Virtualizing Real-World Objects in FRP

Daniel Winograd-Cort
Yale University
dwc@cs.yale.edu

Hai Liu
Intel, Inc.
hai.liu@intel.com

Paul Hudak
Yale University
paul.hudak@yale.edu

July 25, 2011

Yale University
Department of Computer Science
New Haven, CT 06520

Research Report YALEU/DCS/TR-1446

Abstract

We begin with a *functional reactive programming* (FRP) model in which every program is viewed as a *signal function* that converts a stream of input values into a stream of output values. We observe that objects in the real world – such as a keyboard or sound card – can be thought of as signal functions as well. This leads us to a radically different approach to I/O – instead of treating real-world objects as being external to the program, we expand the sphere of influence of program execution to include them within the program. We call this *virtualizing real-world objects*. We explore how even virtual objects, such as GUI widgets, and non-local effects, such as are needed for debugging (using something that we call a "wormhole") and random number generation, can be handled in the same way.

Our methodology may at first seem naïve – one may ask how we prevent a virtualized device from being copied, thus potentially introducing non-determinism as one part of a program competes for the same resource as another. To solve this problem, we introduce the notion of a *resource type* that assures that a virtualized object is not duplicated and that I/O and non-local effects are safe. Resource types also provide a deeper level of transparency: by inspecting the type, one can clearly see exactly what resources are being used. We use arrows, type classes, associated types, and type families to implement our ideas in Haskell, and the result is a safe, effective, and transparent approach to stream-based I/O.

1 Introduction

Every programming language has some way of communicating with the outside world. Usually we refer to such mechanisms as *input/output*, or I/O. In most imperative languages the mechanisms have effects almost entirely outside the program, serving a purpose typically unrelated to the internal computation of an answer to the program. In Haskell, programs engage in I/O by using the *IO monad* [33, 32]. An advantage of Haskell is that we can determine from the type of a function whether or not it is engaged in I/O – if any one part of a program is, then the type of the whole program reflects this. The monadic framework assures us that the overall program is well defined, and in particular, that the I/O operations are executed in a deterministic, sequential manner. However, even in Haskell, the *IO* monad is "special" compared to other monads. I/O commands often represent an awkward disconnect between the internal execution of a program and the objects, devices, and protocols of the real world.

In this paper, we take a different approach. Instead of using an imperative or even monadic basis for overall program execution, we use *arrows* [23]. Specifically, we assume that a program is a "signal function" having the (over-simplified for now) type *SF inp out*, where *inp* is the type of the input to the program and *out* is the output type, both of which are assumed to be *streams* of values (much like data flow). Just as *IO* is a monad, *SF* is an arrow, and like a monad, the arrow framework composes program components in a way that assures us that the streams are well-defined, and that I/O is done in a deterministic, sequential manner.

This approach is the basis for arrow-based versions of *functional reactive programming* (FRP), such as *Yampa* [22, 7] (which has been used for animation, robotics, GUI design, and more), *Nettle* [37] (for networking), and *Euterpea* [21] (for audio processing and sound synthesis). In fact, our work was motivated by Euterpea, and we will use examples from that domain like MIDI devices[1], synthesizers, and keyboards.

Our research is based on three key insights. First, we observe that *objects and devices in the real world can also be viewed as signal functions.* For example, a MIDI keyboard would take note events as input as well as generate note events as output. Similarly, a speaker would take sound data as input, and a microphone would produce sound data as output. So it would only seem natural to simply include these signal functions as part of the program – i.e. to program with them directly and independently rather than merge everything together as one input and one output for the whole program. In this sense, the real-world objects are being *virtualized* for use in the program.

A major problem with this is that one could easily duplicate one of these virtualized objects – after all, they are just values – which would cause the semantics of the program to become unclear. For example, how does a single resource handle multiple event streams linked to each of its virtual duplicates when it only expects one event stream itself? Thus, our second key insight is to realize that *uniqueness of signal functions can be realized at the type level.* In particular, we introduce the notion of a *resource type* to ensure that there is exactly one signal function that represents each real-world device. Because we are using arrows, we can implement this at the type level by re-typing each of the arrow combinators together with introducing type classes that capture the disjoint union of resource types.

Our final insight is to note that *many unsafe functions can be treated as unique signal functions* as well. Examples include GUI widgets, random number generators, and "wormholes"

[1]MIDI = Musical Instrument Digital Interface, a standard protocol for communication between most electronic instruments and computers.

(mutable variables that are written to at one point in a program and safely read from at another).

The advantages of our approach include:

1. *Virtualization.* I/O devices can be treated conveniently and independently as signal functions that are just like any other signal function in a program. I/O is no longer a special case in the language design.

2. *Transparency.* From the type of a signal function, we can determine immediately *all* of the resources that it uses. In particular, this means that we know all the resources that an entire program uses (with monads, all we know is that some kind of I/O is being performed).

3. *Safety.* If used properly, a signal function engaged in I/O or non-local effects is *safe* – despite the side effects, equational reasoning is not compromised.

4. *Modularity.* Certain non-local effects – the lack of which is often cited as a lack of moduarity in functional languages – can be handled safely.

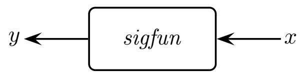
5. *Extensibility.* A user can define his or own signal function that captures a new I/O device or some kind of non-local effect.

In the remainder of this paper we first introduce arrow syntax and the basis of our language design. In Section 3, we concretely present our main ideas and the problem of duplicate resources. We next work through a number of examples using our system in Section 4 before delving into the implementation details in Section 5. Finally, we take a brief look at the limitations of our design (Section 6), the future work we have planned (Section 7), and related work in the field (Section 8).

2 A Signal-Processing Language

The simplest way to understand our language is to think of it as a language for expressing *signal processing diagrams*. We refer to the lines in such a diagram as *signals*, and the boxes (that convert one signal into another) as *signal functions*. Conceptually, signals are continuous, time-varying quantities, but, as mentioned above, they can also be streams of events.

For example, this very simple diagram has two signals, an input x and an output y, and one signal function, *sigfun*:



This is written as a code fragment in our framework as:

$$y \leftarrow \mathit{sigfun} \prec x$$

which uses Haskell's *arrow syntax* [31]. Indeed, the above program fragment cannot appear alone, but rather must be part of a **proc** construct, much like a **do** construct for monads. The expression on the left must always be a variable, whereas the expression on the right can be any well-typed expression that matches the input type of the signal function. Signal functions such as *sigfun* have a type of the form *SF* T_1 T_2, for some types T_1 and T_2; subsequently, x must have type T_1 and y must have type T_2. Although signal functions act on signals, the arrow notation allows one to manipulate the instantaneous values of the signals. For example, here is a definition for *sigfun* that integrates a signal that is one greater than its input:

```
sigfun :: SF Double Double
sigfun = proc x → do
  y ← integral ⤙ x + 1
  returnA ⤙ y
```

The first line is a type signature that declares *sigfun* to be a signal function that converts time-varying values of type *Double* into time-varying values of type *Double*. The notation

proc $x \rightarrow$ **do** ...

introduces a signal function, binding the name x to the instantaneous values of the input. The third line adds one to each instantaneous value, and sends the resulting signal to the integrator whose output is named y. Finally, we specify the output of the signal function by feeding y into *returnA*, a special signal function that returns the final result.

2.1 Streams of Events

With respect to I/O, continuous signals can be useful in several contexts, such as the voltage to a robot motor (as output from a program) or the position of a mouse (as input to a program). However, there are many applications where instead we are interested in programming with *streams of events*. We represent event streams in our language as continuous signals that only contain data at discrete points in time. A signal that periodically carries information of some type T has type *Event* T, whose values are either *NoEvent* or *Event* x, where $x :: T$ (the name *Event* is overloaded). For example, a signal function that converts a stream carrying messages of type M_1 into a stream carrying messages of type M_2 has type *SF* (*Event* M_1) (*Event* M_2). *Event* is an instance of *Functor* allowing us to use *fmap* on events.

3 Introduction to Resource Types

As mentioned in the introduction, we wish to treat I/O devices as signal functions. Consider, for example, a MIDI sound synthesizer with type:

```
midiSynth :: SF (Event Note) ()
```

midiSynth takes a stream of *Note* events as input and synthesizes the appropriate sound of each note. Now consider this code fragment:

```
_ ← midiSynth ⤙ notes₁
_ ← midiSynth ⤙ notes₂
```

We intended for *midiSynth* to represent a single output device, but here we have two occurrences; so what is the effect? Are the event streams $notes_1$ and $notes_2$ somehow interleaved or non-deterministically joined together?

Likewise, here is an example of a similar problem with input. Suppose *randomSF* is intended to be a random number generator initialized with a random seed from the OS (random numbers and probability distributions are commonly used in audio processing and sound synthesis). Its type is:

$$randomSF :: SF\ ()\ Double$$

Now consider this code fragment:

$$\begin{aligned} &rands_1 \leftarrow randomSF \prec ()\\ &rands_2 \leftarrow randomSF \prec () \end{aligned}$$

What is the relationship between $rands_1$ and $rands_2$? Do they return the same result, or are they different? If they are the same, what if we want them to be different?

3.1 Resource Types

The solution to these problems is to somehow prevent duplication of certain signal functions such as those above. To do this, we introduce the notion of a *resource type*. There may be many resource types in a program, and, as we shall see, the user can easily define new ones. For example, in the above cases, we introduce the resource types *MidiSynthRT* and *RandomRT*.

To keep track of *sets* of resource types, we introduce two type-level constructors, S and $\cup$, as well as the *Empty* type to refer to the empty set. The type S *MidiSynthRT* is the singleton set containing *MidiSynthRT*, and the binary operator $\cup$ constructs the union of two sets of resource types (for example $S\ MidiSynthRT \cup S\ RandomRT$).

Finally, we add an extra type parameter representing a collection of resource types to the type signature of each signal function. The type $SF\ r\ a\ b$ is a signal function that accesses the resource represented by resource type r, while converting a signal of type a into a signal of type b. For the two examples above, we would have:

$$\begin{aligned} &midiSynth :: SF\ (S\ MidiSynthRT)\ (Event\ Note)\ ()\\ &randomSF :: SF\ (S\ RandomRT)\quad ()\ Double \end{aligned}$$

The key technical point is that, because we are using arrows, we can re-type each of the combinators in the *Arrow* type class in such a way that the above problematical code fragments *will not type check*. The details of how this is done are described in Section 5.1, but for now the key intuition is to note that whenever two signal functions, say $sf_1 :: SF\ r_1\ a\ b$ and $sf_2 :: SF\ r_2\ b\ c$ are composed, we require that r_1 and r_2 be *disjoint* – otherwise, they may compete for the same resource. Essentially, we use type classes and type families to assure this at the type level.

3.2 From I/O to Resource-Typed Signal Functions

To facilitate working with resource types, we provide three functions to convert I/O actions into signal functions tagged with the appropriate resource type:

$$\begin{aligned} &source :: IO\ c &&\rightarrow SF\ (S\ r)\ ()\ c\\ &sink\ \ :: (b \rightarrow IO\ ()) &&\rightarrow SF\ (S\ r)\ b\ ()\\ &pipe\ \ :: (b \rightarrow IO\ c) &&\rightarrow SF\ (S\ r)\ b\ c \end{aligned}$$

In each case, the resultant signal function is required to have a singleton resource type because these functions are expected to be applied to a monadic I/O action that captures a single I/O device, and thus consumes a single resource.

For event-based signal functions (as described in Section 2.1), we provide three more functions analagous to the above:

$$\begin{array}{lll}
\mathit{sourceE} & :: \mathit{IO}\ c & \rightarrow \mathit{SF}\ (S\ r)\ ()\ \qquad\qquad (\mathit{Event}\ c) \\
\mathit{sinkE} & :: (b \rightarrow \mathit{IO}\ ()) & \rightarrow \mathit{SF}\ (S\ r)\ (\mathit{Event}\ b)\ () \\
\mathit{pipeE} & :: (b \rightarrow \mathit{IO}\ c) & \rightarrow \mathit{SF}\ (S\ r)\ (\mathit{Event}\ b)\ (\mathit{Event}\ c)
\end{array}$$

3.3 Examples

To see these functions in action, let's revisit the *midiSynth* and *randomSF* signal functions from earlier. Suppose that

$$\mathit{midiSynthM} :: \mathit{Note} \rightarrow \mathit{IO}\ ()$$

is the monadic action that sends *Note*s to the synthesizer. Then we can define *midiSynth* as follows:

$$\begin{array}{l}
\mathbf{data}\ \mathit{MidiSynthRT} \\
\mathit{midiSynth} :: \mathit{SF}\ (S\ \mathit{MidiSynthRT})\ (\mathit{Event}\ \mathit{Note})\ () \\
\mathit{midiSynth} = \mathit{sinkE}\ \mathit{midiSynthM}
\end{array}$$

Note that *MidiSynthRT* is an empty data type – all we need is the type name – and that *midiSynth* is a signal function.

randomSF does not access an I/O device, but it is a source of non-local effects from the OS. We can define it from scratch using the *randomIO* function (of type *IO Double*) from Haskell's *Random* library:

$$\begin{array}{l}
\mathbf{data}\ \mathit{RandomRT} \\
\mathit{randomSF} :: \mathit{SF}\ (S\ \mathit{RandomRT})\ ()\ \mathit{Double} \\
\mathit{randomSF} = \mathit{source}\ \mathit{randomIO}
\end{array}$$

Note that *randomSF* is a continuous signal function, and its range, inherited from *randomIO*, is the semi-closed interval $[0, 1)$.

Suppose now that we want *two independent* random number generators. We can construct this simply by defining two different resource types:

$$\begin{array}{l}
\mathbf{data}\ \mathit{RandomRT}_1 \\
\mathbf{data}\ \mathit{RandomRT}_2 \\
\mathit{randomSF}_1 :: \mathit{SF}\ (S\ \mathit{RandomRT}_1)\ ()\ \mathit{Double} \\
\mathit{randomSF}_2 :: \mathit{SF}\ (S\ \mathit{RandomRT}_2)\ ()\ \mathit{Double} \\
\mathit{randomSF}_1 = \mathit{source}\ \mathit{randomIO} \\
\mathit{randomSF}_2 = \mathit{source}\ \mathit{randomIO}
\end{array}$$

Now a slight variation of the problematical example given in Section 3 will properly type check and yield two independent sources of random numbers:

$$\begin{aligned}&\mathit{rands}_1 \leftarrow \mathit{randomSF}_1 \prec\!\!-\, ()\\&\mathit{rands}_2 \leftarrow \mathit{randomSF}_2 \prec\!\!-\, ()\end{aligned}$$

Finally, suppose that we wish to vary the *range* of the random number generator *dynamically* as the program is executing. In other words, we would like a signal function with type:

$$\begin{aligned}&\textbf{data}\ \mathit{RandomRRT}\\&\mathit{randomRSF} :: \mathit{SF}\ (S\ \mathit{RandomRRT})\\&\qquad\qquad\qquad(\mathit{Double}, \mathit{Double})\ \mathit{Double}\end{aligned}$$

where the input pair $(\mathit{Double}, \mathit{Double})$ represents the desired range of the output. To define *randomRSF* we can use the *pipe* function, along with the *randomRIO* function (of type $(\mathit{Double}, \mathit{Double}) \rightarrow \mathit{IO}\ \mathit{Double}$) from the *Random* library:

$$\mathit{randomRSF} = \mathit{pipe}\ \mathit{randomRIO}$$

4 More Examples

In this section we work through a number of examples, each building upon the last, to demonstrate the power and potential of our approach. Most examples are taken from audio processing and sound synthesis, but we have also written examples for robotics, console I/O, GUIs, and networking.

4.1 Composition

A common practice in wiring together MIDI devices is to “daisy chain” them together using MIDI cables – a cable from the computer to the first device, the first device to the second, and so on. In this way, the MIDI events from each are merged together into one large set. By virtualizing these devices, our code reflects precisely the cable wiring.

For example, here is a signal function that daisy chains two MIDI keyboards together and then transposes all of the notes by a given number of steps:

$$\begin{aligned}&\mathit{daisy} :: \mathit{Integer} \rightarrow \mathit{SF}\ (S\ \mathit{MidiKBRT}_1 \cup S\ \mathit{MidiKBRT}_2)\\&\qquad\qquad\qquad\qquad(\mathit{Event}\ [\mathit{Note}])\ (\mathit{Event}\ [\mathit{Note}])\\&\mathit{daisy}\ n = \textbf{proc}\ \mathit{notesIn} \rightarrow \textbf{do}\\&\quad \mathit{notes}_2 \leftarrow \mathit{midiKB}_1 \prec\!\!-\, \mathit{notesIn}\\&\quad \mathit{notes}_3 \leftarrow \mathit{midiKB}_2 \prec\!\!-\, \mathit{notes}_2\\&\quad \mathit{returnA} \prec\!\!-\, \mathit{fmap}\ (\mathit{map}\ \$\ \mathit{transpose}\ n)\ \mathit{notes}_3\end{aligned}$$

The disjoint resource types ensure that the two keyboards are kept distinct, just like in the real world.

A simple variation of this idea brings into play the invariance of resource types in a recursive behavior – i.e. when using the arrow loop combinator (see Section 5.1 for more details). Specifically, we can define an “echo” effect by looping note events back onto themselves, attenuating them by some percentage on each loop:

$$\begin{aligned}&\mathit{echo} :: \mathit{Double} \rightarrow \mathit{Double} \rightarrow\\&\quad \mathit{SF}\ (S\ \mathit{MidiKBRT})\ (\mathit{Event}\ [\mathit{Note}])\ (\mathit{Event}\ [\mathit{Note}])\end{aligned}$$

```
echo rate freq = proc notesIn → do
  rec notesOut ← midiKB −≺ merge notesIn notes
      notes    ← delayt −≺ (1.0/freq,
                            decay rate notesOut)
  returnA −≺ notesOut
```

Here, *echo* takes a decay rate and frequency as static arguments and produces a signal function that adds an echo to the input notes. It uses three helper functions: *merge* takes two [*Note*] events and consolidates them into one; *decay* takes a rate and a [*Note*] event and attenuates the notes; and *delayt* is a signal function that takes a variable amount of time to delay as well as the value to delay (here, the decayed *notesOut*).

4.2 Unions

As discussed earlier, normal signal function composition requires that the resource types of the arguments be *disjoint*. However, for conditionals (i.e. case statements), the proper semantics is to take the *natural union* of the resource types. In this section, we give two examples to justify this.

First, consider the following two functions for sending sound data to stereo speakers:

```
leftSpeaker  :: SF (S LeftRT)  Sound ()
rightSpeaker :: SF (S RightRT) Sound ()
```

We can use these to define a signal function for routing sound to the proper speaker (often called a demultiplexer):

```
data Speaker = Left | Right
routeSound :: SF (S LeftRT ∪ S RightRT)
                 (Speaker, Sound) ()
routeSound = proc (speaker, sound) → do
  case speaker of
    Left  → leftSpeaker  −≺ sound
    Right → rightSpeaker −≺ sound
```

routeSound only makes use of one speaker at a time, but it feels natural that it should acquire both the *LeftRT* and *RightRT* resource types, because we cannot know at compile time which speaker will be used. Thus, conditional (or case) statements acquire the union of the resource types of their branches.

Similarly, recall from Section 3.2 the *midiSynth* function:

```
midiSynth :: SF (S MidiSynthRT) (Event Note) ()
```

And the following variation:

```
midiSynthChord :: SF (S MidiSynthRT)
                     (Event [Note]) ()
```

Based on the resource types, we can see that in this case, both functions use the *same* MIDI resource. However, they have different behaviors: one plays a single note while the other plays a set of notes, or chord. We define a function that uses both:

```
data SoundChoice = Tonic | MajorChord | MinorChord
playNote :: SF  (S MidiSynthRT)
                (SoundChoice, Note) ()
playNote = proc (sc, n) → do
  case sc of
    Tonic      → midiSynth      ─≺ n
    MajorChord → midiSynthChord ─≺ makeMajor n
    MinorChord → midiSynthChord ─≺ makeMinor n
```

makeMajor and *makeMinor* both have type *Note* → [*Note*]. There is no reason why *playNote* should not type check – although it uses the same resource in multiple places, that resource will never be used more than once *simultaneously*.

4.3 Virtual Objects

So far we have focused on physical devices as resources, but in fact, virtual components can be thought of in exactly the same way. In this section we show how to define a simple GUI that allows the user to pick the decay rate and frequency for the echo signal function from Section 4.1 using "sliders" and see it graphed in real time.

To write this program, we use a different type of signal function than used previously. The type *UISF r a b* is designed especially for GUIs, and we can lift ordinary *SF*s to *UISF*s by using the function *toUISF*. In addition, we use two built-in GUI functions: (1) Given a range and initial value, *hslider* creates a horizontal slider; (2) Given some step parameters, a size, and a color, *realTimeGraph* creates a graph that varies in real-time as its input changes.

We begin by defining three signal functions for the three widgets we will use:

```
data DSlider
data FSlider
data Graph
decSlider  :: UISF (S DSlider) () Double
freqSlider :: UISF (S FSlider) () Double
graph      :: UISF (S Graph)   Double ()
decSlider  = title "Decay Rate" $ hSlider (0, 0.9) 0.5
freqSlider = title "Frequency"  $ hSlider (1, 10)  10
graph      = realtimeGraph (400, 300) 400 20 Black
```

Before we can write our GUI, we also need to change our original definition of *echo* so that it can accept time varying values for decay rate and frequency. This is actually as easy as changing its type and declaration to:

```
echo :: SF (S MidiKBRT)
           (Double, Double, Event [Note]) (Event [Note])
echo = proc (rate, freq, notesIn) → do
  ...
```

Finally, we use the signal function:

```
renderNotes :: SF Empty (Event [Note]) Double
```

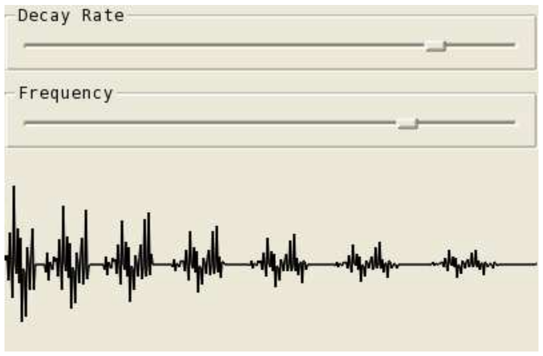


Figure 1: A screenshot of the *echoGUI* signal function just after a note has been played on the MIDI keyboard.

to transform our *Note*s into sound data.

With these functions we can now define our main application, which simply wires the components together:

$$
\begin{array}{l}
echoGUI :: UISF\ (S\ MidiKBRT \cup S\ DSlider \cup \\
\qquad\qquad\qquad S\ FSlider \cup S\ Graph) \\
\qquad\qquad\qquad () \ () \\
echoGUI = \mathbf{proc}\ \_ \rightarrow \mathbf{do} \\
\quad \begin{array}{lll}
notesIn & \leftarrow midiKB & \prec Nothing \\
rate & \leftarrow decSlider & \prec () \\
freq & \leftarrow freqSlider & \prec () \\
notesOut & \leftarrow toUISF\ echo & \prec (notesIn, rate, freq) \\
sound & \leftarrow toUISF\ renderNotes & \prec notesOut \\
\_ & \leftarrow graph & \prec sound
\end{array} \\
\quad returnA \prec ()
\end{array}
$$

Note that the type of *echoGUI* lists all of the resources that it uses: both the physical MIDI keyboard as well as the virtual sliders and graph. If one were to use this module in another GUI, it would be clear from the type what the major components would be.

Figure 1 is a screenshot of the program in action.

4.4 Wormholes and Non-local Effects

Resource types allow us to safely perform I/O actions within signal functions, and although they were designed with physical resources in mind, the idea extends to other kinds of effectful computation as well. For example, mutation and direct memory access, techniques that are typically plagued by difficult-to-find bugs, can be made safe. To demonstrate this, we start by considering an arbitrarily nested signal function:

$$
\begin{array}{lll}
mySF & :: & SF\ ManyResources\ a\ \ b \\
mySF & = & \mathbf{proc}\ \ a\ \rightarrow \mathbf{do}
\end{array}
$$

```
    ...
    y  ← mySF2 -< x
    ...
mySF2 :: SF SomeResources a' b'
mySF2 = proc a' → do
    ...
    y' ← mySF3 -< x'
    ...
```

Perhaps when working with *mySF*, we find that in one of its deeply nested constituents (perhaps, $mySF_5$), there is an *Integer* value that we would like to use back at the top level (i.e. in *mySF*). Seemingly, the only solution is to modify the types of $mySF_2$, $mySF_3$, and so on, so that they return $(b', Integer)$ and $(b'', Integer)$ and so on; in this way the integer value can be threaded back to the top level. This is tedious and cumbersome and an irritating drawback to traditional signal function programming. However, because of the safety that resource types provide, there is an alternative solution: we can use a *wormhole*.

A wormhole is a way to non-locally move data between signal functions. It is essentially a reference in memory that can only be read from or written to with unique reader and writer signal functions. They are kept unique by each having a distinct resource type, so we can guarantee that they are only ever written to in one place and only ever read from in one place. We make use of the following:

```
data Wormhole r1 r2 a =
   Wormhole { whitehole :: SF (S r1) () a,
              blackhole :: SF (S r2) a  () }
makeWormhole :: a → Wormhole r1 r2 a
makeWormhole init = unsafePerformIO $ do
   r ← newIORef init
   return $ Wormhole (source $ readIORef  r)
                     (sink   $ writeIORef r)
```

makeWormhole takes an initial value for the hidden mutable variable and returns a pair of signal functions, the first for reading and the second for writing, with each independently typed. Returning to our example, we can write:

```
data Whitehole
data Blackhole
wormhole :: Wormhole Whitehole Blackhole Integer
wormhole = makeWormhole 0
```

Now, our deeply nested signal function (say $mySF_5$) can write its interesting *Integer* value to the blackhole of this wormhole, and we can read it from the whitehole in *mySF*:

```
mySF  :: SF (ManyResources ∪ S Blackhole ∪
             S Whitehole) a b
mySF  = proc a  → do
   ...
   y ← mySF2 -< x
```

```
    i ← whitehole wormhole ⤙ ()
    ...
mySF_5 :: SF (AFewResources ∪ S Blackhole) a′ b′
mySF_5 = proc a′ → do
    ...
    _ ← blackhole wormhole ⤙ interestingInteger
    ...
```

Without resource types, wormholes would be an unsafe way to transmit data: they could be written to or read from anywhere, and there would be no guarantee that the initial creator and writer of a wormhole controls what is in it. However, with resource types, these seemingly dangerous constructs become both transparent and safe.

4.5 Wormholes Concretely

Now that we have seen how wormholes work and why they are useful, we provide a concrete example of their use. Continuing with our *echo* example from Sections 4.1 and 4.3, suppose we want to add debugging information. There were two values we created in *echo*, *notesOut* and *notes*, but we only return the former. Let's add the ability to view the other value.

If we simply change *echo* to return both note streams, then we need to adjust *echoGUI* and any other functions that rely on *echo* to match, so instead, we use a wormhole. First, we define our wormhole:

```
data DebugW
data DebugB
wormhole :: Wormhole DebugW DebugB (Event [Note])
wormhole = makeWormhole Nothing
```

Then, we update *echo* to use it:

```
echo :: SF (S MidiKBRT ∪ S DebugB)
           (Double, Double, Event [Note]) (Event [Note])
echo = proc (rate, freq, notesIn) → do
    rec notesOut ← midiKB ⤙ merge notesIn notes
        notes    ← delayt ⤙ (1.0/freq,
                              decay rate notesOut)
    _ ← blackhole wormhole ⤙ notes
    returnA ⤙ notesOut
```

The set of resource types for *echo* changes to include *S DebugB*; the set of resource types for *echoGUI* changes similarly, but its implementation can stay the same.

Now, we can make a new *echoGUI* that uses the debug info. Because of the nature of signal functions, this is quite trivial:

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

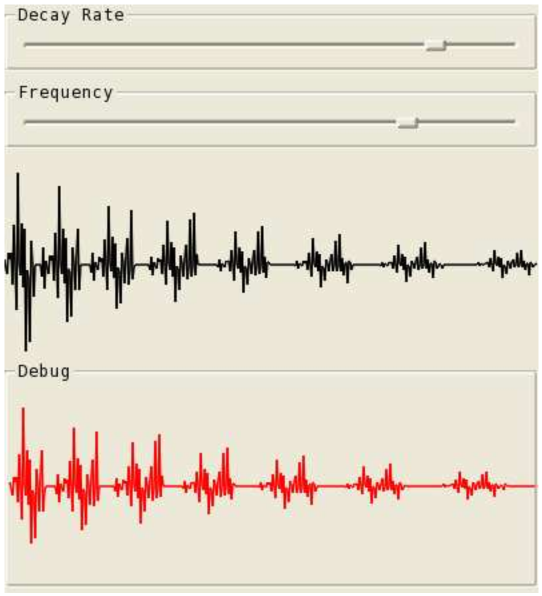


Figure 2: A screenshot of the *echoGUIWithDebug* signal function just after a note has been played on the MIDI keyboard.

```
    toUISF (whitehole wormhole) >>>
    toUISF renderNotes >>> title "Debug" debugGraph
```

Figure 2 is a screenshot of the program in action.

5 Implementation

There are two major parts of our implementation: one that realizes, at the type level, the type-checking rules that allow resource types to work the way they do; and the other that realizes signal functions, and in particular integrates Haskell *IO* actions into the mix. We describe each of these in turn and then describe the implementation of some of our auxiliary functions and our GUI.

5.1 Implementing Resource Type

Before describing our Haskell implementation of resource types, we present type inference rules that capture abstractly what we are trying to achieve. We provide a rule for each of the *Arrow* class operators since the arrow syntax is expanded into precisely this set. However, we shall see shortly that we need to add one extra operator to the *Arrow* class for completeness.

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

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

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

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

$$(init)\frac{\vdash E : \alpha}{\vdash init\ E : SF\ \emptyset\ \alpha\ \alpha}$$

$$(|||)\frac{\begin{array}{c}\vdash E_1 : SF\ \tau'\ \ \alpha\ \gamma \\ \vdash E_2 : SF\ \tau''\ \beta\ \gamma \\ \tau = \tau' \cup \tau''\end{array}}{\vdash E_1 ||| E_2 : SF\ \tau\ (\alpha+\beta)\ \gamma}$$

Figure 3: Resource Type Inference Rules

Type Inference Rules

Figure 3 shows the type inference rules for the standard *Arrow* class operators, as well as those for *ArrowLoop*, *ArrowInit*, and *ArrowChoice*. The $+$ symbol is used here to signify a disjoint (i.e. discriminated) sum type. Set intersection is denoted by $\cap$ and set union by $\cup$. Let's examine each of the rules in turn:

1. The (*arr*) rule states that the set of resource types for a pure function lifted to the arrow level is empty.
2. The (*first*) rule states that transforming a signal function using *first* does not alter the resource type.
3. The ($\ggg$) rule is perhaps the most important; it states that when two signal functions are composed, their resource types must be disjoint, and the resulting resource type is the union of the two.
4. The (*loop*) rule states that the loop combinator must pass the resource type unchanged (i.e. as a loop invariant), reflecting the fact that in a recursively defined signal function, the resource type must be the same at every level of recursion.
5. The (*init*) rule states that the set of resource types for the *init* operator (from the *ArrowInit* class [26]) is empty.
6. The final rule is for the choice operator ($|||$) in the *ArrowChoice* class. The resulting resource type is the union of those of its inputs, which are not required to be disjoint (as discussed in Section 4.2).

Note that the new signal functions created by *init* and *arr* have empty resource types. When defining new signal functions, we need a way to specify their resource types. Thus, we define a function *tag*, whose inference rule is given by:

$$(tag)\frac{\begin{array}{c}\vdash E : SF\ \tau\ \alpha\ \beta \\ \tau \subseteq \tau'\end{array}}{\vdash tag\ E : SF\ \tau'\ \alpha\ \beta}$$

The *tag* function has no run-time effect; it merely adds resource types to the signal function it acts upon.

A nice benefit of the rules described here is that they do not affect the standard arrow laws. All of the changes have to do with resource types, which have no effect on execution. Therefore, a valid instance of the conventional *Arrow* class can be easily extended to a valid one following these rules.

Resource Type Implementation

To implement resource types in Haskell we need a way to represent sets of resource types, integrate them appropriately with our signal functions, and make them consistent with the type inference rules given in the last section. Our implementation is inspired by Haskell's *HList*

```
data Empty
data S a
data a ∪ b

class Disjoint xs ys b | xs ys → b
instance Disjoint Empty    ys HTrue
instance (NotElemOf x ys b) ⇒
         Disjoint (S x)     ys b
instance (Disjoint xs zs b1, Disjoint ys zs b2,
          And b1 b2~b) ⇒
         Disjoint (xs ∪ ys) zs b

class Join xs ys zs | xs ys → zs
instance Join Empty Empty Empty
instance Join Empty ys    ys
instance Join xs    Empty xs
instance (Disjoint xs ys HTrue, Union xs ys zs) ⇒
         Join xs    ys    zs
```

Figure 4: Key Type Classes for Resource Types

library [24] for heterogeneous lists. We use union instead of cons to more easily combine two sets of resource types, but the basic idea is similar.

Appendix A shows the complete code, but in Figure 4 we show two of the more relevant type classes, which we discuss here.

The three empty data types establish the basis for sets of resource types. *Empty* represents the empty set, s a singleton set, and $\cup$ a union of sets. The type families *Or*, *And*, and *Not* are not shown here, but behave as expected.

Disjoint s_1 s_2 *HTrue* declares that s_1 and s_2 are disjoint sets (of resource types). The first instance of the *Disjoint* class declares that the empty set is disjoint from all other sets. The second instance says that if x is not an element of ys, then the singleton set containing x is disjoint from ys. And the final instance says that if both xs and ys are disjoint from zs, then their union is also disjoint from zs.

Join s_1 s_2 s_3 declares that s_3 is the disjoint union of s_1 and s_2. The first instance of the *Join* class declares that the disjoint union of two empty sets is the empty set. The second and third instances declare that the disjoint union of any set s with the empty set is just s. And the final instance says that if two sets are disjoint, then their disjoint union is simply their union.

Re-Typing the Arrow Operators

Now we have a method to represent sets of types as well as type classes for combining them. What remains is to use these types in the typing of the arrow operators, as we did in Section 5.1.

Therefore, our next step is to modify the *Arrow* class itself to obey the inference rules we defined:

```
class Arrow a where
  arr  :: (b → c) → a Empty b c
```

$$\begin{aligned}
&\mathit{first} \;::\; a\ r\ b\ c\ \rightarrow a\ r\ (b, d)\ (c, d)\\
&(\ggg) :: \mathit{Join}\ r_1\ r_2\ r_3 \Rightarrow\\
&\qquad\quad a\ r_1\ b\ c \rightarrow a\ r_2\ c\ d \rightarrow a\ r_3\ b\ d\\
&\mathit{tag} \;\;::\; \mathit{Subset}\ r_1\ r_2 \Rightarrow a\ r_1\ b\ c \rightarrow a\ r_2\ b\ c
\end{aligned}$$

arr and *first* are easily adapted for resource types, as they do not actually affect them. The $(\ggg)$ operator is somewhat more complicated as it needs to perform a disjoint union on the resource types of its arguments. We make use of the *Join* type class we defined in the previous section for this. Lastly, note the addition of the *tag* operator to the class as well.

ArrowChoice

The implementation of *ArrowChoice* involves an interesting issue, and therefore we discuss it in some detail here. The standard *ArrowChoice* class declaration requires instances to define only the function:

$$\mathit{left} :: a\ b\ c \rightarrow a\ (\mathit{Either}\ b\ d)\ (\mathit{Either}\ c\ d)$$

Using this, the default implementation defines a similar function *right*, and then uses them together to define:

$$\begin{aligned}
&(+\!\!+\!\!+) :: a\ b\ c \rightarrow a\ b'\ c' \rightarrow a\ (\mathit{Either}\ b\ b')\ (\mathit{Either}\ c\ c')\\
&f +\!\!+\!\!+ g = \mathit{left}\ f \ggg \mathit{right}\ g
\end{aligned}$$

Finally, $(|||)$ is defined in terms of this.

Adding resource types should be straightforward; *left* and *right* should simply pass their resource types through, and as we discussed in Sections 4.2 and 5.1, $(+\!\!+\!\!+)$ and $(|||)$ should return signal functions with the union of the resource types of their arguments.

However, with just these changes, the class will not compile. The definition of $(+\!\!+\!\!+)$ makes default use of the $(\ggg)$ operator, which requires the disjoint union of resource types of its arguments instead of the natural union that we desire. Rather than force the user to write his or her own definition of $(+\!\!+\!\!+)$ that somehow does not make use of $(\ggg)$, we introduce a new function:

$$\begin{aligned}
&\mathit{unsafeCompose} :: \mathit{Union}\ r_1\ r_2\ r_3 \Rightarrow\\
&\qquad\qquad\qquad a\ r_1\ b\ c \rightarrow a\ r_2\ c\ d \rightarrow a\ r_3\ b\ d
\end{aligned}$$

The *unsafeCompose* function is an unsafe way to compose two signal functions. It is only to be used when one is confident that the composition is safe despite what the resource types are, as in this case where we know that $\mathit{left}\ f \ggg \mathit{right}\ g$ will only ever execute *either f or g*. Because it functions identically to $(\ggg)$, its definition should be the same; the only difference is in its lack of type restricitons.

Now we simply add *unsafeCompose* to the class and update $(+\!\!+\!\!+)$ to make use of it, and *ArrowChoice* is complete. Figure 5 shows the finished version.

5.2 Monadic Signal Functions

With the types prepared, we are ready to move on to an instantiation of the *Arrow* class. We begin with a standard implementation of a signal function, for example as used in Yampa [28],

```
class Arrow a => ArrowChoice a where
  left   :: a r b c -> a r (Either b d) (Either c d)
  right  :: a r b c -> a r (Either d b) (Either d c)
  right f = arr mirror >>> left f >>> arr mirror
            where mirror (Left x)  = Right x
                  mirror (Right y) = Left y
  unsafeCompose :: Union r1 r2 r3 =>
                   a r1 b c -> a r2 c d -> a r3 b d
  (+++) :: Union r1 r2 r3 => a r1 b c -> a r2 b' c' ->
           a r3 (Either b b') (Either c c')
  f +++ g  = left f `unsafeCompose` right g
  (|||)  :: Union r1 r2 r3 => a r1 b d -> a r2 c  d ->
           a r3 (Either b c)  d
  f  |||  g  = f +++ g >>> arr untag
            where untag (Left x)  = x
                  untag (Right y) = y
```

Figure 5: The definition of ArrowChoice with resource types

but with the addition of a resource type parameter:

```
data SigF r a b = SigF
  {sfFunction :: a -> (b, SigF r a b)}
```

Here, a signal function is a function that consumes a value of its input type and produces a value of its output type along with a new function for the next input value.

We are not done yet though, as this definition is lacking: we need to be able to perform monadic *IO* actions within the signal functions. Although our newly adopted model of program execution is based on signal functions, we still have to implement everything in Haskell, which is based on monadic I/O. To address this, we add a monad parameter to the signal function data type. This leads to the following design:

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

which we can use to instantiate the *Arrow* class as follows:

```
instance Arrow (SFM m) where
  arr f = SFM h
    where h x = return (f x, SFM h)
  first (SFM f) = SFM (h f)
    where h f (x, z) = do (y, SFM f') <- f x
                          return ((y, z), SFM (h f'))
  SFM f >>> SFM g = SFM (h f g)
    where h f g x = do (y, SFM f') <- f x
                       (z, SFM g') <- g y
                       return (z, SFM (h f' g'))
```

```
$tag\ (SFM\ f) = SFM\ (h\ f)$
  where $h\ f\ x$ = do $(y, SFM\ f') \leftarrow f\ x$
                       $return\ (y, SFM\ (h\ f'))$
```

The definition of *arr* f is a signal function that, when presented with a value x, returns $f\ x$ along with a copy of itself for use with the next value.

For the definition of *first*, we first extract the *sfmFun* from the argument (through pattern matching) and bind it to f. Then we produce a signal function that takes a pair. It runs f on the first element of the pair producing a value y and a new signal function. It returns y paired with the other half of the input pair untouched along with a version of itself updated with the new signal function that f produced.

For the definition of $(\ggg)$, we extract the *sfmFun*s from both of the arguments as f and g. The input value is run first through f producing the temporary value y and the signal function f' for use later. Then, y is fed into g to produce the output value as well as g'. The signal function returns this output value along with a version of itself updated with f' and g'.

Lastly, as *tag* has no real runtime effect, it produces a new signal function that behaves identically to its argument. The only real change is in the type.

An astute reader may guess at this point the definition of *SF* that we used in Sections 3 and 4:

```
newtype $SF = SFM\ IO$
```

5.3 Auxiliary Functions

Now that we have a complete description of *SF*, we can easily show the definitions of *source*, *sink*, and *pipe* from Section 3.2:

```
$source :: IO\ c \qquad\quad \rightarrow SF\ (S\ r)\ ()\ c$
$sink \quad :: (b \rightarrow IO\ ()) \rightarrow SF\ (S\ r)\ b\ ()$
$pipe \quad :: (b \rightarrow IO\ c)\ \rightarrow SF\ (S\ r)\ b\ c$
$source\ f = SF\ h$ where
  $h\ \_ = f \quad \ggg= return \circ (\lambda\ x \rightarrow (x, SF\ h))$
$sink\ f \quad = SF\ h$ where
  $h\ x = f\ x\ \gg\ return\ ((), SF\ h)$
$pipe\ f \quad = SF\ h$ where
  $h\ x = f\ x \ggg= return \circ (\lambda\ x \rightarrow (x, SF\ h))$
```

We also have a convenient *liftToEvent* function not showcased in this paper which transforms a continuous signal function to an event-based one:

```
$liftToEvent :: SF\ r\ a\ b \rightarrow SF\ r\ (Event\ a)\ (Event\ b)$
$liftToEvent\ sf$ = proc $a \rightarrow$ do
  case $a$ of
    $Event\ a' \rightarrow sf \ggg arr\ Event \prec a'$
    $NoEvent \rightarrow returnA \prec NoEvent$
```

When an event is received, the underlying signal function is activated on the input; otherwise, *NoEvent* is returned with no further work.

Although it may seem obvious to generate *sourceE*, *sinkE*, and *pipeE* by combining their continous versions with *liftToEvent*, this will not work as desired. These functions require more than simple lifting because they are designed to prevent blocking. We want the input function to run in the background while the signal function continues in the main thread. To achieve this, we spawn two helper threads: one to execute the function and another to monitor its progress. Because the three constructor functions each need to do this, we generalize the behavior to a new function. This function, *toSFE*, takes any signal function and turns it into a non-blocking event-based one. *toSFE* makes clever use of *Chan*s to make sure that background threads are performing even when their results are not yet required while the foreground signal function continues as normal. Since the code is fairly complex, we relegate it to Appendix B.

With *toSFE*, we can easily write the three event-based constructor functions:

$$\begin{array}{lll}
\mathit{sourceE} :: \mathit{IO}\ c & \rightarrow \mathit{SF}\ (S\ r)\ () & (\mathit{Event}\ c) \\
\mathit{sinkE} \quad :: (b \rightarrow \mathit{IO}\ ()) & \rightarrow \mathit{SF}\ (S\ r)\ (\mathit{Event}\ b) & () \\
\mathit{pipeE} \quad :: (b \rightarrow \mathit{IO}\ c) & \rightarrow \mathit{SF}\ (S\ r)\ (\mathit{Event}\ b) & (\mathit{Event}\ c)
\end{array}$$

$$\begin{array}{ll}
\mathit{sourceE}\ f & = \mathit{arr}\ (\mathit{const}\ \$\ \mathit{Just}\ ()) \ggg \mathit{toSFE}\ (\mathit{source}\ f) \\
\mathit{sinkE}\ f & = \mathit{toSFE}\ (\mathit{sink}\ f) \ggg \mathit{arr}\ (\mathit{const}\ ()) \\
\mathit{pipeE} & = \mathit{toSFE} \circ \mathit{pipe}
\end{array}$$

5.4 GUI Design

In Section 4.3, we discussed using resource types with virtual objects – we briefly introduced a new type of signal function, *UISF*, and demonstrated a few widgets to use with it. This UI design was adapted from Euterpea's *UI* package [21].

To describe the way in which we created our GUI from Euterpea's *UI* package requires little use of resource types, but it is an interesting reference of how one can adapt monadic functions into signal functions that perform actions (i.e. signal functions that need resource types). Due to both this and its Euterpea-specific nature, we relegate our discussion to Appendices C and D.

6 Limitations

Resource types provide a safe way to manage resources when programming with signal functions, but the system is not without its drawbacks. In this section we discuss the limitations of our approach and suggest possible solutions.

6.1 Reusing Resource Types

We can show that the proper use of resource types results in safe programs, but we cannot enforce their proper use. Even assuming that the user does mark every appropriate signal function with a resource type, he or she may still accidentally *reuse* resource types – that is, use the same resource type for different resource-typed signal functions. This will not cause a program to be unsafe, but it will prevent perfectly safe programs from running.

Alternatively (and much more dangerously), the user could use *different* resource types for signal functions that access the *same* resource. For example, consider the following code:

$midiKB_1 :: SF\ (S\ MidiKBRT_1)\ (Event\ String)\ ()$
$midiKB_2 :: SF\ (S\ MidiKBRT_2)\ (Event\ String)\ ()$
$midiKB_1 = sinkE\ midiKeyboard$
$midiKB_2 = sinkE\ midiKeyboard$

These two signal functions both access the same MIDI keyboard, but they can be used together because they have different resource types. We have no easy way to detect or dissuade this behavior; we simply demand that the programmer take care when assigning resource types.

On the other hand, we should point out that this "flaw" is also a "feature", in that it is what allows us to instantiate the two independent random number generators described in Section 3.3. In general, if one knows that two signal functions will not interfere with each other, even if they access the same resource, then those two signal functions can have different resource types.

6.2 Dynamically Created Types

It is very likely, especially when dealing with virtual objects like widgets, that one would want to create a dynamic number of signal functions each with its own resource. For example, a program could present some variable number of sliders to a user depending on user input. However, despite the fact that any number of signal functions can be created, only the limited number of types declared at compile time are available as resource types. Ideally, dynamically generated types could be created when necessary, but Haskell does not support this.

6.3 Type Explosion

Although resource types provide an elegant solution to managing resources, a lengthy program making use of many resources could become unwieldy. Ideally, we would have some way to hide particular "sets" of types from being displayed, so that, for example, a fully-used wormhole's types would not appear in the signal function's type. One method to achieve this would be to have locally-scoped types that could only be used with similarly scoped signal functions – where the types are not visible, the functions would not be either, so there would be no concern of composing them with each other.

7 Future Work

Although our resource type system is fully formed and functional, there are still features that are desirable. Here we present some ideas that we hope to pursue in the future.

7.1 Parallelism and Asynchrony

Because resource types so clearly show where particular resources are being used and assure that resources will not be accidentally touched in other places, they provide a great setting for safely parallelizing programs. Furthermore, constructs like wormholes (but made thread-safe) could provide an easy way for parallel threads to communicate.

In addition to parallelism, the resource type system allows for elegant asynchronous computation. Rather than the typical parallel model of having one input per output and keeping everything synchronized, we can allow slow performing signal functions to run as event-based ones in separate threads that only supply data when their computations complete.

7.2 Type Families

We would like to use type families instead of functional dependencies in our type code mostly because it fits the functional model of programming better. Unfortunately, if we use only type families and type instances, we run into problems coding type equality.

7.3 Dedicated Language

We have thoughts of extending the resource type idea to its own language rather than it just being an extension of Haskell. At the least, this could allow us to have dynamic types and prettier type hiding (as mentioned in Sections 6.2 and 6.3).

8 Related Work

The idea of using continuous modeling for dynamic, reactive behavior (now often referred to as "functional reactive programming") is due to Elliott, beginning with early work on TBAG, a C++ based model for animation [14]. Subsequent work on Fran ("functional reactive animation") embedded the ideas in Haskell [13, 19]. Other embeddings were explored in [11]. The design of Yampa [7, 22] adopted arrows as the basis for FRP, an approach that is used in most of our research at Yale today, including Euterpea. The use of Yampa to program GUI components was explored in [6, 5], which relates to our work in the use of signal functions to represent GUI widgets. Also related is Elliott's recent work on Eros [12].

There is a long history of programming languages designed specifically for audio processing and computer music applications – indeed, the Wikipedia entry for "Audio Programming Language" currently lists 34 languages, including our original work on *Haskore* [20, 18]. Obviously we cannot mention every language. Noteworthy older efforts include Canon [8] and Fugue [9], the latter of which highlighted the utility of lazy evaluation. Perhaps the most "wide spectrum" of these languages is Nyquist [10]. More recent efforts include Supercollider [27], an object-oriented language, and Chuck [40], a dynamic interactive computer music language. Most closely related to Euterpea is "Switched-on Yampa" [15]. Euterpea has its roots in Haskore and [4]. It is worth noting that, except for our recent work on Euterpea, none of these efforts attempt to address the resource typing necessary for safe virtualization of devices.

With regard to resource types, the work that is most closely related to ours is the I/O system used in *Clean* [3, 34, 2], which has a notion of *uniqueness type*. In Clean, every time an I/O operation is performed on a particular device, a value is returned that represents a new instantiation of that device; this value, in turn, must be threaded as an argument to the next I/O operation, and so on. Inititally, there is a value that represents the whole world, referred to as $*World$. To run a program, it is applied to this value, which, along with those of various I/O devices that are used, must be threaded by the programmer through the program. For example (adapted from [1]):

```
Main world = world₂ where
  (newConsole, world₁)    = stdio world
```

```
wConsole    = fwrites "Type a number" newConsole
(success, y, rConsole)    = freadi wConsole
ansConsole = fwrites (fromInt (y * y))   rConsole
(closingSuccess, world2) = fclose ansConsole world1
```

Note the sequence of console values – *newConsole*, *wConsole*, *rConsole*, and *ansConsole* – as well as the sequence of worlds – *world*, $world_1$, and $world_2$. The Clean type system ensures that the programmer does not make a mistake in threading these values through the program – they must be "single-threaded."

The connection between this and our work is obvious, but there are also significant differences. We do not concern ourselves with single-threadedness since we only have one signal function that represents a particular I/O device. Our focus is on ensuring that resource types do not conflict, a notion that is absent from the Clean type system.

Another type-based approach to the problem of single-threadedness is to use *linear logic* [16]. Various authors have proposed language extensions to incorporate linear types, such as [39, 38, 17, 36]. These approaches are more similar to the problem being addressed by Clean, and we believe the comparison we make above between our work and Clean applies to work on linear logic as well.

As mentioned in Section 1, the Haskell I/O system [32] allows us to determine when a function or program is engaged in I/O, and the monadic framework ensures that the I/O functions are single-threaded and well-defined. But, it is rather imperative in nature, and it does not have the level of transparency that our system has.

It seems clear that a language with dependent types, such as Agda [29], could easily encode the resource type constraints that we have done in this paper. However, Agda and related proof assistants (Coq, Epigram, etc.) are aimed primarily at verification, and not general programming as with Haskell.

Separation logic [30, 35] is also relevant to our work at a theoretical level, in which specifications and proofs of a program component refer only to the portion of memory used by that component, and not the entire global state. It seems that an extension of this idea might provide a theoretical foundation for our work, although we have yet to explore it.

Many researchers have used types in recent years to enhance conventional languages in interesting ways, for example to capture various notions of *security*. Most of this work has been done in the context of sequential imperative languages, and focuses on the issue of information flow. This work seems only peripherally related to ours. Although there are many results, too many to mention here, a good summary can be found in [25].

9 Acknowledgements

This research was supported in part by a gift from Microsoft Research (1014700.00.56099A.622001), and a grant from the National Science Foundation (CCF-0811665 CPA-SEL-T: Domain Specific Languages, Logics, and Proofs for Certified Software Design).

Appendices

A Type Class Definitions

```
module TypeSet where

data HTrue
data HFalse

data Empty
data S a
infixr 5 ∪
data a ∪ b

type family And a b
type instance And HTrue  HTrue  = HTrue
type instance And HTrue  HFalse = HFalse
type instance And HFalse HTrue  = HFalse
type instance And HFalse HFalse = HFalse

type family Or a b
type instance Or  HTrue  HTrue  = HTrue
type instance Or  HTrue  HFalse = HTrue
type instance Or  HFalse HTrue  = HTrue
type instance Or  HFalse HFalse = HFalse

type family Not a
type instance Not HTrue  = HFalse
type instance Not HFalse = HTrue

class IfThenElse b x y z | b x y → z
instance IfThenElse HTrue x y x
instance IfThenElse HFalse x y y

class TypeEq x y b | x y → b
instance (HTrue ∼ b) ⇒ TypeEq x x b
instance (HFalse ∼ b) ⇒ TypeEq x y b

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 ∪ zs) b

class NotElemOf x ys b | x ys → b
instance (ElemOf x ys b', Not b'~b) ⇒
         NotElemOf x ys b

class Disjoint xs ys b | xs ys → b
instance Disjoint Empty    ys HTrue
instance (NotElemOf x ys b) ⇒
         Disjoint (S x)    ys b
instance (Disjoint xs zs b1, Disjoint ys zs b2,
            And b1 b2~b) ⇒
         Disjoint (xs ∪ ys) zs b

class Join xs ys zs | xs ys → zs
instance Join Empty Empty Empty
instance Join Empty ys    ys
instance Join xs    Empty xs
instance (Disjoint xs ys HTrue, Union xs ys zs) ⇒
         Join xs    ys    zs

class AddTo x ys zs | x ys → zs
instance (zs~S x) ⇒
         AddTo x Empty zs
instance (TypeEq ys Empty HFalse, zs~(S x ∪ ys)) ⇒
         AddTo x ys    zs

class Union xs ys zs | xs ys → zs
instance Union Empty      Empty Empty
instance Union Empty      ys    ys
instance Union xs         Empty xs
instance (ElemOf x ys b, IfThenElse b ys ys' zs,
            AddTo x ys ys') ⇒
         Union (S x)      ys    zs
instance (Union xs1 ys zs', Union xs2 zs' zs) ⇒
         Union (xs1 ∪ xs2) ys   zs

class Subset' xs ys b | xs ys → b
instance Subset' Empty    ys HTrue
instance (ElemOf x ys b) ⇒
         Subset' (S x)    ys b
instance (Subset' xs zs b1, Subset' ys zs b2,
            And b1 b2~b) ⇒
         Subset' (xs ∪ ys) zs b

class Subset xs ys | xs → ys
instance (Subset' xs ys HTrue) ⇒ Subset xs ys
```

B Code for *toSFE*

```
toSFE :: SF r a b → SF r (Event a) (Event b)
toSFE (SF f) = SF initFun where
  initFun x = do
    inp  ← newChan
    out  ← newChan
    mon ← newChan
    forkIO $ worker inp out f
    forkIO $ monitor out mon NoEvent
    h inp mon x
  h inp mon x = do
    maybeE (return ()) (writeChan inp) x
    c ← newChan
    writeChan mon c
    y ← getChanContents c ≫= return ∘ head
    return (y, SF (h inp mon))
  worker inp out f = do
    x ← readChan inp
    (y, SF f′) ← f x
    writeChan out y
    worker inp out f′
  monitor out mon v = do
    c ← readChan mon
    writeChan c v
    maybeE (monitor′ c)
           (λ _ → monitor out mon NoEvent) v
    where
      monitor′ c = do
        outEmpty  ← isEmptyChan out
        monEmpty ← isEmptyChan mon
        case (¬ outEmpty, ¬ monEmpty) of
          (True, _)       → do
            v ← readChan out ≫= return ∘ Event
            writeChan c v
            readChan c ≫= monitor out mon
          (False, True) → monitor out mon NoEvent
          (False, False) → threadDelay 100 ≫
                            monitor′ c
```

C *UISF*

We begin by constructing the *UISF* signal function. Unfortunately, *UISF* cannot be defined in terms of *SFM* quite as easily as *SF* could. This is because the *UI* monad used in Euterpea is not as atomic as *IO*:

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

When a *UI* object "runs", it is provided with a static rendering context as well as a stream of input data and system commands. Using this, it produces I/O actions (such as sounds or graphical effects), future system commands, layout information, and values.

The problem is that this *UI* type encapsualtes a primitive signal function within itself in the form of *Signal*s. Rather than having input and output *Signal*s, we would like to use our resource-typed signal functions. Therefore, we would ideally have a monad whose type is:

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

When we lift actions in this monad to our signal functions (using the standard constructors shown previously), we still get the desired stream behavior, but we also get the power of arrows and resource types. Unfortunately, this definition fails to handle the static components of the original *UI*: the context and layout. So, we create a separate monad for these whose type is:

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

Because this contextual information does not depend on any streams of data, it should not be included in the signal function directly. This leaves us with the following:

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

The monad instances of the new *UI* and *UICTX* as well as the function to run a *UISF* follow directly from the original implementation of *UI*. Also, *UISF* can be made to instantiate the *Arrow* class in a straightforward way – by simply lifting the arrow functions of the inner *SFM* through *UICTX*. The only notable operation is composition, where the two signal functions are monadically performed in the logical order, and the result of their composition is returned.

An interesting note about this design is that the order of arrow composition makes a difference to the *UICTX* operations. For example, in the *echoGUI* example from Section 4.3, if we were to switch the order of the lines so that the values from *fSlider* were produced before those from *dSlider*, than the widgets would appear on screen in a different order as well.

D Widgets

Having moved from the *UI* monad to the *UISF* signal function, we next need to adjust the implementation of the widgets themselves to match. For the most part, widgets in Euterpea are created by calling the helper function *mkUI*. Therefore, we will show the basics of how *mkUI* is transformed from *UI* to *UISF*. We begin by examining the type of *mkUI*:

```
mkUI :: Layout →
          (CTX → s → Action)
          (CTX → (a, (Input, Sys)) → (s, Sys, b)) →
          (Signal a) → UI b
```

This function takes, in order, a layout, an action creator, and a computation function to produce the desired widget. To convert it to the new *UISF* standard requires only a small type change; we change the result from (*Signal a*) → *UI b* to *UISF r a b*.

The implementation difference is slightly more complicated. The original *mkUI* manipulates *Signal*s (using functions like *liftS* and *zipS* that perform as expected) to create the *UI* output:

```
mkUI layout mkAction comp x =
  UI aux where
    aux ctx inp =
      let (s, sys, y) = unzip3S $ liftS (comp ctx)
                                        (zipS x inp)
          action      = liftS (mkAction ctx) s
      in  (zip3S action sys y, layout)
```

Just as we needed to split the original *UI* monad into two monads (*UI* and *UICTX*), for the *UISF* version of *mkUI*, we need to split this function to separate the components with different functionality. We end up with three segments: first, we set up the *UICTX* part, then we build a signal function wrapper (using arrow syntax) for the *UI* part, and finally, we build that *UI* part:

```
mkUI layout mkAction comp =
  UISF $ UICTX aux where
    aux ctx = (layout, retSF) where
      retSF = unsafePipe compfun
      compfun x = UI h where
        h inp =
          let (s, sys, b) = comp ctx (x, inp)
          in  return (mkAction ctx s, sys, b)
```

Note the separation of the static and continuous parts (*aux* and *compfun* respectively) as well as the addition of the signal function component (*retSF*). Note also that *unsafePipe* is the same as *pipe* (first introduced in Section 3.2) but with empty resource types.

Bibliography

[1] Philippos Apolinario. *Clean – the Computer Language*. 2010.

[2] E. Barendsen and S. Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 6:579–612, 1996.

[3] T. Brus, M.C.J.D. van Eekelen, M. van Leer, M.J. Plasmeijer, and H.P. Barendregt. CLEAN – A language for functional graph rewriting. In *Proc. of Conference on Functional Programming Languages and Computer Architecture (FPCA '87)*, pages 364–384. Springer-Verlag, LNCS 274, 1987.

[4] Eric Cheng and Paul Hudak. Audio processing and sound synthesis in Haskell. Technical Report YALEU/DCS/RR-1405, Yale University, January 2009.

[5] Antony Courtney. *Modelling User Interfaces in a Functional Language*. PhD thesis, Department of Computer Science, Yale University, May 2004.

[6] Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *2001 Haskell Workshop*, September 2001.

[7] Antony Courtney, Henrik Nilsson, and John Peterson. The Yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN Haskell Workshop (Haskell'03)*, pages 7–18, Uppsala, Sweden, August 2003. ACM Press.

[8] R.B. Dannenberg. The Canon score language. *Computer Music Journal*, 13(1):47–56, 1989.

[9] R.B. Dannenberg, C.L. Fraley, and P. Velikonja. A functional language for sound synthesis with behavioral abstraction and lazy evaluation. In Denis Baggi, editor, *Computer Generated Music*. IEEE Computer Society Press, 1992.

[10] Roger B. Dannenberg. The implementation of Nyquist, a sound synthesis language. *Computer Music Journal*, 21(3):71–82, 1997.

[11] Conal Elliott. Functional implementations of continuous modeled animation. In *Proceedings of PLILP/ALP '98*. Springer-Verlag, 1998.

[12] Conal Elliott. Tangible functional programming. In *International Conference on Functional Programming*, 2007.

[13] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 263–273, June 1997.

[14] Conal Elliott, Greg Schechter, Ricky Yeung, and Salim Abi-Ezzi. Tbag: A high level framework for interactive, animated 3d graphics applications. In *Proceedings of SIGGRAPH '94*, pages 421–434. ACM SIGGRAPH, July 1994.

[15] George Giorgidze and Henrik Nilsson. Switched-on yampa. In *Proc. Practical Aspects of Declarative Languages*, pages 282–298. Springer Verlag LNCS, 2008.

[16] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[17] Chris Hawblitzel. Linear types for aliased resources (extended version). Technical Report MSR-TR-2005-141, Microsoft Research, Redmond, WA, October 2005.

[18] Paul Hudak. Haskore music tutorial. In *Second International School on Advanced Functional Programming*, pages 38–68. Springer Verlag, LNCS 1129, August 1996.

[19] Paul Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia*. Cambridge University Press, New York, NY, 2000.

[20] Paul Hudak. Describing and interpreting music in Haskell. In *The Fun of Programming*, chapter 4. Palgrave, 2003.

[21] Paul Hudak. *The Haskell School of Music – from Signals to Symphonies.* [Version 2.0], January 2011.

[22] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Summer School on Advanced Functional Programming 2002, Oxford University*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer-Verlag, 2003.

[23] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.

[24] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell 2004: Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 96–107. ACM Press, 2004.

[25] Xavier Leroy. Computer security from a programming language and static analysis perspective. In *Proceedings of the 12th European conference on Programming*, ESOP'03, pages 1–9, Berlin, Heidelberg, 2003. Springer-Verlag.

[26] Hai Liu, Eric Cheng, and Paul Hudak. Causal commutative arrows and their optimization. In *Proc. International Conference on Functional Programming*, ICFP '09, pages 35–46. ACM Sigplan, 2009.

[27] James McCartney. SuperCollider: a new real time synthesis language. In *Proceedings of International Computer Music Conference*. Int'l Computer Music Association, 1996.

[28] Henrik Nilsson, Antony Courtney, and John Peterson. Functional Reactive Programming, continued. In *ACM SIGPLAN 2002 Haskell Workshop*, October 2002.

[29] Ulf Norell. Dependently typed programming in Agda. In *Lecture Notes from the Summer School in Advanced Functional Programming*, 2008.

[30] P.W. OHearn, J.C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. *Computer Science Logic*, page 1.

[31] Ross Paterson. A new notation for arrows. In *ICFP'01: International Conference on Functional Programming*, pages 229–240, Firenze, Italy, 2001.

[32] S. Peyton Jones and P. Wadler. Imperative functional programming. In *Proceedings 20th Symposium on Principles of Programming Languages*. ACM, January 1993. 71–84.

[33] Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003.

[34] Rinus Plasmeijer and Marko van Eekelen. Clean – version 2.1 language report. Technical report, Department of Software Technology, University of Nijmegen, November 2002.

[35] J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. Logic in Computer Science (LICS'02)*, pages 55–74, July 2002.

[36] Jesse A. Tov and Riccardo Pucella. Practical affine types. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 447–458, New York, NY, USA, 2011. ACM.

[37] Andreas Voellmy and Paul Hudak. Nettle: Taking the sting out of programming network routers. In *PADL*, pages 235–249, 2011.

[38] P. Wadler. Is there a use for linear logic? In *Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 255–273. ACM/IFIP, 1991.

[39] Philip Wadler. Linear types can change the world! In *Working Conference on Programming Concepts and Methods*, Apr 1990.

[40] G. Wang, R. Fiebrink, and P. Cook. Combining analysis and synthesis in the ChucK programming language. In *Proceedings of the International Computer Music Conference*, 2007.