Virtualizing Real-World Objects in FRP

Daniel Winograd-Cort[1], Hai Liu[2], and Paul Hudak[3]

[1] Yale University
daniel.winograd-cort@yale.edu
[2] Intel, Inc.
hai.liu@intel.com
[3] Yale University
paul.hudak@yale.edu

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. We call this *virtualizing real-world objects*. We explore how *virtual objects* (such as GUI widgets) and even *non-local effects* (such as debugging and random number generation) can be handled in the same way.

The key to our approach is the notion of a *resource type* that assures that a virtualized object cannot be duplicated, and is safe. Resource types also provide a deeper level of transparency: by inspecting the type, one can see exactly what resources are being used. We use arrows, type classes, and type families to implement our ideas in Haskell, and the result is a safe, effective, and transparent approach to stream-based I/O.

Keywords: Functional Programming, Arrows, Functional Reactive Programming, Stream Processing, Haskell, Unique Types, 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* [20,19]. 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

C. Russo and N.-F. Zhou (Eds.): PADL 2012, LNCS 7149, pp. 227–241, 2012.

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* [13]. Specifically, we assume that a program is a *signal function* having the (over-simplified for now) type *SF inp out*, where both the input and output are time-varying signals: *inp* is the type of the instantaneous values of the input, and *out* is the type of the instantaneous values of the output. 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* [12,3] (which has been used for animation, robotics, GUI design, and more), *Nettle* [23] (for networking), and *Euterpea* [11] (for audio processing and sound synthesis). In fact, our work was motivated by Euterpea, and in this paper we use examples from that domain: synthesizers, speakers, keyboards, and MIDI devices.[1]

Our research is based on three insights. First, we observe that *objects and devices in the real world can also be viewed as signal functions*. For example, a MIDI keyboard takes note events as input and generates note events as output. Similarly, a speaker takes a signal representing sound as input and produces no output, and a microphone produces a sound signal as output while ignoring its input. So it would 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 approach 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 concrete device handle the multiple event streams that would result from its virtualized duplicates? This leads to our second insight, namely that *the uniqueness of signal function 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.

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

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

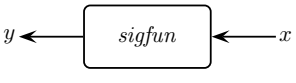
2. *Transparency.* From the type of a signal function, we can determine immediately *all* of the resources that it consumes. In particular, this means that we know all the resources that a complete program uses (with monads, all we know is that some kind of I/O is being performed).
3. *Safety.* As long as each resource is uniquely assigned, a signal function engaged in I/O or non-local effects is *safe* – despite the side effects, equational reasoning is preserved.
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 her own resource type and signal function that capture 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 present our main ideas and the purpose of resource types and then show the type inference rules for them in Section 4. We next work through a number of examples in Section 5 before delving into the implementation details in Section 6. Finally, we discuss limitations and future work in Section 7 and related work in 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, 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:

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

using Haskell's *arrow syntax* [18]. 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 be a variable, whereas the expression on the right can be any well-typed expression that matches the signal function's input type. 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 declares *sigfun* to be a signal function that converts a time-varying value of type *Double* into a time-varying value of type *Double*. The notation "**proc** x → **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 an integrator, whose output is named y. Finally, we specify the output by feeding y into *returnA*, a special signal function that returns the result.

Streams of Events. With respect to I/O, continuous signals can be useful in many contexts, such as the position of a mouse (as input to a program) or the voltage to a robot motor (as output from a program). However, there are many applications where instead we are interested in *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 $x :: T$.[2] For example, a signal function that converts an event stream carrying values of type M_1 into an event stream carrying values of type M_2 has type *SF* (*Event* M_1) (*Event* M_2).

3 Resource Types

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

```
midiSynth :: SF (Event Notes) ()
```

midiSynth takes a stream of *Notes*[3] events as input, synthesizes the appropriate sound of those simultaneous notes, and returns unit values. Now consider this code fragment:

```
_ ← midiSynth —≺ notes1
_ ← midiSynth —≺ notes2
```

We intend for *midiSynth* to represent a single real-world 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, suppose *randomSF* is intended to be a random number generator initialized with a random seed from the OS:

```
randomSF :: SF () Double
```

[2] The name *Event* is overloaded as both the type and data constructor.

[3] The *Notes* type represents a set of simultaneously sounding notes such as a chord or just a single note.

Now consider this code fragment:

$$rands_1 \leftarrow randomSF \multimap ()$$
$$rands_2 \leftarrow randomSF \multimap ()$$

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

A Solution. Our solution to these problems consists of four parts. First, to prevent duplication of signal functions, 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 cases above, we introduce the resource types *MidiSynthRT* and *RandomRT* (by convention, we always use *RT* as the suffix for resource type names).

Second, to keep track of resource types, we introduce three type-level constructors: *Empty*, *S* and ∪. *Empty* is the empty set of resource types; the type *S MidiSynthRT* is the singleton set containing only *MidiSynthRT*; and the binary operator ∪ constructs the union of two sets of resource types.

Third, we add a "phantom" type parameter to each signal function that captures the set of resource types that it uses. A signal function of type *SF r a b* accesses the resources represented by *r*, while converting a signal of type *a* into a signal of type *b*. Following the examples above, this leads to:

$$midiSynth :: SF\ (S\ MidiSynthRT)\ (Event\ Notes)\ ()$$
$$randomSF :: SF\ (S\ RandomRT)\ \ ()\ \ \ \ \ \ \ \ \ \ \ \ Double$$

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

$$source :: IO\ c \rightarrow SF\ (S\ r)\ ()\ c$$
$$sink :: (b \rightarrow IO\ ()) \rightarrow SF\ (S\ r)\ b\ ()$$
$$pipe :: (b \rightarrow IO\ c) \rightarrow SF\ (S\ r)\ b\ c$$

In each case, the resultant signal function has a singleton resource type because it is expected to be applied to a monadic I/O action of a single I/O device, thus consuming a single resource.

For event-based signal functions (as described in Section 2) we provide three analagous functions: *sourceE*, *sinkE*, and *pipeE* with the expected types.

Running Examples. Continuing with our running examples, suppose that: $midiSynthM :: Notes \rightarrow IO\ ()$ is the monadic action that sends a set of notes to the synthesizer. We can then define *midiSynth* as follows:

```
data MidiSynthRT
midiSynth :: SF (S MidiSynthRT) (Event Notes) ()
midiSynth = sinkE midiSynthM
```

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

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

```
data RandomRT

randomSF :: SF (S RandomRT) () Double
randomSF = source randomIO
```

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

Redefining the Arrow Class. Our key technical result is that, because we are using arrows, we can now re-type each of the combinators in the *Arrow* class in such a way that the problematical code fragments given earlier *will not type check*. The details of how this is done are described in the next Section, but for now the key intuition is 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. Both of the problematical code fragments given earlier fall into this category. For example:

```
_ ← midiSynth -≺ notes1
_ ← midiSynth -≺ notes2
```

is essentially the composition of two instances of *midiSynth* – but each of them has the same set of resource types, namely *S MidiSynthRT*; thus they are not disjoint, and not well typed. One way to fix this is to explicitly merge $notes_1$ and $notes_2$:

```
_ ← midiSynth -≺ noteMerge notes1 notes2
```

Now there is one occurence of *midiSynth*, and all is well.

The problematical example involving random numbers leads to a more interesting result if we wish to have *two independent* random number generators. We achieve this by defining two different resource types, and two different versions of *randomSF*:

```
data RandomRT1
data RandomRT2

randomSF1 :: SF (S RandomRT1) () Double
randomSF1 = source randomIO

randomSF2 :: SF (S RandomRT2) () Double
randomSF2 = source randomIO
```

A slight variation of the problematical code yields the desired well-typed result:

```
rands1 ← randomSF1 -≺ ()
rands2 ← randomSF2 -≺ ()
```

(Because each element produced by *randomIO* is independently random, multiple calls will not interfere with each other. Therefore, we can use alternating calls to *randomIO* to produce two independent random streams.)

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

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

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

Fig. 1. Resource Type Inference Rules

4 Type Inference Rules

In Haskell, the arrow syntax is translated into a set of combinators that are captured by the type classes *Arrow*, *ArrowLoop*, *ArrowChoice*, and *ArrowInit*. Space limitations preclude a detailed discussion of this translation process (see [18]). Once translated, the type inference rules that form the basis of our implementation are shown in Figure 1. There is one rule for each of the operators in the above type classes. The + symbol denotes the 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) 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 5).

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

$$(tag)\frac{\vdash E : SF\ \tau\ \alpha\ \beta \qquad \tau \subseteq \tau'}{\vdash tag\ E : SF\ \tau'\ \alpha\ \beta}$$

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

5 More Examples

Recursion. A MIDI keyboard is a stream transformer that adds the notes played on the keyboard in real time to the stream it operates on. It has the type:

$midiKB :: SF\ (S\ MidiKBRT)\ (Event\ Notes)\ (Event\ Notes)$

We can define a signal function that creates an "echo" effect for notes played on the keyboard by delaying and looping them through the keyboard itself, attenuating each note by some percentage on each loop:

```
echo :: SF (S MidiKBRT) (Double, Double) (Event Notes)
echo = proc (rate, freq) → do
  rec notesOut ← midiKB ⪡ notes
      notes    ← delayt ⪡ (1.0/freq, decay rate notesOut)
  returnA ⪡ notesOut
```

Note the use of the **rec** keyword – this will induce the loop rule from Section 4, and everything is well typed.

echo is a signal function that takes a decay rate and frequency as time varying arguments and uses them to add an echo to the notes played on the MIDI keyboard. It uses two helper functions: *decay rate ns* attenuates each note in *ns* by *rate*, and $delayt \prec (t, ns)$ delays each event in *ns* by the time t.

Conditionals. As discussed earlier, 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. Consider the following functions for sending sound data to speakers:

```
playLeft   :: SF (S LeftRT)  Sound ()
playRight  :: SF (S RightRT) Sound ()
playStereo :: SF (S LeftRT ∪ S RightRT) Sound ()
```

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

```
data SpeakerChoice = Left | Right | Stereo
routeSound :: SF (S LeftRT ∪ S RightRT) (SpeakerChoice, Sound) ()
routeSound = proc (sc, sound) → do
```

```
case sc of
  Left   → playLeft   -≺ sound
  Right  → playRight  -≺ sound
  Stereo → playStereo -≺ sound
```

This is well typed, since the case statement in arrow syntax invokes the inference rule for the choice operator (|||) in the *ArrowChoice* class given in Section 4.

Virtual Objects. Virtual (GUI) components can be treated the same as concrete devices in our framework. In this section we extend the echo example given earlier to allow the user to pick the decay rate and frequency using GUI "sliders" and for the echo result to be 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 use:

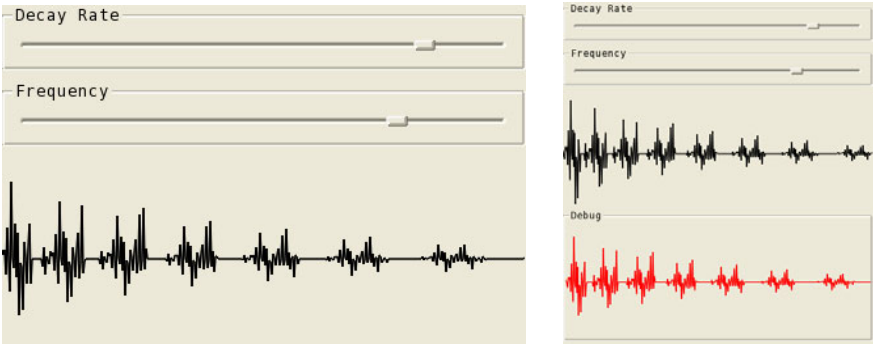
```
decSlider  :: UISF (S DSlider) ()     Double
freqSlider :: UISF (S FSlider) ()     Double
graph      :: UISF (S Graph)   Double ()
decSlider  = title "Decay Rate" $ hSlider (0, 0.9) 0.5
freqSlider = title "Frequency"  $ hSlider (1, 10)  10
graph      = realtimeGraph (400, 300) 400 20 Black
```

We also require *renderNotes*::*SF Empty* (*Event Notes*) *Double*, a signal function that transforms our *Notes* events into sound data. With these functions we can define our main application:

```
echoGUI :: UISF (S MidiKBRT ∪ S DSlider ∪ S FSlider ∪ S Graph) () ()
echoGUI = proc _ → do
  rate   ← decSlider              -≺ ()
  freq   ← freqSlider             -≺ ()
  notes  ← toUISF echo            -≺ (rate, freq)
  sound  ← toUISF renderNotes     -≺ notes
  _      ← graph                  -≺ sound
  returnA -≺ ()
```

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 2a at the end of this section shows a screenshot of the program in action.

Wormholes. 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,



(a) *echoGUI* **(b)** *echoGUIWithDebug*

Fig. 2. Screenshots of the GUI signal functions from Section 5 just after a note has been played on the MIDI keyboard

mutation and direct memory access, techniques that are typically plagued by difficult-to-find bugs, can be made safe. We begin by defining:

$$\textbf{data}\ \mathit{Wormhole}\ r_1\ r_2\ a = \mathit{Wormhole}\ \{\mathit{whitehole} :: \mathit{SF}\ (S\ r_1)\ ()\ a,\ \mathit{blackhole} :: \mathit{SF}\ (S\ r_2)\ a\ ()\}$$
$$\mathit{makeWormhole} :: a \to \mathit{Wormhole}\ r_1\ r_2\ a$$

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.

Continuing with our *echo* example from previous sections, suppose we want to add debugging information. There were two values we created in *echo* – *notesOut* and *notes* – but we only return the former. However, if we try to 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:

$$\mathit{wormhole} :: \mathit{Wormhole}\ \mathit{DebugW}\ \mathit{DebugB}\ (\mathit{Event}\ \mathit{Notes})$$
$$\mathit{wormhole} = \mathit{makeWormhole}\ \mathit{Nothing}$$
$$\mathit{echo} :: \mathit{SF}\ (S\ \mathit{MidiKBRT} \cup S\ \mathit{DebugB})\ (\mathit{Double}, \mathit{Double})\ (\mathit{Event}\ \mathit{Notes})$$
$$\mathit{echo} = \textbf{proc}\ (\mathit{rate}, \mathit{freq}) \to \textbf{do}$$
$$\quad \textbf{rec}\ \mathit{notesOut} \leftarrow \mathit{midiKB} \multimap \mathit{notes}$$
$$\quad\quad \mathit{notes} \leftarrow \mathit{delayt} \multimap (1.0/\mathit{freq}, \mathit{decay}\ \mathit{rate}\ \mathit{notesOut})$$
$$\quad \_ \leftarrow \mathit{blackhole}\ \mathit{wormhole} \multimap \mathit{notes}$$
$$\quad \mathit{returnA} \multimap \mathit{notesOut}$$

The set of resource types for *echo* changes to include $S\ \mathit{DebugB}$; the set of resource types for *echoGUI* changes similarly, but its implementation remains the same.

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

$$\mathit{debugGraph} :: \mathit{UISF}\ (S\ \mathit{DebugGraph})\ \mathit{Double}\ ()$$
$$\mathit{debugGraph} = \mathit{title}\ \texttt{"Debug"}\ \$\ \mathit{realtimeGraph}\ (400, 300)\ 400\ 20\ \mathit{Red}$$
$$\mathit{echoGUIWithDebug} = \textbf{proc}\ \_ \to \textbf{do}$$

$$
\begin{array}{lll}
\_ & \leftarrow \mathit{echoGUI} & \prec () \\
\mathit{debugVal} & \leftarrow \mathit{toUISF}\ (\mathit{whitehole}\ \mathit{wormhole}) & \prec () \\
\mathit{rendered} & \leftarrow \mathit{toUISF}\ \mathit{renderNotes} & \prec \mathit{debugVal} \\
\_ & \leftarrow \mathit{debugGraph} & \prec \mathit{rendered} \\
\mathit{returnA} & \prec () &
\end{array}
$$

Figure 2b shows a screenshot of the program in action.

6 Implementation

Implementing Resource Types. 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 earlier. Our implementation is inspired by Haskell's *HList* library [14] for heterogeneous lists.

We lack the space to show the complete code, but here we show the most relevant type class, *Disjoint*:

class *Disjoint xs ys*
instance *Disjoint Empty ys*
instance $(\mathit{NotElemOf}\ x\ ys\ \mathit{HTrue}) \Rightarrow \mathit{Disjoint}\ (S\ x)\ ys$
instance $(\mathit{Disjoint}\ xs_1\ ys, \mathit{Disjoint}\ xs_2\ ys) \Rightarrow \mathit{Disjoint}\ (xs_1 \cup xs_2)\ ys$

$\mathit{Disjoint}\ s_1\ s_2$ 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_1 and xs_2 are disjoint from ys, then their union is also disjoint from ys.

Re-Typing the Arrow Operators. We now 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 4.

class *Arrow a* **where**
$\mathit{arr} :: (b \to c) \to a\ \mathit{Empty}\ b\ c$
$\mathit{first} :: a\ r\ b\ c \to a\ r\ (b, d)\ (c, d)$
$(\ggg) :: (\mathit{Disjoint}\ r_1\ r_2, \mathit{Union}\ r_1\ r_2\ r_3) \Rightarrow a\ r_1\ b\ c \to a\ r_2\ c\ d \to a\ r_3\ b\ d$
$\mathit{tag} :: \mathit{Subset}\ r_1\ r_2 \Rightarrow a\ r_1\ b\ c \to a\ r_2\ b\ c$

arr and *first* are easily adapted, as the resource types do not actually affect their behavior. The $(\ggg)$ operator is more complex as it needs to perform a disjoint union on the resource types of its arguments. The *Disjoint* type class from the previous section assures the arguments are well-typed, and the *Union* type class behaves like the $\cup$ operator except that it simplifies degenerate cases like $r \cup \mathit{Empty}$ to just r. Lastly, we add the *tag* operator to the class as well.

Monadic Signal Functions. With the types prepared, we can instantiate the *Arrow* class. We begin with a standard implementation of a signal function, such as from Yampa [15], but with an additional resource type parameter:

$$\textbf{data}\ SigF\ r\ a\ b = SigF\ \{\, sfFunction :: a \to (b, SigF\ r\ a\ b)\,\}$$

Here, a signal function 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.

However, this definition does not allow us to perform monadic *IO* actions within the signal function. 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:

$$\textbf{data}\ SFM\ m\ r\ a\ b = SFM\ \{\, sfmFun :: a \to m\ (b, SFM\ m\ r\ a\ b)\,\}$$

Note that this is the automaton arrow transformer specialized to the Kleisli arrow, with an added resource type parameter. The instances for *Arrow*, etc. follow directly. For example, for the *Arrow* class:

$$\begin{aligned}
&\textbf{instance}\ Arrow\ (SFM\ m)\ \textbf{where}\\
&\quad arr\ f = SFM\ h\ \textbf{where}\ h\ x = return\ (f\ x, SFM\ h)\\
&\quad first\ (SFM\ f) = SFM\ (h\ f)\\
&\qquad \textbf{where}\ h\ f\ (x, z) = \textbf{do}\ (y, SFM\ f') \leftarrow f\ x\\
&\qquad\qquad\qquad\qquad\quad return\ ((y, z), SFM\ (h\ f'))\\
&\quad SFM\ f \ggg SFM\ g = SFM\ (h\ f\ g)\\
&\qquad \textbf{where}\ h\ f\ g\ x = \textbf{do}\ (y, SFM\ f') \leftarrow f\ x\\
&\qquad\qquad\qquad\qquad\quad (z, SFM\ g') \leftarrow g\ y\\
&\qquad\qquad\qquad\qquad\quad return\ (z, SFM\ (h\ f'\ g'))\\
&\quad tag\ (SFM\ f) = SFM\ h\ \textbf{where}\ h\ x = \textbf{do}\ (y, sf') \leftarrow f\ x\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad return\ (y, tag\ sf')
\end{aligned}$$

At this point, the astute reader may guess the definition of *SF* that we introduced in Section 3:

$$\textbf{newtype}\ SF = SFM\ IO$$

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:

$$\begin{aligned}
source\ f &= SF\ h\ \textbf{where}\ h\ _ = f \ggg return \circ (\lambda\ x \to (x, SF\ h))\\
sink\ f &= SF\ h\ \textbf{where}\ h\ x = f\ x \gg return\ ((), SF\ h)\\
pipe\ f &= SF\ h\ \textbf{where}\ h\ x = f\ x \ggg return \circ (\lambda\ x \to (x, SF\ h))
\end{aligned}$$

7 Limitations and Future Work

Reusing Resource Types. The benefits of resource types rely on their proper assignment to actual resources, which is not something we can enforce. Even assuming that the user marks every appropriate signal function with a resource type, he or she may still accidentally use the same resource type for different signal functions that don't share a resource. This will not cause a program to be unsafe, but it might prevent perfectly safe programs from type-checking. Alternatively (and more dangerously), the user could use *different* resource types for signal functions that access the *same* resource. This would allow one to use

the same resource multiple times without the type-checker complaining. We have no easy way to detect or dissuade this behavior; we simply demand that the programmer take care when assigning resource types.

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. In general, if two signal functions will not interfere with each other, even if they access the same resource, then they can have different resource types.

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.

Of course, one could create a compound signal function that displays an arbitrary number of sliders yet only has one resource type. Although this is a practical way to deal with the problem, it reduces the effectiveness of resource typing, so we are exploring alternative solutions.

Type Explosion. Although resource types provide an elegant means to managing resources, lengthy programs making use of many resources can 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. Currently, the best way to do this is to group the set of unwanted resource types into a type synonym like so:

$$\textbf{type}\ \mathit{ExtraRTs} = S\ \mathit{Blackhole}_1 \cup S\ \mathit{Whitehole}_1 \cup S\ \mathit{Blackhole}_2 \cup \ldots$$
$$\mathit{mySF} :: \mathit{SF}\ (S\ \mathit{Resource}_1 \cup S\ \mathit{DebugB} \cup \mathit{ExtraRTs})\ a\ b$$

Here, *mySF* uses $\mathit{Resource}_1$ and a debugging black hole and hides the rest of its internal resource types in *ExtraRTs*. However, a more desirable method to achieve this would be to have locally-scoped types that could only be used with similarly scoped signal functions.

Parallelism and Asynchrony. Because resource types 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, resource types allow for elegant asynchronous computation. Rather than the typical parallel synchronous model, where each input correspdons to one output, we can allow slow performing signal functions to run as event-based ones in separate threads that only supply data when their computations complete.

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 [6]. Subsequent work on Fran ("functional reactive animation") embedded the ideas in Haskell [5,9]. The design of Yampa [3,12] adopted arrows as the basis for FRP, an approach that is used in most of our research today, including Euterpea. The use of Yampa to program GUI components was explored in [2,1], which relates to our work in the use of signal functions to represent GUI widgets. So, for example, in Fruit, a model very similar to our *UISF* was proposed, but it does not require resource types. They avoid the problem of reource duplicaton by making their "widgets" essentially pure functions with well defined but restricted output (e.g. *Picture*). Our work allows us to lift this restriction as we address the duplication problem through resource types. Also related is Elliott's recent work on Eros [4].

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* [10]. Obviously we cannot mention every language. It is worth noting that, except for our recent work on Euterpea, none of these efforts attempt to address the safe virtualization of devices.

With regard to types, the idea of linear typing is somewhat similar to our work. For example, the language *Clean* [21] has a notion of *uniqueness types*. In Clean, when an I/O operation is performed on a 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. This single-threadedness can also be tackled using *linear logic* [7]. In fact, various authors have proposed language extensions to incorporate linear types, such as [24,8]. In contrast, we do not concern ourselves with single-threadedness since we only have one signal function to represent any particular I/O device. Our focus is on ensuring that resource types do not conflict.

It seems clear that a language with dependent types, such as Agda [16], could easily encode the resource type constraints that we showed in this paper. However, Agda and related proof assistants (Coq, Epigram, etc.) are aimed primarily at verification, and not general programming as Haskell is.

Separation logic [17,22] is also relevant, 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. An extension of this idea might provide a theoretical basis for our work, although we have yet to explore it.

Acknowledgements. This research was supported in part by a gift from Microsoft Research and a grant from the National Science Foundation (CCF-0811665).

References

1. Courtney, A.: Modelling User Interfaces in a Functional Language. Ph.D. thesis, Department of Computer Science, Yale University (May 2004)
2. Courtney, A., Elliott, C.: Genuinely functional user interfaces. In: 2001 Haskell Workshop (September 2001)

3. Courtney, A., Nilsson, H., Peterson, J.: The Yampa arcade. In: Proceedings of the 2003 ACM SIGPLAN Haskell Workshop (Haskell 2003), pp. 7–18. ACM Press, Uppsala (2003)
4. Elliott, C.: Tangible functional programming. In: International Conference on Functional Programming (2007), http://conal.net/papers/Eros/
5. Elliott, C., Hudak, P.: Functional reactive animation. In: International Conference on Functional Programming, pp. 263–273 (June 1997)
6. Elliott, C., Schechter, G., Yeung, R., Abi-Ezzi, S.: Tbag: A high level framework for interactive, animated 3d graphics applications. In: Proceedings of SIGGRAPH 1994, pp. 421–434. ACM SIGGRAPH (July 1994)
7. Girard, J.Y.: Linear logic. Theoretical Computer Science 50, 1–102 (1987)
8. Hawblitzel, C.: Linear types for aliased resources (extended version). Tech. Rep. MSR-TR-2005-141, Microsoft Research, Redmond, WA (October 2005)
9. Hudak, P.: The Haskell School of Expression – Learning Functional Programming through Multimedia. Cambridge University Press, New York (2000)
10. Hudak, P.: Describing and interpreting music in Haskell. In: The Fun of Programming, ch. 4. Palgrave (2003)
11. Hudak, P.: The Haskell School of Music – from Signals to Symphonies (Version 2.0) (January 2011), http://haskell.cs.yale.edu/?post_type=publication&p=112
12. Hudak, P., Courtney, A., Nilsson, H., Peterson, J.: Arrows, Robots, and Functional Reactive Programming. In: Jeuring, J., Jones, S.L.P. (eds.) AFP 2002. LNCS, vol. 2638, pp. 159–187. Springer, Heidelberg (2003)
13. Hughes, J.: Generalising monads to arrows. Science of Computer Programming 37, 67–111 (2000)
14. Kiselyov, O., Lämmel, R., Schupke, K.: Strongly typed heterogeneous collections. In: Haskell 2004: Proceedings of the ACM SIGPLAN Workshop on Haskell, pp. 96–107. ACM Press (2004)
15. Nilsson, H., Courtney, A., Peterson, J.: Functional Reactive Programming, continued. In: ACM SIGPLAN 2002 Haskell Workshop (October 2002)
16. Norell, U.: Dependently Typed Programming in Agda. In: Koopman, P., Plasmeijer, R., Swierstra, D. (eds.) AFP 2008. LNCS, vol. 5832, pp. 230–266. Springer, Heidelberg (2009)
17. OHearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. Computer Science Logic, p. 1
18. Paterson, R.: A new notation for arrows. In: ICFP 2001: International Conference on Functional Programming, Firenze, Italy, pp. 229–240 (2001)
19. Peyton Jones, S., Wadler, P.: Imperative functional programming. In: Proceedings 20th Symposium on Principles of Programming Languages, pp. 71–84. ACM (January 1993)
20. Peyton Jones, S., et al.: The Haskell 98 language and libraries: The revised report. Journal of Functional Programming 13(1), 0–255 (January 2003)
21. Plasmeijer, R., van Eekelen, M.: Clean – version 2.1 language report. Tech. rep., Department of Software Technology, University of Nijmegen (November 2002)
22. Reynolds, J.: Separation logic: A logic for shared mutable data structures. In: Proc. Logic in Computer Science (LICS 2002), pp. 55–74 (July 2002)
23. Voellmy, A., Hudak, P.: Nettle: Taking the Sting Out of Programming Network Routers. In: Rocha, R., Launchbury, J. (eds.) PADL 2011. LNCS, vol. 6539, pp. 235–249. Springer, Heidelberg (2011)
24. Wadler, P.: Is there a use for linear logic? In: Symposium on Partial Evaluation and Semantics Based Program Manipulation, pp. 255–273. ACM/IFIP (1991)