Virtualizing Real-World Objects in FRP

Daniel Winograd-Cort

Department of Computer Science

Yale University

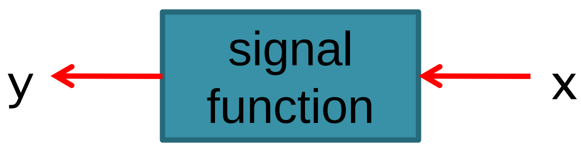
Haskell Implementors' Workshop

September 23, 2011

The Context:
Functional Reactive Programming

- Programming with *continuous values* and *streams of events*.
- Like drawing *signal processing diagrams*:

signal processing diagram — equivalent arrow syntax in Haskell



≡

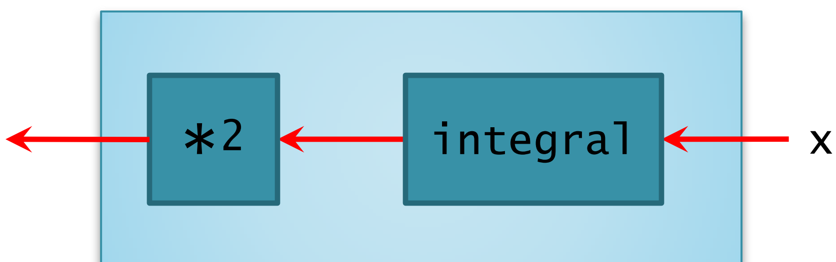
```
y <- sigfun -< x
```

- Previously used in:
  - Yampa: robotics, vision, animation
  - Nettle: networking
  - Euterpea: sound synthesis and audio processing

Understanding arrow syntax

- Let's write a program that integrates a signal and then doubles it:

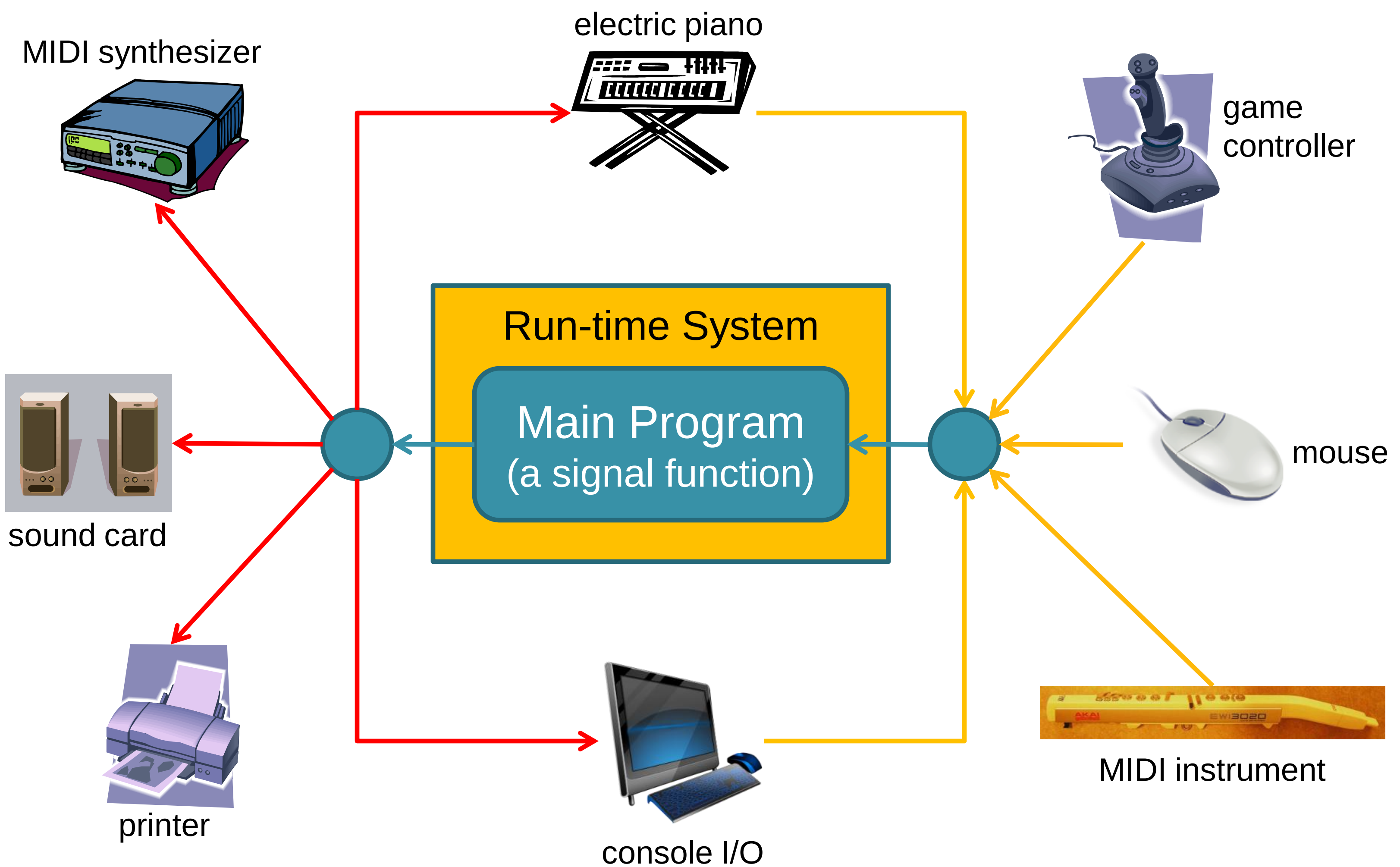
<u>signal processing diagram</u>



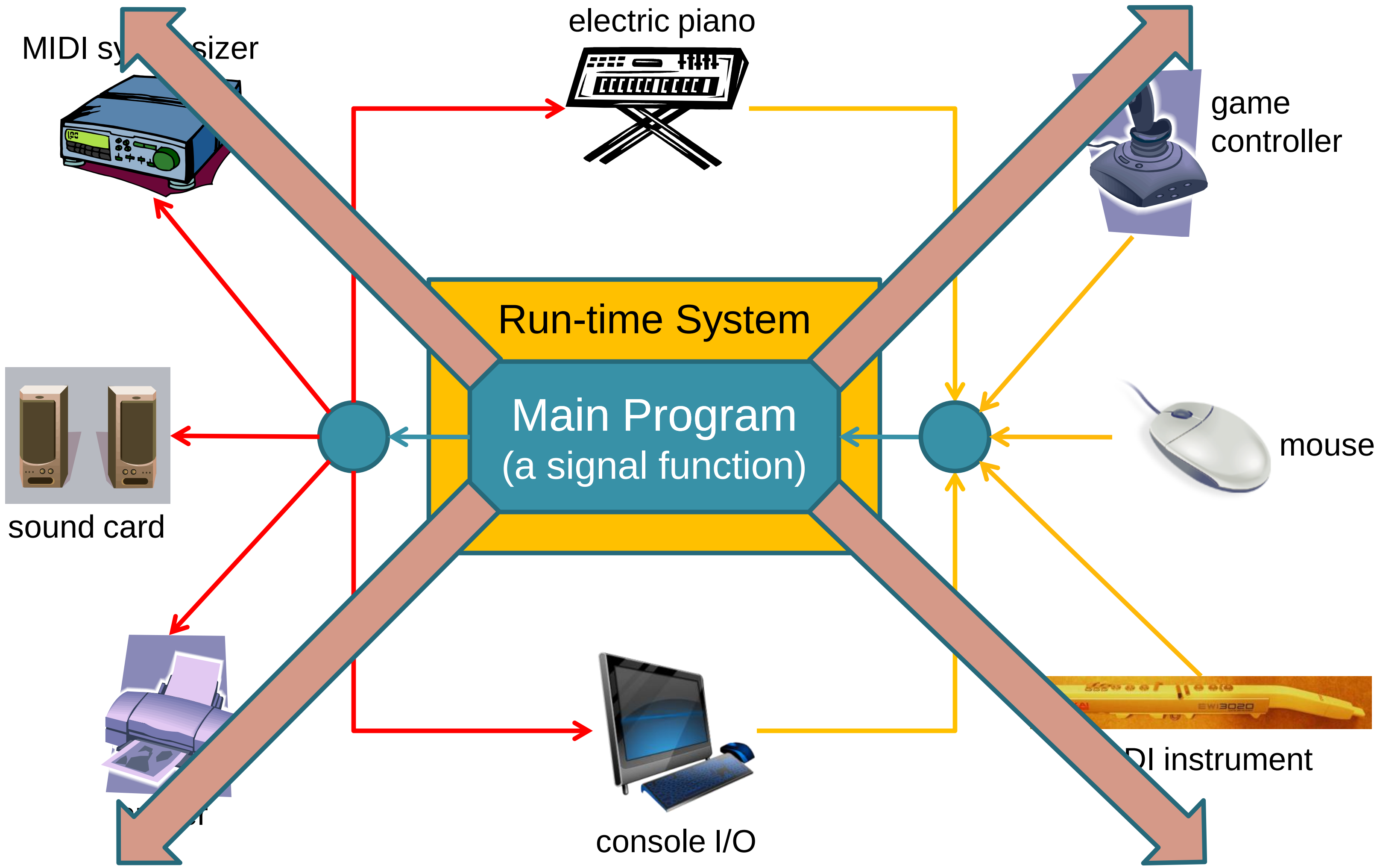
<u>arrow syntax in Haskell</u>

```
sigfun :: SF Double Double
sigfun = proc x -> do
    y <- integral -< x
    returnA -< 2 * y
```

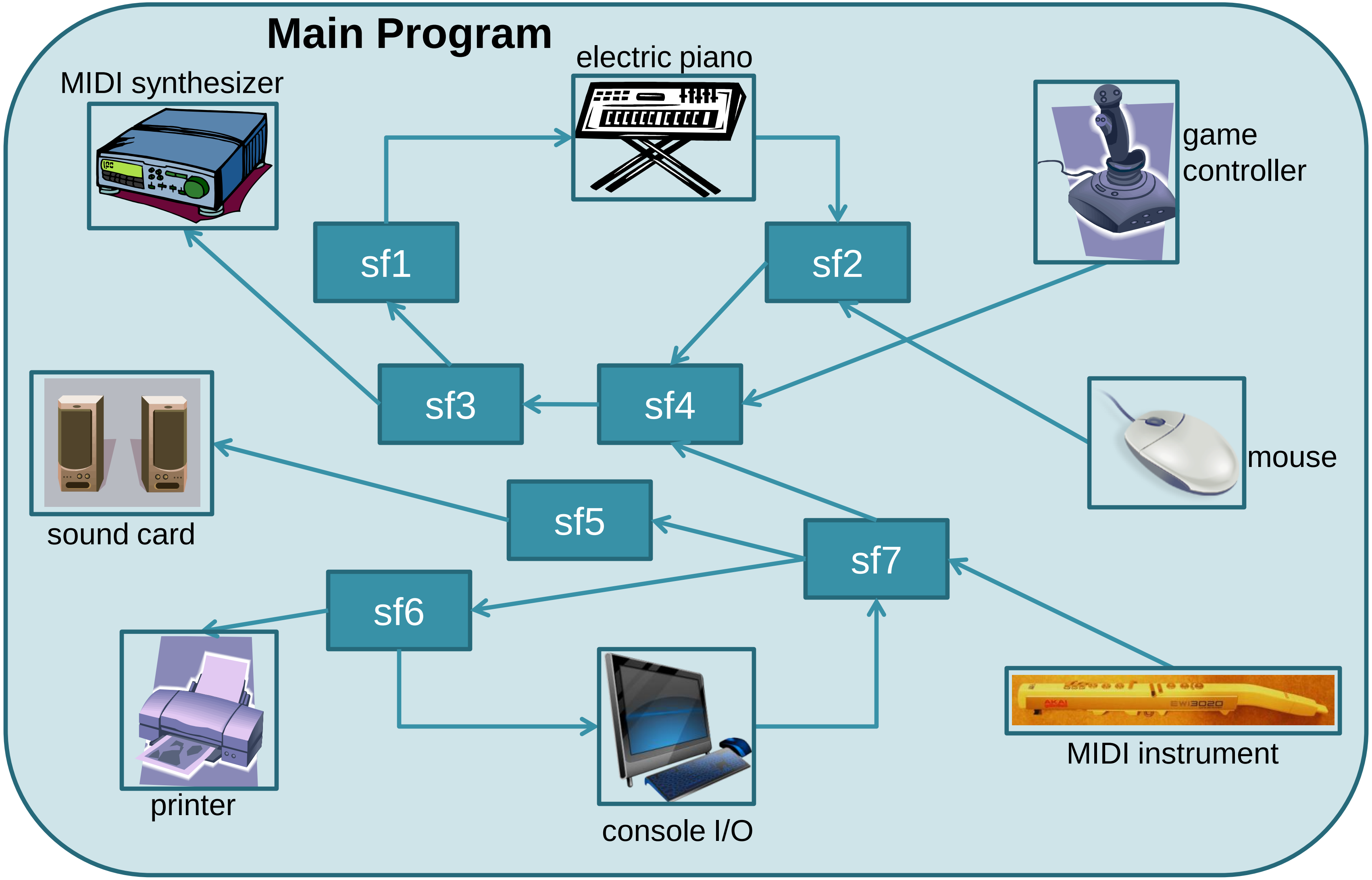
The IO bottleneck of FRP



Add transparency by moving the devices into the signal function



An IO-transparent Signal Function



An IO-transparent Signal Function

- IO devices are now treated just like other signal functions.
- The concept extends further
  - We can virtualize virtual objects (e.g. widgets)
  - We can use “wormhole” signal functions to perform non-local effects.

The Problem of Resource Duplication

- Consider this code fragment:

```
_ <- midiSynth <- noteList1
_ <- midiSynth <- noteList2
```

`midiSynth` is a single output device, but there are two occurrences -- what happens?

Interleaving? Non-determinism?

- Likewise, here is an example of input:

```
rands1 <- randomSF <- ()
rands2 <- randomSF <- ()
```

Do `rands1` and `rands2` return the same result, or are they different?

Duplication resolved with *Resource Types*

- Tag each virtualized object with a unique *resource type* to prevent duplication.

```
midiSynth :: SF (S MidiSynth) (Event Notes) ()
randomSF  :: SF (S RandomRT)  ()            Double
```

- The first argument to SF is a *set* of resource types; `S MidiSynth` and `S RandomRT` are *singleton* sets.
- With these types, the previous code fragments *will not type-check* – resource types of composed signal functions must be *disjoint*.
- Arrows, higher-order types, and type families allow us to implement all this in Haskell.

Implementing Resource Types

- We need:
  - Resource types
  - A way to add resource types
  - Restrictions on composition
- We cannot redefine function application in general, so we use arrows.

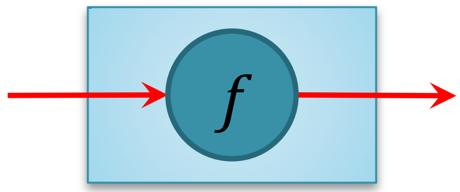
Arrows

- The standard Arrow class:

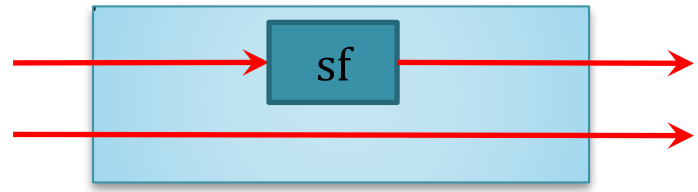
```
class Arrow a where
  arr    :: (b -> c) -> a b c
  first  :: a b c -> a (b,d) (c,d)
  (>>>)  :: a b c -> a c d -> a b d
  loop   :: a (b,d) (c,d) -> a b c
```

- All arrow syntax is translated into these functions.

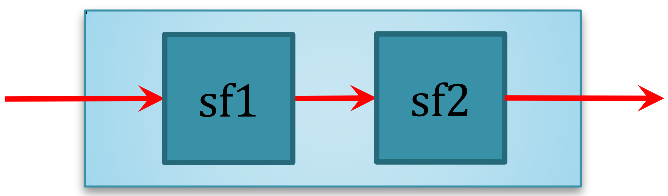
Arrows in use



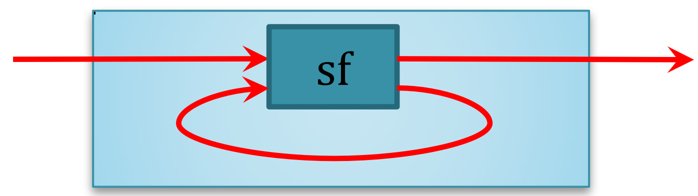
`arr f`



`first sf`



`sf1 >>> sf2`



`loop sf`

Resource Type Inference Rules

$$(arr)\ \frac{\vdash E : \alpha \rightarrow \beta}{\vdash arr\ E : SF\ \emptyset\ \alpha\ \beta}$$

$$(first)\ \frac{\vdash E : SF\ \tau\ \alpha\ \beta}{\vdash first\ E : SF\ \tau\ (\alpha, \gamma)\ (\beta, \gamma)}$$

$$(>>>)\ \frac{\begin{array}{c}\vdash E_1 : SF\ \tau'\ \alpha\ \beta \\ \vdash E_2 : SF\ \tau''\ \alpha\ \beta \\ \emptyset = \tau' \cap \tau'' \\ \tau = \tau' \cup \tau''\end{array}}{\vdash\ E_1 >>> E_2 : SF\ \tau\ \alpha\ \beta}$$

$$(loop)\ \frac{\vdash E : SF\ \tau\ (\alpha, \gamma)\ (\beta, \gamma)}{\vdash loop\ E : SF\ \tau\ \alpha\ \beta}$$

Arrows with resource types

- We add a type parameter to Arrow:

```
class Arrow a where
  arr    :: (b -> c) -> a Empty b c
  first  :: a r b c -> a r (b,d) (c,d)
  (>>>)  :: (Disjoint r1 r2, Union r1 r2 r3) =>
              a r1 b c -> a r2 c d -> a r3 b d
  loop   :: a r (b,d) (c,d) -> a r b c
```

Arrows with resource types

- We add a type parameter to Arrow:

```
class Arrow a where
  arr    :: (b -> c) -> a Empty b c
  first  :: a r b c -> a r (b,d) (c,d)
  (>>>)  :: (Disjoint r1 r2, Union r1 r2 r3) =>
                a r1 b c -> a r2 c d -> a r3 b d
  loop   :: a r (b,d) (c,d) -> a r b c
```

- The `Disjoint` class assures that `r1` and `r2` are disjoint.

Sets at the Type Level

- We represent type sets as either Empty, Singleton sets, or Unions:

```
data Empty
data S a
data a `U` b
```

- Unioning sets is easy, but testing disjointness is not.

Sets at the Type Level

- Set disjointness:

```
class Disjoint xs ys

instance Disjoint Empty ys
instance (ElemOf x ys HFalse) =>
    Disjoint (S x) ys
instance (Disjoint xs zs, Disjoint ys zs) =>
    Disjoint (xs `U` ys) zs
```

Sets at the Type Level

- Set disjointness:

```
class Disjoint xs ys

instance Disjoint Empty ys
instance (ElemOf x ys HFalse) =>
    Disjoint (S x) ys
instance (Disjoint xs zs, Disjoint ys zs) =>
    Disjoint (xs `U` ys) zs
```

- … which requires set membership:

```
class ElemOf x ys b | x ys -> b

instance ElemOf x Empty HFalse
instance (TypeEq x y b) =>
    ElemOf x (S y) b
instance (ElemOf x ys b1, ElemOf x zs b2, OR b1 b2 b) =>
    ElemOf x (ys `U` zs) b
```

Sets at the Type Level

- … which requires set membership:

```
class ElemOf x ys b | x ys -> b

instance ElemOf x Empty HFalse
instance (TypeEq x y b) =>
    ElemOf x (S y) b
instance (ElemOf x ys b1, ElemOf x zs b2, OR b1 b2 b) =>
    ElemOf x (ys `U` zs) b
```

- … which requires type equality:

```
class TypeEq x y b | x y -> b

instance (HTrue ~ b)  => TypeEq x x b
instance (HFalse ~ b) => TypeEq x y b
```

Arrows into Signal Functions

- We instantiate arrows with the following signal function definition

```
data SF r a b = SF
  { sfFun :: a -> IO (b, SF r a b) }

instance Arrow SF where
  arr g = SF h
    where h x = return (f x, SF h)

  first (SF f) = SF (h f)
    where h f (x, z) = do (y, SF f') <- f x
                          return ((y, z), SF (h f'))

  SF f >>> SF g = SF (h f g)
    where h f g x = do (y, SF f') <- f x
                       (z, SF g') <- g y
                       return (z, SF (h f' g'))
```

From I/O to Resource Types

- How do we make these SFs?
  - Continuous SFs

```
source   :: IO c ->            SF (S r) () c
sink     :: (b -> IO ()) -> SF (S r) b  ()
pipe     :: (b -> IO c) ->  SF (S r) b  c
```

  - Event-based SFs

```
sourceE :: IO c ->            SF (S r) () (Event c)
sinkE   :: (b -> IO ()) -> SF (S r) (Event b)  ()
pipeE   :: (b -> IO c) ->  SF (S r) (Event b) (Event c)
```

From I/O to Resource Types

- These functions can be easily defined:
  - ```
    source f = SF h where
          h _  = f   >>= return . (\x -> (x, SF h))
    ```
  - ```
    sink   f = SF h where
          h x  = f x >>  return ((), SF h)
    ```
  - ```
    pipe   f =  SF h where
          h x  = f x >>= return . (\x -> (x, SF h))
    ```
- The event-based ones are more subtle due to blocking and are outside the scope of this talk.
    ```

From I/O to Resource Types

- With Haskell `IO`, we might have:

```
mSynth  :: Notes -> IO ()
```

- Using resource typed SFs, we have:

```
data MIDISynth
midiSynth  :: SF (S MidiSynth) (Event Notes) ()
midiSynth  =  sinkE mSynth
```

- Now our example from before won't even type check:

```
_ <- midiSynth <- noteList1
_ <- midiSynth <- noteList2
```

Making a GUI with Resource Types

- For virtual objects, we use a modified version of Euterpea's UI.
- We first make some widgets

```
ampSlider  :: UISF (S ASlider) ()     Double
freqSlider :: UISF (S FSlider) ()     Double
graph      :: UISF (S Graph)   Double ()

ampSlider  = title "Amplitude“ $ hSlider (0, 1)     0.5
freqSlider = title "Frequency“ $ hSlider (20, 2000) 400
graph      = realtimeGraph (400,300) 400 20 Black
```

(`UISF` is a special signal function to handle UI.)

Making a GUI with Resource Types

- It's trivial to bind the widgets together:

```
type sinWavRTs = S FSlider `U` S ASlider `U` S Graph

sinGraph :: UISF sinWavRTs () ()
sinGraph = proc _ -> do
    f  <-  freqSlider  -< ()
    a  <-  ampSlider   -< ()
    s  <-  freqToSin   -< f
    graph -< s * a

freqToSin :: SF Empty Double Double
```

- Here is this program in action

Adding Debugging data

- Perhaps we want to show debug data generated by `freqToSin`.
- We can update it to have type:

```
freqToSin :: SF Empty Double (Double, Double)
```

- But now all functions depending on `freqToSin` will have type errors!

Wormholes

- We can use a wormhole to fix this.

```
data Wormhole r1 r2 a =
    Wormhole { whitehole :: SF (S r1) () a,
               blackhole :: SF (S r2) a  () }
makeWormhole :: a -> Wormhole r1 r2 a
```

  - Wormholes are basically just mutable variables (i.e. memory locations).

```
makeWormhole init = unsafePerformIO $ do
    r <- newIORef init
    return $ Wormhole (source $ readIORef  r)
                      (sink   $ writeIORef r)
```

Wormholes

- We can use a wormhole to fix this.

```
data Wormhole r1 r2 a =
    Wormhole { whitehole :: SF (S r1) () a,
               blackhole :: SF (S r2) a  () }
makeWormhole :: a -> Wormhole r1 r2 a
```

  - Wormholes are basically just mutable variables (i.e. memory locations).
  - With resource types, we can guarantee that they are only ever written to in one place and only ever read from in one place.
  - This assures safety.

Wormholes

- Wormholes are tagged with one resource type for reading and one for writing

```
data DebugW
data DebugB
wormhole :: WormHole DebugW DebugB Double
wormhole = makeWormhole 0
```

- Now, `freqToSin` writes to the wormhole, and only its resources:

```
freqToSin :: SF (S DebugB) Double Double
```

Wormholes

- We don't even need to change `sinGraph`. We simply read from the wormhole for the stored debug info:

```
data DebugGraph
debugGraph :: UISF (S DebugGraph) Double ()
debugGraph = realtimeGraph (400,300) 400 20 Red

sinGraphWithDebug
  :: UISF (sinWavRTs `U` S DebugB  `U`
           S DebugW  `U` S DebugGraph) () ()
sinGraphWithDebug = proc _ -> do
    _ <- sinGraph -< ()
    d <- toUISF (whiteHole wormhole) -< ()
    _ <- title "Debug" debugGraph -< d
    returnA -< ()
```

- Another Demo

Future work

- Running signal functions in parallel
  - SF work can be easily pushed to threads
  - Perhaps we can use something like wormholes to create safe communication between threads
- Rebindable Syntax for Arrows
  - Currently, arrow syntax in GHC doesn’t accept resource types properly
- Local Resource Types
  - Existential types for wormholes
  - Type level counters for arbitrarily many virtual resources

Conclusions

- Resource types clearly show what resources are being used.
- They safely permit seemingly dangerous non-local effects.
- They are straightforward and effective.

Questions

Extra Slides

Event-Based Signal Functions

- Transforming a continuous signal function to an event based one is easy.

```
liftToEvent :: SF r a b -> SF r (Event a) (Event b)
liftToEvent sf = proc a -> do
    case a of
        Event a' -> sf >>> arr Event -< a'
        NoEvent  -> returnA -< NoEvent
```

- But this doesn’t help if the signal function blocks on input.

Running SFs in Parallel

- We need to run the blocking action in parallel in a separate thread
- We use `toSFE` to do that:

```
toSFE :: SF r a b -> SF r (Event a) (Event b)
```

  - `toSFE` cleverly uses `Chans` to make sure that data is available as soon as it's ready.
  - `toSFE` has an interesting sister function:

```
fromSFE :: SF r (Event a) (Event b) -> SF r a b
```

  - `par = fromSFE . toSFE :: SF r a b -> SF r a b`

UISF

- We based UISF on the Euterpea UI.
- How do we make UISF without redoing all our Euterpea UI work?

UISF

- There is no reason to pin SF to the IO monad. In practice, it has a monadic argument:

```
data SFM m r a b = SFM
  { sfmFun :: a -> m (b, SFM m r a b) }
newtype SF = SFM IO
```

- So, all we need is a UI monad that fits nicely into SFM.

UISF

- Euterpea's UI monad:

```
newtype UI a = UI
  { unUI :: CX -> Signal (Input, Sys) ->
              (Signal (Action, Sys, a), Layout) }
newtype Signal a = Signal { unS :: [a] }
```

- This encapsulates a primitive signal function with itself.
- It also has a static rendering context.

UISF

- Ideally, we want something like:

```
newtype UI a = UI
  { unUI :: (Input, Sys) -> (Action, Sys, a) }
```

- This is the signal portion, but we also need the context portion:

```
newtype UICTX a = UICTX
  { unUICTX :: CTX -> (Layout, a) }
```

- Together, we achieve:

```
newtype UISF r a b =
  UISF (UICTX (SFM UI r a b))
```