Modeling User Interfaces in a Functional Language

Antony Courtney

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

\[
\text{putStrLn} :: \text{String} \rightarrow \text{IO} ()
\]

\[
\text{getStrLn} :: \text{IO} \text{ String}
\]

we read:  \( f :: \text{IO} a \)

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:

\[ [\text{IO } a] = \text{World} \rightarrow (\text{World}, a) \]

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.
  - 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:
    \[
    \text{Signal } \alpha = \text{Time } \rightarrow \alpha
    \]
  - **Signal Function**: function from Signal to Signal:
    \[
    \text{SF } \alpha \beta = \text{Signal } \alpha \rightarrow \text{Signal } \beta
    \]

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): 
  \[ \text{arr} :: (a \to b) \to SF\ a\ b \]
  \[
  \begin{array}{c}
  \text{arr} f \\
  \hline
  b \quad f \quad c \\
  \end{array}
  \]
  \[
  \text{arr}\ f = \lambda s.\lambda t.\mathbb{[}f\mathbb{]}(s\ t)
  \]

- Serial Composition:
  \[
  \text{gf} :: SF\ b\ c \to SF\ c\ d \to SF\ b\ d
  \]
  \[
  \begin{array}{c}
  \text{fa} \quad \text{ga} \\
  b \quad b \quad c \quad c \quad d \quad d \\
  \end{array}
  \]
  \[
  \mathbb{[}fa \ggg ga\mathbb{]} = \mathbb{[}ga\mathbb{]} \circ \mathbb{[}fa]\]
Other Arrow Combinators

- Use *tuples* to group signals:

\[ \text{first} :: SF \ a \ b \rightarrow SF \ (a, c) \ (b, c) \]

\[ \text{first}\ sf \]

\[ \llbracket \text{first} \ sf \rrbracket = \lambda s. \ \text{pair}Z \ (\llbracket sf \rrbracket \ (\text{fst}Z\ s)) \ (\text{snd}Z\ s) \]

- Other (derived) combinators to form arbitrary digraphs:
Feedback

- Can define cyclic graphs with \textit{loop}:

\[
\text{loop} :: SF \ (a, c) \ (b, c) \rightarrow SF \ a \ b
\]

\[
\text{loop sf}
\]

\[
\llbracket \text{loop fa} \rrbracket = \lambda s. \text{fstZ} \ (Y (\lambda r. \llbracket sf \rrbracket (\text{pairZ} \ s \ (\text{sndZ} \ r))))
\]

- 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:
  ```haskell
data Event a = EvOcc a  
                 | NoEvent
```
- Some basic operations (used point-wise):
  ```haskell
tag       :: b → Event a → Event b
mergeE    :: Event a → Event a → Event a
gate      :: Event a → Bool → Event a
```
Event Processors

\[ \text{hold} :: a \rightarrow SF (\text{Event } a) \ a \]

\[ \hspace{1cm} \text{hold 3} \hspace{1cm} \]

\[ \text{accum} :: a \rightarrow SF (\text{Event } (a \rightarrow a)) (\text{Event } a) \]

\[ \hspace{1cm} \text{tag incr} \rightarrow \text{accum 1} \hspace{1cm} \]
Example: A Bounded Counter

\[ \text{bc} :: \text{Int} \to \text{Int} \to \text{SF (Event ())) Int} \]
\[ \text{bc \hspace{2pt} x0 \hspace{2pt} max = \ldots} \]

- Initial value: \( x0 \)
- Increment on each event until \( \text{max} \) reached

Implementation:

```
boundedCounter x0 max =
```
Arrows Syntax [Paterson 2001]

```
boundsCounter x0 max =

bc :: Int → Int → SF (Event ()) Int
bc x0 max =
    proc incReq → do
        let updE = (incReq ∘ gate ∘ (x < max)) ∘ tag ∘ incr
            updE ⊢ dAccumHold x0 → x
            x ⊢ returnA
```
Arrows Syntax [Paterson 2001]

boundedCounter \( x_0 \) max =

\[
\begin{align*}
bc & \colon \text{Int} \rightarrow \text{Int} \rightarrow \text{SF (Event (\)) Int} \\
b \colon & \text{Int} x_0 \text{ max } = \\
\text{proc } incReq \rightarrow \text{ do } \\
& \text{let } updE = (\text{incReq } \text{ gate} \ (x < \text{ max})) \text{ tag } \text{ incr} \\
& \text{updE } \leftarrow \text{dAccumHold } x_0 \rightarrow x \\
x & \leftarrow \text{returnA}
\end{align*}
\]
Arrows Syntax [Paterson 2001]

```plaintext
boundedCounter x0 max =

bc :: Int \rightarrow Int \rightarrow SF (Event ()) Int
bc x0 max =

proc incReq \rightarrow do
  let updE = (incReq `gate` (x < max)) `tag` incr
  updE \leftarrow dAccumHold x0 \rightarrow x
  x \leftarrow returnA
```
Arrows Syntax [Paterson 2001]

boundedCounter x0 max =

\[ \begin{align*}
\text{bc} & : \text{Int} \to \text{Int} \to \text{SF} \left( \text{Event} () \right) \text{Int} \\
\text{bc x0 max} & = \\
\text{proc incReq} \to \text{do} \\
\text{let updE} & = (\text{incReq} \text{ `gate` (x < max)}) \text{ `tag` incr} \\
\text{updE} & \supset \text{dAccumHold x0} \to x \\
x & \supset \text{returnA}
\end{align*} \]
Arrows Syntax [Paterson 2001]

boundedCounter x0 max =

\[
bc :: \text{Int} \to \text{Int} \to SF (\text{Event}()) \text{Int}
\]

\[
b \leq x0 \leq max =
\]

\[
\text{proc incReq } \rightarrow \text{do}
\]

\[
\text{let updE } = (\text{incReq} \ 'gate' (x \ < \ max))'tag' \text{ incr}
\]

\[
\text{updE } \succ \text{dAccumHold x0 } \rightarrow x
\]

\[
x \succ \text{returnA}
\]
Arrows Syntax [Paterson 2001]

boundedCounter x0 max =

code

bc :: Int → Int → SF (Event ()) Int
bc x0 max =
  proc incReq → do
  let updE = (incReq ‘gate‘ (x < max)) ‘tag‘ incr
  updE ≫ dAccumHold x0 → x
  x ≫ returnA

Page: 24
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:

\[
\text{switch} :: SF \ a \ (b, \ Event \ c) \\
\rightarrow (c \rightarrow SF \ a \ b) \\
\rightarrow SF \ a \ b
\]

![Diagram of switch combinator]

```
switch sf0 kf
```
Basic Switching

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

\[
\text{switch} :: SF \ a \ (b, \ Event \ c) \\
\rightarrow (c \rightarrow SF \ a \ b) \\
\rightarrow SF \ a \ b
\]
Basic Switching

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

\[
\text{switch} :: SF \ a \ (b, \text{Event} \ c) \\
\rightarrow (c \rightarrow SF \ a \ b) \\
\rightarrow SF \ a \ b
\]
Basic Switching

- \textit{switch} combinator switches from one SF to another on event occurrence:

\[
\text{switch} :: SF \ a \ (b, \text{Event} \ c) \\
\rightarrow (c \rightarrow SF \ a \ b) \\
\rightarrow SF \ a \ b
\]
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"):
  \[ \text{Behavior } a \approx \text{Time} \rightarrow a \]

- SOE FRP [Hudak 2000] [Wan & Hudak 2000]:
  Behaviors are functions of start time ("computations"):
  \[ \text{Behavior } a \approx \text{Time} \rightarrow \text{Time} \rightarrow a \]

**Motivation:** Avoid inherent space-time leak in:
\[ x = y \ `\text{switch}` \ (e \ -\rightarrow \ z) \]
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:**
  \[ \text{Signal } \alpha = \text{Time } \rightarrow \alpha \]

  and **Signal Functions:**
  \[ \text{SF } \alpha \beta = \text{Signal } \alpha \rightarrow \text{Signal } \beta \]

  ...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
  
  $$\text{type} \ Image = (\text{Point} \rightarrow \text{Color}, \text{Region})$$

  - 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:**  
  \[ \text{fillShape} :: \text{Color} \rightarrow \text{Shape} \rightarrow \text{Image} \]

  e.g.:  
  \[ \text{fillShape \ red \ (circle \ 20)} = \]

- **Haven provides:**  
  \[ \text{monochrome} :: \text{Color} \rightarrow \text{Image} \]
  \[ \text{imgCrop} :: \text{Shape} \rightarrow \text{Image} \rightarrow \text{Image} \]

  \[ \text{monochrome \ red} = \]

  \[ \text{imgCrop \ (circle \ 20) \ (monochrome \ red)} = \]
**fillShape vs. imgCrop**

*imgCrop* is far more versatile than *fillShape*:

- **Use *imgCrop* on any image:**
  - color gradients:
    
    \[
    \text{gradient} :: \text{Point} \rightarrow \text{Color} \rightarrow \text{Point} \rightarrow \text{Color} \rightarrow \text{Image}
    \]

- **Compose crop operations:**
  
  \[
  \text{imgCrop } s1 \ ((\text{imgCrop } s2 \ ... ) \ \over\ \text{over}\ (\text{imgCrop } s3 \ (\text{imgCrop } ...)))
  \]

\[
\begin{array}{ccc}
\text{imgCrop} & \circ & \text{Triangle} = \text{Image} \\
\end{array}
\]
Fruit

What is a GUI?

- GUIs are Signal Functions:
  \[
  \text{type } GUI\ a\ b = SF\ (\text{GUIInput},\ a)\ (\text{Picture},\ b)
  \]

- Signal types:
  \[
  \text{GUIInput} \quad \text{– keyboard and mouse state}
  
  \text{Picture} \quad \text{– visual display (Image)}
  
  a, b \quad \text{– auxiliary semantic input and output signals}
  \]

- \text{GUIInput}:
  \[
  \text{data}\ Mouse = Mouse\{\ mpos :: Point, \}
  
  lbDown,\ rbDown :: \text{Bool}
  
  \}
  
  \text{type}\ \text{GUIInput} = (\text{Maybe Kbd, Maybe Mouse})
Fruit: Components and Layout

- Aux. signals connect GUI to rest of application.
- Components (slightly simplified interfaces):
  - Text Labels:
    \[ \text{label} :: \text{GUI String} () \]
  - Buttons:
    \[ \text{button} :: \text{String} \rightarrow \text{GUI Bool (Event ()}) \]
  - Text fields:
    \[ \text{textField} :: \text{String} \rightarrow \text{GUI (Event String)} (Event String) \]
- Layout Combinators:
  \[ \text{beside GUI} :: \text{GUI b c} \rightarrow \text{GUI d e} \rightarrow \text{GUI (b, d) (c, e)} \]
  \[ \text{above GUI} :: \text{GUI b c} \rightarrow \text{GUI d e} \rightarrow \text{GUI (b, d) (c, e)} \]
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:
  ![Classic VCR-style media controller](image)

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

- Represent media state with:
  
  \[
  \text{data } \text{MediaState} = \text{Playing} \mid \text{Paused} \mid \text{Stopped}
  \]
Design

• just a simple FSM:

  ![Finite State Machine Diagram]

  - playE: state /= Playing
  - pauseE: state == Playing
  - stopE: state /= Stopped

• Derive time-invariant *constraints* by inspection:
Fruit Specification (Visual)

- Visually:
Fruit Specification (Textual)

\[
\text{playerCtrl} :: \text{GUI (}) \text{ MediaState}
\]
\[
\text{playerCtrl} = \text{ hbox (proc } _ \rightarrow \text{ do}
\]
\[
(\text{state } \not\equiv \text{ Playing}) \quad \triangleright \text{ button "Play" } \rightarrow \text{ playE}
\]
\[
(\text{state } \equiv \text{ Playing}) \quad \triangleright \text{ button "Pause" } \rightarrow \text{ pauseE}
\]
\[
(\text{state } \not\equiv \text{ Stopped}) \quad \triangleright \text{ button "Stop" } \rightarrow \text{ stopE}
\]
\[
(\text{mergeEs } [\text{ tag playE Playing, tag pauseE Paused,}
\text{ tag stopE Stopped}])
\]
\[
\triangleright \text{ boxSF (dHold Stopped) } \rightarrow \text{ state}
\]
\[
\text{state } \triangleright \text{ returnA}
\]
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:
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 $\always$ (“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:
  
  \[
  \text{oneRow} :: \text{GUI} () \text{ MsgAttr}
  \]

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

  \[
  \text{addRow} :: \text{GUI} () [\text{MsgAttr}] \rightarrow \text{GUI} () [\text{MsgAttr}]
  \]

  \[
  \text{addRow} \text{ curGrid} = \text{proc} (\text{gin}, _) \rightarrow \text{do}
  \]

  \[
  (\text{gin}, ()) \leftarrow \text{curGrid} \{ \text{aboveGUI} \} \text{ oneRow} \rightarrow (\text{pic}, (\text{mas}, \text{ma}))
  \]

  \[
  (\text{pic}, \text{ma} : \text{mas}) \leftarrow \text{returnA}
  \]

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

  \[
  \text{mkGrid} :: \text{GUI} () [\text{MsgAttr}] \rightarrow \text{GUI} (\text{Event} ()) [\text{MsgAttr}]
  \]

  \[
  \text{mkGrid} \text{ prevGrid} = \text{switch aux}
  \]

  \[
  (\lambda _\_ \rightarrow \text{mkGrid} (\text{addRow prevGrid}))
  \]

  \[
  \text{where aux} = \ldots
  \]
Search Pane – First Attempt

![Search Messages Window]

- Search for messages in: Inbox on Mail for aac28@aac28.mail.yale.edu
- Search subfolders
- Match any of the following:
  - Subject: contains yampa
  - Sender: contains Antony
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`:

\[
\text{swTest} :: SF () \rightarrow SF () \rightarrow \text{Float}
\]
\[
\text{swTest sf} = \text{switch} (sf \&\& \text{after 2})
\]
\[
(\lambda_\rightarrow \text{swTest sf})
\]

\[
\text{swTest (constant 1 \gggg integral)} :
\]
Continuation-Based Switching

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

- Operational Semantics of SFs (Chapters 4,5):
  \[
  \text{data } SF \ a \ b = SF (DTime \rightarrow a \rightarrow (SF \ a \ b, b))
  \]

- Internal state of running SF in its continuation.

- Expose this SF continuation during switching:
  \[
  kSwitch :: SF \ a \ b \\
  \rightarrow SF (a, b) (Event \ c) \\
  \rightarrow (SF \ a \ b \rightarrow c \rightarrow SF \ a \ b) \\
  \rightarrow SF \ a \ b
  \]
Fun with kSwitch

\[ kswTest1 :: SF () Float \rightarrow SF () Float \]
\[ kswTest1 \ sf = kSwitch \ sf \ (\text{after } 2) \]
\[ (\lambda \sf \rightarrow kswTest1 \ sf) \]

\[ kswTest1 \ (\text{constant } 1 \implies \text{integral}) : \]

\[ kswTest2 :: SF () Float \rightarrow SF () Float \]
\[ kswTest2 \ sf = kSwitch \ sf \ (\text{after } 2) \]
\[ (\lambda \sf \rightarrow kswTest2 \ sf) \]

\[ kswTest2 \ (\text{constant } 1 \implies \text{integral}) : \]
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...
Yampa provides pSwitch(B) (parallel switch):

\[
pSwitchB :: \text{Functor } \text{col} \Rightarrow \text{col} (\text{SF} \ a \ b) \rightarrow \text{SF} (a, \text{col} \ b) \ (\text{Event} \ c) \rightarrow (\text{col} (\text{SF} \ a \ b) \rightarrow c \rightarrow \text{SF} \ a \ (\text{col} \ b)) \rightarrow \text{SF} \ a \ (\text{col} \ b)
\]

- reshape function type is key to flexible updates.
Yampa provides \texttt{pSwitch(B)} (parallel switch):

\[
pSwitchB :: \text{Functor } \text{col} \Rightarrow \text{col (SF a b)} \\
\rightarrow SF (a, \text{col b}) (Event c) \\
\rightarrow (\text{col (SF a b)} \rightarrow c \rightarrow SF a (\text{col b})) \\
\rightarrow SF a (\text{col b})
\]

- 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

browser:

navPane
- Back
- Forward
- Location:

htmlPane
- nextUrlES
- setPageES
- linkClickES
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'])

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

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

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

in

---

The diagram illustrates the history list functionality, with arrows indicating how different events (e.g., goBack, goFwd, goUrl) affect the state of the history list.
History List Semantics

- Essence of semantics is a few equations:

```haskell
type HistState = (Int, [URL])

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

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

getUrl :: String -> HistState -> HistState
getUrl 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:

\[
\text{simpleGun} \,(\text{Point2} \ x_0 \ y_0) \equiv \text{proc} \ gin \rightarrow \text{do}
\]

-- Desired position:
\[
gin \leftarrow \text{mouseSF} \rightarrow (\text{Point2} \ xd \ _) \]

-- Desired acceleration:
\[
\text{let} \ ad \equiv 10 \ast (xd - x) - 5 \ast v
\]

-- basic physics:
\[
\text{ad} \leftarrow \text{integral} \rightarrow v
\]
\[
v \leftarrow \text{integral} \rightarrow x
\]
\[
\ldots
\]
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.
  - 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]

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