Wormholes: Introducing Effects to FRP

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

Paul Hudak
Yale University
paul.hudak@yale.edu

Abstract

Functional reactive programming (FRP) is a useful model for programming real-time and reactive systems in which one defines a *signal function* to process a stream of input values into a stream of output values. However, performing side effects (e.g. memory mutation or input/output) in this model is tricky and typically unsafe. In previous work, Winograd-Cort et al. [2012] introduced *resource types* and *wormholes* to address this problem.

This paper better motivates, expands upon, and formalizes the notion of a wormhole to fully unlock its potential. We show, for example, that wormholes can be used to define the concept of causality. This in turn allows us to provide behaviors such as looping, a core component of most languages, without building it directly into the language. We also improve upon our previous design by making wormholes less verbose and easier to use.

To formalize the notion of a wormhole, we define an extension to the simply typed lambda calculus, complete with typing rules and operational semantics. In addition, we present a new form of semantic transition that we call a *temporal* transition to specify how an FRP program behaves over time and to allow us to better reason about causality. As our model is designed for a Haskell implementation, the semantics are lazy. Finally, with the language defined, we prove that our wormholes indeed allow side effects to be performed safely in an FRP framework.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics; D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages

General Terms Design, Languages

Keywords Functional Reactive Programming, Arrows, Resource Types, Stream Processing, Side Effects, Causality

1. Introduction

Functional reactive programming (FRP) is based on the notion of a *signal*, i.e. a time-varying value. Although signals are invariably represented as streams of data, FRP allows one to think of them as having instantaneous values for any given moment in time, and to think of programs as running to completion on each of those values in an infinitesimal period of time.[1] In practice, because computers cannot process instantaneously, this is typically implemented as a loop that proceeds at a given or variable clock rate, whose mechanics are an abstraction of the language. In this way, FRP programs are similar to circuit, or signal processing, diagrams, thus facillitating reasoning about program behavior.

However, standard FRP systems (such as Fran [Elliott and Hudak 1997]) lend themselves far too easily to space and time leaks [Liu and Hudak 2007]. One can address these leaks by using an *arrow-based* [Hughes 2000] design such as used in *Yampa* [Hudak et al. 2003; Courtney et al. 2003] (which has been used for animation, robotics, GUI design, and more), *Nettle* [Voellmy and Hudak 2011] (for networking), and *Euterpea* [Hudak 2011] (for audio processing and sound synthesis). Instead of treating signals as first class values, the *signal function* becomes the core component. By using arrows, one can compose and manipulate signal functions fairly easily.

An arrow-based FRP program is still a pure functional program. That is, the signal-based computations are performed using pure functions, and the input and output of the program—which may include I/O commands—are handled separately, i.e. outside of the program. In this sense, there is an *I/O bottleneck* on either end of the signal function that represents a complete program. All of the input data must be separated from its source so that it can be fed purely into the appropriate signal function, and all of the output data must be separately piped to the proper output devices. We see this as an imperfect system, as ideally the sources and sinks would be directly connected to their data.

1.1 Background and Motivation

A purely functional language does not admit side effects. Indeed, the original Haskell Report (Version 1.0) released in 1990, as well as the more widely publicized Version 1.2 [Hudak et al. 1992] specified a pure language, and the I/O system was defined in terms of both streams and continuations, which are equivalent (one can be defined straightforwardly in terms of the other). In 1989 the use of monads to capture abstract computations was suggested by Moggi [1989], subsequently introduced into Haskell by Wadler [1992], and further popularized by Peyton Jones and Wadler [1993].

Originally conceived as a pure algebraic structure, and captured elegantly using Haskell's type classes, it was soon realized that monads could be used for I/O and other kinds of side effects. Indeed, Version 1.3 of Haskell, released in 1996, specifies a monadic I/O system. The inherent data dependencies induced by the operators in the monad type class provide a way to sequence I/O actions in a predictable, deterministic manner (often called "single threaded"). The Haskell I/O monad is simply named *IO*, and primitive I/O operations are defined with this monadic type to allow

[1] This is consistent with the family of *synchronous* languages such as Lustre [Caspi et al. 1987], Esterel [Berry and Cosserat 1984], and Signal [Gautier et al. 1987]

Haskell'12, September 13, 2012, Copenhagen, Denmark.

essentially any kind of I/O. A monadic action that returns a value of type *a* has type *IO a*.

To make this approach sound, a program engaged in I/O must have type $IO\ ()$, and there can be no function, say $runIO :: IO\ a \to a$, that allows one to "escape" from the I/O monad. It's easy to see why this would be unsound. Consider the expression:

$$runIO\ m_1 + runIO\ m_2$$

If both m_1 and m_2 produce I/O actions, then it is not clear in which order the I/O actions will occur, since a pure language does not normally express an order of evaluation for $(+)$, and in general we would like $(+)$ to be commutative.

I/O is, of course, just one form of effect. For example, one might want to have mutable arrays (meaning that updates can be done "in-place" in constant time). A purely functional approach cannot provide constant-time performance for both reads and writes. Haskell has two solutions to this problem: First, Haskell defines an *IOArray* that can be allocated and manipulated in an imperative style. Predefined operations on the array are defined in terms of the I/O monad, and thus manipulating a mutable array becomes part of the single-threaded flow of control induced by the *IO* monad, as discussed earlier.

A problem with this approach is that it is common to want to define some local computation using an array and hide the details of how the array is implemented. Requiring that each such local computation inject the array allocation and subsequent mutations into the global I/O stream is thus not modular, and seems unnatural and restrictive.

What we would like is a monad within which we can allocate and manipulate mutable arrays (and not perform any I/O), and then "escape" from that monad with some desired result. Haskell's *ST* monad [Launchbury and Peyton Jones 1994] does just that. Haskell further defines a type constructor *STArray* that can be used to define arrays that can be allocated and manipulated just like an *IOArray*. Once the programmer is done with the local computation, the *ST* monad can be escaped using the function:

$$runST :: (\mathsf{forall}\ s.\ ST\ s\ a) \to a$$

The "trick" that makes this sound is the use of the existential (phantom) type variable *s* within the *ST* monad and the operations defined on the arrays. For example, returning the value of an array reference would be unsound—it would mean that the mutable array could be further mutated in other contexts, with potentially unpredictable results. However, this is not possible in Haskell's *ST* monad, because the type of the array reference contains the hidden existential type, thus resulting in a type error.

1.2 Effects in FRP

Monads can be used for many pure computations as well as other kinds of effects, but the above has focused on two kinds of effects: I/O and mutable data structures. It is important to distinguish these two, since there are inherent differences: I/O devices are generally fixed—each printer, monitor, mouse, database, MIDI device, and so on, is a unique physical device—and they cannot be created on the fly. With a mutable data structure, the situation is different: such data structures can be created on the fly and allocated dynamically as required by the program. It is also worth noting that for both I/O devices and mutable data structures, the sequence of actions performed on each of them must generally be ordered, as it would be in an imperative language, but conceptually, at least, actions on a printer, a MIDI device, or some number of separately allocated mutable data structures, could be performed concurrently.

So the question now is, how do we introduce these kinds of effects into FRP? Indeed, do these kinds of effects even make sense in an FRP language? Allowing side effects directly in a signal function has been explored (as in FrTime [Cooper and Krishnamurthi 2006]), but results in an imperative, impure design (equational reasoning is lost).

A normal Haskell variable is time-invariant, meaning that its value in a particular lexical context and in a particular invocation of a function that contains it, is fixed. In a language based on FRP, a variable is conceptually time-varying—its value in a particular lexical context and in a particular invocation of a function that contains it, is not fixed, but rather depends on the time.

A key insight for our work is that the sequencing provided by a monad can be achieved in FRP by using the ordering of events in an event stream. In the case of I/O, another key insight is that each of the I/O devices can be viewed as a signal function that is a "virtualized" version of that device. To guarantee soundness, *resource types* can be defined that guarantee uniqueness, as Winograd-Cort et al. [2012] described. Resource types assure that an FRP program remains deterministic despite I/O effects by restricting the access of any given real-world device to only one point in the program. For example, the keyboard could be represented as a signal function that produced keystroke events. Any given keystroke should only produce a single event, but if this signal function were used in multiple places in the program, each instance might produce a distinct event. Therefore, the signal function itself would be tagged with a *Keyboard* resource type, and if a programmer attempted to use it more than once in the same program, the program would produce a type error.

In the case of mutable data structures, a similar approach can be taken. For example, we could define a function:

$$sfArray :: Size \to \mathsf{SF}\ (Event\ Request)\ (Event\ Response)$$

such that *sfArray n* is a signal function encapsulating a mutable array of size *n*. (*SF a b* is the type of signal function whose input is a signal carrying values of type *a*, and whose output is a signal carrying values of type *b*.) That signal function would take as input a stream of *Request* events (such as read or write) and return a stream of *Response* events (such as the value returned by a read, acknowledgement of a successful write, or an index-out-of-bounds error). Note the similarity of this approach to the original stream I/O design in early Haskell [Hudak et al. 1992].

This design is also analogous to the *STArray* design, in that in-place updates of the array are possible in a sound way, and every invocation of *sfArray* creates a new mutable array. However, no changes to the type system are required to ensure soundness (in particular, no hidden existential types are needed, nor are resource types). Using this idea, many kinds of mutable data structures are possible, as well as, for example, a random number generator. (Winograd-Cort et al. [2012] described a random number generator that was resource typed, but in fact, as with the mutable array above, no resource types are needed to ensure soundness. Every invocation of a suitably defined random number generator will create a fresh stream of random numbers.)

1.3 Wormholes

Can we do more? What other kinds of effects might be desired for FRP? The remainder of this paper focuses on the notion of a *wormhole*, which can be viewed in two ways: (1) as a non-local one-way communication channel through which one can transfer signal values from one part of a program to another, in a completely safe manner, or (2) a mutable variable that can be written to and read from independently. By analogy, wormholes are a bit like *MVAR*s in Haskell, but in the FRP framework, the details are considerably different. The main insight is that to have such a feature in FRP, we need to separate the reads from the writes. Thus a wormhole consists of *two* signal functions, one for reading, and one for writing. We refer to these as the *whitehole* and *blackhole*, respectively. To

make this approach sound, resource types are used to ensure that each whitehole and blackhole is used just once.

Wormholes will be discussed in much more detail later, but here is a simple example of their use. Suppose *wh* and *bh* are the signal functions for the whitehole and blackhole, respectively, of a wormhole. Assuming they are executing in the same arrow and lexical scope, the following two signal functions communicate to each other non-locally through the wormhole:

$$
\begin{aligned}
sf_1 = \ & \textbf{proc}\ () \to \textbf{do} \\
& \ldots \text{ -- create some local data} \\
& \_ \leftarrow bh \prec localData \\
& \ldots \\
& returnA \prec 7
\end{aligned}
$$

$$
\begin{aligned}
sf_2 = \ & \textbf{proc}\ () \to \textbf{do} \\
& \ldots \\
& dataFromSF_1 \leftarrow wh \prec () \\
& \ldots \\
& returnA \prec 42
\end{aligned}
$$

Note that the data does not affect the signal function types—the data passes through the wormhole as if by magic. If one did not have wormholes, one would have to add the type of *localData* to the output of sf_1 and input of sf_2, and in a larger context ensure that the two were composed properly. This example is a bit contrived, and it is not advisable to program in this style all the time—types are useful, and one of the hallmarks of functional programming—but one can imagine using this technique when debugging, for example.

The astute reader will note that this approach is seemingly unsound—what if *bh* is used by some other part of the program, thus creating write conflicts? The answer is that resource types are used to ensure that this does not happen.

1.4 Contributions

In previous work, Winograd-Cort et al. [2012] sketched the idea of a wormhole as an alternative method for general kinds of effects. In this paper we expand on these ideas significantly, as described in this section.

Our first contribution is recognizing that the order of execution of a wormhole affects program behavior. One could allow the read and write from a wormhole to happen in either order, but this allows two nearly identical programs to potentially have very different behaviors. We show that restricting wormholes such that the read always happens before the write allows sounder reasoning as well as introduces a new possibility for control flow. Intuitively, regardless of the structure of a program, we want the read to be immediate while the write takes place "between" time steps. In this way, we can be sure that any data read from a wormhole was generated in the previous time step, allowing us to use wormholes to create causal connections.

In fact, our second contribution is to show a connection between wormholes and causal commutative arrows (CCA) [Liu et al. 2011]. In FRP applications, looping is achieved by feeding the output of a signal function back into the input. When expressed in CCA an extra restriction is placed on the fedback data: it must be from the past. This idea of causal, or temporal, looping fits well into our model, obviating the need for a primitive operator for looping. Indeed, causal loops are a higher level construction in our language rather than a core requirement.

Our third contribution is a formal specification and semantics for a lazy, resource-typed FRP based on the Haskell [Peyton Jones et al. 2003] implementation that Winograd-Cort et al. [2012] presented. We show that since FRP programs act over time, the transitions that govern their semantics should have a temporal component. Thus, we define program execution as an infinite trace through a "temporal" transition. The input and output of the program is handled through the resources, which are represented as streams built into the environment that resource types allow us to track. The key to this model's success is the subtle interconnection of the components: the temporal transition is meaningless without the resources to represent side effects, and the resources' inherent real-world quality makes reasoning temporally a necessity. With the semantics well specified, we are able to substantiate our previous work's claims of side effect safety.

Finally, in addition to formalizing the semantics, we improve upon the original design. In previous work, not only did each wormhole need to be defined at the top level of a program, but for every wormhole that a program used, the program's type would bloat with more terms. By slightly modifying the way wormholes are introduced, we are able to remove both of these restrictions without any significant loss in usability.

In the next section we introduce our core language features, which we follow up with a formal definition of the language in Section 3. Section 4 illustrates some examples of the various ways wormholes can be used, and Section 5 describes our work in implementing wormholes in Haskell. In Section 6 we describe the operational semantics and in Section 7 we prove that our language properly implements the desired features. Finally, Sections 8 and 9 discuss our conclusions and related work.

2. Language Features

In the introduction we described the basis for our language: an arrow-based implementation of functional reactive programming that uses wormholes and resource types to handle side effects. Here we discuss in more detail what these terms mean.

2.1 Signal Processing

The easiest way to conceptualize arrow-based FRP is to think of it as a language for expressing *signal processing diagrams*. The lines in these diagrams can be thought of as *signals*, and the boxes, that act on those signals, as *signal functions*. In general, the signals should be thought of as continuous, time-varying quantities, although they can also represent streams of events.

Haskell is an excellent language to consider coding with arrow-based FRP due to its *arrow syntax* [Paterson 2001]. For example, the following is a simple signal processing diagram that has two signals, an input *x* and an output *y*, as well as one signal function, *sigfun*.

In Haskell this diagram would be coded as:

$$y \leftarrow sigfun \prec x$$

This code fragment cannot appear alone, but instead must be part of a **proc** construct. The expression in the middle must be a signal function, whose type we write as $T_1 \rightsquigarrow T_2$ for some types T_1 and T_2. The expression on the right may be any well-typed expression with type T_1, and the expression on the left must be a variable or pattern of type T_2.

The purpose of the arrow notation is to allow the programmer to manipulate the instantaneous values of the signals. For example, the following is a definition for *sigfun* that integrates a signal and

$$
\begin{array}{ll}
\textbf{arr} & :: (a \to b) \to (a \rightsquigarrow b) \\
\textbf{first} & :: (a \rightsquigarrow b) \to ((a \times c) \rightsquigarrow (b \times c)) \\
(\ggg) & :: (a \rightsquigarrow b) \to (b \rightsquigarrow c) \to (a \rightsquigarrow c) \\
\textbf{loop} & :: ((c \times a) \rightsquigarrow (c \times b)) \to (a \rightsquigarrow b) \\
(|||) & :: (a \rightsquigarrow c) \to (b \rightsquigarrow c) \to ((a + b) \rightsquigarrow c) \\
\textbf{app} & :: ((a \rightsquigarrow b) \times a) \rightsquigarrow b
\end{array}
$$

Figure 1. The types of the arrow operators.

multiplies the output by two:

$$
\begin{array}{l}
\mathit{sigfun} :: \mathit{Double} \rightsquigarrow \mathit{Double} \\
\mathit{sigfun} = \textbf{proc}\ x \to \textbf{do} \\
\quad y \leftarrow \mathit{integral} \prec x \\
\quad \mathit{returnA} \prec y * 2
\end{array}
$$

The first line gives the type of *sigfun*, a signal function that converts a stream of type *Double* into a stream of type *Double*. The notation "**proc** $x \to$ **do** ..." introduces a signal function, binding the name x to the instantaneous values of the input. The third line sends the input signal into an integrator, whose output is named y. Finally, we multiply the value y by two and feed it into a special signal function, *returnA*, that returns the result.

Of course, one can use arrows without Haskell's arrow syntax. Arrows are made up of three basic operators: construction (**arr**), partial application (**first**), and composition ($\ggg$). Furthermore, arrows can be extended with looping (**loop**) [Paterson 2001], choice ($|||$) [Hughes 2000], and application (**app**) [Hughes 2000]. The types of these operators are shown in Figure 1. To simplify the discussion, we omit further details about looping and choice, other than their typing rules given in Section 3.3.

For example, the signal function *sigfun* defined earlier can be written without arrow syntax as follows:

$$\mathit{sigfun} = \mathit{integral} \ggg \textbf{arr}(\lambda x.x * 2)$$

2.2 Resource Tracking and Management

When signal functions become effectful, an insidious problem develops. We want to think of signal functions as ordinary, pure functions, and as such, we should have the power to duplicate them at will. However, if the signal functions can perform side effects, then they may not behave properly when duplicated. Consider, for example, a signal function to play sound in real time:

$$\mathit{playSound} :: \mathit{SoundData} \rightsquigarrow ()$$

playSound takes a stream of *SoundData*, plays it to the computer's speakers, and returns unit values. Now consider the following code snippet in arrow syntax:

$$
\begin{array}{l}
\_ \leftarrow \mathit{playSound} \prec \mathit{sound}_1 \\
\_ \leftarrow \mathit{playSound} \prec \mathit{sound}_2
\end{array}
$$

We intend for *playSound* to represent a single real-world device, but here we have two occurrences—what is the effect? Are the sounds somehow merged together? Is one sound stream ignored? A similar situation can be constructed for input where the input device provides a stream of events to multiple listeners. If a new event appears, should all listeners receive a copy of it or just one, and if only one, which?

In previous work, Winograd-Cort et al. [2012] proposed *resource types* as a solution to this problem. By adding a phantom type parameter to each signal function, we were able to represent what resources that signal function accesses. This set of resources is then statically checked whenever two signal functions are composed—if the sets of resources of the two signal functions are not disjoint, then the composition results in a type error. Adding resource types to our previous example yields this type for *playSound*:

$$\mathit{playSound} :: \mathit{SoundData} \overset{\{\mathit{Speakers}\}}{\rightsquigarrow} ()$$

With this type, the code snippet does not type-check. We discuss the typing rules in more detail in Section 3.

2.3 Wormholes

In addition to having resource types that represent physical resources, we can have resource types that represent arbitrary side effects. Notably, we can consider using resource types to represent mutable memory.

In particular, we can create a *wormhole* as a reference in memory that comes with two fresh virtual resources, one for the input end and one for the output end, which we affectionately refer to as the *blackhole* and *whitehole*, respectively.[2] We access the ends of the wormhole in the same manner that we might access any real resource, and the same machinery that makes resource types work for real resources makes mutation and direct memory access safe.

2.4 Causality

Functional reactive programming itself does not need to be causal. That is, values along a signal can, in fact, depend on future values. Of course, in real-time systems, causality is forced to be preserved by the nature of the universe. For example, a program's current output cannot depend on a user's future input. Thus, in the world of effectful FRP, we limit ourselves to causal signal functions.

The main impact of this limitation has to do with fixed points and looping in the signal function domain. We restrict signal functions so that they cannot perform limitless recursion without moving forward in time. That is, all loops must contain some sort of delay such that the input only depends on past outputs. We call this *strictly causal looping*.

Liu et al. [2011] introduced the **init** operator as an abstract form of causal computation:

$$\textbf{init} :: a \to (a \rightsquigarrow a)$$

Technically, the current output of **init** i can depend on the current and previous inputs; however, the typical definition is as a delay operator, and as such, the current output would depend on only the previous inputs. Used in tandem with the arrow **loop** operator from Figure 1, one can define strictly causal loops. We offer just that:

$$\textbf{dLoop} :: c \to ((c \times a) \rightsquigarrow (c \times b)) \to (a \rightsquigarrow b)$$

The **dLoop** operator takes an initial value for the looping parameter, which will update in time but always be slightly delayed. Notice that when **dLoop** is given the simple swapping function $(\lambda(x,y).(y,x))$ as its second argument, it reduces to an instance of the **init** function acting as a unit delay.

3. The Formal Language

We specify our language in a similar manner to Lindley et al. [2010]. We start with the lambda calculus extended with a product type and general recursion, which when necessary, we will refer to as $\mathscr{L}\{\to\times\}$. We show the abstract syntax for this language in Figure 2. We let τs range over types, vs over variable names, es over expressions, and Γs over environments. A type judgment $\Gamma \vdash e : \tau$ indicates that that it follows from the mappings in the environment Γ that expression e has type τ. Sums, products, and functions satisfy β- and η-laws. This is a well established language, so rather

[2] This is a reference to the theoretical astronomical oddity, the "Einstein-Rosen bridge," a one-directional path through space-time such that matter can only flow in through the black hole and out through the white hole.

Typ	τ	::=	$()$	unit
		$\mid$	$\tau_1 \times \tau_2$	binary product
		$\mid$	$\tau_1 \to \tau_2$	function
Var	v			
Exp	e	::=	v	variable
		$\mid$	(e_1, e_2)	pair
		$\mid$	**fst** e	left-pair projection
		$\mid$	**snd** e	right-pair projection
		$\mid$	$\lambda v.e$	abstraction
		$\mid$	$e_1\ e_2$	application
Env	Γ	::=	$v_1 : \tau_1, ..., v_n : \tau_n$	type environment

Figure 2. The abstract syntax of $\mathscr{L}\{\to\times\}$.

than repeat the typing rules, it suffices to say that they are as expected. We also borrow an expected operational semantics that utilizes lazy evaluation.

From there, we add the type for resource-typed, arrow-based signal functions, and we add expressions for the three standard operators for them (**arr**, **first**, and $\ggg$). In the process, we also add resources as a new component to the language, complete with a resource type, resource operators, and a resource environment. Finally, we connect the resources to the expressions with a form of resource interaction (**rsf**), and we provide an operator for creating new virtual resources (**wormhole**).

We show our extension to the abstract syntax in Figure 3 and the typing rules for resources and resource operators in Figure 4 and for newly added expressions in Figure 5. In addition to the previous syntax, we let rs range over resources, ts over resource types, ρs over resource operators, and $\mathscr{R}$s over resource environments. A type judgment $\mathscr{R} \vdash r : t$ indicates that resource environment $\mathscr{R}$ contains an entry mapping resource r to resource type t. Typically, we will combine judgments to the form $\Gamma, \mathscr{R} \vdash ...$ indicating that both environments may be used.

Lastly, we make the following definition of programs that our language supports at the top level:

Definition 1. *An expression p is a **program** if it has type $() \overset{R}{\rightsquigarrow} ()$ for some set of resources R.*

This restriction is actually rather minor. As our language is defined for FRP, it is reasonable to require that the expression being run is a signal function. Furthermore, as all input and output for a program should be handled through resources, the input and output streams of a program need not contain any information.

3.1 Resources and Resource Operators

Resources should be thought of as infinite streams of data that correspond with real world objects. The default resource environment, $\mathscr{R}_o$, is essentially the real world (i.e. user and outside data interaction) split up into discrete, quantized pieces, but new "virtual" resources can be added to resource environments via wormholes.

In our language, resources are basically "black boxes". We can interact with them via the resource operators (**put** and **next**), but as they represent external interaction, we do not examine them more closely. Resources each have a type of the form $\langle \tau_{in}, \tau_{out} \rangle$ that indicates that the resource accepts expressions of type τ_{in} and produces expressions of type τ_{out}.

Resource operators are functions that take a resource and interact with it in some way. They are distinctly not expressions and are not used by expressions, but they are necessary for defining resource interaction in the operational semantics. The two operators we introduce are for examining a resource's current state and for updating the resource. The typing rules for resources and their operators are shown in Figure 4.

Res	r			
ROp	ρ	::=	**next**(r)	query resource
		$\mid$	**put**(r, e)	set resource
RTp	t	::=	$\langle \tau_{in}, \tau_{out} \rangle$	resource type
Typ	τ	::=	...	
		$\mid$	$\tau_1 \overset{\{r_1,...\}}{\rightsquigarrow} \tau_2$	resource typed SF
Exp	e	::=	...	
		$\mid$	**arr**(e)	SF construction
		$\mid$	**first**(e)	SF partial application
		$\mid$	$e_1 \ggg e_2$	SF composition
		$\mid$	**rsf**$[r]$	SF resource interaction
		$\mid$	**wormhole**$[r_w, r_b](e_i;\ e)$	wormhole introduction
REn	$\mathscr{R}$	::=	$r_1 : t_1, ..., r_n : t_n$	resource environment

Figure 3. The abstract syntax additions to $\mathscr{L}\{\to\times\}$ that describe our language.

$$\textsc{Ty-Res}\ \frac{}{\Gamma, \mathscr{R}(r : t) \vdash r : t}$$

$$\textsc{Ty-R-Next}\ \frac{}{\Gamma, \mathscr{R}(r : \langle \_, \tau \rangle) \vdash \mathbf{next}(r) : \tau}$$

$$\textsc{Ty-R-Put}\ \frac{\Gamma, \mathscr{R} \vdash e : \tau_{in}}{\Gamma, \mathscr{R}(r : \langle \tau_{in}, \tau_{out} \rangle) \vdash \mathbf{put}(r, e) : \langle \tau_{in}, \tau_{out} \rangle}$$

Figure 4. The typing rules for resources and resource operators.

The Ty-R-Next rule shows that the **next** value from a resource has the same type as the resource's output type. The Ty-R-Put rule says that an expression e can be **put** into a resource if it matches the input type of the resource, and the result is a resource of the same type as the original resource.

Resources are used in the language at both the type level and the expression level. At the type level, resources are associated with the signal functions that use them. Specifically, they are included in the set of resources that is part of the type of signal functions.

At the expression level, resources can be accessed for input and output via the **rsf** expression. Given a resource, it essentially lifts the resource into a signal function. The input type of the signal function is the input type of the resource, and the output type is similarly the output type of the resource. Furthermore, the signal function is tagged with the given resource at the type level. All resource interaction, and thus all I/O, is done via **rsf** expressions.

New virtual resources are created by **wormhole** expressions. A **wormhole** expression takes an initial value to be contained in the wormhole's memory, and produces two fresh virtual resources representing either end of it. In practice, it works similarly to how one might use a **let** expression in another language—the **wormhole** expression takes two names for the fresh resources as well as an expression in which those resources are available. Note that although **wormhole** expressions do take two names for the resources they produce, the resources are guaranteed to always be fresh even if there are naming conflicts. That is, typical scoping rules apply.

The purpose of resources is to track I/O; therefore, despite the fact that they are "usable" at the expression level, we do not want them to escape through an abstraction and so we do not even allow them as first class values.

$$\textsc{Ty-Arr}\ \frac{\Gamma,\mathscr{R} \vdash e : \alpha \to \beta}{\Gamma,\mathscr{R} \vdash \mathbf{arr}(e) : \alpha \overset{\emptyset}{\rightsquigarrow} \beta}$$

$$\textsc{Ty-First}\ \frac{\Gamma,\mathscr{R} \vdash e : \alpha \overset{R}{\rightsquigarrow} \beta}{\Gamma,\mathscr{R} \vdash \mathbf{first}(e) : (\alpha \times \gamma) \overset{R}{\rightsquigarrow} (\beta \times \gamma)}$$

$$\textsc{Ty-Comp}\ \frac{\Gamma,\mathscr{R} \vdash e_1 : \alpha \overset{R_1}{\rightsquigarrow} \beta \quad \Gamma,\mathscr{R} \vdash e_2 : \beta \overset{R_2}{\rightsquigarrow} \gamma \quad R_1 \cup R_2 = R \quad R_1 \cap R_2 = \emptyset}{\Gamma,\mathscr{R} \vdash e_1 \ggg e_2 : \alpha \overset{R}{\rightsquigarrow} \gamma}$$

$$\textsc{Ty-RSF}\ \frac{}{\Gamma,\mathscr{R}(r : \langle \tau_{in}, \tau_{out} \rangle) \vdash \mathbf{rsf}[r] : \tau_{in} \overset{\{r\}}{\rightsquigarrow} \tau_{out}}$$

$$\textsc{Ty-WH}\ \frac{\Gamma,\mathscr{R}(r_w : \langle (), \tau \rangle, r_b : \langle \tau, () \rangle) \vdash e : \alpha \overset{R'}{\rightsquigarrow} \beta \quad \Gamma,\mathscr{R} \vdash e_i : \tau \quad R = R' \setminus \{r_w, r_b\}}{\Gamma,\mathscr{R} \vdash \mathbf{wormhole}[r_w, r_b](e_i;\ e) : \alpha \overset{R}{\rightsquigarrow} \beta}$$

Figure 5. The typing rules for the new expressions of our language.

3.2 Signal Function Expressions

Here, we examine each of the typing rules for new expressions we have added to the language (shown in Figure 5):

- The Ty-Arr rule states that the set of resource types for a pure function lifted to a signal function is empty.
- The Ty-First rule states that transforming a signal function using **first** does not alter the resource type.
- The Ty-Comp rule 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.
- The Ty-RSF rule is for resource interaction. It says that the input and output types of the signal function that interacts with a given resource must match the input and output types given by the form of the resource. Furthermore, the signal function created will have the singleton resource type set containing the used resource.
- The Ty-WH rule is for wormhole introduction. It says that the body of the wormhole is a signal function provided that two resources are added to $\mathscr{R}$: one of the form $\langle (), \tau \rangle$ (the whitehole) and one of the form $\langle \tau, () \rangle$ (the blackhole) where τ is the type of the initializing expression. The result of the whole expression is the same as that of the body except that the resources r_w and r_b are removed from the resource set. This omission is valid because the virtual resources cannot escape the wormhole expression.[3]

A more complete analysis of the reasoning for these typing rules is covered by Winograd-Cort et al. [2012].

3.3 Choice and Application

In Section 2.1, we mentioned the arrow extensions for choice and application. They have little impact on the focus of this paper, so we omit them from the language for simplicity. However, it is worth mentioning that our language has no problem with them and can fully support them. Therefore, we provide their typing rules to demonstrate how they function in the presence of resource types:

[3] This is similar to a trick used in Haskell to hide monadic effects by using the universal type quantifier forall to constrain the scope. Here, the resources are only available inside the body of the wormhole.

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

$$\textsc{Ty-App}\ \frac{}{\Gamma,\mathscr{R} \vdash \mathbf{app} : ((\alpha \overset{R}{\rightsquigarrow} \beta) \times \alpha) \overset{R}{\rightsquigarrow} \beta}$$

The Ty-Chc rule is for the choice operator. When choosing, we can be certain that only one branch will be chosen, so the resulting resource type set is the union of those of its inputs, which are not required to be disjoint. The Ty-App rule, for the application operator, allows for arbitrary evaluation of signal functions, but it is restricted such that those signal functions must all have the same resource types.

4. Examples

We have introduced wormholes as a means to achieve side effects and non-local communication in FRP programs. The usefulness of these concepts can be best demonstrated with a few examples.

4.1 Loops

One may wonder at the absense of looping in our language. We mentioned in Section 2.4 that we would not adhere to the standard arrow loop, but our language has no built-in delay loop either.

We start by showing that a strictly causal implementation of **init** (also mentioned in Section 2.4) can be produced as syntactic sugar with a wormhole:

$$\textsc{Ty-Init}\ \frac{\Gamma,\mathscr{R} \vdash e_i : \alpha}{\Gamma,\mathscr{R} \vdash \mathbf{init}\ e_i : \alpha \overset{\emptyset}{\rightsquigarrow} \alpha}$$

$$\mathbf{init}\ i \overset{\mathbf{def}}{=} \mathbf{wormhole}[r_w, r_b](i; \mathbf{rsf}[r_b] \ggg \mathbf{rsf}[r_w])$$

By attaching the blackhole and whitehole of a wormhole back to back, we create a signal function that accepts present input and returns output delayed by one step. Essentially, we see that the **init** operator is the connection of two ends of a wormhole.

Interestingly, we can attach the wormhole ends the other way too. Obviously, this can lead to a trivial signal function of type $() \overset{\emptyset}{\rightsquigarrow} ()$ that does nothing, but if we provide a signal function to be run in between the connection, we can build the following:

$$\textsc{Ty-dLoop}\ \frac{\Gamma,\mathscr{R} \vdash e_i : \gamma \quad \Gamma,\mathscr{R} \vdash e : (\gamma \times \alpha) \overset{R}{\rightsquigarrow} (\gamma \times \beta)}{\Gamma,\mathscr{R} \vdash \mathbf{dLoop}(e_i; e) : \alpha \overset{R}{\rightsquigarrow} \beta}$$

$$\begin{aligned}\mathbf{dLoop}(i; e) \overset{\mathbf{def}}{=} \mathbf{wormhole}[r_w, r_b](i;\ & \\ & \mathbf{arr}(\lambda x.\ ((), x)) \ggg \mathbf{first}(\mathbf{rsf}[r_w]) \ggg e \\ & \ggg \mathbf{first}(\mathbf{rsf}[r_b]) \ggg \mathbf{arr}(\lambda (\_, x).\ x))\end{aligned}$$

We are able to achieve delay looping by a clever use of a wormhole. We first produce a new wormhole and provide the loop's initialization value as its initial value. The **arr** and **first** commands together arrange the input so that the wormhole's whitehole output is paired with the external input just as e is expecting. After that input is processed by e, the resultant loop argument is fed into the wormhole's blackhole, and the output value is returned. Due to the causal behavior of wormholes, values that are output from e become new input values to e on the next iteration. Thus, the input on the n^{th} iteration is given by the output on the $n-1^{\text{st}}$ iteration.

In fact, even a built-in delay loop would not be able to perform better. The above loop delays by exactly one iteration. If it were any less delayed, we would no longer satisfy our strict causality requirement.

4.2 Data transfer

One strength of wormholes is their ability to transfer data between two disparate parts of a program. Typically, this would involve rewriting signal functions so that they consume or produce more streams so that one can create a stream link between the two components to be connected. However, this work is unnecessary with wormholes.

First, we will assume that our language is extended with an *Integer* data type; this will help us keep track of the data moving through the wormhole. Next, we wil consider the following two programs:

$$P_1 \ : \ R_1' \subseteq R_1 \Rightarrow (\mathit{Integer} \overset{R_1'}{\rightsquigarrow} \mathit{Integer}) \rightarrow (() \overset{R_1}{\rightsquigarrow} ())$$
$$P_2 \ : \ R_2' \subseteq R_2 \Rightarrow (\mathit{Integer} \overset{R_2'}{\rightsquigarrow} \mathit{Integer}) \rightarrow (() \overset{R_2}{\rightsquigarrow} ())$$

We will assume that as long as R_1' and R_2' are disjoint, then R_1 and R_2 are disjoint also. These two programs both do almost the same thing: they acquire a stream of *Integer*s from a source, apply a given signal function to them, and then send the result to an output device.

Our goal is to connect these two programs in order to cross their streams. That is, we would like the stream from P_1 to go to the output device of P_2 and vice versa. Without wormholes, we would be forced to examine and change the implementation and type of at least one of these two programs. However, instead, we can define:

$$\begin{aligned} \mathit{main} = \ & \mathbf{wormhole}[r_{w_1}, r_{b_1}](0; \\ & \quad \mathbf{wormhole}[r_{w_2}, r_{b_2}](0; \\ & \quad\quad P_1 \ (\mathbf{rsf}[r_{b_1}] \ggg \mathbf{rsf}[r_{w_2}]) \ggg \\ & \quad\quad P_2 \ (\mathbf{rsf}[r_{b_2}] \ggg \mathbf{rsf}[r_{w_1}]) \end{aligned}$$

We pair two wormholes together almost like two **init** expressions, except that we swap the inputs and outputs. This provides us with two functions that are able to communicate even when no streams seem readily available.

5. Wormholes in Haskell

Previously, Winograd-Cort et al. [2012] provided a working implementation of an arrow-based FRP system with wormholes that utilized resource types. First, we noticed that since the resources of a signal function are statically determinable, they should be implemented through Haskell's type system. Thus, we let each resource have an empty type associated with it, and we leveraged Haskell's complex data types, type classes with functional dependencies, and type families to interact with them. Thus, the type of a signal function is represented in Haskell by the three argument data type $\mathsf{SF}\ r\ a\ b$, which translates to $a \overset{r}{\rightsquigarrow} b$ in the abstract language of this paper.

Ideally, we would like a data type to encode sets at the type level, but we were unable to achieve this. Two identical sets can have different representations, and the type checker is unable to unify them. Fortunately, between work on heterogeneous lists [Kiselyov et al. 2004] and Haskell's new data kinds extension [Yorgey et al. 2012], type level lists are straightforward to implement. In fact, our previous work showed that unioning as well as the property of disjointness is implementable with heterogeneous lists. Where we previously used an un-kinded list, we have since updated to employ the standard notation for type level lists presented by the data kinds extension.

Our last step was to incorporate Haskell's *IO* monad directly into the signal function framework to allow side effects to be performed during signal function execution. Signal function resource interaction (the **rsf** operator in our language here) was achieved by programmer-level tagging of the appropriate resources along with the *pipe* (and *source* and *sink*) command.

Unfortunately, the Haskell implementation of our system is not as powerful and robust as the theory we have presented. Notably, wormholes are conspicuously absent from the previous section's implementation discussion, and in fact, we currently believe that a proper implementation may not be feasible without new extensions to Haskell's type system.

In previous work, wormholes were not as dynamic as we have presented them here. The programmer was required to declare all wormhole resources at the top level, and as such, only a finite, pre-determined number of wormholes could be made. Even then, wormholes could not be generated with a loop as each resource had to be attached manually.

One of our contributions in this paper is to show a better way to make wormholes: the type signature shown in Figure 5 allows new, unique resources can be created dynamically. For the implementation, we can use the same strategy of existential types as employed by the *ST* monad [Launchbury and Peyton Jones 1994]. Thus, the type for a function to make wormholes should be:

```
wormhole :: forall t r a b.
  t → (forall rw rb r'. SetDiff r' '[rw, rb] r ⇒
       SF '[rw] () t → SF '[rb] t () → SF r' a b)
  → SF r a b
```

The class *SetDiff xs ys zs* would have instances to define that the set *zs* would contain all of the elements of the set *xs* except those from the set *ys*.

The problem is that it does not seem possible to define the *SetDiff* type class for the same reason that type level sets cannot be constructed. In this case, there are multiple correct types for *r* when given r', and the type system is unable to properly unify.

One option is to find a canonical representation for our sets so that they can be reduced to lists. For instance, by associating each resource with a type-level number, we can require that a resource set is always sorted. In this way, there will be only one representation of any given type set, and the type checker will be able to unify two sets. However, as *wormhole* can be recursively called, there is no way to assign numbers to the existential wormhole resource types. Thus, we must restrict our functionality: sets become lists, unioning becomes concatentation, and set difference requires an order. In addition, the type of *wormhole* becomes:

```
wormhole :: forall t r a b.
  t → (forall rw rb.
       SF '[rw] () t → SF '[rb] t () → SF (rw' : rb' : r) a b)
  → SF r a b
```

With this definition, we are forced to have the wormhole resources ordered so that they come first, and because unioning, which happens during signal function composition, concatenates resource type sets, this applies many more restrictions to using wormholes than we intend.

6. Operational Semantics

The operational semantics for resource typed signal functions are somewhat complex, and in an effort to demystify them, we separate the functionality into three distinct transitions. At the highest level, we apply a temporal transition. This transition details how resources behave over time and explains how the signal function itself is "run". (Recall from Definition 1 that only expressions with type $() \overset{R}{\rightsquigarrow} ()$ are allowed as "runnable" programs.) Because our language is lazy and evaluation is performed when necessary, expressions may be able to simplify themselves over time. Therefore, this transition will return an updated (potentially more evaluated) version of the input program.

The temporal transition makes use of a functional transition to interpret the flow of data through the component signal functions

$$\text{ET-ARR}\ \frac{}{\mathbf{arr}(e)\ \mathsf{val}}$$

$$\text{ET-FIRST}\ \frac{}{\mathbf{first}(e)\ \mathsf{val}}$$

$$\text{ET-COMP}\ \frac{}{(e_1 \ggg e_2)\ \mathsf{val}}$$

$$\text{ET-RSF}\ \frac{}{\mathbf{rsf}[r]\ \mathsf{val}}$$

$$\text{ET-WH}\ \frac{}{(\mathbf{wormhole}[r_w, r_b](e_i;\ e))\ \mathsf{val}}$$

Figure 6. The evalution transition judgments for our extension to $\mathscr{L}\{\rightarrow\times\}$.

of the program at a given point in time. Thus, the judgments in the functional transition handle how the instantaneous values of the signals are processed by signal functions.

Because the expressions to be run can contain arbitrary lambda calculus, the functional transition judgments make use of an evaluation transition when necessary to evaluate expressions when strictness points are reached. This is a fairly simple transition that performs as a typical, lazy semantics of a lambda calculus.

A top-down view of the three transitions is the most intuitive way to describe their functionality. However, to define them, it is easier to start with the evaluation transition and work up from there. Therefore, we present the following transitions:

$$\begin{array}{rcll} e & \mapsto & e' & \text{Evaluation transition} \\ (\mathscr{V}, x, e) & \Rrightarrow & (\mathscr{V}', y, e', \mathscr{W}) & \text{Functional transition} \\ (\mathscr{R}, \mathscr{W}, P) & \overset{t}{\mapsto} & (\mathscr{R}', \mathscr{W}', P') & \text{Temporal transition} \end{array}$$

where

e and e'	are expressions
$\mathscr{V}$ and $\mathscr{V}'$	are sets of triples
x and y	are values
$\mathscr{W}$ and $\mathscr{W}'$	are sets of wormhole data
$\mathscr{R}$ and $\mathscr{R}'$	are resource environments, and
P and P'	are programs

In the following subsections, we discuss these transitions in more detail.

6.1 Evaluation transition

The evaluation transition is used to evaluate the non-streaming components of the language. In an effort to conserve space, we take as given the evaluation semantics for $\mathscr{L}\{\rightarrow\times\}$. That is, we assume a classic, lazy semantics for lambda expressions and application, product-type pairs and projection, and sum-type case analysis and injection. We show our additional rules for the five additional expressions of our language in Figure 6.

We use the notation $e\ \mathsf{val}$ to denote that expression e is a value and needs no further evaluation.

Obviously, these rules are very straightforward: no evaluation is done on signal functions in this transition. This transition is important for the operations of $\mathscr{L}\{\rightarrow\times\}$, but it is strictly a formality here.

The language $\mathscr{L}\{\rightarrow\times\}$ has a standard Canonical Forms Lemma associated with it that explains that for each type, there are only certain expressions that evaluate to a value of that type. By simple examination of these new rules to the transition, we can extend the lemma as follows:

Lemma 1 (Canonical Forms)**.** *If $e\ \mathsf{val}$ and $e : \alpha \overset{R}{\rightsquigarrow} \beta$, then e is either an SF constructor, an SF partial application, an SF composition, an SF resource interaction, or a wormhole introduction.*

6.2 Functional transition

The functional transition details how a signal function behaves when given a single step's worth of input. It is a core component of the temporal transition described in the next section as it essentially drives the signal function for an instant of time. The functional transition judgments are shown in Figure 7.

Before we discuss the judgments themselves, it is important to examine the components being used. First, one will notice the set $\mathscr{V}$. $\mathscr{V}$ represents the state of the resources (both real and virtual) in the world at the particular moment in time that this transition is taking place. Each element of $\mathscr{V}$ is actually a triple of a resource, the value that resource is providing at this moment, and the value to be returned to that resource. At the start, we assume that all of the elements have the form $(r, x, \cdot)$, which indicates that resource r provides the value x and has no value to receive. It should be no surprise that the only judgments that read from or modify this set are FT-RSF and FT-WH, the judgments for resource interaction and virtual resource creation.

The second argument to each of the judgments (typically x in Figure 7) represents the streaming value being piped into the signal function. However, since the functional transition is only defined for an instant of time, rather than this value being an actual stream, it is the instantaneous value on the stream at this time step. Its partner is the second result, or the instantaneous value of the streaming output of the input signal function.

The third argument is the expression being processed. The purpose of the functional transition is to describe how signal functions behave when given values from their streaming input, and as such, it is only defined for signal functions (that is, expressions that have the type $\alpha \overset{R}{\rightsquigarrow} \beta$ for some set R). Notably, there are only judgments corresponding to the forms given in the updated canonical forms lemma (Lemma 1). On the output end, this term represents the potentially further evaluated form of the input expression. We prove later in Theorem 2 that this output expression is functionally equivalent to the input one.

The first three terms of the output correspond to the three terms of the input, but there is also an additional term $\mathscr{W}$, which contains data about any wormholes processed during this transition. In addition to adding the two virtual resources created by a wormhole expression to the resource environment, we need to separately keep track of the fact that they are a pair. Therefore, $\mathscr{W}$ contains elements of the form $[r_b, r_w, e]$ where r_b is the name of the blackhole end of the wormhole, r_w is the name of the whitehole end, and e is the value in the wormhole. We will use this information later to properly update wormholes over time in the temporal transition.

Note also that we use the term $e \mapsto^* e'$ to denote continued application of the evaluation transition $\mapsto$ on e until it is evaluated to a value. That value is e'

As this is a critical piece of the overall semantics, we examine each of the judgments individually:

- The FT-ARR judgment does not touch the resources, so the input $\mathscr{V}$ is returned untouched in the output. The expression $e\ x$ does not need to be evaluated due to the lazy semantics, but it is the streaming output nonetheless. The final two outputs reveal that no further evaluation of the expression has been done and no wormhole data was created.
- The FT-FIRST judgment is only applicable when the input streaming value is a pair (which is assured by the type checker by using the TY-FIRST rule). The first element of the pair is recursively processed with the argument to **first**, and the output

$$\textsc{Ft-Arr}\ \frac{}{(\mathscr{V},x,\mathbf{arr}(e)) \Rrightarrow (\mathscr{V},e\ x,\mathbf{arr}(e),\emptyset)}$$

$$\textsc{Ft-First}\ \frac{e \mapsto^* e' \quad (\mathscr{V},x,e') \Rrightarrow (\mathscr{V}',y,e'',\mathscr{W})}{(\mathscr{V},(x,z),\mathbf{first}(e)) \Rrightarrow (\mathscr{V}',(y,z),\mathbf{first}(e''),\mathscr{W})}$$

$$\textsc{Ft-Comp}\ \frac{e_1 \mapsto^* e_1' \quad (\mathscr{V},x,e_1') \Rrightarrow (\mathscr{V}',y,e_1'',\mathscr{W}_1) \quad e_2 \mapsto^* e_2' \quad (\mathscr{V}',y,e_2') \Rrightarrow (\mathscr{V}'',z,e_2'',\mathscr{W}_2)}{(\mathscr{V},x,e_1 \ggg e_2) \Rrightarrow (\mathscr{V}'',z,e_1'' \ggg e_2'',\mathscr{W}_1 \cup \mathscr{W}_2)}$$

$$\textsc{Ft-RSF}\ \frac{}{(\mathscr{V} \cup \{(r,y,\cdot)\},x,\mathbf{rsf}[r]) \Rrightarrow (\mathscr{V} \cup \{(r,\cdot,x)\},y,\mathbf{rsf}[r],\emptyset)}$$

$$\textsc{Ft-WH}\ \frac{e \mapsto^* e' \quad (\mathscr{V} \cup \{(r_w,e_i,\cdot),(r_b,(),\cdot)\},x,e') \Rrightarrow (\mathscr{V}',y,e'',\mathscr{W})}{(\mathscr{V},x,\mathbf{wormhole}[r_w,r_b](e_i;\ e)) \Rrightarrow (\mathscr{V}',y,e'',\mathscr{W} \cup \{[r_b,r_w,e_i]\})}$$

Figure 7. The functional transition judgments.

is formed by the updated $\mathscr{V}'$ and by re-pairing the output y. As the body of the **first** expression, e, was evaluated, its updated form is returned along with any wormhole data the recursion generated.

- The FT-COMP judgment first sends the streaming argument x through e_1 recursively. Then, with the updated $\mathscr{V}'$, it sends the result y through e_2. The resulting $\mathscr{V}''$ and z are returned. Once again, the updated expression is returned in the output. Lastly, the wormhole data from both recursive calls of the transition are unioned together and returned.
- The FT-RSF judgment requires $\mathscr{V}$ to contain an element of the form $(r,y,\cdot)$, where r is the resource being accessed, y is the value the resource currently has, and no output has been sent to this resource yet. The streaming value x is put into the resource, and the result is the streaming value y from what was in the resource. The set $\mathscr{V}$ is updated, replacing the triple used here with a new one of the form $(r,\cdot,x')$ showing that this resource has essentially been "used up".
- The FT-WH judgment first evaluates its body e to the value e'. For its recursive call, it updates the set $\mathscr{V}$ with two new triples corresponding to the two new resources created in the wormhole operation: $(r_w,e_i,\cdot)$ and $(r_b,(),\cdot)$. These are two fresh, unused triples that **rsf** operators can make use of in the body e'. As triples are never removed, $\mathscr{V}'$ will include these two triples as well. The result is this $\mathscr{V}'$ with the new triples, the streaming value y, the updated body e'', and the wormhole data from the recursion updated with the element $[r_b,r_w,e_i]$ corresponding to this wormhole. Note that the returned expression is no longer a wormhole but has been replaced with the body of the wormhole. This is because now that this wormhole has been evaluated, its values live inside $\mathscr{V}$ and it has been cataloged in $\mathscr{W}$—it is no longer needed in the expression.

The following theorems provide some extra information about the overall functionality of this transition.

Theorem 1. *If* $(\mathscr{V},x,e) \Rrightarrow (\mathscr{V}',y,e',\mathscr{W})$, *then* $\forall(r,a,b) \in \mathscr{V}$, $\exists(r,a',b') \in \mathscr{V}'$ *and* $\forall[r_b,r_w,i] \in \mathscr{W}$, $\exists(r_b,a_b,b_b) \in \mathscr{V}'$ *and* $\exists(r_w,a_w,b_w) \in \mathscr{V}'$.

This theorem states that the elements in the input $\mathscr{V}$ are preserved in the output. In fact, there is a direct correspondence between them such that if the input set has an element with resource r, then the output will too. Furthermore, when new values are added (as in FT-WH), they correspond to values in $\mathscr{W}$. The proof is straightforward and proceeds by induction on the functional transition judgments. It has been omitted for brevity.

Theorem 2. *If* $e : \alpha \overset{R}{\rightsquigarrow} \beta$ *and* $(\mathscr{V},x,e) \Rrightarrow (\mathscr{V}',y,e',\mathscr{W})$, *then* $e' : \alpha \overset{R'}{\rightsquigarrow} \beta$ *and* e' *has the same structure of sub-expressions as* e *with the exception that wormhole expressions may have been replaced by their bodies. For each so replaced, there is a corresponding element in* $\mathscr{W}$ *of the form* $[r_b,r_w,i]$ *such that* r_b *and* r_w *are the virtual resources of said wormhole. Furthermore,* $R \subseteq R'$ *and* $\forall r \in (R' \setminus R)$, *either* $[r,\_,\_] \in \mathscr{W}$ *or* $[\_,r,\_] \in \mathscr{W}$.

This theorem states exactly how the output expression e' can be different from the input expression e. Notably, it will still be a signal function with the same input and output types and it will still behave in essentially the same way, but its set of resource types may grow. Specifically, if the resource type set does grow, it is because a wormhole expression was reduced to its body and the virtual resources it introduced are now visible at a higher level. A notable corollary of this theorem is that if $\mathscr{W} = \emptyset$, then $e = e'$.

Proof. The proof follows by induction on the judgments and the typing rule TY-WH for wormholes. A cursory examination of the judgments reveals that the only one to change the form of the expression from input to output is FT-WH, which replaces the input expression with the body of the wormhole. The typing rule tells us that if $e : \alpha \overset{R}{\rightsquigarrow} \beta$ and e is a wormhole, then the body of e has type $\alpha \overset{R'}{\rightsquigarrow} \beta$ where $R = R' \setminus \{r_w,r_b\}$. Although the resource type set may have grown, it could only have grown by the addition of r_b, r_w, or both. Furthermore, the element $[r_b,r_w,e_i]$ is added to the output $\mathscr{W}$. □

Lastly, it may appear that multiple **rsf** commands on the same resource could be problematic; after all, the FT-RSF judgment initially requires the resource r to have a triple of the form $(r,y,\cdot)$, but it results in the third element of the triple being filled in. That is, there is no **rsf** command judgment where the triple has a value in the third element. However, as we prove later in Theorem 3, if the program has type $\alpha \overset{R}{\rightsquigarrow} \beta$, then it must have at most one **rsf** command for any given resource r.

6.3 Temporal transition

Because signal functions act over time, we need a transition to show their temporal behavior. At each time step, we process the program, taking in the state of the world (i.e. all the resources) and returning it updated. There is only one temporal transition, but it is quite complicated. It is shown in Figure 8.

This transition says that the resource environment $\mathscr{R}$, the set of wormhole data $\mathscr{W}$, and a program P transition into an updated resource environment, an updated set of wormhole data, and a potentially more evaluated program.

$$\mathscr{V}_{in} = \{(r, \mathbf{next}(r), \cdot) \mid r \in \mathscr{R}\} \cup \{(r_w, i, \cdot) \mid [r_b, r_w, i] \in \mathscr{W})\} \cup \{(r_b, (), \cdot) \mid [r_b, r_w, i] \in \mathscr{W})\}$$
$$(\mathscr{V}_{in}, (), P) \Rrightarrow (\mathscr{V}_{out}, (), P', \mathscr{W}_{new})$$
$$\mathscr{R}' = \{\mathbf{put}(r, o') \mid r \in \mathscr{R}, (r, \_, o) \in \mathscr{V}_{out}, o \mapsto^* o'\}$$
$$\mathscr{W}' = \{[r_b, r_w, \mathbf{if}\ o = \cdot\ \mathbf{then}\ i\ \mathbf{else}\ o] \mid (r_b, \_, o) \in \mathscr{V}_{out}, [r_b, r_w, i] \in (\mathscr{W} \cup \mathscr{W}_{new})\}$$
$$(\mathscr{R}, \mathscr{W}, P) \overset{t}{\mapsto} (\mathscr{R}', \mathscr{W}', P')$$

Figure 8. The temporal transition.

The first precondition extracts data from the resources and wormholes and compiles it into a form that the functional transition can use. For the resources, we create triples of the form $(r, \mathbf{next}(r), \cdot)$ meaning that the resource r provides the value $\mathbf{next}(r)$ and is waiting for a return value. For wormholes, we actually create two triples, one for the blackhole and one for the whitehole. The whitehole uses the whitehole resource name r_w and the current value in the wormhole, and the blackhole uses r_b and produces only $()$.

This data is provided to the functional transition along with the program P. Because P has type $() \overset{R}{\rightsquigarrow} ()$ by definition, the streaming argument is set to $()$. The result of the functional transition is the updated value set ($\mathscr{V}_{out}$), the streaming output of P (given by the type to be $()$), the updated program, and a set of any new wormhole data encountered during execution.

The last two preconditions are analogous to the first one: they extract the resource and wormhole data from $\mathscr{V}_{out}$. For every element in $\mathscr{V}_{out}$ that corresponds to a resource in $\mathscr{R}$, we take the output value o, evaluate it, and push it to the resource. The resulting updated resources make up the new set $\mathscr{R}'$. It may be that o was never filled and is still empty—the **put** operation is executed regardless in order to push the resource one time step into the future. Note that because of the use of the evaluation transition, this step acts as a strictness point for the streaming values of the signal functions.

The wormhole data is extracted in much the same way. For every element in $\mathscr{V}_{out}$ that corresponds to a blackhole in either the original wormhole data set $\mathscr{W}$ or in the new additions $\mathscr{W}_{new}$, we examine the output value o. If o was filled in, then the updated wormhole entry contains the new value; otherwise, the wormhole keeps its old value.

In total, we see that the temporal transition uses the program P to update the resources $\mathscr{R}$ and the wormhole data $\mathscr{W}$. Because of Lemma 1, we can see that $\mathscr{R}'$ contains all the resources that $\mathscr{R}$ did, and similarly, $\mathscr{W}'$ contains all of the elements from both $\mathscr{W}$ and $\mathscr{W}_{new}$. Therefore, if $(\mathscr{R}, \mathscr{W}, P) \overset{t}{\mapsto} (\mathscr{R}', \mathscr{W}', P')$, then this transition can repeat indefinitely. That is, the next step would be $(\mathscr{R}', \mathscr{W}', P') \overset{t}{\mapsto} (\mathscr{R}'', \mathscr{W}'', P'')$ and so on. Since each pass through the transition represents one moment in time, this makes sense as a valid way to represent program execution over time.

We can use the temporal transition to establish an overall semantics for a program P in our language. Recall that $\mathscr{R}_o$ is the default resource environment containing all the resources of the real world.

Definition 2. *If P is a program (that is, an expression of the form $() \overset{R}{\rightsquigarrow} ()$ for some set R), then P will have the infinite trace starting at state $(\mathscr{R}_o, \emptyset, P)$ that uses only the temporal transition $\overset{t}{\mapsto}$.*

7. Safety

Here we show the safety that resource typing provides. We intend to show that if a program is well typed, then no two components will compete for the same resource. To express this, we must first define what it means to interact with a resource.

Definition 3 (Resource interaction)**.** *A program P interacts **once** with a resource r at a given time step if it reads the value produced by r at that time step, returns a value to r at that time step, or does both simultaneously.*

With this definition, we can state our resource safety theorem:

Theorem 3 (Resource safety)**.** *If a program $P : \alpha \overset{R}{\rightsquigarrow} \beta$, then P will interact only with resources in R, and for each resource it interacts with, it will do so at most once per time step.*

This theorem tells us that any program that type checks will only use the resources in its type and never have the problem where two components are vying for the same resource. The program will be entirely deterministic in its resource management, and from the type alone, one will be able to see which resources it has the potential to interact with while it runs.

Proof. The proof of resource safety begins by examining the temporal transition. Because each element in $\mathscr{R}$ is a unique resource, we know that interacting once each with different elements in $\mathscr{R}$ will never cause a problem. Furthermore, as all we do to create $\mathscr{R}'$ is exactly one **put** operation on each resource, $\mathscr{R}'$ will likewise have unique resources. The concern, then, comes from the functional transition. We must prove that updates in $\mathscr{V}_{out}$ are not being overwritten by future updates during the functional transition.

Therefore, the bulk of the proof proceeds by induction on the functional transition where we must show that any elements in $\mathscr{V}$ are only being updated at most once. Based on the updated Canonical Forms Lemma (Lemma 1), we know that since $P : \alpha \overset{R}{\rightsquigarrow} \beta$, it must be one of the five SF operators. We examine each in turn:

- *SF constructor:* If P is of the form $\mathbf{arr}(e)$, then by typing rule TY-ARR, $R = \emptyset$ and it will use judgment FT-ARR. There are no other transitions nor resource interaction being performed in this judgment, and since $R = \emptyset$, we trivially satisfy our conditions.
- *SF partial application:* If P is of the form $\mathbf{first}(e)$, then by typing rule TY-FIRST, we know that if e has type $\alpha \overset{R'}{\rightsquigarrow} \beta$, then $R = R'$. Furthermore, we know that P will proceed via judgment FT-FIRST. By our inductive hypothesis, we know that e will interact with each resource in R' at most once, and since no resource interaction happens in this judgment, we satisfy our conditions.
- *SF composition:* When P is of the form $e_1 \ggg e_2$, it will proceed by the FT-COMP judgment. By typing rule TY-COMP, we know that e_1 has resource type set R_1 and e_2 has resource type set R_2 such that $R_1 \cup R_2 = R$ but $R_1 \cap R_2 = \emptyset$. By our inductive hypothesis, e_1 evaluates interacting with at most the resources in R_1 and e_2 evaluates interacting with at most the resources in R_2. However, R_1 and R_2 share no common resources, and together, they make up R. Therefore, P does not interact with any

more resources than those in R, and any in R that it interacts with, it does so at most once.

- *SF resource interaction:* If P is of the form $\mathbf{rsf}[r]$, then it will proceed by the FT-RSF judgment. Typing rule TY-RSF tells us that its type must be $\alpha \overset{\{r\}}{\rightsquigarrow} \beta$. The transition completes in one step with no preconditions making use of no further calls, but in fact, $\mathscr{V}$ is being modified, so resource interaction is taking place. We see that the element in $\mathscr{V}$ for resource r is the only one being accessed and it happens precisely once. The access is allowed because trivially $r \in \{r\}$.
- *wormhole introduction:* P will proceed by the FT-WH judgment when it is of the form $\mathbf{wormhole}[r_w, r_b](e_i; e)$. Typing rule TY-WH tells us that e has type $\alpha \overset{R}{\rightsquigarrow} \beta$ the same as P. First, we recognize that no resource interaction can be performed by e_i because it is never evaluated as a expression by the functional transition. Even though we add values to $\mathscr{V}$, we do not modify and existing values, so we are not doing any true resource interaction in this transition. Therefore, our inductive hypothesis tells us that only acceptable resource interaction is done in the transition of the precondition. □

This proof takes the progress and preservation of our semantics for granted. The proofs for these can be located in Appendix A.

8. Conclusion

We have expanded upon the concept of wormholes, providing a clearer picture of their functionality than previous work. Not only have we improved their design, allowing dynamic wormhole creation with existential resource types, but we have solidified their theoretical foundation. This led us to new insights about the capabilities of wormholes, which has allowed us to draw a connection between them and causal loops. In fact, we show that in the presense of wormholes, other looping structures are superfluous.

Additionally, we have presented a novel way to conceptualize the program flow of an arrow-based FRP language by separating the various components of the semantic transition and introducing a temporal one. In doing so, we define the formal semantics that allow us to prove that resource types provide the safety that we claim they do. That is, no well-typed resource-typed signal function can access resources that are not in its resource type set, and furthermore, any that it does access will be accessed from only a single component. Therefore, a proper implementation of the resource type system should allow functional reactive programming with side effects without fear of the typical bugs that plague impure programming languages.

9. Related Work

The idea of using continuous modeling for dynamic, reactive behavior (now usually referred to as "functional reactive programming," or FRP) is due to Elliott, beginning with early work on TBAG, a C++ based model for animation [Elliott et al. 1994]. Subsequent work on Fran ("functional reactive animation") embedded the ideas in Haskell [Elliott and Hudak 1997; Hudak 2000]. The design of Yampa [Courtney et al. 2003; Hudak et al. 2003] adopted arrows as the basis for FRP, which is what we use here.

Liu et al. [2011] introduced an *ArrowInit* class to capture causality in their work on causal commutative arrows. Indeed, we drew our inspiration for the design of our delay loops from that structure. Although our work is somewhat more expressive since it is not limited to being only first-order, it does lack the benefits of being commutative, and as such, the optimizations for CCA are not applicable here.

Krishnaswami et al. [2012] also explore causaliity at the type level. They describe a language that uses non-arrow-based FRP yet still manages to restrict space-leaks statically. This language is somewhat more expressive than ours as it allows a more generic loop operator, but it is not clear whether it can be easily adapted to allow mutation or other side effects.

Cooper and Krishnamurthi [2006] embed an effectful implementation of FRP into PLT Scheme (now Racket) in FrTime. Although similar in content and behavior, this system cannot provide the resource safety that we do. Furthermore, the semantics that Cooper [2008] presents are quite different from ours, as he uses an imperative style with heap updates where we introduce the idea of resources.

The language *Clean* [Brus et al. 1987; Plasmeijer and van Eekelen 2002] 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* [Girard 1987], and various authors have proposed language extensions to incorporate linear types, such as Wadler [1991]; Hawblitzel [2005]; Tov and Pucella [2011]; Wadler [1990]. 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.

Recent work in linear-time temporal logic (LTL) [Jeffrey 2012; Jeltsch 2012] explores the Curry-Howard correspondence between LTL and FRP. This has led to another way to constrain the temporal behavior of reactive programs. Indeed, Jeffrey [2012] lays out the basis for an implementation of a constructive LTL in a dependently typed language such that reactive programs form proofs of LTL properties.

Acknowledgments

This research was supported by a gift from Microsoft Research and a grant from the National Science Foundation (CCF-0811665). Thanks to Shu-chun Weng for support and motivation.

References

G. Berry and L. Cosserat. The Esterel synchronous programming language and its mathematical semantics. In *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 389–448. Springer-Verlag, July 1984.

T. Brus, M. van Eekelen, M. van Leer, M. Plasmeijer, and H. Barendregt. CLEAN – A language for functional graph rewriting. In *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 364–384. Springer-Verlag, September 1987.

P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: A declarative language for programming synchronous systems. In *14th Symposium on Principles of Programming Languages*, pages 178–188. ACM, January 1987.

G. H. Cooper. *Integrating dataflow evaluation into a practical higher-order call-by-value language*. PhD thesis, Brown University, Providence, RI, USA, May 2008.

G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *Programming Languages and Systems*, volume 3924 of *Lecture Notes in Computer Science*, pages 294–308. Springer-Verlag, March 2006.

A. Courtney, H. Nilsson, and J. Peterson. The Yampa arcade. In *Haskell Workshop*, Haskell '03, pages 7–18. ACM, August 2003.

C. Elliott and P. Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 263–273. ACM, June 1997.

C. Elliott, G. Schechter, R. Yeung, and S. Abi-Ezzi. TBAG: A high level framework for interactive, animated 3D graphics applications. In *21st Conference on Computer Graphics and Interactive Techniques*, pages 421–434. ACM, July 1994.

T. Gautier, P. L. Guernic, and L. Besnard. Signal: A declarative language for synchronous programming of real-time systems. In *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 257–277. Springer-Verlag, November 1987.

J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

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

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

P. Hudak. *The Haskell School of Music – from Signals to Symphonies*. [Version 2.0], January 2011. URL `http://haskell.cs.yale.edu/?p=112`.

P. Hudak, S. Peyton Jones, and P. Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.

P. Hudak, A. Courtney, H. Nilsson, and J. 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, August 2003.

J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, May 2000.

A. Jeffrey. LTL types FRP: Linear-time temporal logic propositions as types, proofs as functional reactive programs. In *Sixth Workshop on Programming Languages meets Program Verification*, pages 49–60. ACM, January 2012.

W. Jeltsch. Towards a common categorical semantics for linear-time temporal logic and functional reactive programming. In *28th Conference on the Mathematical Foundations of Programming Semantics*, pages 215–228. Elsevier, June 2012.

O. Kiselyov, R. Lämmel, and K. Schupke. Strongly Typed Heterogeneous Collections. CWI Technical Report SEN-E 0420, CWI, August 2004.

N. R. Krishnaswami, N. Benton, and J. Hoffmann. Higher-Order Functional Reactive Programming in Bounded Space. In *39th Symposium on Principles of Programming Languages*, pages 45–58. ACM, January 2012.

J. Launchbury and S. Peyton Jones. Lazy functional state threads. In *Conference on Programming Language Design and Implementation*, pages 24–35. ACM, June 1994.

S. Lindley, P. Wadler, and J. Yallop. The arrow calculus. *Journal of Functional Programming*, 20(1):51–69, January 2010.

H. Liu, E. Cheng, and P. Hudak. Causal commutative arrows. *Journal of Functional Programming*, 21(4–5):467–496, September 2011.

P. Liu and P. Hudak. Plugging a space leak with an arrow. *Electronic Notes in Theoretical Computer Science*, 193(1):29–45, November 2007.

E. Moggi. Computational lambda-calculus and monads. In *Symposium on Logic in Computer Science*, pages 14–23. IEEE, June 1989.

R. Paterson. A new notation for arrows. In *Sixth International Conference on Functional Programming*, pages 229–240. ACM, September 2001.

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

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

R. Plasmeijer and M. van Eekelen. Clean – version 2.1 language report. Technical report, Department of Software Technology, University of Nijmegen, November 2002.

J. A. Tov and R. Pucella. Practical affine types. In *38th Symposium on Principles of Programming Languages*, pages 447–458. ACM, January 2011.

A. Voellmy and P. Hudak. Nettle: Taking the sting out of programming network routers. In *Practical Aspects of Declarative Languages*, volume 6539 of *Lecture Notes in Computer Science*. Springer-Verlag, January 2011.

P. Wadler. Linear types can change the world! In *Working Conference on Programming Concepts and Methods*, pages 347–359. IFIP TC 2, April 1990.

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

P. Wadler. The essence of functional programming. In *19th Symposium on Principles of Programming languages*, pages 1–14. ACM, January 1992.

D. Winograd-Cort, H. Liu, and P. Hudak. Virtualizing Real-World Objects in FRP. In *Practical Aspects of Declarative Languages*, volume 7149 of *Lecture Notes in Computer Science*, pages 227–241. Springer-Verlag, January 2012.

B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a Promotion. In *8th Workshop on Types in Language Design and Implementation*, pages 53–66. ACM, January 2012.

A. Proofs of Preservation and Progress

In order to prove preservation and progress for our semantics, we must show these properties for each of the transitions we have defined. Here we state and prove the relevant theorems.

Evaluation Transition

The evaluation transition is mostly lifted from a standard lazy semantics for $\mathscr{L}\{\rightarrow\times\}$. The additions presented in Figure 6 simply explain that the new expressions are all values. Therefore, preservation and progress follow trivially.

Functional Transition

Preservation for the functional transition proceeds in a straightforward manner making sure that the streaming input is appropriately transitioned into a streaming output.

Theorem 4 (Preservation during functional transition)**.** *If* $e : \alpha \overset{R}{\rightsquigarrow} \beta$, $x : \alpha$, *and* $(\_, x, e) \Rrightarrow (\_, y, \_, \_)$, *then* $y : \beta$.

Proof. The proof of preservation proceeds by induction on the derivation of the transition judgment along with the knowledge of preservation for the evaluation transition. Each of the judgments can be proved trivially with a brief examination of the typing rules, so we omit the details. □

Progress for the functional transition is a somewhat more interesting concept. Because of the complexity of the transition, we are forced to make a few assumptions about the input data:

Theorem 5 (Progress during functional transition)**.** *If* $e : \alpha \overset{R}{\rightsquigarrow} \beta$, $x : \alpha$, *and* $\mathscr{V}$ *contains elements such that* $\forall r \in R, (r, a, \cdot) \in \mathscr{V}$ *where* $r : \langle \tau_{in}, \tau_{out} \rangle$ *and* $a : \tau_{in}$, *then* $\exists y : \beta, e' : \alpha \overset{R'}{\rightsquigarrow} \beta, \mathscr{V}', \mathscr{W}$ *such that* $(\mathscr{V}, x, e) \Rrightarrow (\mathscr{V}', y, e', \mathscr{W})$.

We require that in addition to the expression e being well-formed and the streaming argument x being of the right type, the set $\mathscr{V}$ must also be "well-formed". That is, for every resource that e might interact with (all resources in R), there is a triple in $\mathscr{V}$ corresponding to that resource that contains values of the appropriate types. Notably, they must all be resources that have not seen any interaction. This is not an unreasonable requirement as

we proved in Theorem 3 that at any point during the functional execution, no resources see more than one interaction.

Proof. The proof of progress proceeds by induction on the derivation of the functional transition judgment. Based on the Canonical Forms Lemma (Lemma 1), we know that the functional transition need only apply to the five forms of a signal function, and we see by inspection that it does. We examine each judgment in turn:

- *SF constructor* (FT-ARR): When e is of the form $\mathbf{arr}(e')$, typing rule TY-ARR tells us that $e' : \alpha \to \beta$. As $x : \alpha$, the streaming output $e\ x$ is of type β as necessary. The other outputs exist regardless of the form of e'.
- *SF partial application* (FT-FIRST): If e is of the form $\mathbf{first}(e')$, then the typing rule TY-FIRST tells us that e' has resource type set R just as e does. Our inductive hypothesis tells us that outputs are available for our recursive transition. The streaming output (y,z) has the appropriate type, and the expression output, formed by applying **first** to the expression output of the recursive transition has the same type as e.
- *SF composition* (FT-COMP): e may be of the form $e_1 \ggg e_2$. By typing rule TY-COMP, we know that $e : \alpha \overset{R}{\rightsquigarrow} \gamma$, $e_1 : \alpha \overset{R_1}{\rightsquigarrow} \beta$, and $e_2 : \beta \overset{R_2}{\rightsquigarrow} \gamma$. The evaluation transitions progress, and by our inductive hypothesis, the functional transitions in the precondition progress as well. The output is formed from the results of the precondition with the streaming value z being of type γ as required. The expression output, made by composing the two expressions e_1'' and e_2'' has the same type as e.
- *SF resource interaction* (FT-RSF): If e is of the form $\mathbf{rsf}[r]$, then the typing rule TY-RSF tells us that its type must be $\alpha \overset{\{r\}}{\rightsquigarrow} \beta$ and $r : \langle \alpha, \beta \rangle$. By the conditions of our theorem, $\mathscr{V}$ must contain an element $(r, y, \cdot)$ such that $y : \beta$. Therefore, the streaming output y is of the right type. Lastly, the output expression is identical to the input expression.
- *Wormhole introduction* (FT-WH): We use typing rule TY-WH when e is of the form $\mathbf{wormhole}[r_w, r_b](e_i; e_{body})$; it tells us that e_{body} has type $\alpha \overset{R'}{\rightsquigarrow} \beta$ where $R = R' \setminus \{r_w, r_b\}$. Before using our inductive hypothesis, we must prove that the value set for the recursive call meets our requirements. We know that $(R \cup \{r_w, r_b\}) \supseteq R'$, so $\mathscr{V} \cup \{(r_w, e_i, \cdot), (r_b, (), \cdot)\}$ clearly satisfies the condition. Therefore, the streaming output y will be of type β. Furthermore, the output expression e'' must have the same type as e_{body} which satisfies our output requirement. □

Temporal Transition

By the definition of the overall operational semantics (Definition 2), we know that the trace of any program P is infinite. As long as we can prove progress, preservation is irrelevant. We make use of the preservation and progress theorems for the evaluation and functional transitions shown earlier to prove the following:

Theorem 6 (Progress of overall semantics)**.** *If P is a program with type $\alpha \overset{R}{\rightsquigarrow} \beta$ and $R \subseteq \mathscr{R}_o$ then the trace of P will always be able to progress via the temporal transition $\overset{t}{\mapsto}$ when starting from $(\mathscr{R}_o, \emptyset, P)$.*

Proof. The judgment for the temporal transition allows the input to progress so long as the preconditions are met. The first condition defines $\mathscr{V}_{in}$ to contain elements for each resource in $\mathscr{R}$ as well as for each whitehole and blackhole pair in $\mathscr{W}$. This is used in the second condition, which will progress only if we can prove that $(\mathscr{V}_{in}, (), P)$ will progress through the functional transition. P may access resources in R as well as any virtual resources introduced through wormholes. In the base case, the functional transition has never been run, and R does not contain any virtual resources. Then, because $R \subseteq \mathscr{R}_o$, $\mathscr{V}_{in}$ contains elements for every resource in R, so we meet the conditions of the functional progress theorem (Theorem 5). In the inductive case, we are dealing with a potentially further evaluated program P' with resources R', which may contain virtual resources. Then, all virtual resources will have been generated from previous passes through the functional transition, and all of the virtual resources will be represented by $\mathscr{W}$. Once again, $\mathscr{V}_{in}$ will contain elements for each resource in R', and the functional transition can progress.

The last two preconditions are simply defintions of $\mathscr{R}'$ and $\mathscr{W}'$ such that R' contains the same number of elements keyed by the same resource names as $\mathscr{R}$ and that $\mathscr{W}'$ contains the same whitehole and blackhole resource names as $\mathscr{W}$ as well as any new wormhole data entries from $\mathscr{W}_{new}$.

The output program P' is not the same as P. Notably, its type may have changed to $() \overset{R'}{\rightsquigarrow} ()$. From Theorem 2, we know that R' is the set R with up to two new virtual resources for each element of $\mathscr{W}_{new}$ corresponding to the whiteholes and blackholes of the elements of $\mathscr{W}_{new}$. This is fine for exactly the reason that these new resources are "documented" in $\mathscr{W}_{new}$ and $\mathscr{W}_{new}$ is unioned with $\mathscr{W}$ for the output of the transition. Therefore, when $\mathscr{V}$ is being generated in the next iteration, all of the resources of R' will be represented, both the original ones in $\mathscr{R}$ and any virtual ones created and documented in $\mathscr{W}$.

Finally, we must consider the overall base case. On the first iteration through the temporal transition, there can be no virtual resources because no wormhole expressions have been executed by the functional transition yet. Therefore, the initial wormhole set $\mathscr{W}$ can be the empty set. □