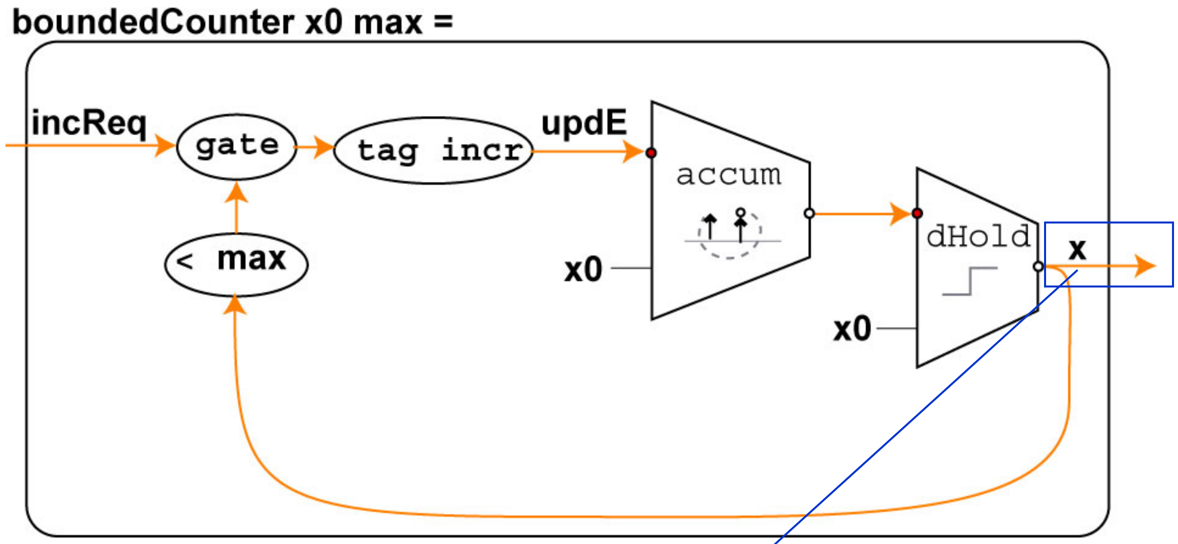
$$\quad \mathbf{proc}\ incReq \rightarrow \mathbf{do}$$

$$\quad\quad \mathbf{let}\ updE = (incReq\ `gate`\ (x < max))\ `tag`\ incr$$

$$\quad\quad \boxed{updE \succ\!\!- dAccumHold\ x0 \rightarrow x}$$

$$\quad\quad x \quad \succ\!\!- returnA$$

Arrows Syntax [Paterson 2001]



$$bc :: Int \rightarrow Int \rightarrow SF\ (Event\ ())\ Int$$

$$bc\ x0\ max\ =$$

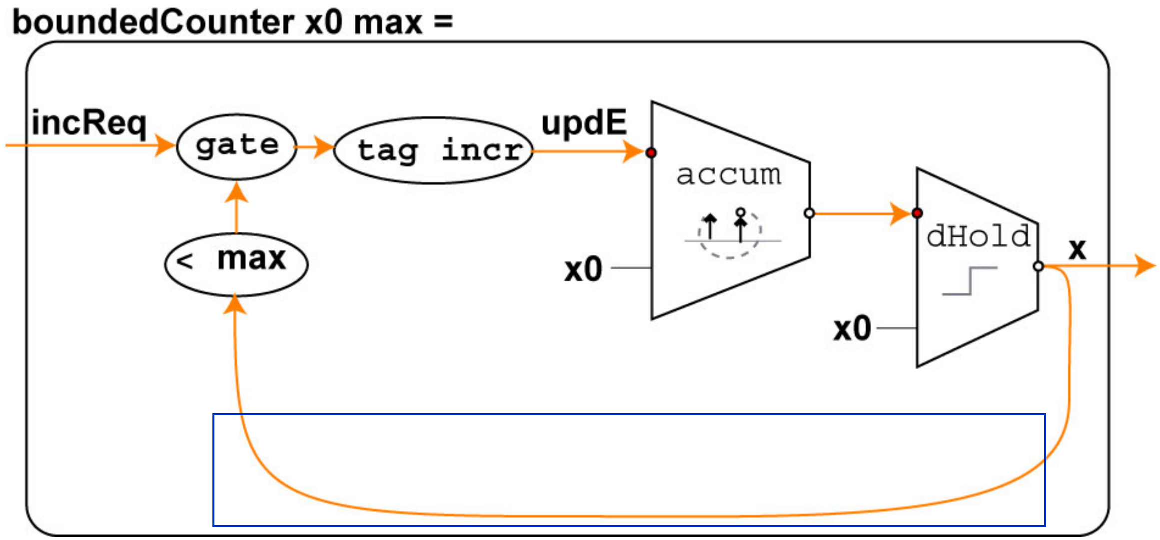
$$\quad \mathbf{proc}\ incReq \rightarrow \mathbf{do}$$

$$\qquad \mathbf{let}\ updE = (incReq\ `gate`\ (x < max))\ `tag`\ incr$$

$$\qquad updE \succ\!\!- dAccumHold\ x0 \rightarrow x$$

$$\qquad \boxed{x \qquad \succ\!\!- returnA}$$

Arrows Syntax [Paterson 2001]



$$bc :: Int \to Int \to SF\ (Event\ ())\ Int$$

$$bc\ x0\ max\ =$$

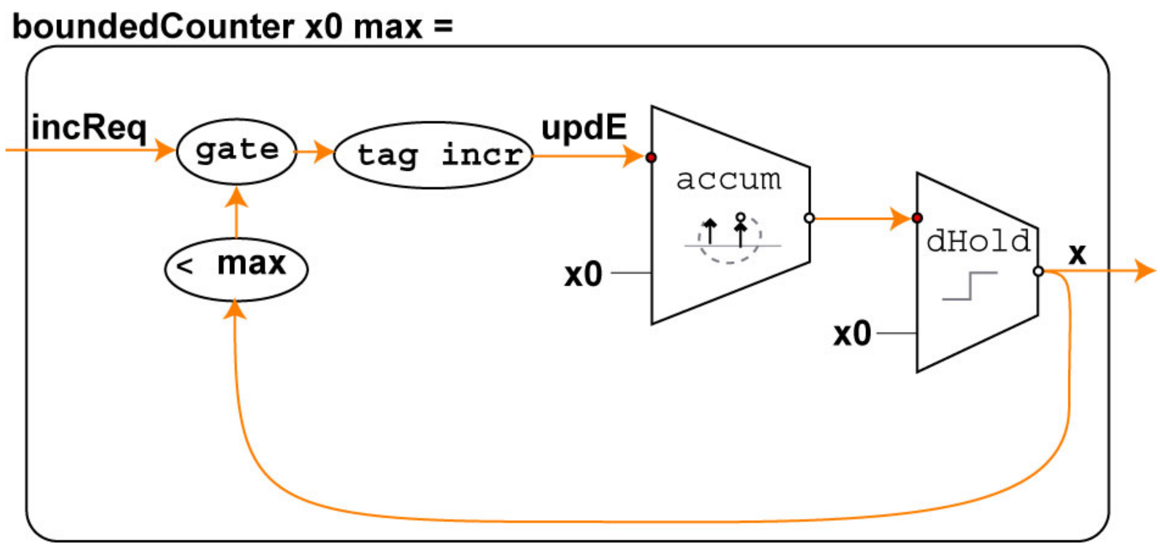
$$\quad \mathbf{proc}\ incReq \to \mathbf{do}$$

$$\qquad \mathbf{let}\ updE = (incReq\ \text{`}gate\text{`}\ (x < max))\ \text{`}tag\text{`}\ incr$$

$$\qquad updE \succ\!\!- dAccumHold\ x0 \to x$$

$$\qquad x \succ\!\!- returnA$$

Concrete Syntax

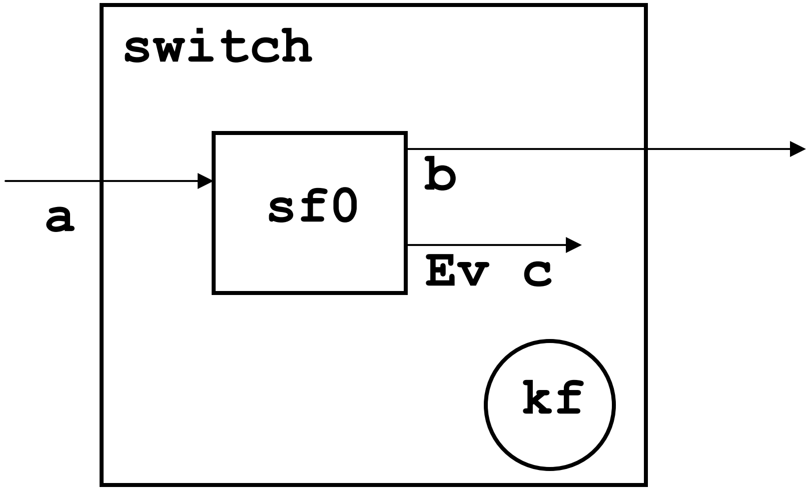



```
bc :: Int -> Int -> SF (Event ()) Int
bc x0 max = loop (arr gateReq >>>
                     dAccumHold x0 >>> arr dup)
  where gateReq :: (Event (),Int) -> Event (Int -> Int)
        gateReq (incReq,n) =
                     (incReq `gate` (n < max)) `tag` incr
        dup x = (x,x)
```

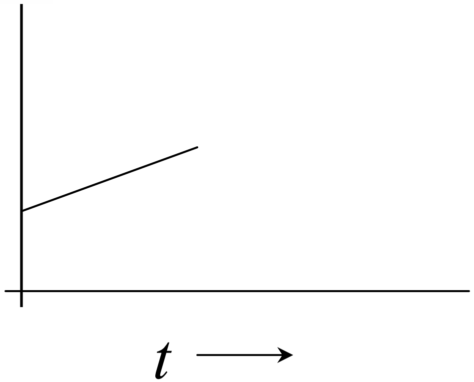
Basic Switching

- *switch* combinator switches from one SF to another on event occurrence:

$$\begin{aligned} switch :: & \; SF \;\; a \; (b, Event \;\; c) \\ & \rightarrow (c \rightarrow SF \;\; a \;\; b) \\ & \rightarrow SF \;\; a \;\; b \end{aligned}$$



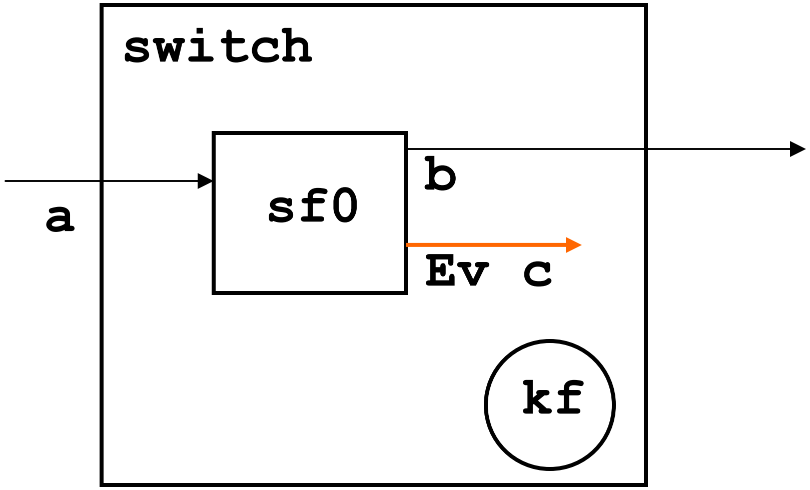
switch sf0 kf

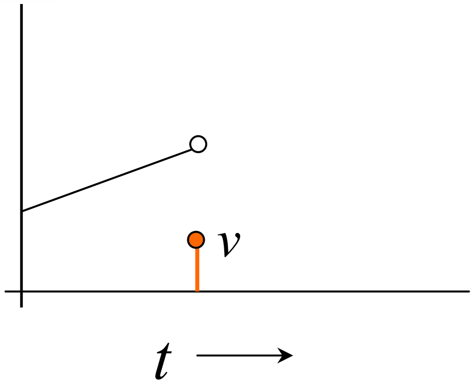


Basic Switching

- *switch* combinator switches from one SF to another on event occurrence:

$$\begin{aligned} switch :: SF\ a\ (b, Event\ c) \\ \rightarrow (c \rightarrow SF\ a\ b) \\ \rightarrow SF\ a\ b \end{aligned}$$

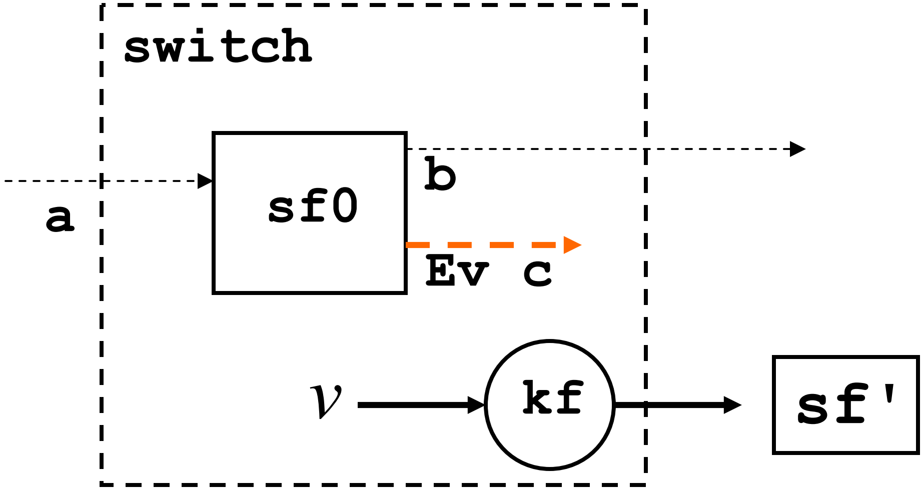


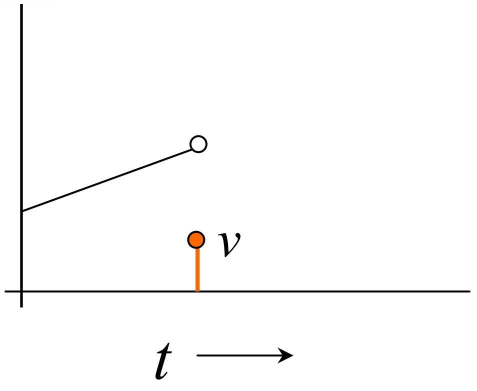


Basic Switching

- *switch* combinator switches from one SF to another on event occurrence:

$$
\begin{aligned}
switch :: & \; SF \; a \; (b, Event \; c) \\
& \rightarrow (c \rightarrow SF \; a \; b) \\
& \rightarrow SF \; a \; b
\end{aligned}
$$

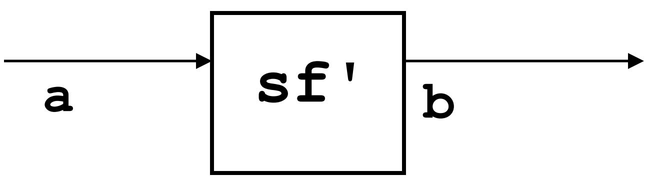


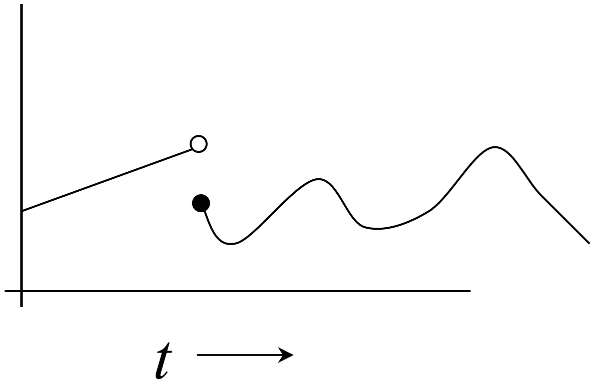


Basic Switching

- *switch* combinator switches from one SF to another on event occurrence:

$$
\begin{aligned}
switch :: SF\ a\ (b, Event\ c) \\
\rightarrow (c \rightarrow SF\ a\ b) \\
\rightarrow SF\ a\ b
\end{aligned}
$$





A Brief History of Time (in FRP)

Evolution of the FRP semantic model (Chapter 2):

- Fran [Elliott & Hudak 1997]:

  Behaviors are time-varying values ("**signals**"):

  ```
  Behavior a ≈ Time -> a
  ```

- SOE FRP [Hudak 2000] [Wan & Hudak 2000]:

  Behaviors are functions of start time ("**computations**"):

  ```
  Behavior a ≈ Time -> Time -> a
  ```

  Motivation: Avoid inherent space-time leak in:

  ```
  x = y `switch` (e -=> z)
  ```

Evolution of Yampa

- Fran's Behavior semantics:
  - highly expressive
  - difficult to implement efficiently (**space/time leaks**)
- SOE FRP's Behavior semantics:
  - Efficient, but basic model limited in expressive power
  - Attempt to recover expressive power: *runningIn*
    - Captures a running signal as a Behavior
- SOE FRP + *runningIn*:
  - No type level distinction between *signals* and *signal computations (Behaviors).*
    - *very* confusing in practice.
  - Implementation couldn't handle recursive definitions.

What Yampa Gives Us

- A clear distinction between

  Signals:

  $$\text{Signal } \alpha \quad = \quad Time \;\rightarrow\; \alpha$$

  and **Signal Functions:**

  $$\text{SF } \alpha\ \beta \quad = \quad Signal\ \alpha \;\rightarrow\; Signal\ \beta$$

  …and ways to express both.
- Arrows framework:
  - Arrow laws for reasoning about programs
  - Std. library of combinators for specifying plumbing
    - Explicit combinators help avoid time/space leaks.
- Arrows syntactic sugar:
  - Concrete syntax for data flow diagrams.
  - Alleviates syntactic awkwardness of combinator-based design.

Overview

- Background / Motivation
- Foundations:
  - Yampa – adaptation of FRP to Arrows framework
  - Fruit – GUI model based on Yampa
- Small Example
- Extensions
  - Continuations and Dynamic Collections
- Larger Examples
- Conclusions

Brief Aside: Graphics Model

Haven (Chapter 6):

- Typed, functional interface to 2D vector graphics

$$\textbf{type}\ Image = (Point \to Color, Region)$$

  - Programming model owes much to *Pan* [Elliott 2001]

Main Idea:

- Try to provide minimal set of primitives
- Provide higher-level functionality by composing primitives.

Portable, Functional *Interface:*

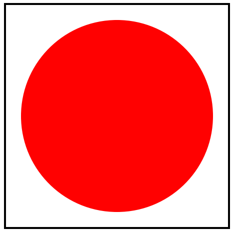
- Implementations: Java2D, FreeType

Compositionality in Haven

- **Instead of:** $fillShape :: Color \rightarrow Shape \rightarrow Image$

e.g.:

$$fillShape\ red\ (circle\ 20) =$$



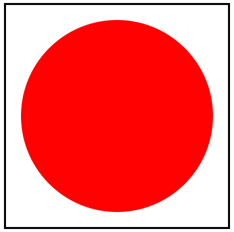
- **Haven provides:**

$$monochrome :: Color \rightarrow Image$$

$$imgCrop :: Shape \rightarrow Image \rightarrow Image$$

$$monochrome\ red =$$

$$imgCrop\ (circle\ 20)\ (monochrome\ red) =$$



fillShape vs. imgCrop

imgCrop is far more versatile than *fillShape*:

- Use *imgCrop* on *any* image:
  - color gradients:

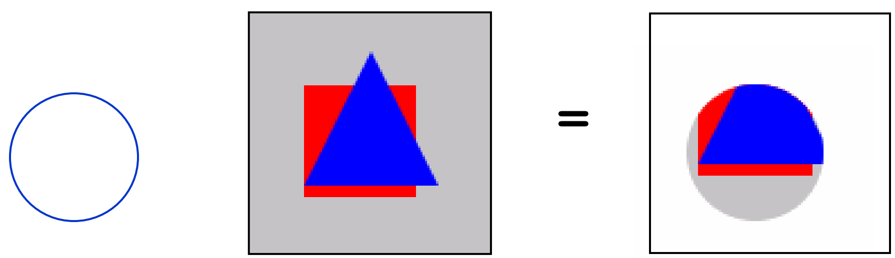
$$gradient :: Point \rightarrow Color \rightarrow Point \rightarrow Color \rightarrow Image$$

- Compose crop operations:

imgCrop s1 ((imgCrop s2 …) `over`

(imgCrop s3 (imgCrop …)))

`imgCrop`



Fruit

What is a GUI?

- GUIs are Signal Functions:

$$\textbf{type}\ GUI\ a\ b = SF\ (GUIInput, a)\ (Picture, b)$$

- Signal types:

  $GUIInput$ – keyboard and mouse state

  $Picture$ – visual display ($Image$)

  a,b – auxiliary semantic input and output signals

- $GUIInput$:

$$\begin{aligned}\textbf{data}\ Mouse = Mouse\{\, & mpos :: Point, \\ & lbDown, rbDown :: Bool \\ \}&\end{aligned}$$

$$\textbf{type}\ GUIInput = (Maybe\ Kbd, Maybe\ Mouse)$$

Fruit: Components and Layout

- Aux. signals connect GUI to rest of application.
- Components (slightly simplified interfaces):
  - Text Labels:

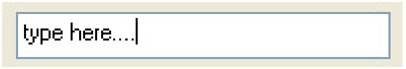
    $$label :: GUI\ String\ ()$$

  - Buttons:

    $$button :: String \to GUI\ Bool\ (Event\ ())$$

  - Text fields:

    $$textField :: String \to GUI\ (Event\ String)\ (Event\ String)$$



- Layout Combinators:

$$besideGUI :: GUI\ b\ c \to GUI\ d\ e \to GUI\ (b, d)\ (c, e)$$
$$aboveGUI :: GUI\ b\ c \to GUI\ d\ e \to GUI\ (b, d)\ (c, e)$$

Overview

- Background / Motivation
- Foundations:
  - Yampa – adaptation of FRP to Arrows framework
  - Fruit – GUI model based on Yampa
- Small Example
- Extensions
  - Continuations and Dynamic Collections
- Larger Examples
- Conclusions / Future Work

Basic Fruit Example

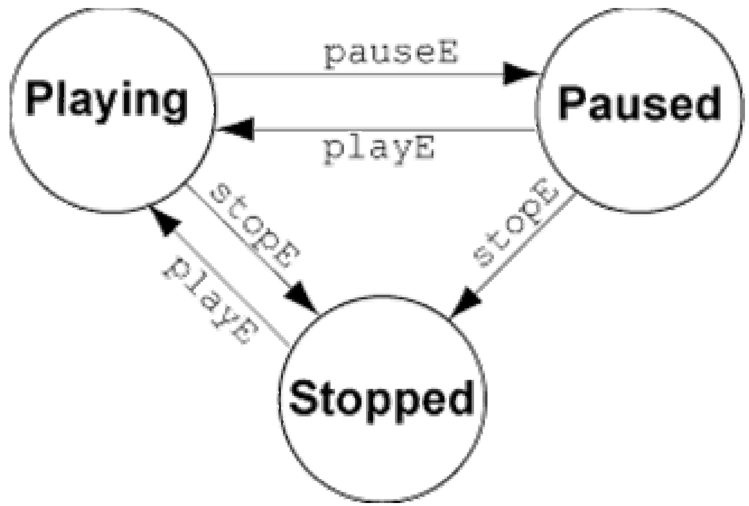
- Classic VCR-style media controller:

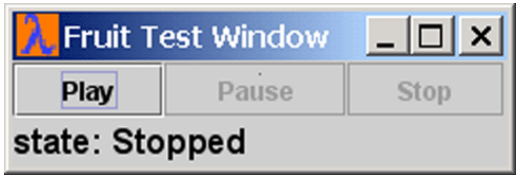
- Only enable buttons when action is valid:
  - i.e. "pause" only enabled when media is playing.
- Represent media state with:

$$\mathbf{data}\ \mathit{MediaState} = \mathit{Playing} \mid \mathit{Paused} \mid \mathit{Stopped}$$

Design

- just a simple FSM:

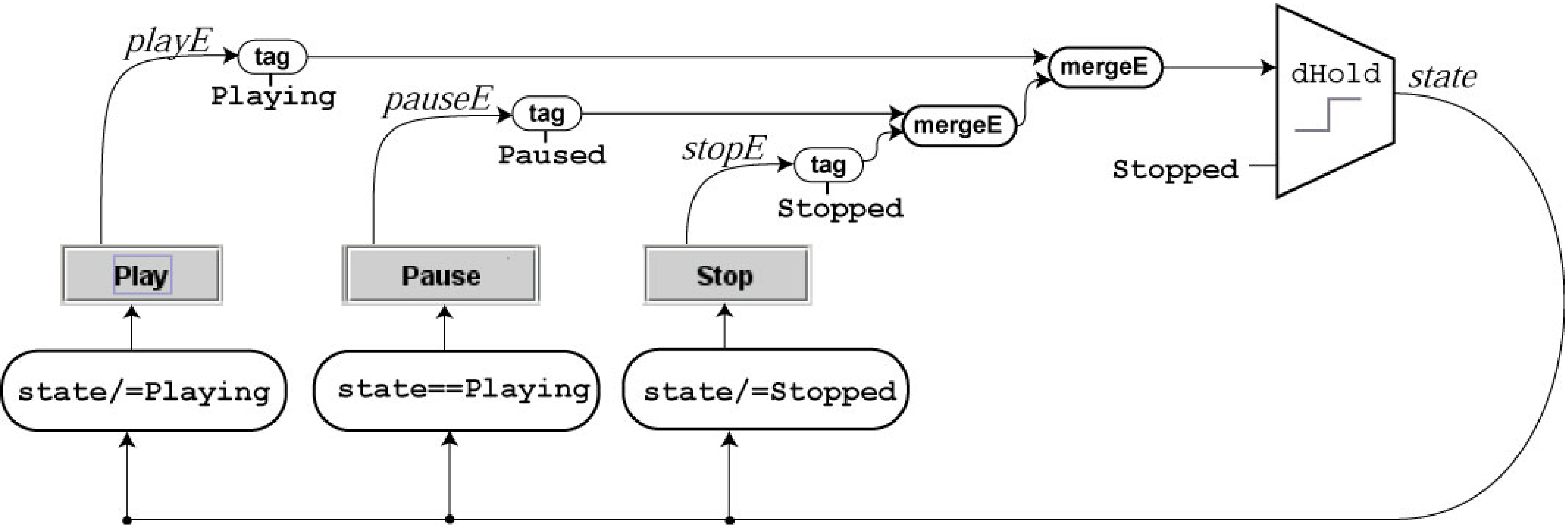


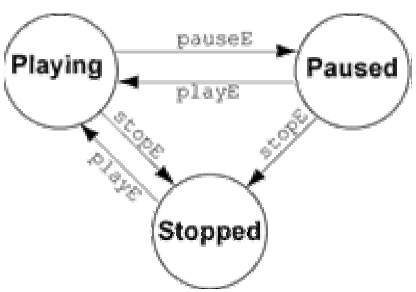


- Derive time-invariant *constraints* by inspection:
  - playE: **`state /= Playing`**
  - pauseE: **`state == Playing`**
  - stopE: **`state /= Stopped`**

Fruit Specification (Visual)

- Visually:





Fruit Specification (Textual)

$$
\begin{array}{l}
playerCtrl :: GUI\ ()\ MediaState \\
playerCtrl = hbox\ (\mathbf{proc}\ \_ \rightarrow \mathbf{do} \\
\quad (state \not\equiv Playing) \ \succ\!\!- button\ \texttt{"Play"} \ \rightarrow playE \\
\quad (state \equiv Playing) \ \succ\!\!- button\ \texttt{"Pause"} \rightarrow pauseE \\
\quad (state \not\equiv Stopped) \ \succ\!\!- button\ \texttt{"Stop"} \ \rightarrow stopE \\
\quad (mergeEs\ [tag\ playE\ Playing, tag\ pauseE\ Paused, \\
\qquad\qquad\qquad tag\ stopE\ Stopped]) \\
\qquad\qquad\qquad\qquad \succ\!\!- boxSF\ (dHold\ Stopped) \rightarrow state \\
\qquad\qquad\qquad state \succ\!\!- returnA)
\end{array}
$$

Evaluation

- The Fruit specification looks rather complicated:
  - explicit hold operator to accumulate state
  - **feedback loop!**

We can easily implement the media controller in our favorite (imperative) language and toolkit.

So we should ask:

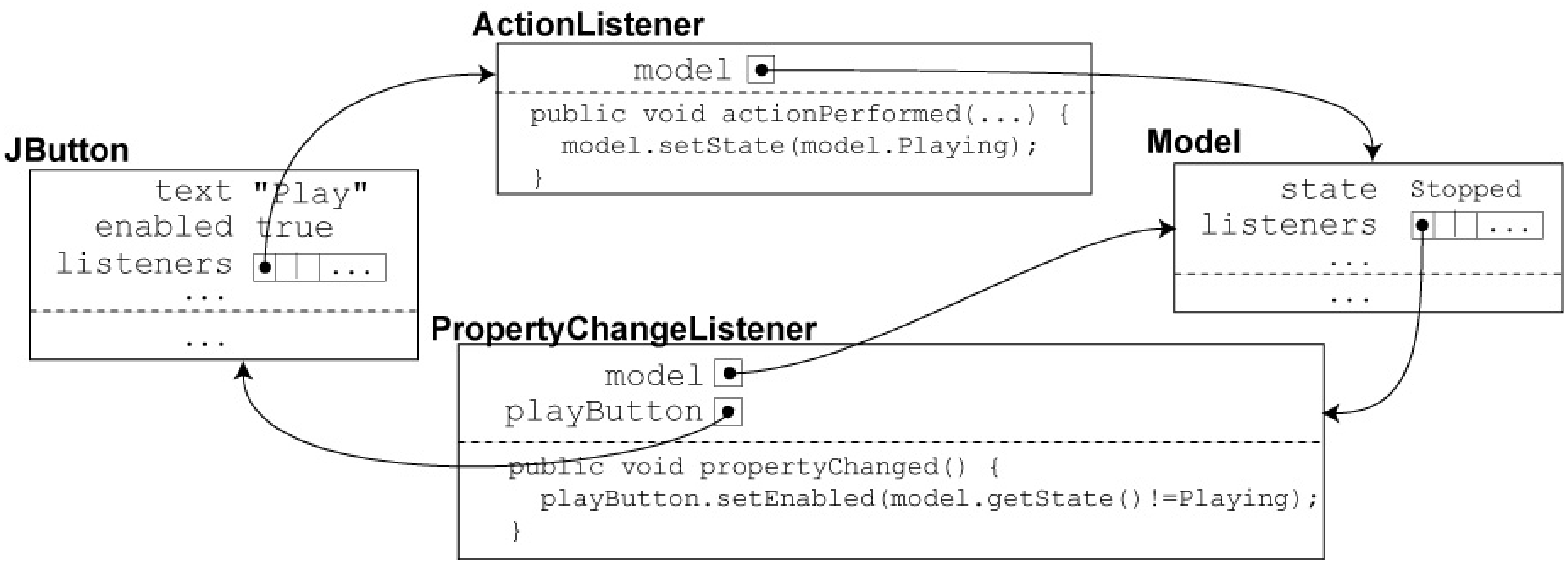
How does a Fruit specification compare to an imperative implementation?

Imperative, OO Implementation

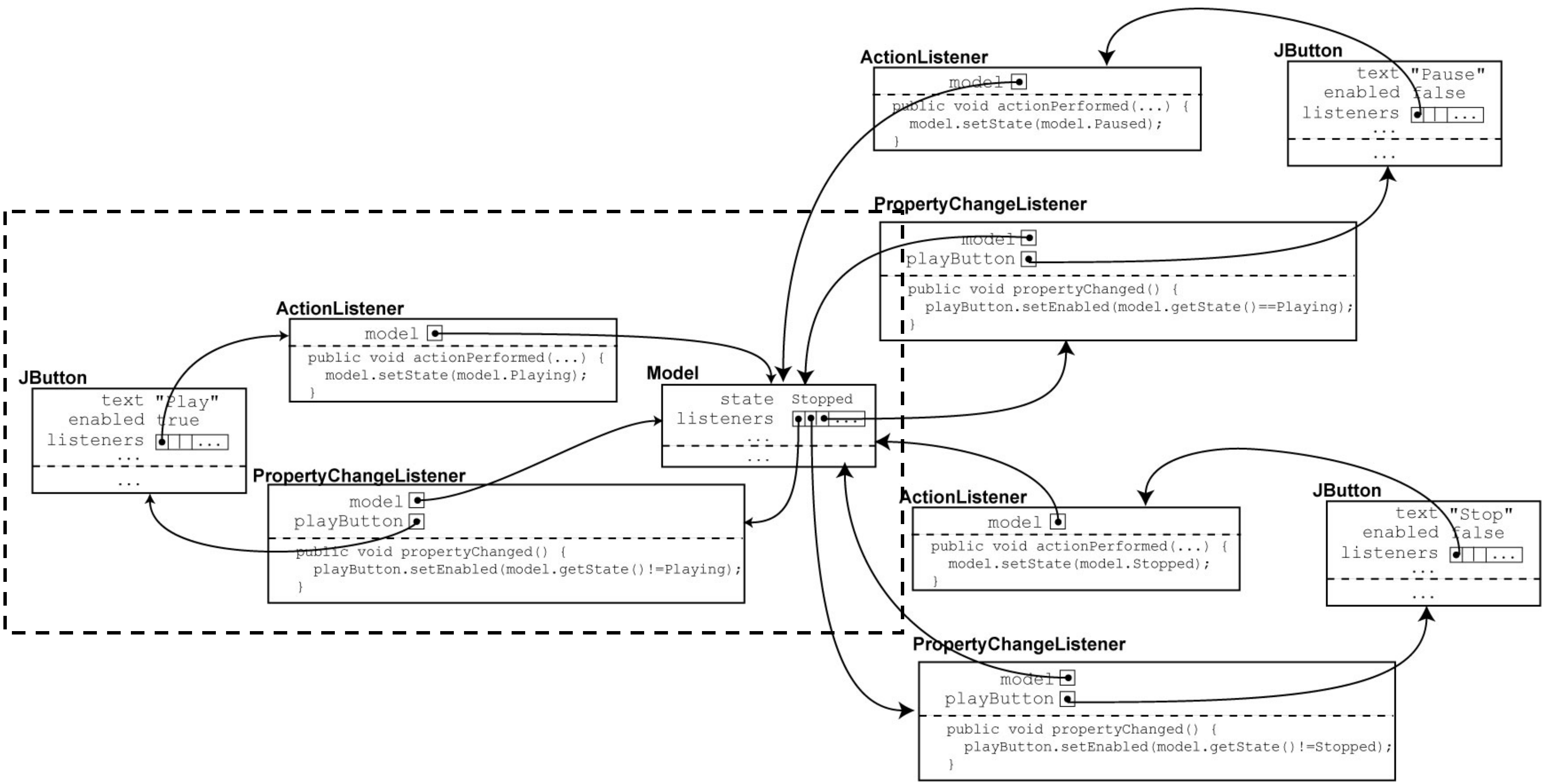
- Using Java/Swing and MVC design pattern:
  - Implement time-varying *state* as a mutable field.
  - Encapsulate state in a *model* class that supports registration of *listeners*
    - *Listeners* are notified when state is updated
  - Implement a model listener for each button that updates that button's enabled state.
  - Implement action listeners for each button that update the model's state.
- At program initialization time:
  - construct objects, register listeners.
  - relinquish control to the toolkit.

Visualising Java/Swing solution

- *partial* snapshot of heap at runtime:



Java/Swing Heap – Big Picture



Some Observations

From the heap snapshot, we can see:

A feedback loop exists in Swing implementation just as it did in Fruit specification.

However:

- In Java, dataflow graph created implicitly and dynamically by mutating objects.
  - Error-prone! easy to update a field, but forget to invoke listeners…
- Java diagram is a snapshot of heap at one particular instant at runtime.
  - Can't derive such pictures from the program text.
- In contrast:
  - **Fruit diagram *is* the specification**.
    (or at least isomorphic…)

Being able to *see* complex relationships (feedback) enables reasoning…

Reasoning with Specifications

Some questions we can ask / answer just by inspection of (visual) specification:

- What effect does pressing "play" have on *state*?
- What GUI components affect *state*?
- How does the "play" button affect the "pause" button?

In Yampa/Fruit these relationships are all made explicit in the data flow diagram.

- purely functional model $\Rightarrow$ no hidden side effects.

Reasoning: Proofs?

Q: If Yampa/Fruit provide a formal model, can we use them to prove properties of reactive programs?

A: Of course! See Chapter 10:

- *runSF_* based on *scanl*, operational semantics.
- TLA's □ ("always") operator [Lamport 1994] for SF's.
- An *invariance theorem* for SFs.
  - Serves as a simple coinduction principle.
  - Proof: induction on length of input sequence.
- Example proof: bounded counter is always bounded.
  - Uses: case analysis over internal state & input, inv. thm.

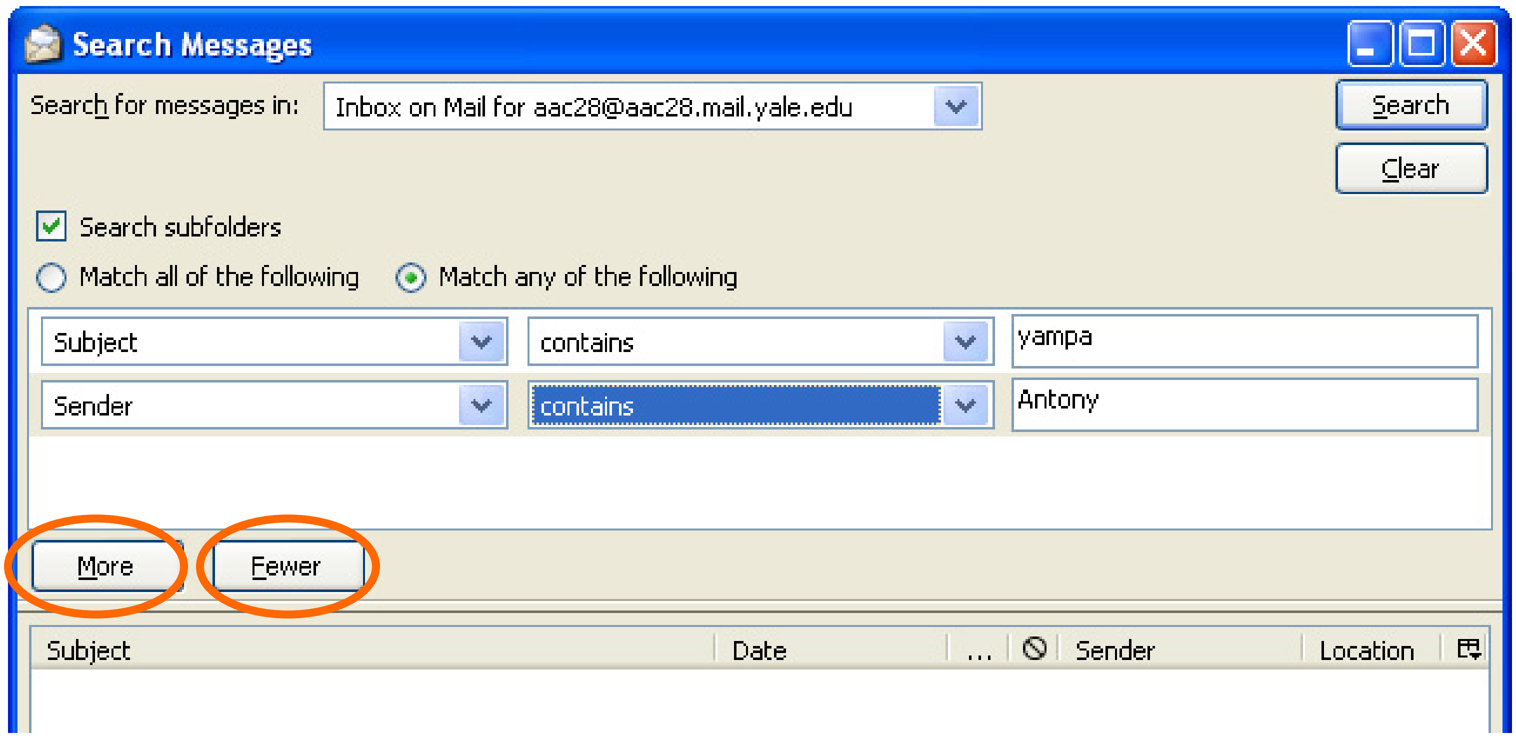
- **But..:**
  - We gain much reasoning power just by having a precise type for GUIs.
  - Simple data flow analysis by inspection.

Overview

- Background / Motivation
- Foundations:
  - Yampa – adaptation of FRP to Arrows framework
  - Fruit – GUI model based on Yampa
- Small Example
- Extensions
  - Continuations and Dynamic Collections
- Larger Examples
- Conclusions

Motivation

- Give a *modular* account of *dynamic* user interfaces in Fruit/Yampa.
- Example:

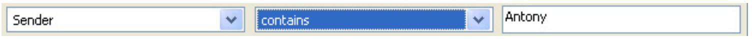


Search Pane – First Attempt

Basic Ideas: GUIs as first-class values, switching.

- Represent each row of search pane as a GUI:

$$oneRow :: GUI\ ()\ MsgAttr$$



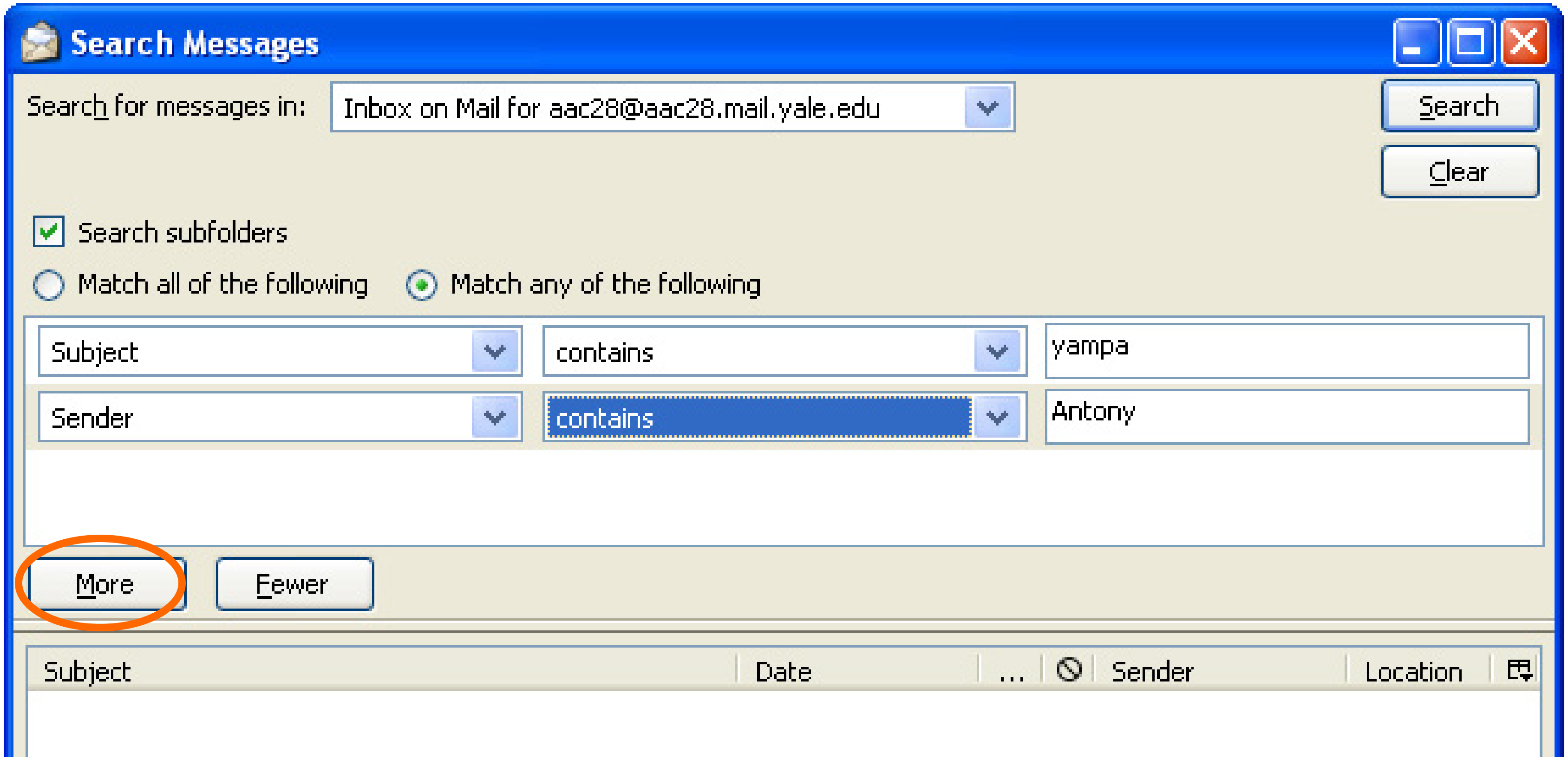
- Can compose rows into a *grid* with $aboveGUI$:

$$\begin{array}{l} addRow :: GUI\ ()\ [MsgAttr] \rightarrow GUI\ ()\ [MsgAttr] \\ addRow\ curGrid = \mathbf{proc}\ (gin, \_) \rightarrow \mathbf{do} \\ \quad (gin, ()) \qquad\quad \succ\!\!- curGrid\ `aboveGUI`\ oneRow \rightarrow (pic, (mas, ma)) \\ \quad (pic, ma : mas) \succ\!\!- returnA \end{array}$$

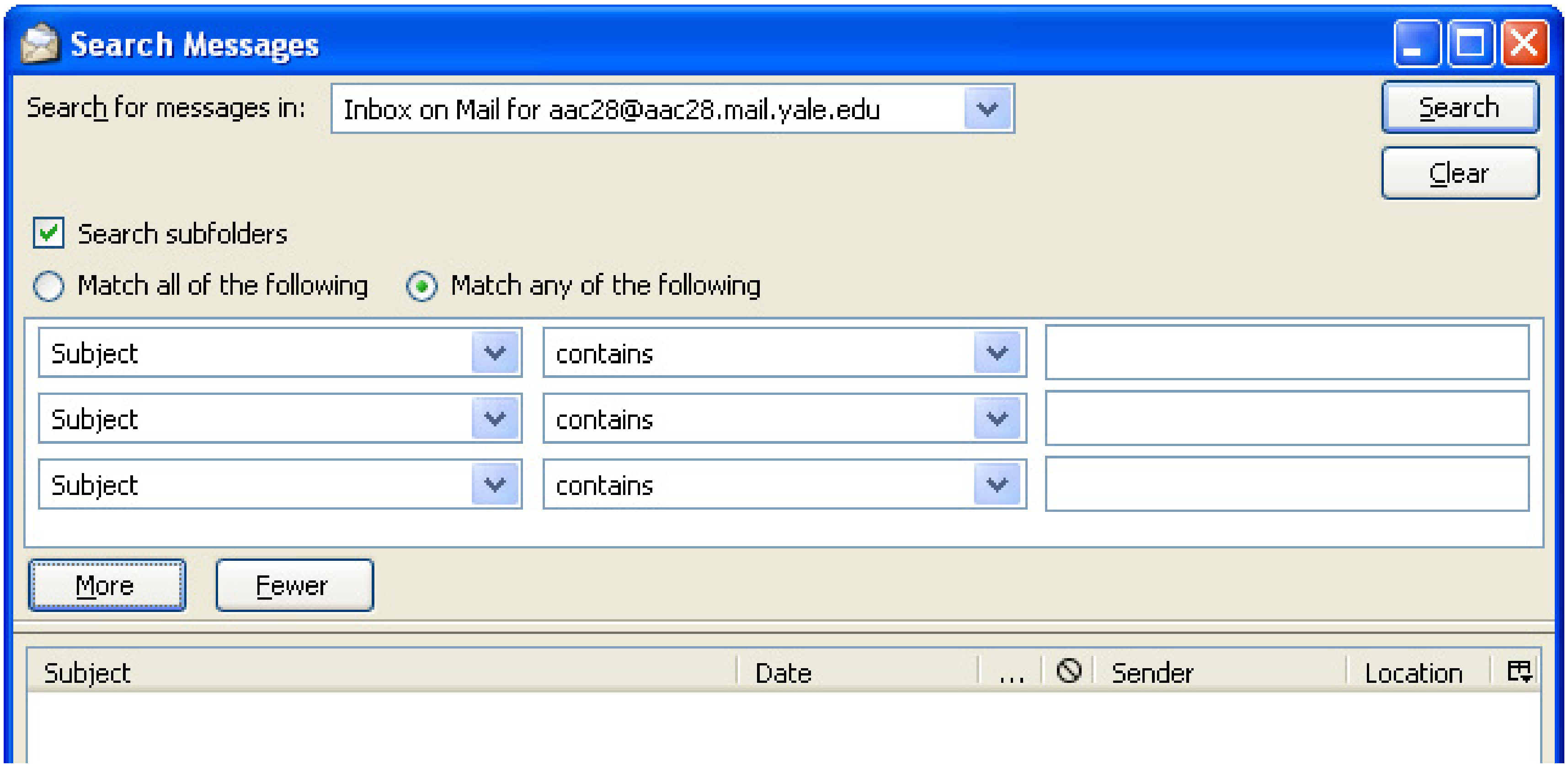
- On *more* button, switch recursively into new grid:

$$\begin{array}{l} mkGrid :: GUI\ ()\ [MsgAttr] \rightarrow GUI\ (Event\ ())\ [MsgAttr] \\ mkGrid\ prevGrid = switch\ aux \\ \qquad\qquad\qquad\qquad\qquad (\lambda\_ \rightarrow mkGrid\ (addRow\ prevGrid)) \\ \quad \mathbf{where}\ aux = \ldots \end{array}$$

Search Pane – First Attempt



Search Pane – First Attempt



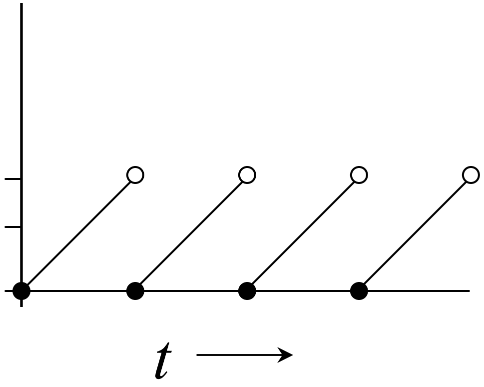
The Problem with Switching

What happened?!

- As they execute, SFs may accumulate internal state.
- But this internal state is discarded on a *switch*:

$$
\begin{aligned}
&swTest :: SF\ ()\ Float \rightarrow SF\ ()\ Float\\
&swTest\ sf = switch\ (sf\ \&\&\&\ after\ 2)\\
&\qquad\qquad\qquad\qquad\qquad (\lambda\_ \rightarrow swTest\ sf)
\end{aligned}
$$

$swTest\ (constant\ 1 \ggg integral)$:



Continuation-Based Switching

Solution: A “call/cc” for Signal Functions:

- Operational Semantics of SFs (Chapters 4,5):

$$\mathbf{data}\ SF\ a\ b = SF\ (DTime \to a \to (\underline{SF\ a\ b}, b))$$

- Internal state of running SF in its continuation.

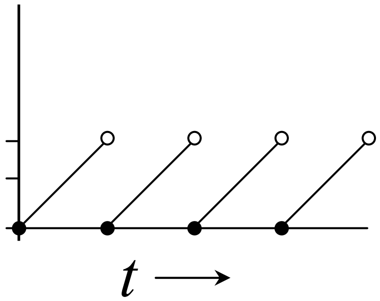
- Expose this SF continuation during switching:

$$\begin{aligned} kSwitch :: SF\ a\ b & \\ & \to SF\ (a, b)\ (Event\ c) \\ & \to (\underline{SF\ a\ b} \to c \to SF\ a\ b) \\ & \to SF\ a\ b \end{aligned}$$

Fun with kSwitch

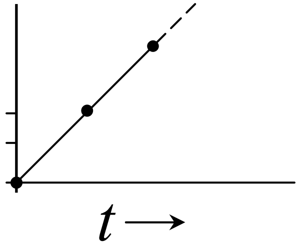
$$kswTest1 :: SF\ ()\ Float \rightarrow SF\ ()\ Float$$
$$kswTest1\ \ sf = kSwitch\ sf\ (after\ 2)$$
$$(\lambda ksf\ \_ \rightarrow kswTest1\ \ sf)$$

$kswTest1\ (constant\ 1 \ggg integral)$:



$$kswTest2 :: SF\ ()\ Float \rightarrow SF\ ()\ Float$$
$$kswTest2\ \ sf = kSwitch\ sf\ (after\ 2)$$
$$(\lambda ksf\ \_ \rightarrow kswTest2\ \ ksf)$$

$kswTest2\ (constant\ 1 \ggg integral)$:



Dynamic Collections

Back to our dynamic search pane GUI:

- “More” button:
  - kSwitch is sufficient.
  - Compose “current grid” (SF continuation) with GUI for another row.
- “Fewer” button?
  - **Problem:** Can’t “invert” a >>> operation!
  - **Solution:** Allow switching over *collections* of SFs running in parallel…

pSwitch(B)

- Yampa provides pSwitch(B) (parallel switch):

$$
\begin{aligned}
pSwitchB ::\ & Functor\ col \Rightarrow \\
& col\ (SF\ a\ b) \\
& \rightarrow SF\ (a, col\ b)\ (Event\ c) \\
& \rightarrow (col\ (SF\ a\ b) \rightarrow c \rightarrow SF\ a\ (col\ b)) \\
& \rightarrow SF\ a\ (col\ b)
\end{aligned}
$$

- reshape function type is key to flexible updates.

pSwitch(B)

- Yampa provides pSwitch(B) (parallel switch):

$$
\begin{aligned}
pSwitchB &:: Functor\ col \Rightarrow \\
&\quad col\ (SF\ a\ b) \\
&\quad \rightarrow SF\ (a, col\ b)\ (Event\ c) \\
&\quad \rightarrow (col\ (SF\ a\ b) \rightarrow c \rightarrow SF\ a\ (col\ b)) \\
&\quad \rightarrow SF\ a\ (col\ b)
\end{aligned}
$$

initial collection

switching event

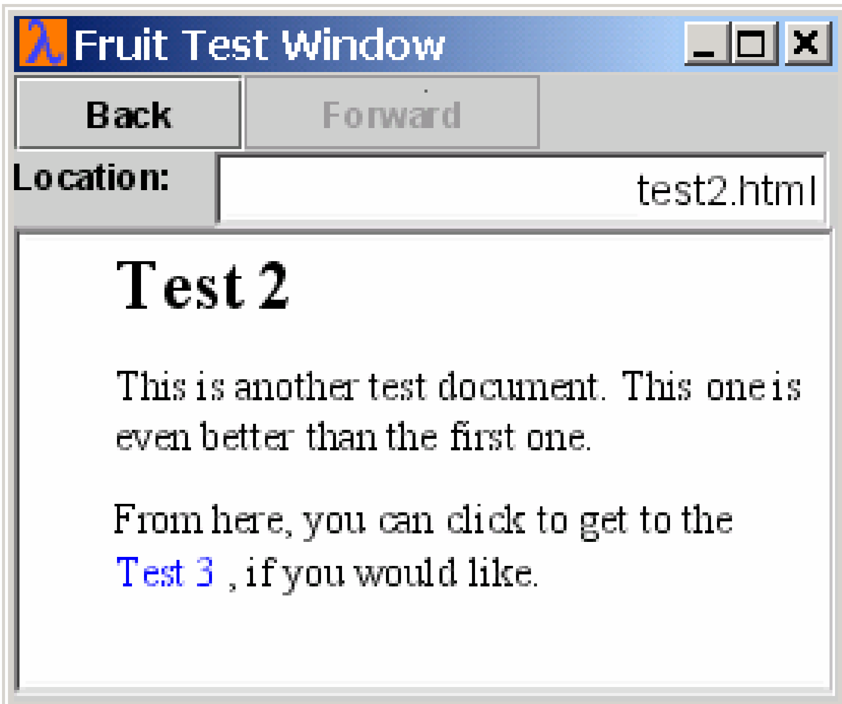
reshape function

- reshape function type is key to flexible updates.

Overview

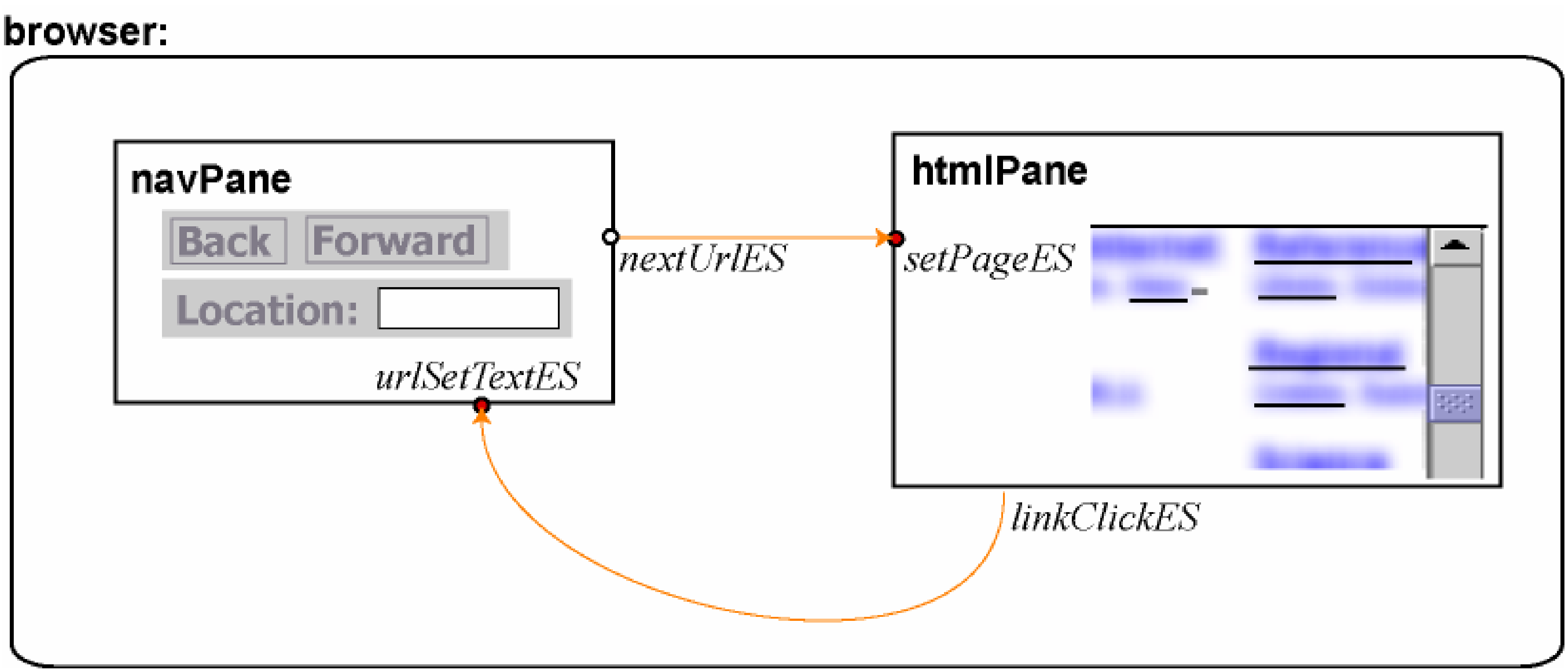
- Background / Motivation
- Foundations:
  - Yampa – adaptation of FRP to Arrows framework
  - Fruit – GUI model based on Yampa
- Small Example
- Extensions
  - Continuations and Dynamic Collections
- Larger Examples
- Conclusions

Web Browser with History



- Fwd, Back buttons navigate history
- Buttons, location field and clicking links update history.
- Location field, buttons updated in response to navigation events.

Web Browser w/ History

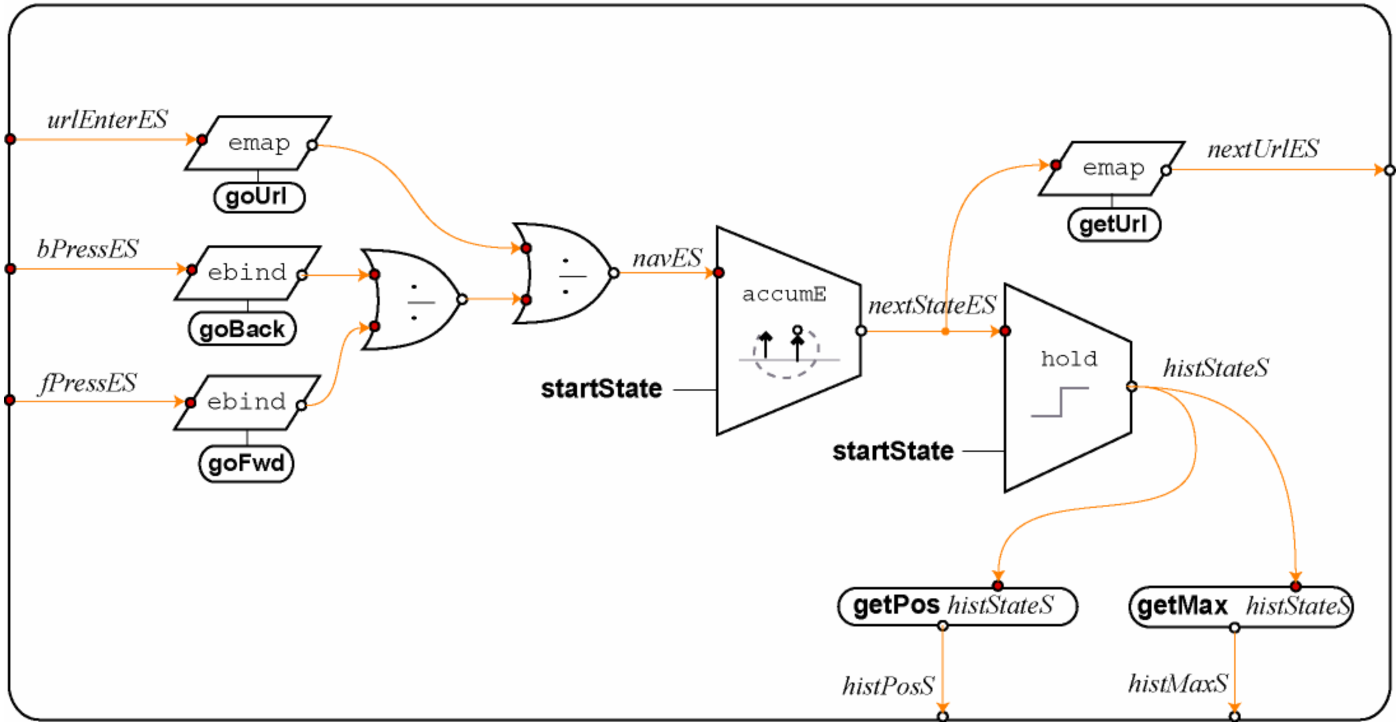


History List

histList:

```
let goBack :: HistState -> HistState
           goBack (pos,hList) = (pos+1,hList)

    goFwd :: HistState -> HistState
    goFwd (pos,hList) = (pos-1,hList)

    goUrl :: String -> HistState -> HistState
    goUrl url (pos,hList) = (0,(url:(drop pos hList))

    startState :: HistState
    startState = (0, ["http://www.haskell.org"])
in
```

```
getUrl :: HistState -> String
getUrl (pos,hList) = hList !! pos

getPos :: HistState -> Int
getPos (pos,hList) = pos

getMax :: HistState -> Int
getMax (_,hList) = (length hList)-1
```



History List Semantics

- Essence of semantics is a few equations:

$$\textbf{type}\ HistState = (Int, [URL])$$

$$goBack :: HistState \rightarrow HistState$$

$$goBack\ (pos, hList) = (pos + 1, hList)$$

$$goFwd\ :: HistState \rightarrow HistState$$

$$goFwd\ (pos, hList) = (pos - 1, hList)$$

$$goUrl\ \ :: String \rightarrow HistState \rightarrow HistState$$

$$goUrl\ \ url\ (pos, hList) = (0, url : drop\ pos\ hList)$$

Space Invaders

Demonstrates:

- Physical simulation
- Control systems
- Animation
- **Dynamic Collections:**
  - Bullets, Invaders

…and of course:

- Fun!

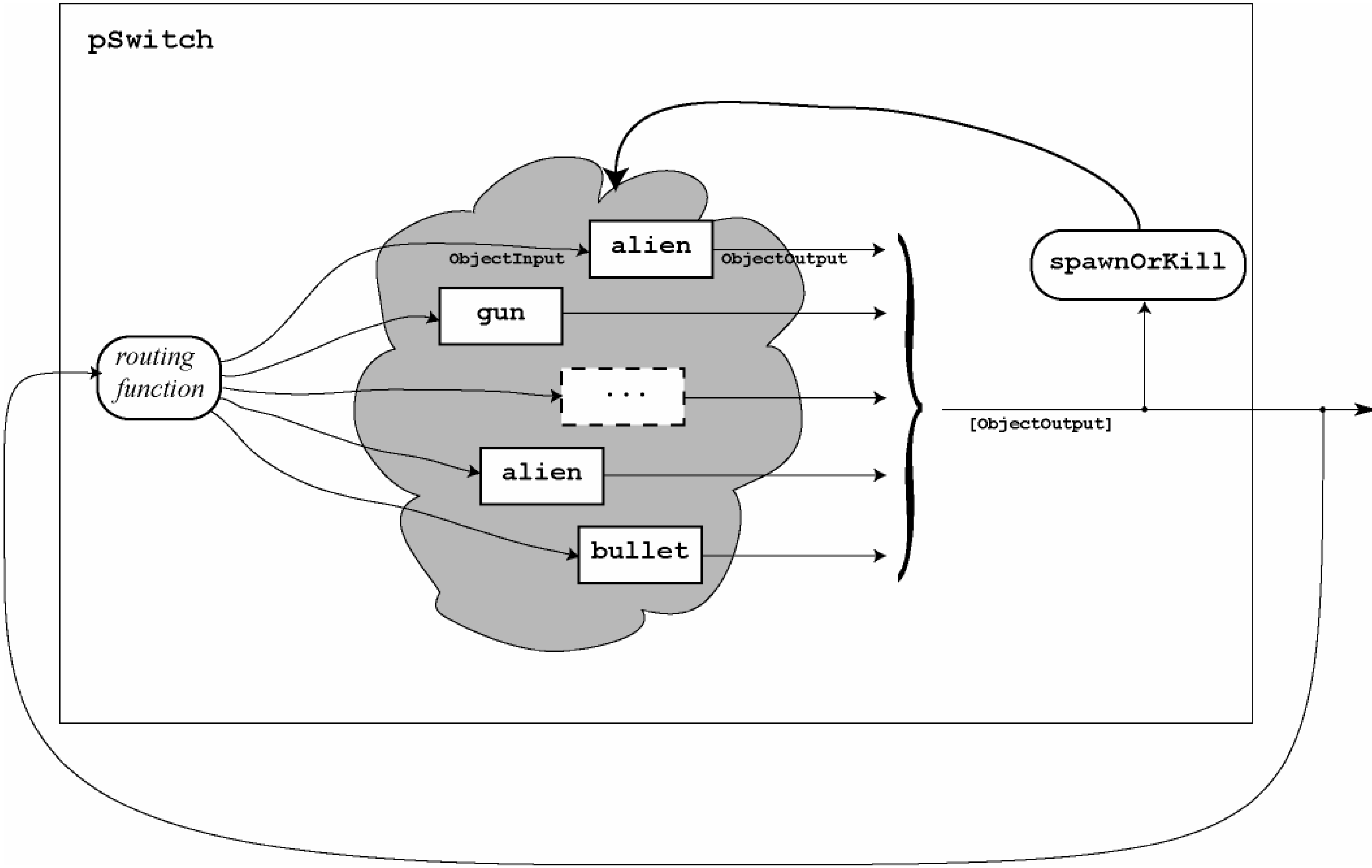
Implementing Game Objects

- Model each game object as a signal function:

$$
\begin{aligned}
&\mathit{simpleGun}\ (\mathit{Point2}\ \mathit{x0}\ \mathit{y0}) = \mathbf{proc}\ \mathit{gin} \rightarrow \mathbf{do} \\
&\qquad \text{-- Desired position:} \\
&\quad \mathit{gin} \succ\!\!-\ \mathit{mouseSF} \rightarrow (\mathit{Point2}\ \mathit{xd}\ \_) \\
&\qquad \text{-- Desired acceleration:} \\
&\quad \mathbf{let}\ \mathit{ad} = 10 * (\mathit{xd} - x) - 5 * v \\
&\qquad \text{-- basic physics:} \\
&\quad \mathit{ad}\ \succ\!\!-\ \mathit{integral} \rightarrow v \\
&\quad v\ \ \succ\!\!-\ \mathit{integral} \rightarrow x \\
&\quad \ldots
\end{aligned}
$$

Simulated World

- "World" is a dynamic collection of SFs: (demo)



Overview

- Background / Motivation
- Foundations:
  - Yampa – adaptation of FRP to Arrows framework
  - Fruit – GUI model based on Yampa
- Small Example
- Extensions
  - Continuations and Dynamic Collections
- Larger Examples
- Conclusions

Summary of Contributions

- Yampa (Chapters 3-5, [Courtney & Elliott 2001], [Nilsson, Courtney, Peterson 2002]):
  - A purely functional model of reactive systems based on synchronous dataflow.
  - Based on adapting Fran [Elliott & Hudak 1997] and FRP [Wan & Hudak 2001] to Arrows Framework [Hughes 2000].
  - Simple denotational and operational semantics.
- Haven (Chapter 6):
  - A functional model of 2D vector graphics.
- Fruit (Chapters 7, 10, 11, [Courtney & Elliott 2001], [Courtney 2003]):
  - A GUI library defined solely using Yampa and Haven.
- Dynamic Collections (Ch. 8, [Nilsson, Courtney, Peterson 2002]):
  - Continuation-based and parallel switching primitives

Conclusions

- With Yampa, we can write rigorous executable specifications of GUIs without appealing to imperative programming or I/O.
- Purely functional model of GUIs enables:
  - Precise reasoning about GUI program behavior.
  - Clear account of GUI programming idioms.

- Prototype implementation embedded in Haskell:

  http://www.haskell.org/yampa

  http://www.haskell.org/haven

  http://www.haskell.org/fruit

Related Work: Fudgets

- [Carlsson & Hallgren 1993], [Carlsson & Hallgren 1998]

Fudgets	Fruit
`F hi ho =` `  SP (hi+li) (ho+lo)`	`GUI b c =` `  SF (GUIIn,b)(Pic,c)`
Stream Processors (discrete, asynchronous)	Signal Functions (continuous, synchronous)
Extends stream-based I/O	Defined denotationally
`F()()` may perform I/O	`GUI()()` performs no I/O
Uses Xlib protocol requests / responses	Explicit, functional model of input devices, graphics