EFFECTS, ASYNCHRONY, AND CHOICE IN ARROWIZED FUNCTIONAL REACTIVE PROGRAMMING

Daniel Winograd-Cort

Department of Computer Science
Yale University

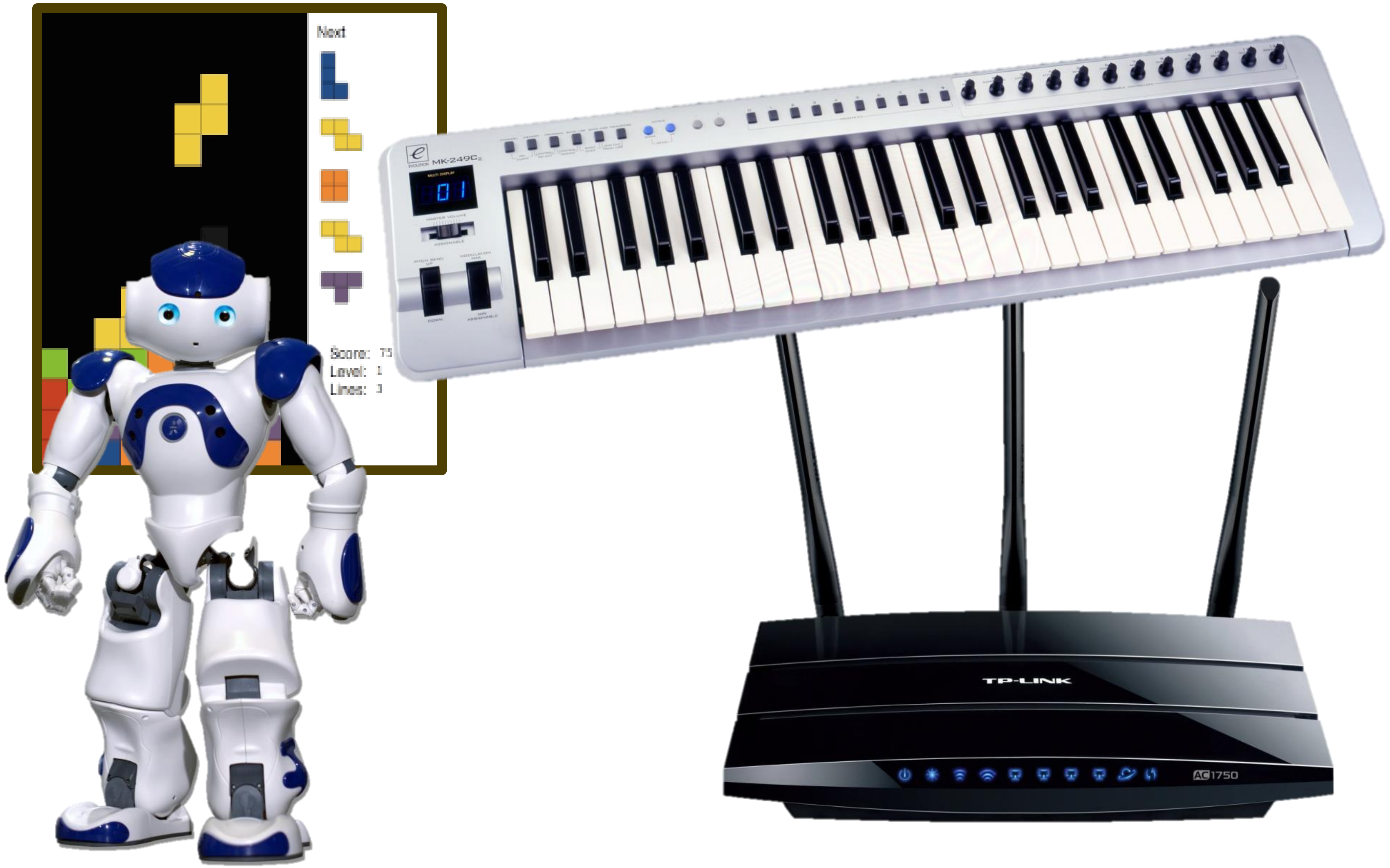
Dissertation Defense

New Haven, CT
Thursday, June 11, 2015

Functional Reactive Programming

- Functional programming that can react to change.
  - Time is a built-in aspect of the design.
- One programs with *continuous values* and *streams of events*.
  - Values themselves are time-dependent.
  - The computation is time-independent.
- FRP is required to be …
  - Causal by default.
  - Synchronous by default.
- Already in major use.

Functional Reactive Programming



GUI Example

- We would like a graphical user interface:
  - One textbox displays a temperature in Celsius.
  - Another displays the temperature in Fahrenheit.
- Updating one value should automatically update the other.

GUI Example

- We would like a graphical user interface:
  - One textbox displays a temperature in Celsius.
  - Another displays the temperature in Fahrenheit.
- Updating one value should automatically update the other.
- -Demo-
- We will explore this with and without FRP.

Java 7 with Swing

```java
public class TemperatureConverter extends JFrame {
  JTextField celsiusField;
  JTextField fahrenheitField;

  public TemperatureConverter(String name) {
    super(name);
    initGUI();
    initListeners();
  }

  private void initGUI() {
    celsiusField = new JTextField(5);
    fahrenheitField = new JTextField(5);

    Container pane = this.getContentPane();
    pane.setLayout(new FlowLayout());
    pane.add(celsiusField);
    pane.add(new JLabel("Celsius"));
    pane.add(new JLabel("="));
    pane.add(fahrenheitField);
    pane.add(new JLabel("Fahrenheit"));
  }

  public static void main(String[] args) {
    javax.swing.SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        TemperatureConverter frame =
          new TemperatureConverter("Temperature Converter");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
      }
    });
  }
```

```java
  private void initListeners() {
     celsiusField.getDocument().addDocumentListener(
      new DocumentListener() {
       public void insertUpdate(DocumentEvent e) { update(); }
       public void removeUpdate(DocumentEvent e) { update(); }
       public void changedUpdate(DocumentEvent e) { update(); }

       private void update() {
         if (!celsiusField.isFocusOwner() ||
                 !isNumeric(celsiusField.getText())) return;
         double celsius =
           Double.parseDouble(celsiusField.getText().trim());
         double fahrenheit = cToF(celsius);
         fahrenheitField.setText(
                String.valueOf(Math.round(fahrenheit)));
       }
     });
     fahrenheitField.getDocument().addDocumentListener(
      new DocumentListener() {
       public void insertUpdate(DocumentEvent e) { update(); }
       public void removeUpdate(DocumentEvent e) { update(); }
       public void changedUpdate(DocumentEvent e) { update(); }

       private void update() {
         if (!fahrenheitField.isFocusOwner() ||
                 !isNumeric(fahrenheitField.getText())) return;
         double fahrenheit =
           Double.parseDouble(fahrenheitField.getText().trim());
         double celsius = fToC(fahrenheit);
         celsiusField.setText(
                String.valueOf(Math.round(celsius)));
       }
     });
   }
}
```

* Code from https://github.com/eugenkiss/7guis

Java 7 with Swing

```java
public class TemperatureConverter extends JFrame {
  JTextField celsiusField;
  JTextField fahrenheitField;

  public TemperatureConverter(String name) {
    super(name);
    initGUI();
    initListeners();
  }

  private void initGUI() {
    celsiusField = new JTextField(5);
    fahrenheitField = new JTextField(5);

    Container pane = this.getContentPane();
    pane.setLayout(new FlowLayout());
    pane.add(celsiusField);
    pane.add(new JLabel("Celsius"));
    pane.add(new JLabel("="));
    pane.add(fahrenheitField);
    pane.add(new JLabel("Fahrenheit"));
  }

  public static void main(String[] args) {
    javax.swing.SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        TemperatureConverter frame =
          new TemperatureConverter("Temperature Converter");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
      }
    });
  }
```

```java
  private void initListeners() {
    celsiusField.getDocument().addDocumentListener(
     new DocumentListener() {
      public void insertUpdate(DocumentEvent e) { update(); }
      public void removeUpdate(DocumentEvent e) { update(); }
      public void changedUpdate(DocumentEvent e) { update(); }

      private void update() {
        if (!celsiusField.isFocusOwner() ||
                !isNumeric(celsiusField.getText())) return;
        double celsius =
          Double.parseDouble(celsiusField.getText().trim());
        double fahrenheit = cToF(celsius);
        fahrenheitField.setText(
               String.valueOf(Math.round(fahrenheit)));
      }
    });
    fahrenheitField.getDocument().addDocumentListener(
     new DocumentListener() {
      public void insertUpdate(DocumentEvent e) { update(); }
      public void removeUpdate(DocumentEvent e) { update(); }
      public void changedUpdate(DocumentEvent e) { update(); }

      private void update() {
        if (!fahrenheitField.isFocusOwner() ||
                !isNumeric(fahrenheitField.getText())) return;
        double fahrenheit =
          Double.parseDouble(fahrenheitField.getText().trim());
        double celsius = fToC(fahrenheit);
        celsiusField.setText(
               String.valueOf(Math.round(celsius)));
      }
    });
  }
}
```

* Code from https://github.com/eugenkiss/7guis

Java 8 with ReactFX (FRP)

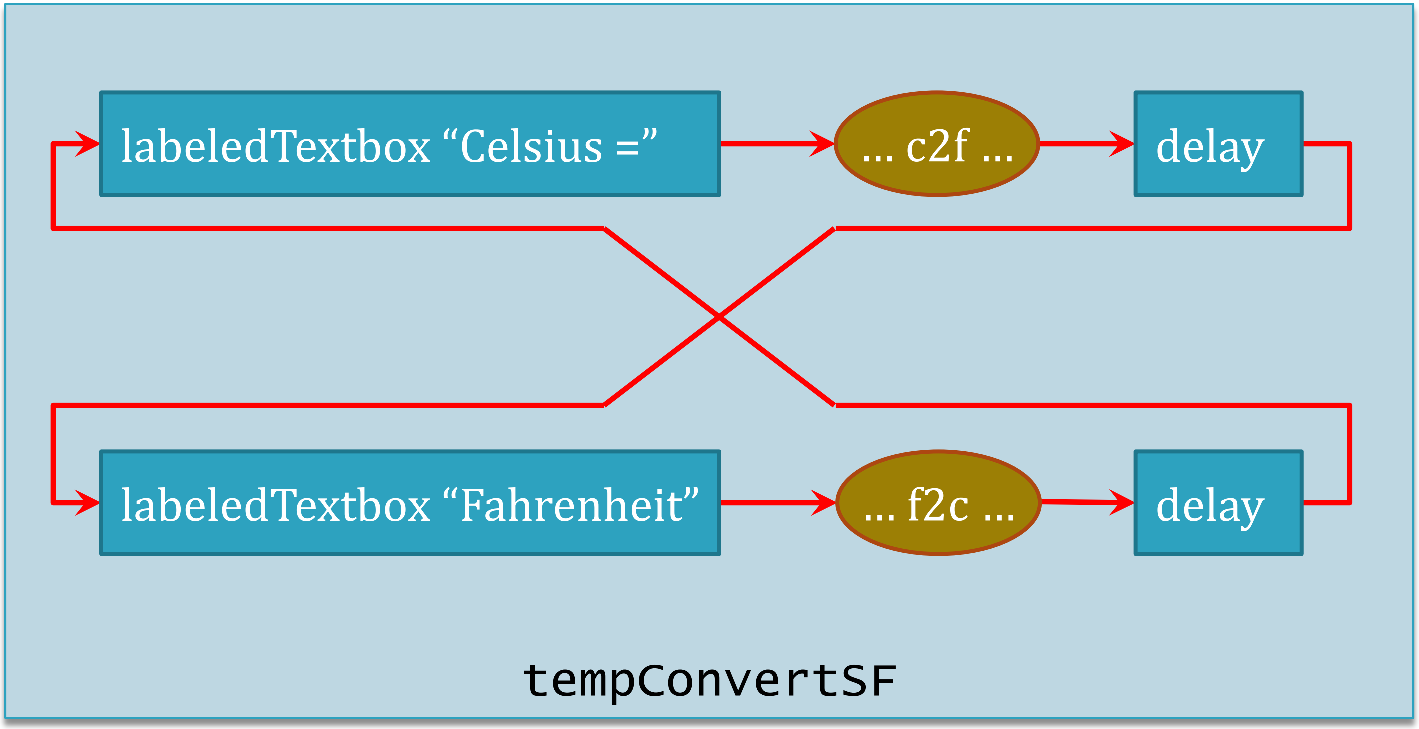
```
public class TemperatureConverterReactFX extends Application {

    public void start(Stage stage) {
        TextField celsius = new TextField();
        TextField fahrenheit = new TextField();

        EventStream<String> celsiusStream =
                EventStreams.valuesOf(celsius.textProperty()).filter(Util::isNumeric);
        celsiusStream.map(Util::cToF).subscribe(fahrenheit::setText);
        EventStream<String> fahrenheitStream =
                EventStreams.valuesOf(fahrenheit.textProperty()).filter(Util::isNumeric);
        fahrenheitStream.map(Util::fToC).subscribe(celsius::setText);

        HBox root =
          new HBox(10, celsius, new Label("Celsius ="), fahrenheit, new Label("Fahrenheit"));
        root.setPadding(new Insets(10));

        stage.setScene(new Scene(root));
        stage.setTitle("Temperature Converter");
        stage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

* Code from https://github.com/eugenkiss/7guis

Arrows …

- Are a well-founded concept inspired by category theory.
- Create a tighter semantic connection between data.
- Enforce the appropriate abstraction of time.
  - By removing direct access to streams, we eliminate certain memory leaks and non-causal behaviors.
- Have a static structure, which makes them …
  - More suitable for resource constrained systems.
  - Highly amenable to optimizations (e.g. CCA).
- Have been used in Yampa, Nettle, Euterpea, etc.
- Look like *signal processing diagrams*.

AFRP (as a Diagram)



Haskell with UISF (AFRP)

```
tempConvertSF = leftRight $ proc () -> do
  rec c <- labeledTextbox "Celsius = " -< updateC
      f <- labeledTextbox "Fahrenheit" -< updateF
      updateF <- delay Nothing -< fmap (show . c2f) (c >>= readMaybe)
      updateC <- delay Nothing -< fmap (show . f2c) (f >>= readMaybe)
  returnA -< ()

main = runUI (defaultUIParams
                   {uiSize=(400, 24), uiTitle="Temp Converter"})
             tempConvertSF
```

* http://hackage.haskell.org/package/UISF

Drawbacks of (Arrowized) FRP

- Data varies over time, but arrows cannot.
  - This lack of dynamic behavior limits expressivity.
- I/O Bottleneck
  - Pure FRP cannot perform effects.
  - All inputs and outputs must be routed manually.
  - This is a potential security leak.
- Synchrony can be restrictive.

My Contributions

- Extend arrows to allow "predictably dynamic" behavior [ICFP '14].
  - Non-interfering choice adds expressivity to arrows.
- Add concurrency and asynchrony [submitted '15].
  - Wormholes allow communication for concurrency.
    - https://github.com/dwincort/CFRP
- Safe effects such as physical resource interaction memory access [PADL '12, HS '12].
  - Resource types address safety.

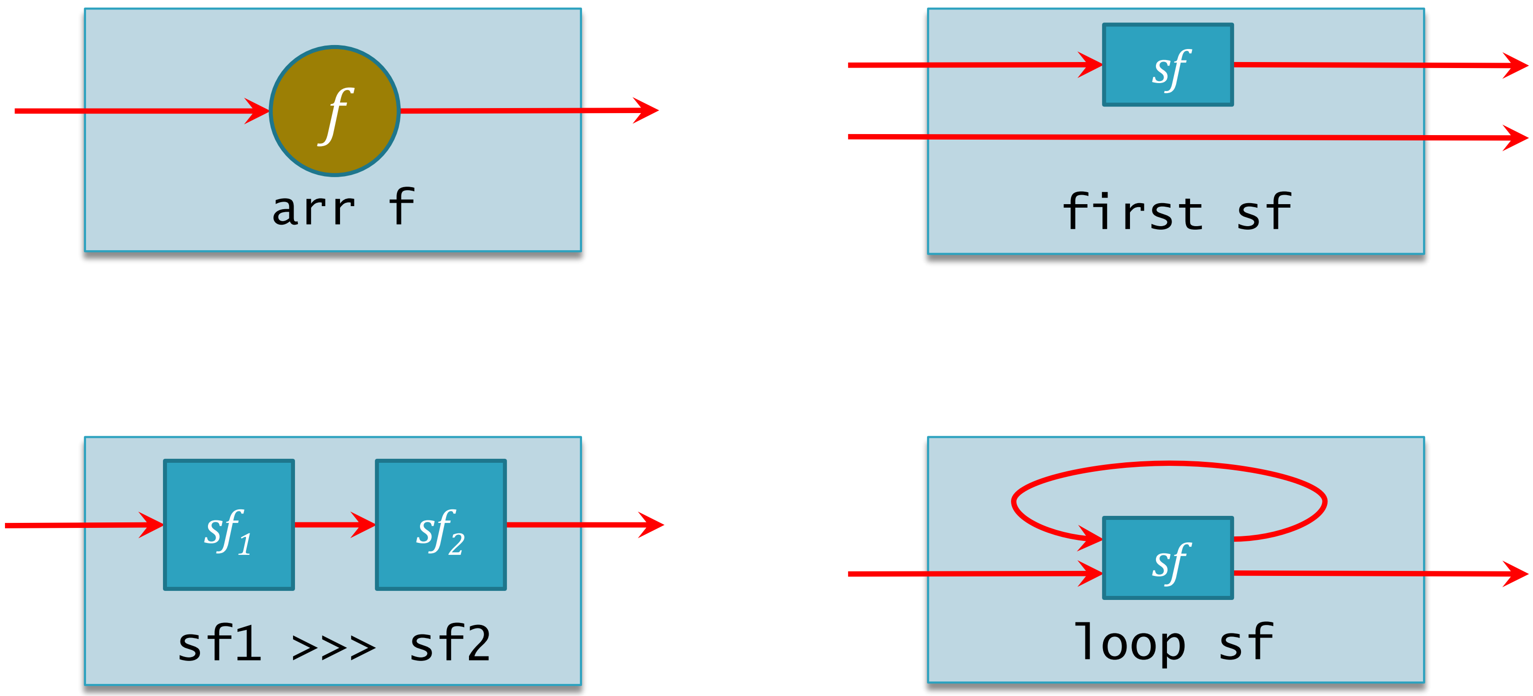
My Contributions

- Extend arrows to allow "predictably dynamic" behavior [ICFP '14].
  - Non-interfering choice adds expressivity to arrows.
- Add concurrency and asynchrony [submitted '15].
  - Wormholes allow communication for concurrency.
    - https://github.com/dwincort/CFRP
- Safe effects such as physical resource interaction memory access [PADL '12, HS '12].
  - Resource types address safety.

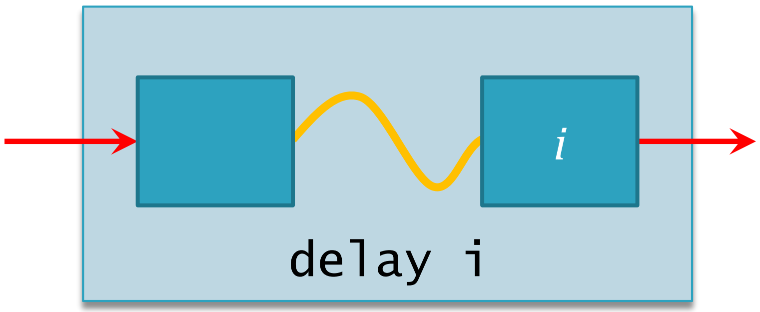
Expressing Arrows

How arrows work and what we need to express interesting computations

Standard Arrow Operators



Stateful Arrows



- With continuous semantics, the length of the delay approaches zero.
- When used in conjunction with loop, delay allows one to create stateful signal functions.

Dynamic Behavior

- Can we get more dynamic power for arrows?
- Why would we want that?

Example: Mind Map

Exploring predictably dynamic behavior

Example

- We would like a GUI to help a user build and navigate a "mind map."
  - A mind map is a mapping from keywords to values.
  - A user can look up a key to see its values, and then add new values.
- The GUI's appearance should dynamically update based on how many values the given key has.

Example

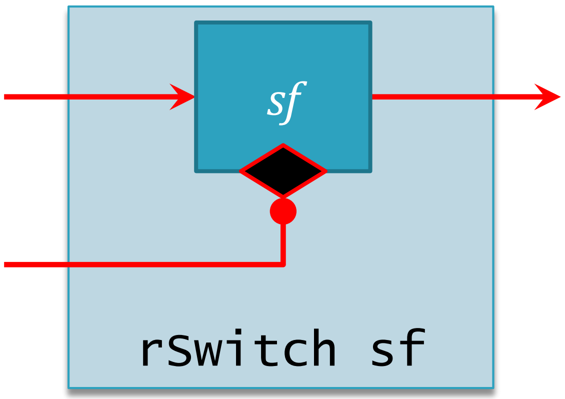
- We would like a GUI to help a user build and navigate a “mind map.”
  - A mind map is a mapping from keywords to values.
  - A user can look up a key to see its values, and then add new values.
- The GUI’s appearance should dynamically update based on how many values the given key has.
- -Demo-

Mindmap in code

```
mindmap :: MindMap -> UISF () ()
mindmap iMap = proc () -> do
    l <- textEntryField "Lookup" -< ()
    a <- textEntryField "Add"    -< ()
    key <- accum ""   -< fmap const l
    m   <- accum iMap -< fmap (\v -> insertWith (++) key [v]) a
    title "Key = " displayStr -< key
    runDynamic displayStr -< Map.findWithDefault [] key m
    returnA -< ()
```

- How do we write runDynamic?

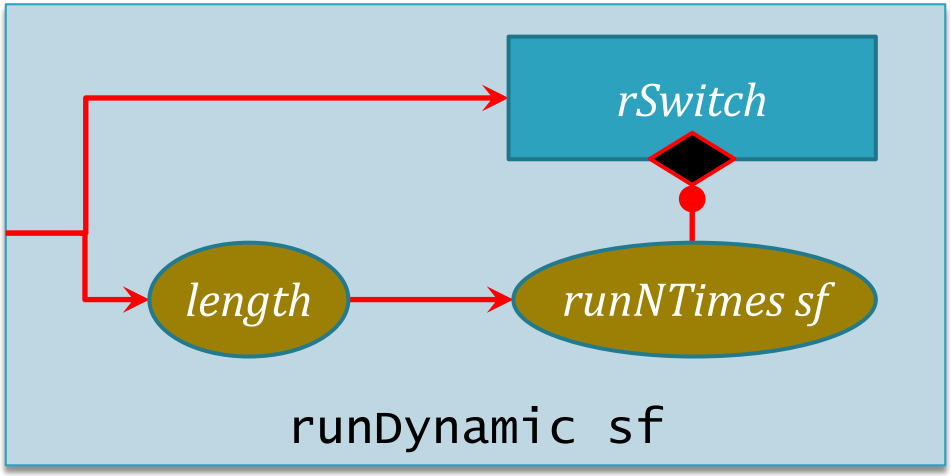
Higher Order Arrows



- The control signal determines the overall behavior.
  - This allows highly dynamic programs.
- Switched out signal functions are permanently off.
  - Switching can be used to increase performance.

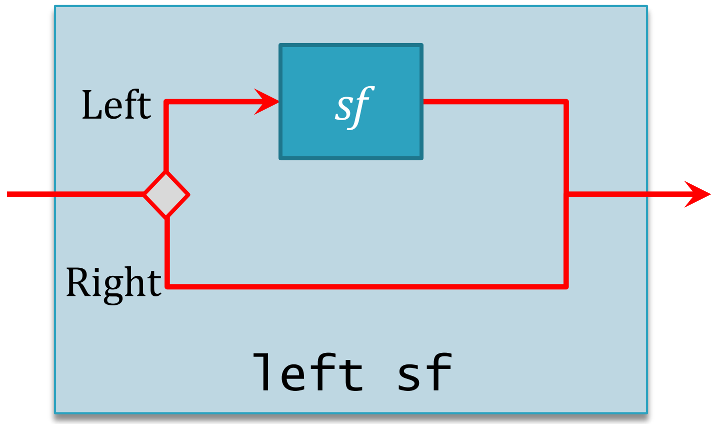
Implementing runDynamic

- We can create a new compound-widget when necessary and then switch into it:



- But this approach voids our static guarantees!
  - Arrows with switch are equivalent to Monads.
- It seems unnecessary – we are not running unknown functions.

Arrow Choice



- With choice, running the signal function is a dynamic decision.
- This seems to help, but it’s not enough.
  - We get fixed branching, but not true recursion.

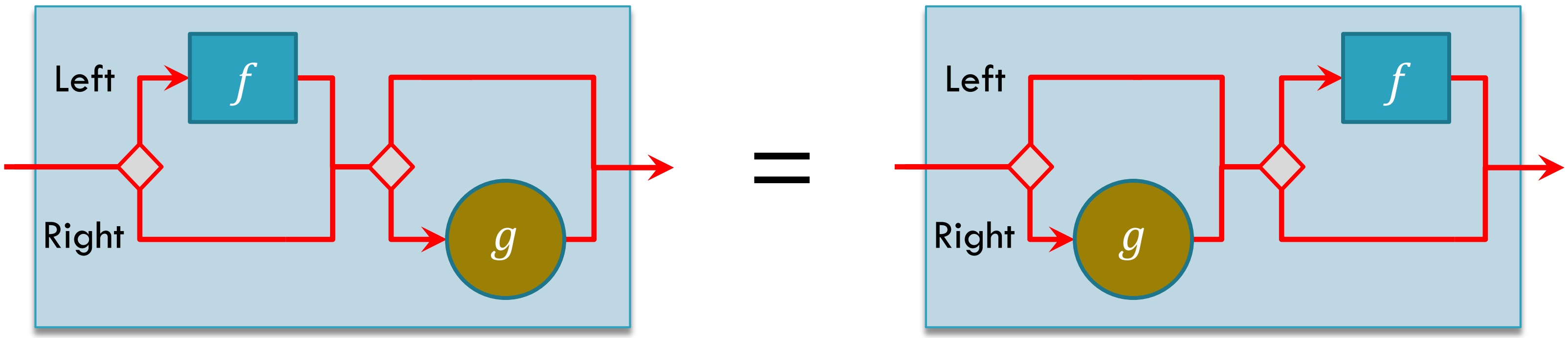
Arrow Choice Laws

Extension	`left (arr f) = arr (left f)`
Functor	`left (f >>> g) = left f >>> left g`
Exchange	`left f >>> arr (right g) =` `arr (right g) >>> left f`
Unit	`f >>> arr Left = arr Left >>> left f`
Assoc	`left (left f) >>> arr` $\texttt{assoc}_+$ `=` `arr` $\texttt{assoc}_+$ `>>> left f`

Arrow Choice Laws

Extension	`left (arr f) = arr (left f)`
Functor	`left (f >>> g) = left f >>> left g`
Exchange	`left f >>> arr (right g) = arr (right g) >>> left f`
Unit	`f >>> arr Left = arr Left >>> left f`
Assoc	$\texttt{left (left f) >>> arr assoc}_{+}\texttt{ = arr assoc}_{+}\texttt{ >>> left f}$

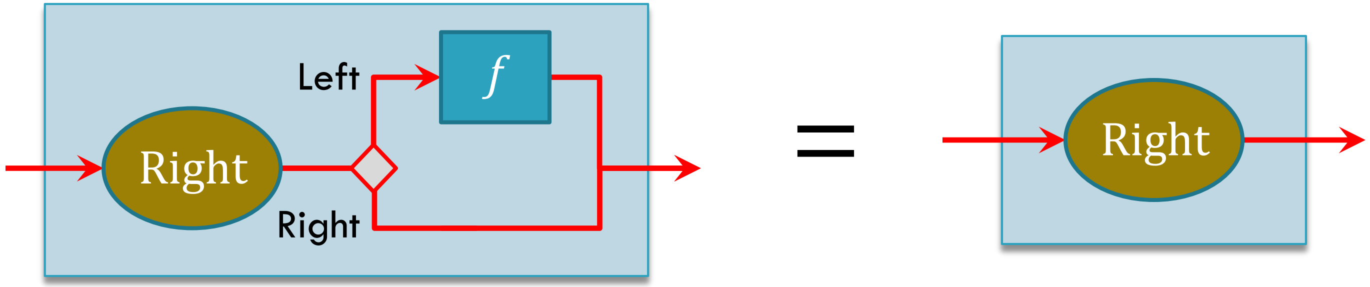
Exchange



- Why isn't this commutative?
  - Some arrows have effects.
  - For instance, UISF uses arrow order to determine widget layout.
- These effects make recursion impossible.
- In general, arrows are not commutative, but for choice in FRP, they can be.

Non-Interference

- We strengthen exchange into **non-interference**

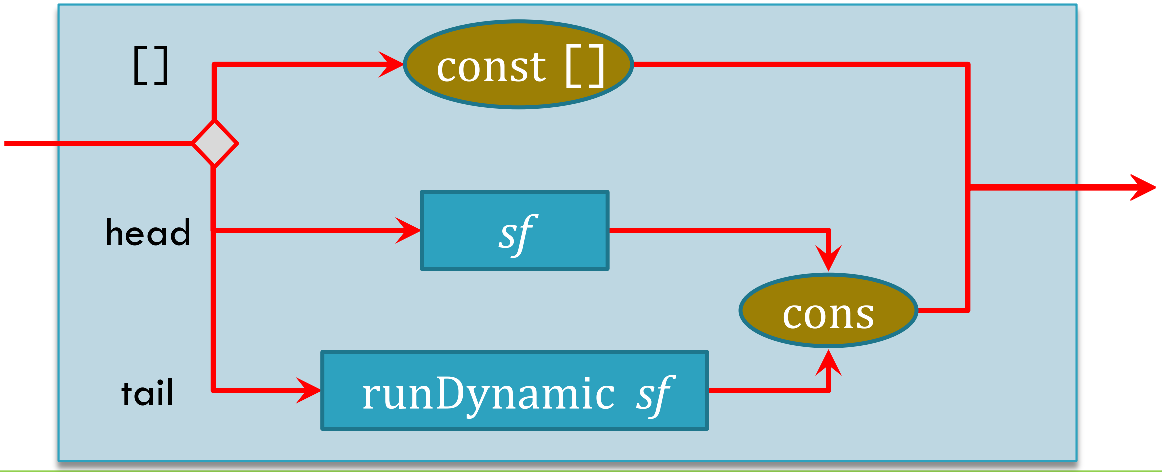


- If the input value is Right, then the program will behave the same whether there is a left function after it or not.
- The unused branch is now guaranteed to not run.
- Now we can use Arrow Choice for recursion!

runDynamic Revisited

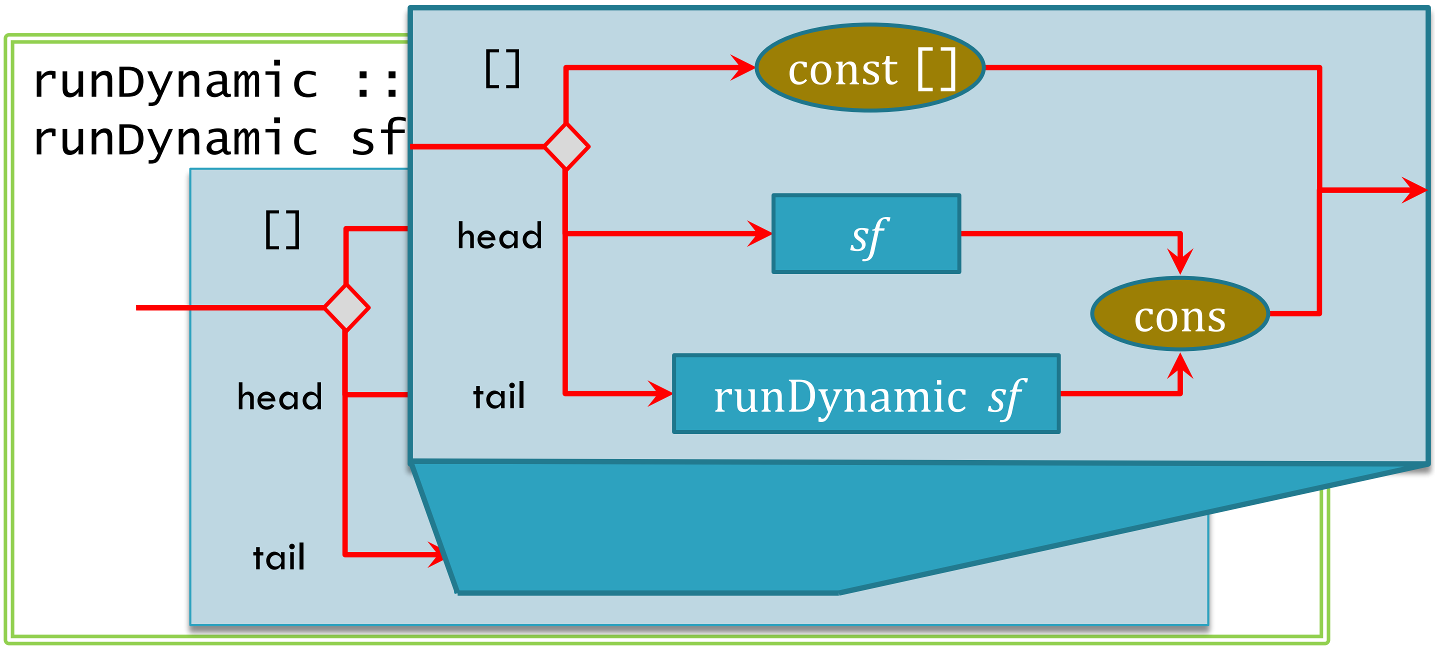
- Arrowized recursion allows us to write this without using switch.

```
runDynamic :: (a ~> b) -> ([a] ~> [b])
runDynamic sf =
```



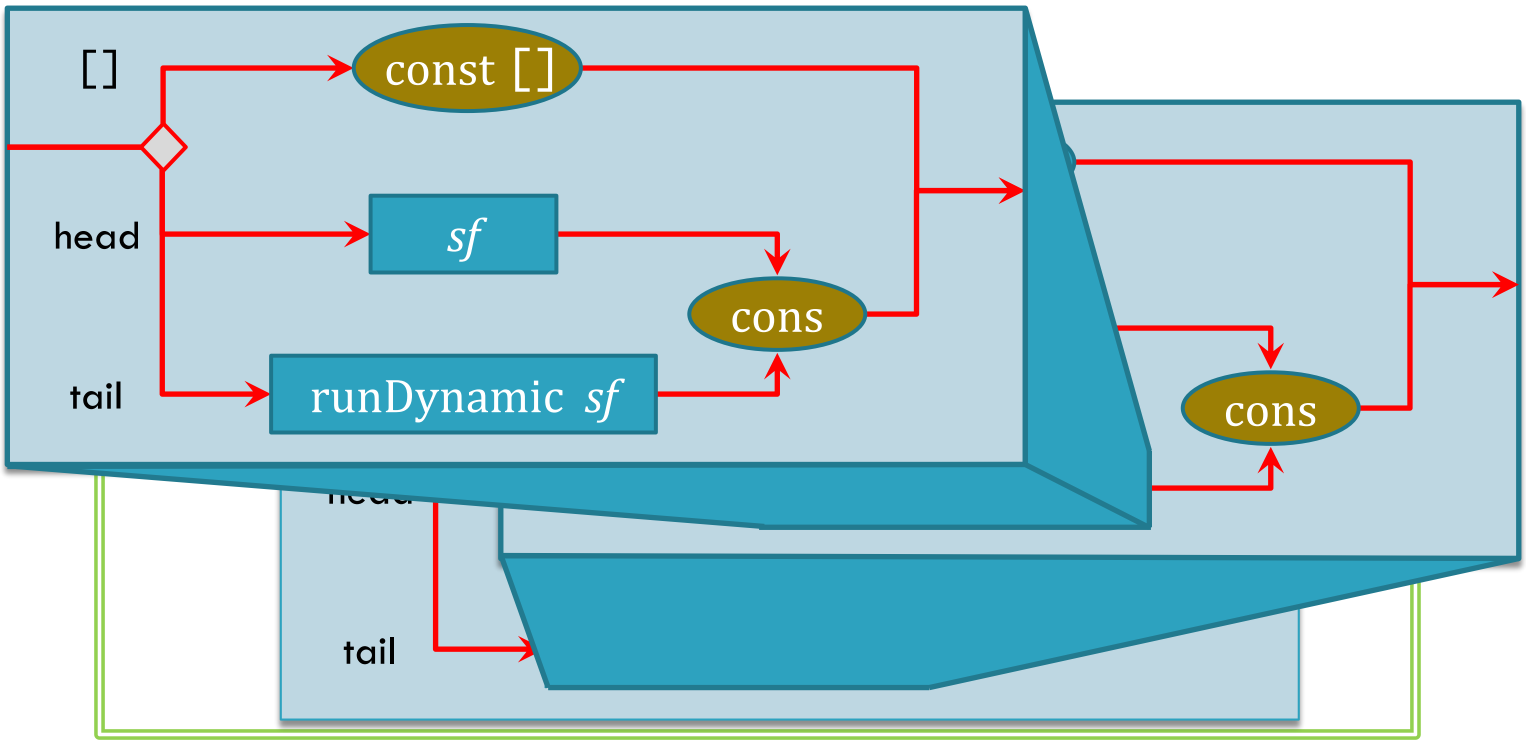
runDynamic Revisited

- Arrowized recursion allows us to write this without using switch.



runDynamic Revisited

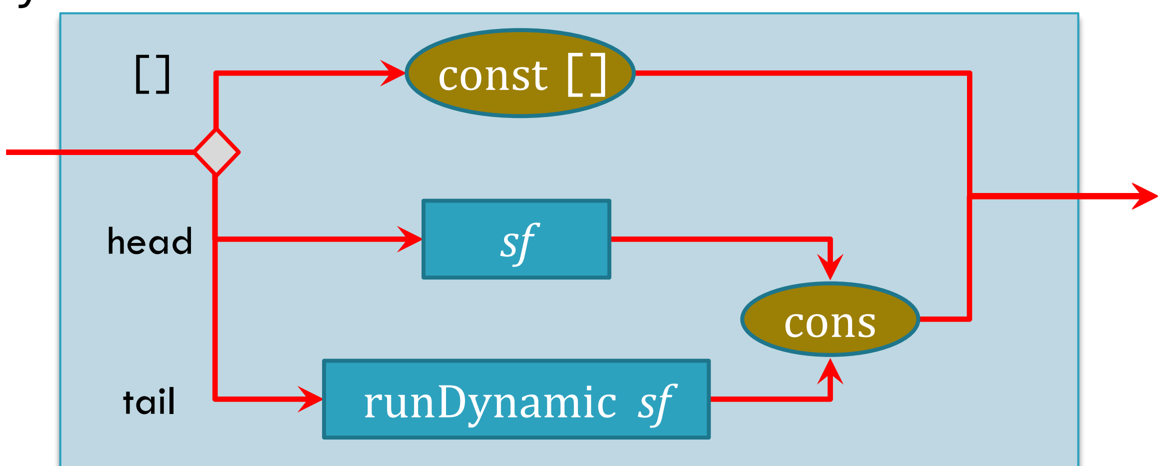
- Arrowized recursion allows us to write this without



runDynamic Revisited

- Arrowized recursion allows us to write this without using switch.

```
runDynamic :: (a ~> b) -> ([a] ~> [b])
runDynamic sf =
```



- The arrow structure is not technically static, but it is predictably dynamic.

Non-Interfering Choice Wrap-Up

- Like switch, non-interfering choice (and thus arrowized recursion) only computes when needed.
- The predictable nature of non-interfering choice does not interfere with optimizations.
  - The CCA transformation is still applicable.
- Time complexity can now be variable, but resource allocation is still static (arrow dependent).

My Contributions

- Extend arrows to allow "predictably dynamic" behavior [ICFP '14].
  - Non-interfering choice adds expressivity to arrows.
- Add concurrency and asynchrony [submitted '15].
  - Wormholes allow communication for concurrency.
    - https://github.com/dwincort/CFRP
- Safe effects such as physical resource interaction memory access [PADL '12, HS '12].
  - Resource types address safety.

Example: Connect Four

Allowing local asynchronous concurrency

Example

- We would like a GUI to play a game of Connect 4.
  - It should follow the rules of the game.
  - After the user makes a play, an AI should play.

Example

- We would like a GUI to play a game of Connect 4.
  - It should follow the rules of the game.
  - After the user makes a play, an AI should play.
- -Demo-

Connect Four GUI

```
connectFour = proc () -> do
  rec aiLevel <- title "AI Level" (hiSlider 1 (0, 5) 2) -< ()
      select  <- displayBoard numCols 10 -< board
      board   <- hold initBoard -< fmap (makeMove board) $
        case (turn board) of
          X -> fmap (,X) select
          O -> findBestMove O aiLevel board
  case (isWin board) of
    Nothing -> label "" -< ()
    Just X  -> label "You win!" -< ()
    Just O  -> label "You lose!" -< ()
```

Connect Four GUI

```
connectFour = proc () -> do
  rec aiLevel <- title "AI Level" (hiSlider 1 (0, 5) 2) -< ()
      select  <- displayBoard numCols 10 -< board
      board   <- hold initBoard -< fmap (makeMove board) $
        case (turn board) of
          X -> fmap (,X) select
          O -> findBestMove O aiLevel board
  case (isWin board) of
    Nothing -> label "" -< ()
    Just X  -> label "You win!" -< ()
    Just O  -> label "You lose!" -< ()
```

- When we ramp up the AI level, we find a problem.
  - -Demo-

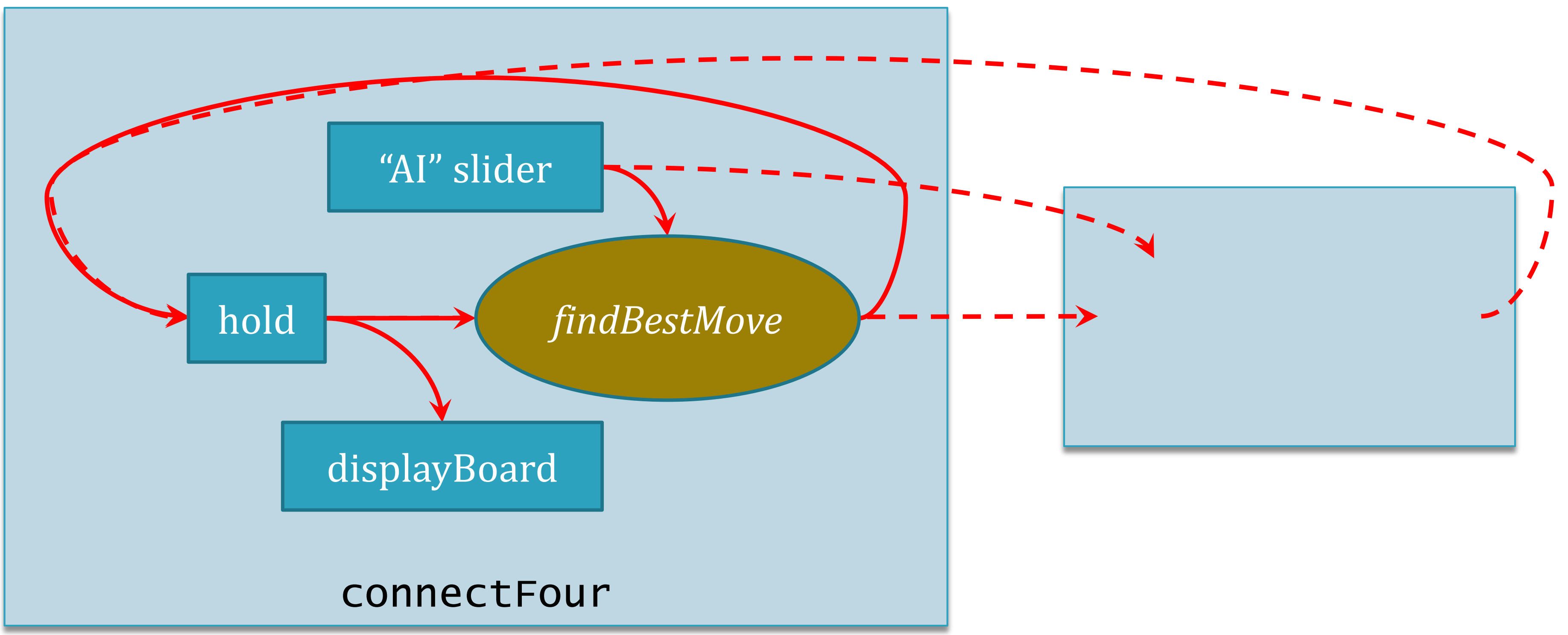
Synchrony Can Be a Burden

- The two parts would like to run at different rates.
  - The GUI should continue running at ~60FPS.
  - The AI should be allowed to run as slow as it needs to.
- The synchronous assumption of FRP is too strong.
- Other examples include …
  - Memory reads together with hard drive seeks.
  - Packet routing together with network map updating.
  - Sound synthesis together with a GUI interface.

Asynchrony

- Let us allow multiple processes, each with its own notion of time.
  - Each will individually remain synchronous and causal.
  - However, they will no longer synchronize.

Connect Four GUI Diagram

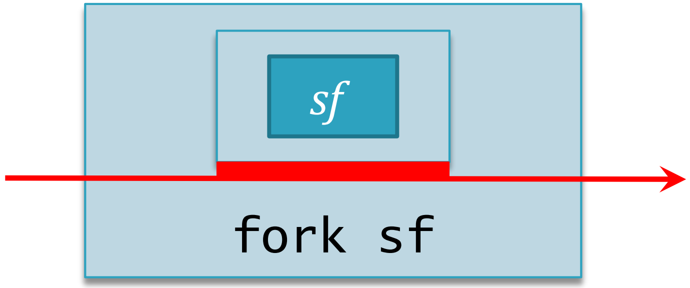


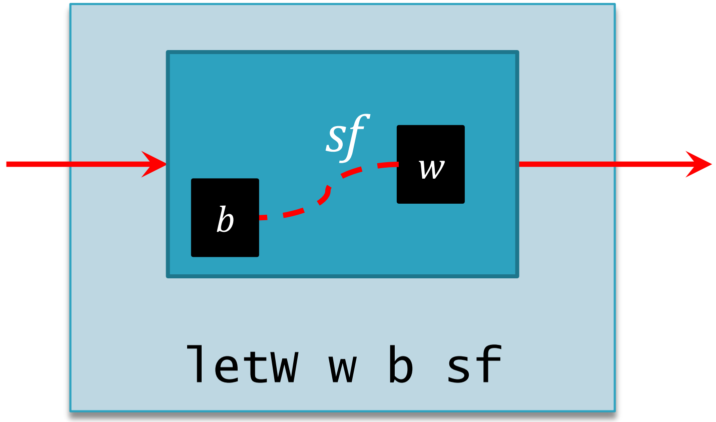
- But what are those dashed lines?

Inter-Process Communication

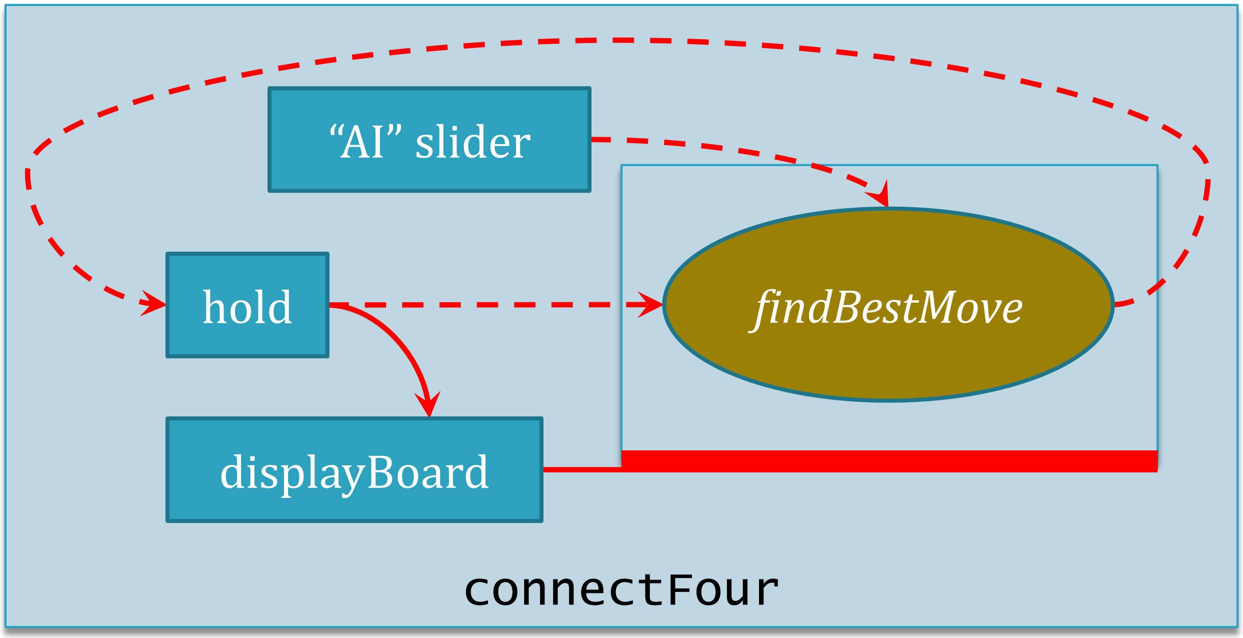
- We need a way to communicate data from one time stream to another.
- Data needs to get time dilated – either stretched or compressed.
- A special form of channel: Wormholes
  - Wormholes have a blackhole for writing to and a whitehole for reading from.
  - Wormholes automatically dilate their data.

New Operators





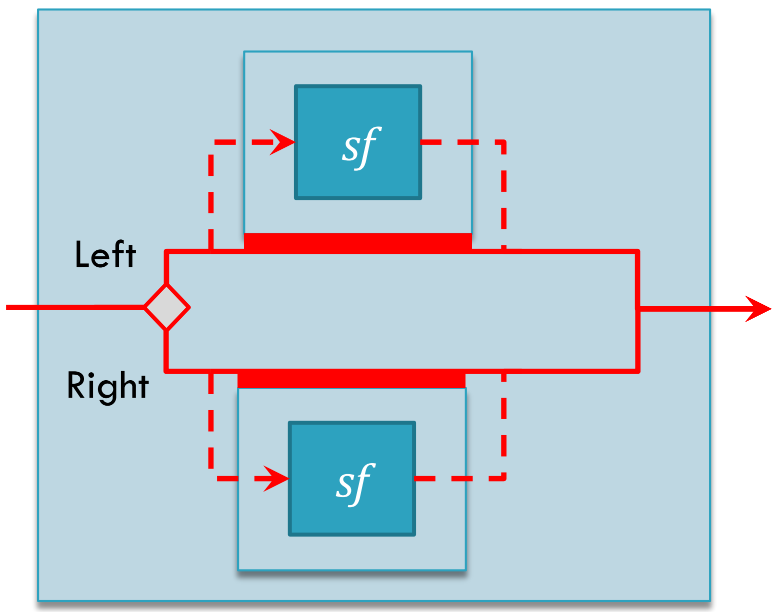
Connect Four GUI Diagram 2



- Now, findBestMove can run with its own clock.
- The data is communicated clearly via wormholes.

Maintaining Modular Consistency

- How can we control forked processes?



Asynchronous Choice

- Remember that data is time-dependent.
  - When a signal function has no incoming data, it must freeze.
  - Likewise, if a fork has no incoming data, it freezes its forked process.
- We achieve this while guaranteeing consistency.
  - Treat every moment in time as a transaction.
  - Freezing may occur between transactions.

Asynchrony Wrap-Up

- We can create multiple time streams for different FRP components.
  - Each time stream is internally synchronous and deterministic.
- We can communicate between time streams in a clear way with wormholes.
  - Data is automatically time dilated.
- We can govern time streams using non-interfering choice.

My Contributions

- Extend arrows to allow "predictably dynamic" behavior [ICFP '14].
  - Non-interfering choice adds expressivity to arrows.
- Add concurrency and asynchrony [submitted '15].
  - Wormholes allow communication for concurrency.
    - https://github.com/dwincort/CFRP
- Safe effects such as physical resource interaction memory access [PADL '12, HS '12].
  - Resource types address safety.

Example: MIDI Echo Player

Allowing effects in a meaningful yet safe manner

Example

- We would like a GUI to control the parameters of an echo effect that we can add to a MIDI stream.
  - MIDI stands for Musical Instrument Digital Interface.
  - An echo decays and loops the sound.
- The program should read from and write to a MIDI port.

Example

- We would like a GUI to control the parameters of an echo effect that we can add to a MIDI stream.
  - MIDI stands for Musical Instrument Digital Interface.
  - An echo decays and loops the sound.
- The program should read from and write to a MIDI port.
- -Demo-

Echo GUI

```
echo :: UISF () ()
echo = proc () -> do
  m <- midiIn -< ()
  r <- title "Decay rate"          (hSlider (0, 0.9) 0.6) -< ()
  f <- title "Echoing frequency" (hSlider (1, 10)  3)   -< ()
  rec let m' = m <> s
      s <- vdelay -< (1.0 / f, decay 0.1 r m')
  midiOut -< m'
```

- Let's also add a metronome tick to this.

Echo GUI

```
echo :: UISF () ()
echo = proc () -> do
  m <- midiIn -< ()
  r <- title "Decay rate"         (hSlider (0, 0.9) 0.6) -< ()
  f <- title "Echoing frequency" (hSlider (1, 10)  3)   -< ()
  rec let m' = m <> s
      s <- vdelay -< (1.0 / f, decay 0.1 r m')
  midiOut -< m'

metronomeTick :: UISF () ()
metronomeTick = proc () -> do
  bpm <- title "Metronome BPM" (hSlider (40, 200) 100) -< ()
  e <- timer -< 60 / bpm
  midiOut -< makeTick e
```

Echo GUI

```
echo :: UISF () ()
echo = proc () -> do
  m <- midiIn -< ()
  r <- title "Decay rate"         (hSlider (0, 0.9) 0.6) -< ()
  f <- title "Echoing frequency" (hSlider (1, 10)  3)    -< ()
  rec let m' = m <> s
      s <- vdelay -< (1.0 / f, decay 0.1 r m')
  midiOut -< m'

metronomeTick :: UISF () ()
metronomeTick = proc () -> do
  bpm <- title "Metronome BPM" (hSlider (40, 200) 100) -< ()
  e <- timer -< 60 / bpm
  midiOut -< makeTick e

runUI defaultUIParams (echo >>> metronomeTick)
```

Multiple midiOut Effects

- What happens when we send MIDI output twice in one program?
  - The two input streams merge in some way?
  - The top input stream processes first?
- This may break our functional guarantee.
  - Blocks of code are no longer modular.
  - The UISF layout is determined by program structure.
    - Layout is determined statically ("predictably dynamic").
    - Computation and layout are totally separate.

Adding Effects

- To make effects safe, we must limit how we use effectful signal functions.
  - If an effect is used, it can only be used in one place.
- We achieve this by tagging signal functions at the type level with *resource types* and restricting their composition.

Resource Typed Arrow Operators

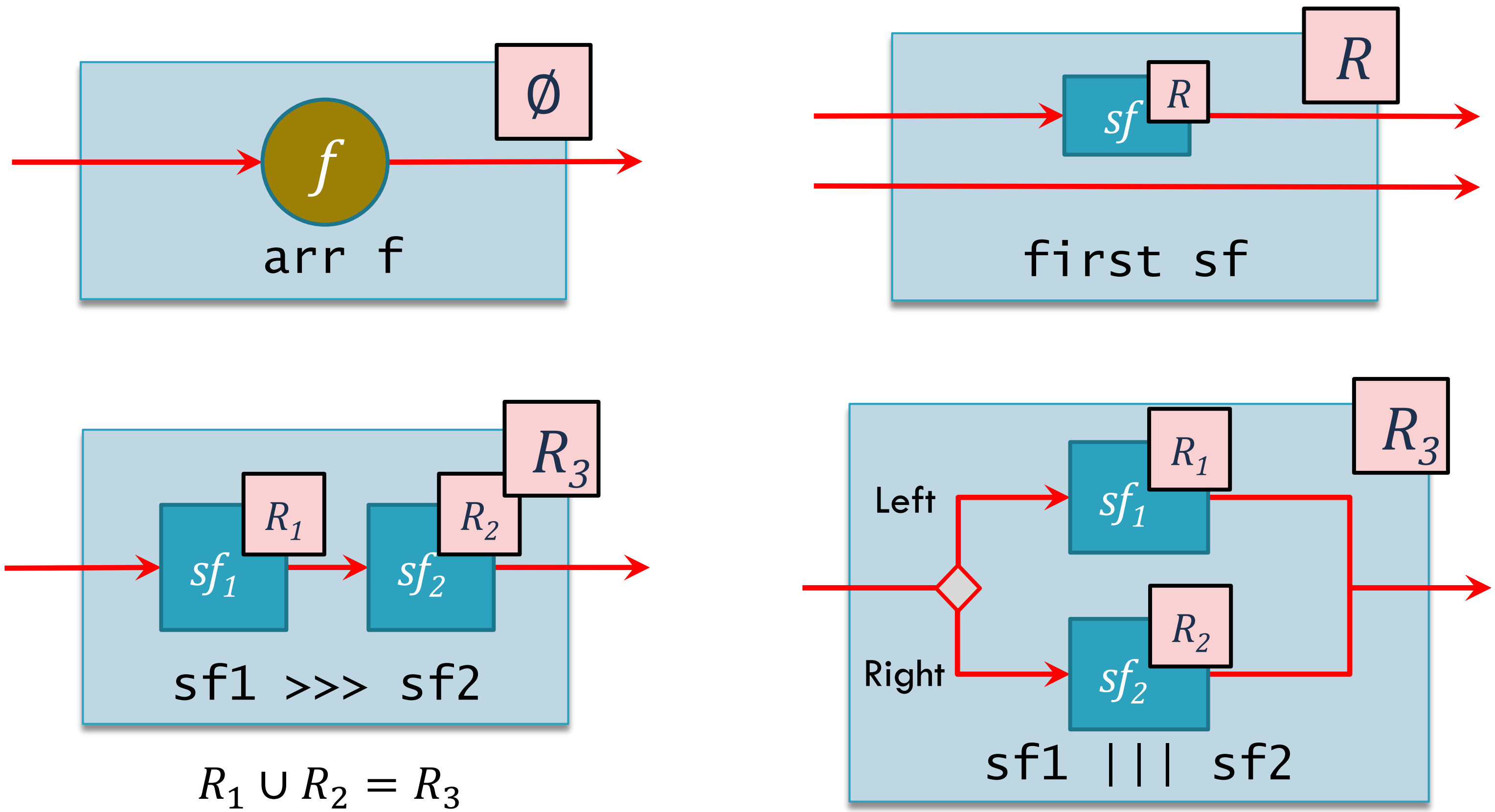
Ty-Arr $$\frac{\Gamma \vdash e : \alpha \rightarrow \beta}{\Gamma;\Psi \vdash arr\ e : \alpha \overset{\emptyset}{\rightsquigarrow} \beta}$$

Ty-First $$\frac{\Gamma;\Psi \vdash e : \alpha \overset{R}{\rightsquigarrow} \beta}{\Gamma;\Psi \vdash first\ e : (\alpha \times \gamma) \overset{R}{\rightsquigarrow} (\beta \times \gamma)}$$

Ty-Comp $$\frac{\Gamma;\Psi \vdash e_1 : \alpha \overset{R_1}{\rightsquigarrow} \beta \quad \Gamma;\Psi \vdash e_2 : \beta \overset{R_2}{\rightsquigarrow} \gamma \quad R_1 \uplus R_2 = R}{\Gamma;\Psi \vdash e_1 >>> e_2 : \alpha \overset{R}{\rightsquigarrow} \gamma}$$

Ty-Chc $$\frac{\Gamma;\Psi \vdash e_1 : \alpha \overset{R_1}{\rightsquigarrow} \gamma \quad \Gamma;\Psi \vdash e_2 : \beta \overset{R_2}{\rightsquigarrow} \gamma \quad R_1 \cup R_2 = R}{\Gamma;\Psi \vdash e_1 ||| e_2 : (\alpha + \beta) \overset{R}{\rightsquigarrow} \gamma}$$

Resource Typed Arrow Operators



$R_1 \cup R_2 = R_3$
$R_1 \cap R_2 = \emptyset$

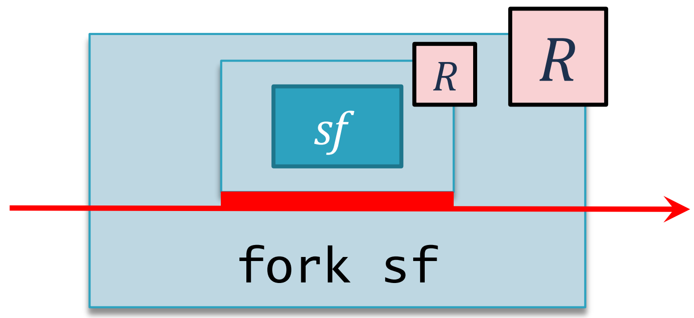
$R_1 \cup R_2 = R_3$

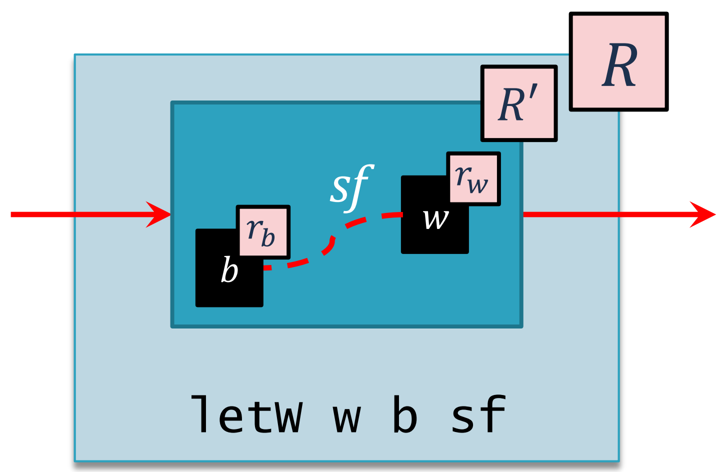
Resource Typed Arrow Operators

Ty-Fork
$$\frac{\Gamma;\Psi\vdash e : () \overset{R}{\rightsquigarrow} ()}{\Gamma;\Psi\vdash fork\ e : \alpha \overset{R}{\rightsquigarrow} \alpha}$$

Ty-LetW
$$\frac{\begin{array}{c}\Gamma;\Psi,r_w : \langle (),List\ \tau\rangle,r_b : \langle \tau,()\rangle\vdash e : \alpha \overset{R'}{\rightsquigarrow} \beta \\ \Gamma;\Psi\vdash e_i : List\ \tau \quad R=R' \setminus \{r_w,r_b\}\end{array}}{\Gamma;\Psi\vdash \mathrm{letW}\ r_w\ r_b\ e_i\ \mathrm{in}\ e : \alpha \overset{R}{\rightsquigarrow} \beta}$$

Resource Typed Arrow Operators





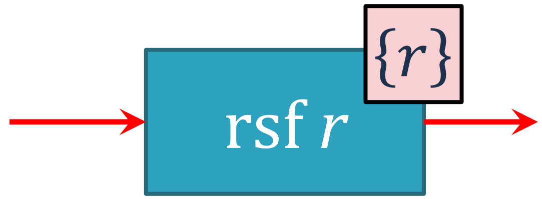
$$R = R' \setminus \{r_b, r_w\}$$

Resource Signal Function

$$\text{Ty-RSF} \quad \frac{(r : \langle \tau_{in}, \tau_{out} \rangle) \in \Psi}{\Gamma ; \Psi \vdash rsf\ r : \tau_{in} \overset{\{r\}}{\rightsquigarrow} \tau_{out}}$$

- All physical devices have an associated virtual resource.

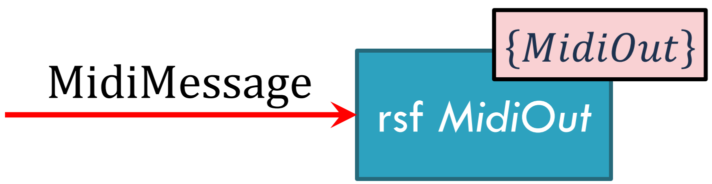
Resource Signal Function



- All physical devices have an associated virtual resource.

FRP I/O Effects

- Back to our example:
  - We can send MIDI data by using the MidiOut resource:



- We are assured that the input stream is unique.
- The *type* of a program shows its resource usage:
  - Our poorly-defined metronome/echo program will no longer type check.

```
echo          :: UISF {MidiIn, MidiOut} () ()
metronomeTick :: UISF {MidiOut}         () ()

echo >>> metronomeTick :: TYPE ERROR
```

Formalism

- Operational semantics describe the behavior of fork and wormholes with arrows.
- The semantics proceed in a 3-phase set of transitions:

$$
\begin{array}{rcll}
e & \mapsto & e' & \text{Evaluation transition} \\
(S, T, \mathscr{R}, \mathscr{W}) & \Rrightarrow & (S', T', \mathscr{R}', \mathscr{W}') & \text{Functional transition} \\
(T, \mathscr{R}, \mathscr{W}) & \Downarrow & (T', \mathscr{R}', \mathscr{W}') & \text{Executive transition}
\end{array}
$$

- The evaluation transition is a classic, non-strict, functional semantics.

Formalism – Functional Transition

$$\textsc{Ft-Fork}\ \frac{p \text{ fresh}, \quad T' = T[p \mapsto \varepsilon \rhd (e, (), \emptyset)]}{(K \rhd (\mathit{fork}\ e, x, U), T) \Rrightarrow (K \lhd (\mathit{fork}\ e\ p, x, U), T')}$$

$$\textsc{Ft-Fork}_p\ \frac{T' = \textbf{if } p \in \mathit{Dom}(T) \textbf{ then } T \textbf{ else } T[p \mapsto \varepsilon \rhd (e, (), \emptyset)]}{(K \rhd (\mathit{fork}\ e\ p, x, U), T) \Rrightarrow (K \lhd (\mathit{fork}\ e\ p, x, U), T')}$$

$$\textsc{Ft-RSF}_r\ \frac{r \in \mathscr{R} \quad U' = (r, x) :: U \quad y = \text{read } r\ \mathscr{R}(r)}{(K \rhd (\mathit{rsf}\ r, x, U), \mathscr{R}, \mathscr{W}) \Rrightarrow (K \lhd (\mathit{rsf}\ r, y, U'), \mathscr{R}, \mathscr{W})}$$

$$\textsc{Ft-RSF}_w\ \frac{r \in \mathscr{W} \quad U' = (r, x) :: U \quad y = \text{read } r\ \mathscr{R}(\mathit{fst}\ \mathscr{W}(r))}{(K \rhd (\mathit{rsf}\ r, x, U)\mathscr{R}, \mathscr{W}) \Rrightarrow (K \lhd (\mathit{rsf}\ r, y, U')\mathscr{R}, \mathscr{W})}$$

$$\textsc{Ft-LetW}\ \frac{r \text{ fresh} \quad \mathscr{R}' = \mathscr{R}[r \mapsto (\varepsilon, e_i)] \quad \mathscr{W}' = \mathscr{W}[r_b \mapsto (r, \mathrm{B}), r_w \mapsto (r, \mathrm{W})]}{(K \rhd (\textbf{letW}\ r_w\ r_b\ e_i \textbf{ in } e, x, U), \mathscr{R}, \mathscr{W}) \Rrightarrow (K \rhd (e, x, U), \mathscr{R}', \mathscr{W}')}$$

$$\textsc{Ft-Time}\ \frac{\begin{array}{l} \mathscr{R}_1 = \mathscr{R}\ [r \mapsto \text{update } r\ \ \mathscr{R}(r)\ x \mid\ (r, x)\ \in U, r \in \mathscr{R}] \\ \mathscr{R}_2 = \mathscr{R}_1\ [r \mapsto \text{update } r_b\ \mathscr{R}_1(r)\ x \mid (r_b, x) \in U, \mathscr{W}(r_b) = (r, \mathrm{B})] \\ \mathscr{R}_3 = \mathscr{R}_2\ [r \mapsto \text{update } r_w\ \mathscr{R}_2(r)\ x \mid (r_w, x) \in U, \mathscr{W}(r_w) = (r, \mathrm{W})] \end{array}}{(\varepsilon \lhd (e, (), U), \mathscr{R}, \mathscr{W}) \Rrightarrow (\varepsilon \rhd (e, (), \emptyset), \mathscr{R}_3, \mathscr{W})}$$

Formalism – Functional Transition

- Choice is specially designed to handle freezing:

$$\textsc{Ft-Chc}_e\ \frac{x \mapsto x'}{(K \rhd (e_1 \mid\mid\mid e_2, x, U), T) \Rrightarrow (K \rhd (e_1 \mid\mid\mid e_2, x', U), T)}$$

$$\textsc{Ft-Chc}_{l1}\ \frac{T' = T \setminus (\mathsf{getChildrenOf}\ T\ e_2)}{(K \rhd (e_1 \mid\mid\mid e_2, \mathit{Left}\ x, U), T) \Rrightarrow (K; (\cdot \mid\mid\mid e_2) \rhd (e_1, x, U), T')}$$

$$\textsc{Ft-Chc}_{l2}\ \frac{}{(K; (\cdot \mid\mid\mid e_2) \lhd (e_1, z, U), T) \Rrightarrow (K \lhd (e_1 \mid\mid\mid e_2, z, U), T)}$$

$$\textsc{Ft-Chc}_{r1}\ \frac{T' = T \setminus (\mathsf{getChildrenOf}\ T\ e_1)}{(K \rhd (e_1 \mid\mid\mid e_2, \mathit{Right}\ y, U), T) \Rrightarrow (K; (e_1 \mid\mid\mid \cdot) \rhd (e_2, y, U), T')}$$

$$\textsc{Ft-Chc}_{r2}\ \frac{}{(K; (e_1 \mid\mid\mid \cdot) \lhd (e_2, z, U), T) \Rrightarrow (K \lhd (e_1 \mid\mid\mid e_2, z, U), T)}$$

Formalism – Executive Transition

- The executive transition runs the program.

$$\text{EXEC}\quad \frac{(p,S)\in T \qquad (S, T\setminus\{(p,S)\}, \mathscr{R}, \mathscr{W}) \Rightarrow (S', T', \mathscr{R}', \mathscr{W}')}{(T, \mathscr{R}, \mathscr{W}) \Downarrow (T' \cup \{(p,S')\}, \mathscr{R}', \mathscr{W}')}$$

- It chooses a process p non-deterministically and fairly and runs it.

Program execution is the application of the reflexive transitive closure over the EXEC transition $\Downarrow$ starting with initial parameters $T = \{(p, \varepsilon \rhd (e, (), \varepsilon))\}$, $\mathcal{R} = \mathcal{R}_0$, and $W = \emptyset$ where p is a fresh process ID, e is a process, and $\mathcal{R}_0$ is an initial mapping of resources representing those of the real world.

Theorem: Safety

For a program P: $\longrightarrow \boxed{sf}^{\boxed{R}} \longrightarrow$, we know:
- No program states will ever interact with a resource $r \notin R$.
- No two processes in P can interact with the same resource.
- No moment of time in P will ever interact with a resource more than once.

- The type reveals which resources a program can interact with when run.
- Forked processes will respect each others' resources.
- All resource streams are guaranteed unique.

Theorem: Resource Commutativity

For any S and r, if $(S, \mathcal{R}, \mathcal{W}) \hookrightarrow_p (S', \mathcal{R}', \mathcal{W}')$ is the set of states $S_0 \ldots S_n$ and there exists $i < n$ such that $S_i = \left(K \rhd (rsf\ r, \_, U_i)\right)$ and $S_{i+1} = \left(K \lhd (rsf\ r, x, U_{i+1})\right)$, then x will be the same for all S regardless of i.

- Resource types *enforce* data commutativity.
- Programs stay functional and modular.
- Reasoning about behavior through diagrams remains clear.

Effects Wrap-Up

- Effects can be inserted directly into FRP programs.
  - Resource types assure safety and data commutativity.
  - Invalid effect interactions are eliminated statically.
- Formal semantics demonstrate features.
  - Proofs are in the dissertation.

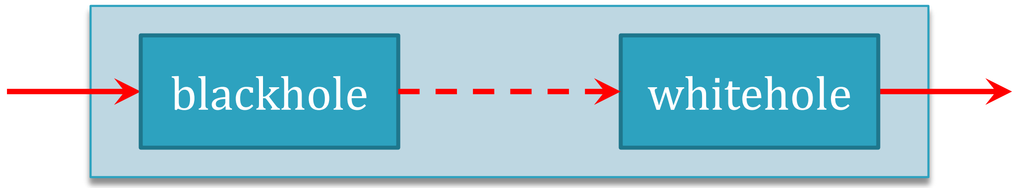
Other FRP Enhancing Efforts

More uses for Wormholes

- Wormholes provide communication between processes, but what if both ends are in the same process?
  - What kind of time dilation occurs?

More uses for Wormholes

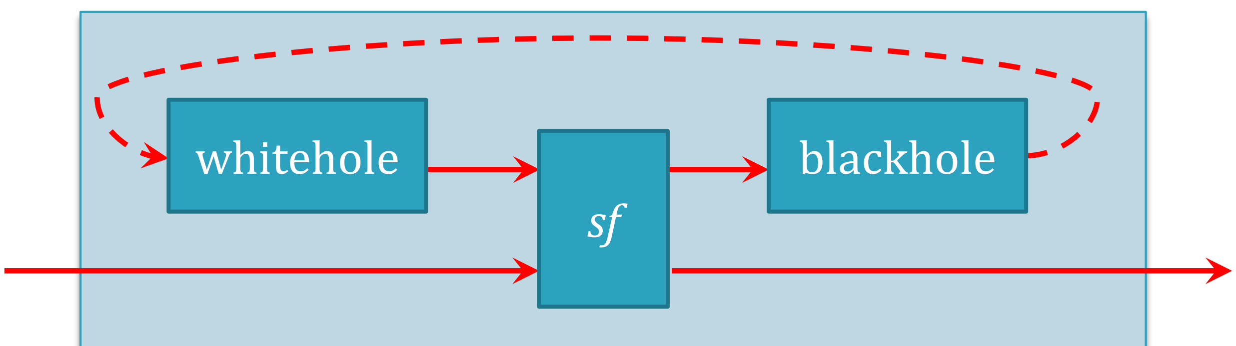
- Wormholes provide communication between processes, but what if both ends are in the same process?
  - What kind of time dilation occurs?
- A blackhole into a whitehole:



- We create delay.

More uses for Wormholes

- Wormholes provide communication between processes, but what if both ends are in the same process?
  - What kind of time dilation occurs?
- A whitehole into a blackhole:



- We create a strictly causal form of loop.

More uses for Wormholes

- Wormholes provide communication between processes, but what if both ends are in the same process?
  - What kind of time dilation occurs?
- In arbitrary locations:
  - We achieve non-local memory mutation.

Other Results

- Settability – A transformation applicable to AFRP that creates access to internal state.
  - https://github.com/dwincort/SettableArrow
- A non-interfering choice extension to CCA with comparable performance.
  - https://github.com/dwincort/CCA
- An alternate back-end for rec-delay syntax that uses wormholes to statically prevent infinite loops.

Conclusions

Contributions

- Safer FRP
  - Resource types track and limit effects.
- More Efficient FRP
  - Static arrows can be greatly optimized.
  - Concurrent processing can leverage multiple cores.
- More Expressive FRP
  - Non-interfering choice provides predictably dynamic behavior.
  - Effects can be used within the computation.
  - Concurrency allows multiple simultaneous clock rates.

Future Work

- Dynamic Resource Types
  - Wormhole resources cannot be fully implemented in GHC without a significant extension.
- Deterministic Parallelism
  - Can we make deterministic guarantees about predictable concurrent programs?
- Optimization
  - CCA transformation with Non-Interfering Choice needs to be more robust.

Thank you!

Questions?

Contributions

- Safer FRP
  - Resource types track and limit effects.
- More Efficient FRP
  - Static arrows can be greatly optimized.
  - Concurrent processing can leverage multiple cores.
- More Expressive FRP
  - Non-interfering choice provides predictably dynamic behavior.
  - Effects can be used within the computation.
  - Concurrency allows multiple simultaneous clock rates.

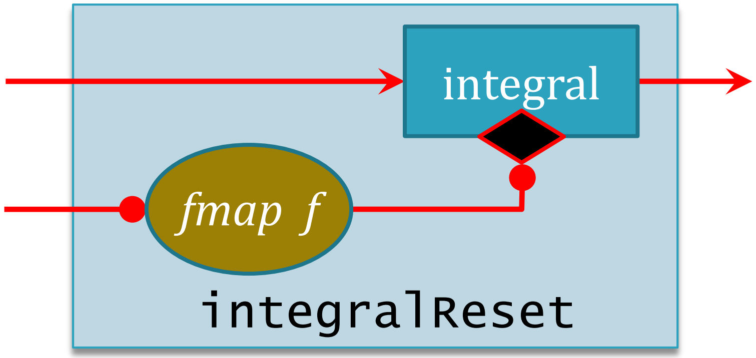
EXTRA SLIDES

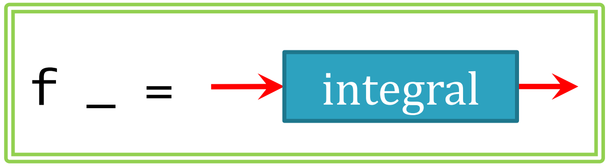
Settability

Saving, loading, and resetting signal functions

Example: IntegralReset

- A signal function that calculates an integral but can be reset with an event.

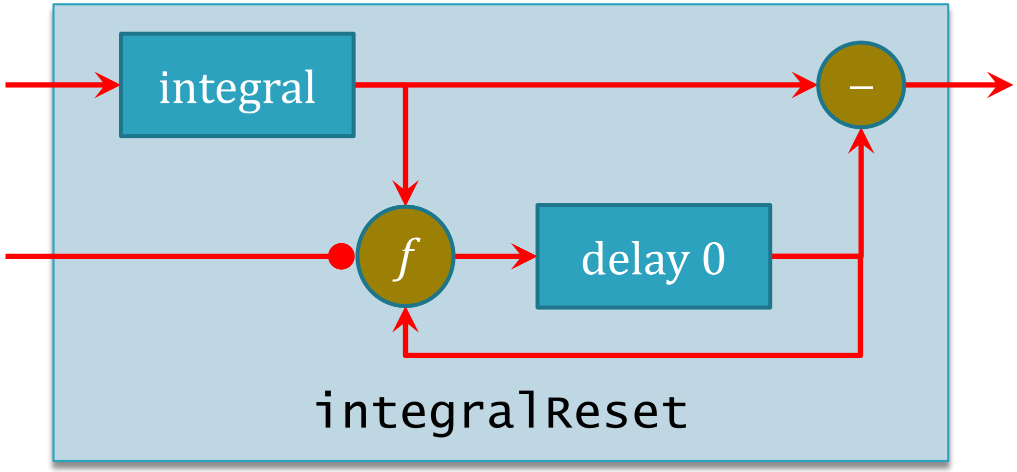




- Can we even do this without switch?

Example: IntegralReset

- Without switch, we can simulate a reset, but we can't modify integral itself.




```
f v e k = if isEvent e then v else k
```

- This solution is inelegant and does not scale.

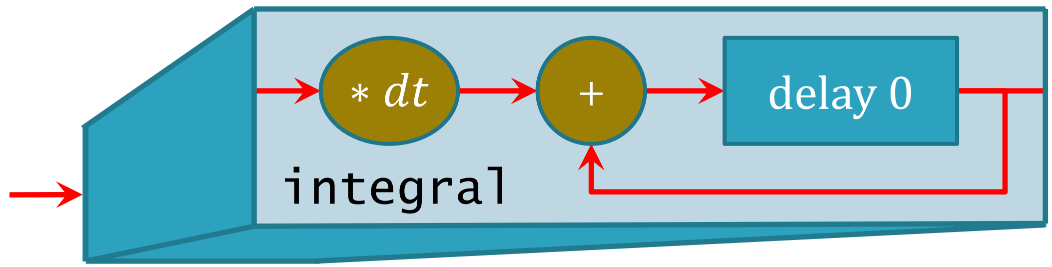
Resetting State

- We want to access the state inside a signal function.

- But what's inside of an arbitrary signal function?
- All state is saved with loop and delay.

Resetting State

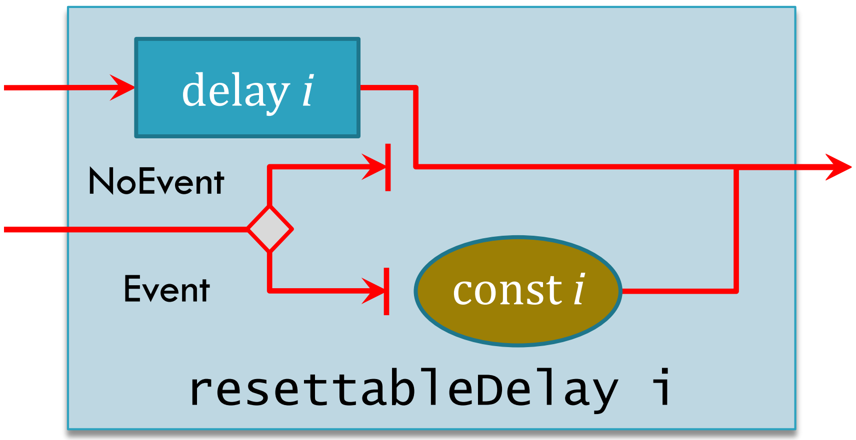
- We want to access the state inside a signal function.



- If we could reach in and restart the delay, then integral would behave as if it just started.

Resettable Delay

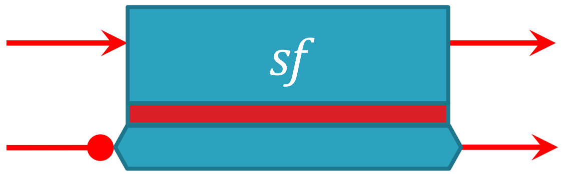
- Let's consider a new delay that can be reset directly.



- When the event is given, resettableDelay reverts to its starting state.
- Does this scale? YES

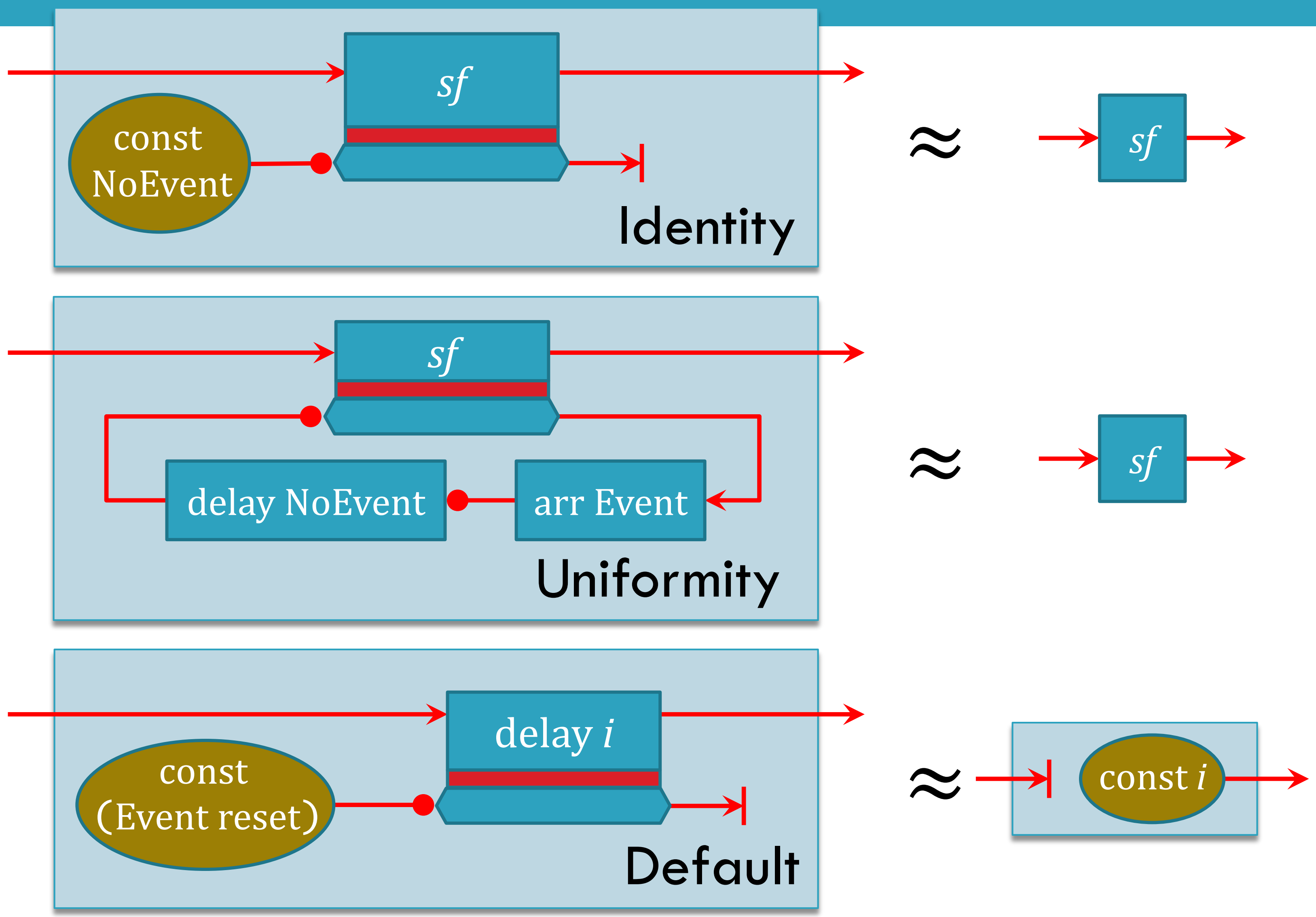
General Settability

- We can take any signal function and transform it into a settable signal function:



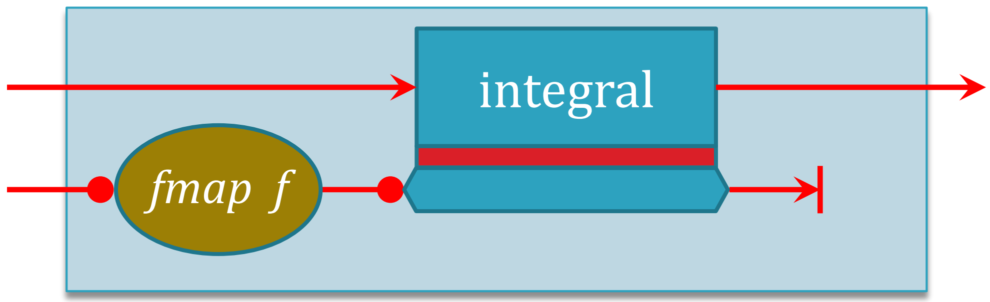
- The top wires are the standard signals.
- The bottom wires are State signals.
  - The input Event State can be used to change *sf*'s internal state.
  - The output State is used to capture the current internal state.

Settable Laws



Example: IntegralReset

- Settability makes our original problem trivial:




```
f _ = reset
```

- We no longer need the overkill of lifting a signal function to the signal level.

Optimization

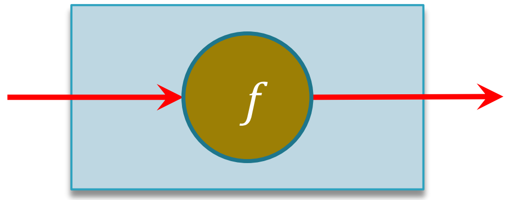
The benefit of static arrows over dynamic arrows

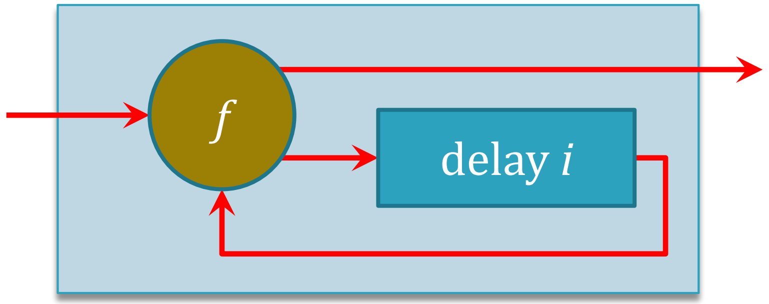
Causal Commutative Arrows

- Liu, Cheng, Hudak [JFP '11] introduced CCA
  - CCAs can be heavily optimized.
  - Performance increases 10-40 times.
  - CCAs do not allow switch but do allow choice.
- CCAs can allow Non-Interfering choice.
  - Arrowized recursion is not supported by default, but it can be added.

How CCA Works

- The CCA optimization reduces arrows to one of two forms:





- We extend this with the ability to handle arrowized recursion and call it CCA*.

Performance Results

	GHC	CCA* + Stream
Chained Adder	1.0	4.06
Chained Integral	1.0	13.27
Dynamic Counters	1.0	10.91

- 3 sample programs using arrowized recursion.
- The 10x performance increase is comparable to Liu et al's results.
  - The Chained Adder is stateless, and thus more optimized by GHC.