Higher-Order Functional Reactive Programming without Spacetime Leaks

Neelakantan R. Krishnaswami
Max Planck Institute for Software Systems (MPI-SWS)
neelk@mpi-sws.org

Abstract

Functional reactive programming (FRP) is an elegant approach to declaratively specify reactive systems. However, the powerful abstractions of FRP have historically made it difficult to predict and control the resource usage of programs written in this style.

In this paper, we give a new language for higher-order reactive programming. Our language generalizes and simplifies prior type systems for reactive programming, by supporting the use of streams of streams, first-class functions, and higher-order operations. We also support many temporal operations beyond streams, such as terminatable streams, events, and even resumptions with first-class schedulers. Furthermore, our language supports an efficient implementation strategy permitting us to eagerly deallocate old values and statically rule out *spacetime leaks*, a notorious source of inefficiency in reactive programs. Furthermore, these memory guarantees are achieved *without* the use of a complex substructural type discipline.

We also show that our implementation strategy of eager deallocation is safe, by showing the soundness of our type system with a novel step-indexed Kripke logical relation.

Categories and Subject Descriptors D.3.2 [*Dataflow Languages*]

Keywords Functional reactive programming; Kripke logical relations; temporal logic; guarded recursion; dataflow; capabilities; comonads

1. Introduction

Interactive programs engage in an ongoing dialogue with the environment. An interactive program receives an input event from the environment, and computes an output event, and then waits for the next input, which may in turn be affected by the earlier outputs the program has made. Examples of such systems range from embedded controllers and sensor networks, up to graphical user interfaces, web applications, and video games.

Programming interactive applications in general purpose programming languages can be very confusing, since the different components of the program do not typically interact via structured control flow (such as loops or direct procedure calls), but instead operate by registering state-manipulating callbacks with one another, which are then implicitly invoked by an event loop. Reasoning about such programs is difficult, since each of these features – higher-order functions, aliased imperative state, and concurrency – is challenging on its own, and their combination takes us to the outer limits of what verification can cope with.

These difficulties, plus the critical nature of many reactive systems, have inspired a great deal of research into languages, libraries and analysis techniques for reactive programming. Two of the main strands of work on this problem are the *synchronous dataflow languages*, and *functional reactive programming*.

Synchronous dataflow languages, such Esterel [5], Lustre [7], and Lucid Synchrone [39], implement a computational model inspired by Kahn networks. Programs are fixed networks of stream-processing nodes that communicate with each other, each node consuming and producing a statically-known number of primitive values at each clock tick. These languages deliver strong guarantees on space and time usage, and so see wide use in applications such as hardware synthesis and embedded control software.

Functional reactive programming, introduced by Elliott and Hudak [15], also works with time-varying values (rather than mutable state) as a primitive abstraction. However, it provides a much richer model than synchronous languages do. Signals are first-class values, and can be used freely, including in higher-order functions and signal-valued signals, which permits writing programs which dynamically grow and alter the dataflow network. FRP has been applied to problem domains such as robotics, games, music, and GUIs, illustrating the power of the FRP model.

However, this power comes at a steep price. Modeling A-valued signals as streams A^ω (in the real-valued case, $A^{\mathbb{R}}$) and reactive systems as functions $\mathsf{Input}^\omega \to \mathsf{Output}^\omega$ has several problems. First, this model does not enforce causality (that is, output at time n depends only upon inputs at earlier times), nor does it ensure that feedback (used to define signals recursively) is well-founded. Second, since the model of FRP abstracts away from resource usage, it is easy to write programs with significant resource leaks by inadvertently accumulating the entire history of a signal – that is,"space leaks".[1] For example, consider the simple program below:

$$\mathsf{bad\_const} \;:\; \mathsf{S}\,\mathbb{N} \to \mathsf{S}\,(\mathsf{S}\,\mathbb{N}) \qquad 1$$
$$\mathsf{bad\_const}\; n s = \mathsf{cons}(n s, \mathsf{bad\_const}\; n s) \qquad 2$$

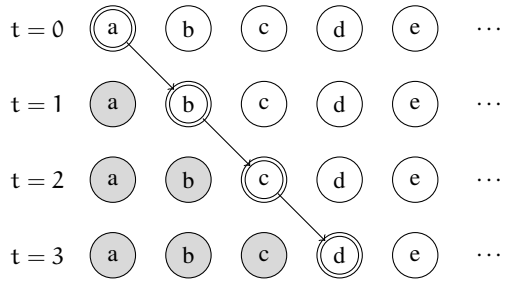
The bad_const function takes a stream of numbers as an argument, and then returns that stream constantly. So if it receives the stream $(a, b, c, \ldots)$, then it returns a stream of streams with $(a, b, c, \ldots)$ as its first element, with $(a, b, c, \ldots)$ as its second element, and so on

[1] In reactive programming, the phrase "space leak" usually refers only to memory leaks arising from capturing too much history, since they form the species of memory leak that functional programmers are unused to debugging. Relatedly, there are also "time leaks", which occur when a signal is sampled infrequently. Under lazy evaluation, infrequent sampling can lead to the head of a stream growing to become a large computation.

ICFP '13, September 25-27, Boston, MA, USA..

ACM 978-1-4503-2326-0/13/09...$15.00.
http://dx.doi.org/10.1145/2500365.2500588

indefinitely. This is a perfectly natural, even boring, *mathematical* function on streams, but many *implementations* leak memory. Consider the following diagram, illustrating how a stream data structure evolves over time:



Here, a stream is a computation incrementally producing values. At time 0, the stream yields the value "a", and at time 1, it yields the value "b", and so on. Each value in a double-circle represents the stream's *current value* at time t, and the values in white circles represent the stream's *future values*, and the values in gray circles represent the stream's *past values*. (So there are n gray circles at time n.)

If a pointer to the head of a stream persists, then at time n all of the first $n + 1$ values of the stream will be accessible, and so we will need to store all the values the stream has produced – all of the grey nodes, plus the double-circled node. Thus, at time n, we will have to store n elements of history, which means that the memory usage of the bad_const function will be $O(n)$ at time n. So the accidental use of a function like bad_const in a long-running program will eventually consume all of the memory available.

To avoid this problem, a number of alternative designs, such as such as *event-driven FRP* [43] and *arrowized FRP* [35], have been proposed. The common thread in this work is to give up on treating signals as first-class values, and instead offer a collection of combinators to construct stream transformer functions (that is, functions of type $S\,A \to S\,B$) from smaller ones. Streams of streams (and other time-dependent values) are forbidden, and each of the exported combinators is designed to ensure that only efficient stream transformers can be constructed.

Restricting programmers to indirect access to streams is akin to a first-order programming style, since it makes it difficult to abstract over stream-manipulating operations. This restriction is also quite similar to those synchronous dataflow languages impose, and Liu et al. [30] exploit this resemblance to develop a compilation scheme (reminiscent of the Bohm-Jacopini theorem) to compile arrow programs into efficient single-loop code, much as compilers for Esterel and Lucid do. Sculthorpe and Nilsson [40] extend the arrowized approach further, by using dependent types to enforce causality.

However, dynamic modification of the dataflow graph is very natural for interactive programs. For example, in a graphical program, we may wish to associate dataflow networks with windows, which are created and destroyed as a program executes. To support this, arrowized FRP libraries need to add additional combinators to restore dynamic behavior. These combinators make unwanted memory leaks possible once more (though less easily than in the original FRP). They also have a somewhat *ad-hoc* flavor: since first-class and higher-order stream types are unavailable, somewhat complex and strange types are needed to encode typical usage patterns.

To recover a more natural programming style, Krishnaswami and Benton [27] proposed a lambda-calculus for stream programming based on the guarded recursion calculus of Nakano [34], which ensured by typing that all definitions are causal and well-founded. Jeffrey [19] and Jeltsch [22] further observed that it is possible to use the formulas of linear temporal logic [38] as types for reactive programs. However, while all of these papers offer a coherent account of causality (not using values from the future, and ensuring that all recursive definitions are guarded), none of them have much to say about the memory behavior of their programs.

To resolve this problem, Krishnaswami et al. [28] gave a calculus which retains the causality checking and fully higher order style of [19, 22, 27], but which also uses linear types to control the memory usage of reactive programs. This calculus passes around linearly-typed tokens representing the right to allocate stream values, which ensures that no program can ever allocate a dataflow graph which is larger than the total number of permissions passed in. This solution works, but is *too precise* to be useful in practice. The exact size of the dataflow graph is revealed in a program's type, and so even modest refactorings lead to massive changes in the type signature of program components.

Contributions Our contributions are as follows:

1. We give a new implementation strategy, described using operational semantics, for higher-order reactive programming. Our semantics rules out space and time leaks *by construction*, making it *impossible* to write programs which accidentally retain temporal values over long periods of time.

   To accomplish this, we describe the semantics of reactive programs with two operational semantics. The first semantics explains how to evaluate program terms in the current clock tick, and the second is a "tick relation" for advancing the clock.

   Expressions in a reactive language can be divided into two classes, those which must be evaluated now, and those which must be evaluated later, on future clock ticks. We evaluate current expressions using a call-by-value strategy, and we suspend future expressions by placing them in mutable thunks on the heap, in a style reminiscent of call-by-need. Evaluating current expressions in a call-by-value style prevents time leaks, by making streams head-strict [42].

   Future expressions can only be evaluated in the future, and the tick relation describes how to advance the clock into the future. This semantics works by going through the heap of thunks, and forcing each expression scheduled for execution on the current time step. Furthermore, we achieve our goal of blocking space leaks through the simple but drastic measure of *deleting all old values*, thereby making it operationally impossible to accidentally retain a reference to a value past its lifetime.

2. We give a simple type theory for a higher-order reactive programming language.

   Our type system generalizes earlier work on using temporal logic [19, 22, 27] to type reactive programs. Instead of using formulas of temporal logic as types, we take a standard simply-typed lambda calculus, and introduce a *delay modality* $\bullet A$ of terms of type A that can be evaluated "on the next tick of the clock", and add to that the new notion of a *temporal recursive type* $\hat{\mu}\alpha.\ A$, which allows defining types which are recursive through time.

   This recursive type construct allows *encoding* all of the standard temporal operators as derived constructions within our calculus – streams, events, and even resumptions are all definable within our system, as are higher-order functions to construct and manipulate elements of these types.

   To retain control over space leaks, we simplify the work of Krishnaswami et al. [28] on using linear types to prevent space leaks in reactive systems. We introduce a new kind of *type-based capability* which grants the *right* to allocate memory, without having to enumerate the exact *amount* of memory used. As a result, we do not need the complexities of a substructural type

system, and moreover we no longer reflect exact memory usage in types, which improves the modularity of reactive programs.

3. We show that our type discipline is sound by means of a novel step-indexed Kripke logical relation.

   Deleting old values naturally raises the question of what happens if we accidentally reference a deleted value. Our logical relation lets us prove that no well-typed program will ever dereference a deallocated location, and that expressions whose types say they do not allocate memory, do not actually allocate memory. Furthermore, the logical relation shows that the expression relation is total for well-typed terms, despite the presence of a term-level fixed point. This demonstrates that we can only define well-founded and causal loop structures.

Supplementary Material Full proofs of the main results in this paper, and statements and proofs of all the supporting lemmas, can be found in the accompanying technical report.

2. Programming Language

We begin with a description of our language design. We give the syntax of types, terms, and values in Figure 1, the syntax of contexts in Figure 2, and the typing rules in Figures 3, 4, and 5. The operational semantics of expressions is in Figure 6, and the semantics of advancing the clock is in Figure 7.

Overview In a reactive language, an explicit notion of time is exposed to the programmer, and so we will need to extend the operational semantics to account for time, and then give a modified type system which properly reflects the semantics.

Our operational semantics consists of two relations. The expression relation $\langle \sigma; e\rangle \Downarrow \langle \sigma'; v\rangle$ describes how to evaluate an expression within a single timestep. This relation, given in Figure 6, is a standard big-step, call-by-value operational semantics for a functional language with a store. As we will see, the use of a call-by-value semantics rules out time leaks, since they are inherently an artifact of lazy evaluation.

However, the stores are not unrestricted heaps in the style of ML. Since reactive languages allow programmers to schedule *when* different expressions should be executed, there are program expressions which should not be executed right away. We put these expressions into a store, with the idea that the code stored in the heap will be evaluated later. In terms of reactive programming idioms, the store represents *the dataflow graph* of the program, which contains the nodes that will supply values on later ticks of the clock. In terms of functional programming idioms, the store implements *lazy evaluation*, in a variant of call-by-need where thunks are explicitly scheduled for later execution.

To actually advance the clock, we give the *tick relation* $\sigma \Longrightarrow \sigma'$, which describes (Figure 7) how the store σ is transformed into a store σ' when the clock ticks. As expected, our relation evaluates all of the computations scheduled for evaluation on the next tick. The tick semantics also makes space leaks impossible, by construction: brutally but expediently it *deletes all values* that are more than one tick old.

It is now trivially the case that it is impossible to inadvertently store a history too long – but, at what price? Surprisingly, we can show that the price is low. We give a type discipline, which not only ensures that all well-typed programs are *safe* (in that they never try to access a deleted value), but which is also *expressive*, in that natural reactive programs (higher-order or not) are still well-typed.

Since the passage of time is a central feature of our operational semantics, we introduce three *temporal qualifiers*, now, later, and stable, to explain *when* some expression is well-typed, and when a variable may be used. The now qualifier means "in the present time step", the later qualifier means "on the next time step", and the stable qualifier means "at any time step, present or future".

As a result, our typing judgment takes the form $\Gamma \vdash e : A\ q$, where q is a qualifier. So the informal reading of the typing judgment is "under hypotheses Γ, the expression e has type A at time q." Just as expressions have temporal qualifiers, so too do the hypotheses in the context. The context Γ (with a grammar given in Figure 2) consists of a list of hypotheses of the form $x : A\ q$, which gives not just a type A for each variable x, but also gives a qualifier q controlling when we may use the variable x.

Types, Terms, and Expression Evaluation The core of our programming language is essentially the simply-typed lambda calculus. The basic types include product types $A \times B$ with pairs (e, e') and projections $\mathsf{fst}\ e$ and $\mathsf{snd}\ e$, disjoint unions $A + B$ with injections $\mathsf{inl}\ e$ and $\mathsf{inr}\ e$ and a case statement $\mathsf{case}(e, \mathsf{inl}\ x \to e', \mathsf{inr}\ y \to e'')$, and function types $A \to B$ with lambda-abstractions $\lambda x.\ e$ and applications $e\ e'$, as well as a variety of primitive types (such as numbers $\mathbb{N}$ and booleans bool). The typing rules for these constructs are all given in Figure 3, and are all standard. Similarly, the reduction rules for these terms are given in the first half of Figure 6, and are also standard.

To this, we add a number of features to deal with time.

First, we have the next-step operator $\bullet A$. The intuitive reading of this type is that an inhabitant of $\bullet A$ is a term that, when evaluated on the next time step, will yield a value of type A – the type of "A's tomorrow". The introduction form is the delay term $\delta_{e'}(e)$, which says that e is a term to evaluate on the next time step. Consequently, its typing rule $\bullet$I requires e to be typechecked later. The elimination form is a binding-style elimination form, $\mathsf{let}\ \delta(x) = e\ \mathsf{in}\ e'$. The typing rule $\bullet$E asserts that e is of type $\bullet A$, and the variable x is of type A, but with the later qualifier. This ensures that the body e' does not use e's value before it is available.

Operationally, we do not evaluate the body e of a delayed expression $\delta_{e'}(e)$ right away. Instead, we extend the store with a pointer $l : e\ \mathsf{later}$ to a thunk e, which will be evaluated on the next time step. These pointers implement a form of call-by-need, ensuring that even if we use a delayed variable multiple times, the delayed expression will itself only be evaluated once. This can be seen in the reduction rule for $\mathsf{let}\ \delta(x) = e\ \mathsf{in}\ e'$. Here, if e evaluates to the pointer l, the reduction rule substitutes $!l$, a dereferencing of the pointer, for x in e'.

The allocation of a thunk extends the reactive program's dataflow graph, and the growth of the dataflow graph is something we must control. To achieve this, we use the subscripted argument e' in the delay introduction form $\delta_{e'}(e)$. This argument must be of the allocation type alloc, and indicates a permission to allocate heap storage. Since the type alloc represents a pure capability [32], we have no introduction or elimination forms for it; a token (denoted by $\diamond$ in our syntax) representing this capability must be passed in to a closed program by the runtime system of the language. Thus, programmers may control whether or not a function allocates by controlling whether or not they pass it an allocation capability.

Pointer expressions l, dereference expressions $!l$, and resource tokens $\diamond$ are all internal forms of our language, and cannot be written by a programmer. (As a result, there are no typing rules for them.)

Values of $\bullet A$ type are time-dependent, in the sense that their representation changes over time: when the clock ticks, the dataflow graph/store is evaluated, and pointers are updated. This means that they should only be used on particular time steps. Other types, such as natural numbers or booleans, consist of values whose representation does not change over time. These *stable values* may safely be used at many different times. Values of other types, such as functions $A \to B$, are either time-dependent or stable, depending on whether or not they capture any time-dependent values in their environment.

So we also introduce a modal type $\Box A$, which contains only those values of type A which are *stable*. The introduction form $\mathsf{stable}(e)$ ensures that e has the type A under the stable qualifier, and the elimination form $\mathsf{let\ stable}(x) = e\ \mathsf{in}\ e'$ takes a term e of type $\Box A$, and binds x to a stable variable. Since values of certain types (base types, and products and sums of same) are always stable, we also introduce a term $\mathsf{promote}(e)$, which takes a term of an inherently stable type A (as judged by the judgment $A\ \mathsf{stable}$, defined in Figure 5) and returns a value of type $\Box A$.

In order to get interesting temporal data structures, we introduce a *temporal recursive type*, $\hat{\mu}\alpha.\ A$. This type has the expected introduction $\mathsf{into}\ e$ and elimination $\mathsf{out}\ e$ forms, but their types are slightly nonstandard. When constructing a term $\mathsf{into}\ e$ of recursive type $\hat{\mu}\alpha.\ A$, we require e to have the type $[\bullet(\hat{\mu}\alpha.\ A)/\alpha]A$. That is, every occurrence of α in A is substituted with $\bullet(\hat{\mu}\alpha.\ A)$, which is the recursive type *on the next time step*. This lets us use the next-step modality to define data whose structure repeats over time, rather than proceeding only a constant number of steps into the future.

In addition to type-level recursion, we also have a term-level recursion operator $\mathsf{fix}\,x.\ e$. There are no restrictions on the types we may take fixed points at; we only require that if $\mathsf{fix}\,x.\ e$ is of type A, then we assume x is a later variable. This ensures that we can never construct an unguarded loop.

We also include a stream type $\mathsf{S}\,A$, with an introduction form $\mathsf{cons}(e, e')$ and an elimination form $\mathsf{let\ cons}(x, xs) = e\ \mathsf{in}\ e'$. The stream type can be encoded using the other constructs of the language (see Section 3.2), but we include it in order to give a more readable syntax to our examples. However, the $\mathsf{cons}(e, e')$ form does help illustrate why we chose a call-by-value evaluation strategy. Under call-by-value, the head of the stream e is always evaluated to a value. As a result, "time leaks" are impossible, since the head gets reduced to a value on every tick, and so it doesn't matter how frequently a stream is sampled.

Finally, the variable rule HYP says that we can only use a variable now, if it is either stable, or if it is available now. Then, there are the TSTABLE and TLATER rules, which show how to derive terms with the stable and later qualifiers, in terms of the now qualifier. The key to this are the context-clearing operations defined in Figure 2. The $\Gamma^{\Box}$ context operation deletes every now and later hypothesis, ensuring that stable terms may only depend on stable variables. The $\Gamma^{\bullet}$ operation takes a context, and deletes every hypothesis marked now, and changes every later hypothesis to now, ensuring that terms which are typechecked later (1) view the later variables as if they were available in the present, and (2) do not access any variables containing possibly out-of-date values.

Dataflow Graphs The store σ is used to represent the dataflow graph of the program, and contains a collection of pointers, which can be viewed as the nodes in a dataflow graph. Each node is either a suspended expression $l : e$ later to be evaluated on the next tick of the clock, or points to a presently available value $l : v$ now, or is undefined $l : \mathsf{null}$ and points to nothing.

Advancing Time We give the tick relation $\sigma \Longrightarrow \sigma'$ in Figure 7, which explains how to advance time for a dataflow graph.

If the dataflow graph σ is empty, then of course the updated store is also empty. In order to evaluate a store $\sigma, l : e$ later, we first update the rest of the dataflow graph σ to σ', and then evaluate e in the result store σ' to a value v, updating the pointer to $l : v$ now. However, when we see a store $\sigma, l : v$ now containing a value, we evaluate the store, but null out the pointer, setting it to $l : \mathsf{null}$. Finally, once nulled out, null pointers stay nulled out.

This feature of the operational semantics suffices to ensure that FRP-specific space leaks are *impossible* – because our dataflow graph never stores values for more than a single tick, it follows that values can never accumulate and build up into a memory leak. As a result, values in dataflow nodes can never persist in memory unless the programmer explicitly writes code to retain them. (Of course, all the memory leaks traditional to functional programming are still possible – we have simply ensured that reactivity has not added any *new* sources of leaks.)

It is worth reiterating that it is the operational semantics, and not the type system, which ensures the absence of space leaks: because we *delete* everything old, it is simply impossible to remember the past unless the programmer explicitly writes the code to do so. The type system merely acts as a set of guard rails, ensuring that we do not accidentally follow any invalid pointers.

Making dataflow nodes transient is a significant departure from existing imperative implementation strategies for FRP libraries, such as Scala.React [31] or Racket's FrTime [9, 10]. In these libraries, dataflow nodes are persistent, and last across many time steps, and clever heuristics are used to order updates.

Abandoning this strategy pays many dividends. We have already observed that accidental space leaks are no longer possible, but there are also further practical benefits. A real implementation needs to garbage collect null nodes. Fortunately, the structure of our typing rules ensures that well-typed program terms never contain locations that outlive their tick, and as a result, the usual reachability heuristic of standard garbage collectors will work unchanged.

In contrast, persistent dataflow nodes need to manage dependencies explicitly, and as a result, each dataflow cell knows both which cells it reads, and which cells read it. This bidirectional linkage means that if one cell is reachable, all cells are reachable, defeating garbage collection unless special (and often expensive) measures such as weak references are used.

Complete Programs Finally, we need to say a word about what complete programs in our language are. Since we use an object-capability style to control the growth of the dataflow graph, a user program must be a function type, so that it can receive a capability as an argument. Furthermore, since alloc is not a stable type, we will need to supply a fresh capability on each tick.

For this reason, we take user programs to be closed terms of type $\mathsf{S\,alloc} \to A$. If e is such a term, then we will begin evaluation of the program with the call $e\ (\mathsf{fix}\,xs.\ \mathsf{cons}(\diamond, \delta_{\diamond}(xs)))$. The term $\mathsf{fix}\,xs.\ \mathsf{cons}(\diamond, \delta_{\diamond}(xs))$ computes into a term which produces a stream yielding an allocation token $\diamond$ at each tick, but this term cannot itself be typed under our type system, Modeling the fact that the runtime system of the language possesses certain capabilities that user programs do not.

As a result, our metatheory needs to be done in a "semantic" style, using a step-indexed Kripke logical relation, rather than showing soundness through the usual syntactic progress and preservation lemmas.

3. Examples

3.1 Stream Functions

Basic Examples We begin with the constant function on streams. It takes a natural number as an input, and returns a constant stream of that number.

```
const : S alloc → ℕ → S ℕ                                  1
const us n =                                               2
  let cons(u, δ(us′)) = us in                              3
  let stable(x) = promote(n) in                            4
  cons(x, δ_u(const us′ x))                                5
```

In this example, we have a function const which receives a stream of permissions us to allocate, and a number n on line 2. On line 3, we take the head and tail of the permission stream, with a pattern-matching-style nested delay elimination to bind us' to

$$
\begin{array}{llll}
\text{Types} & A & ::= & b \mid A \times B \mid A + B \mid A \to B \\
& & \mid & \bullet A \mid \Box A \mid \hat{\mu}\alpha.\, A \mid S\, A \mid \mathsf{alloc} \\
\text{Terms} & e & ::= & \mathsf{fst}\, e \mid \mathsf{snd}\, e \mid (e, e') \\
& & \mid & \mathsf{inl}\, e \mid \mathsf{inr}\, e \mid \mathsf{case}(e, \mathsf{inl}\, x \to e', \mathsf{inr}\, y \to e'') \\
& & \mid & \lambda x.\, e \mid e\ e' \\
& & \mid & \delta_{e'}(e) \mid \mathsf{let}\ \delta(x) = e\ \mathsf{in}\ e' \\
& & \mid & \mathsf{stable}(e) \mid \mathsf{let\ stable}(x) = e\ \mathsf{in}\ e' \\
& & \mid & \mathsf{into}\, e \mid \mathsf{out}\, e \\
& & \mid & \mathsf{cons}(e, e') \mid \mathsf{let\ cons}(x, xs) = e\ \mathsf{in}\ e' \\
& & \mid & \mathsf{fix}\, x.\, e \mid x \mid \mathsf{promote}(e) \\
& & \mid & l \mid\ !l \mid \diamond \\
\text{Values} & v & ::= & (v, v') \mid \mathsf{inl}\, v \mid \mathsf{inr}\, v' \mid \lambda x.\, e \\
& & \mid & l \mid \diamond \mid \mathsf{stable}(v) \mid \mathsf{into}\, v \mid \mathsf{cons}(v, v') \\
\text{Stores} & \sigma & ::= & \cdot \mid \sigma, l : v\ \mathsf{now} \mid \sigma, l : e\ \mathsf{later} \mid \sigma, l : \mathsf{null}
\end{array}
$$

Figure 1. Syntax

$$
\begin{array}{llll}
\text{Qualifiers} & q & ::= & \mathsf{now} \mid \mathsf{stable} \mid \mathsf{later} \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : A\ q
\end{array}
$$

$$
\begin{array}{lcl}
(\cdot)^{\bullet} & = & \cdot \\
(\Gamma, x : A\ \mathsf{later})^{\bullet} & = & \Gamma^{\bullet}, x : A\ \mathsf{now} \\
(\Gamma, x : A\ \mathsf{stable})^{\bullet} & = & \Gamma^{\bullet}, x : A\ \mathsf{stable} \\
(\Gamma, x : A\ \mathsf{now})^{\bullet} & = & \Gamma^{\bullet} \\
\\
(\cdot)^{\Box} & = & \cdot \\
(\Gamma, x : A\ \mathsf{stable})^{\Box} & = & \Gamma^{\Box}, x : A\ \mathsf{stable} \\
(\Gamma, x : A\ \mathsf{later})^{\Box} & = & \Gamma^{\Box} \\
(\Gamma, x : A\ \mathsf{now})^{\Box} & = & \Gamma^{\Box}
\end{array}
$$

Figure 2. Hypotheses, Contexts and Operations on Them

$\boxed{\Gamma \vdash e : A\ q}$

$$\frac{\Gamma \vdash e : A\ \mathsf{now} \qquad \Gamma \vdash e' : B\ \mathsf{now}}{\Gamma \vdash (e, e') : A \times B\ \mathsf{now}}\ \times\mathrm{I}$$

$$\frac{\Gamma \vdash e : A \times B\ \mathsf{now}}{\Gamma \vdash \mathsf{fst}\, e : A\ \mathsf{now}}\ \times\mathrm{LE} \qquad \frac{\Gamma \vdash e : A \times B\ \mathsf{now}}{\Gamma \vdash \mathsf{snd}\, e : B\ \mathsf{now}}\ \times\mathrm{RE}$$

$$\frac{\Gamma \vdash e : A\ \mathsf{now}}{\Gamma \vdash \mathsf{inl}\, e : A + B\ \mathsf{now}}\ +\mathrm{LI} \qquad \frac{\Gamma \vdash e : B\ \mathsf{now}}{\Gamma \vdash \mathsf{inr}\, e : A + B\ \mathsf{now}}\ +\mathrm{RI}$$

$$\frac{\Gamma \vdash e : A + B\ \mathsf{now} \qquad \Gamma, x : A\ \mathsf{now} \vdash e' : C\ \mathsf{now} \qquad \Gamma, y : B\ \mathsf{now} \vdash e'' : C\ \mathsf{now}}{\Gamma \vdash \mathsf{case}(e, \mathsf{inl}\, x \to e', \mathsf{inr}\, y \to e'') : C\ \mathsf{now}}\ +\mathrm{E}$$

$$\frac{\Gamma, x : A\ \mathsf{now} \vdash e : B\ \mathsf{now}}{\Gamma \vdash \lambda x.\, e : A \to B\ \mathsf{now}}\ \to\mathrm{I}$$

$$\frac{\Gamma \vdash e : A \to B\ \mathsf{now} \qquad \Gamma \vdash e' : A\ \mathsf{now}}{\Gamma \vdash e\ e' : B\ \mathsf{now}}\ \to\mathrm{E}$$

Figure 3. Standard Typing Rules

$\boxed{\Gamma \vdash e : A\ q\ \text{continued}}$

$$\frac{\Gamma \vdash e : A\ \mathsf{later} \qquad \Gamma \vdash e' : \mathsf{alloc}\ \mathsf{now}}{\Gamma \vdash \delta_{e'}(e) : \bullet A\ \mathsf{now}}\ \bullet\mathrm{I}$$

$$\frac{\Gamma \vdash e : \bullet A\ \mathsf{now} \qquad \Gamma, x : A\ \mathsf{later} \vdash e' : C\ \mathsf{now}}{\Gamma \vdash \mathsf{let}\ \delta(x) = e\ \mathsf{in}\ e' : C\ \mathsf{now}}\ \bullet\mathrm{E}$$

$$\frac{\Gamma \vdash e : [\bullet(\hat{\mu}\alpha.\, A)/\alpha]A\ \mathsf{now}}{\Gamma \vdash \mathsf{into}\, e : \hat{\mu}\alpha.\, A\ \mathsf{now}}\ \mu\mathrm{I}$$

$$\frac{\Gamma \vdash e : \hat{\mu}\alpha.\, A\ \mathsf{now}}{\Gamma \vdash \mathsf{out}\, e : [\bullet(\hat{\mu}\alpha.\, A)/\alpha]A\ \mathsf{now}}\ \mu\mathrm{E}$$

$$\frac{\Gamma \vdash e : A\ \mathsf{stable}}{\Gamma \vdash \mathsf{stable}(e) : \Box A\ \mathsf{now}}\ \Box\mathrm{I}$$

$$\frac{\Gamma \vdash e : \Box A\ \mathsf{now} \qquad \Gamma, x : A\ \mathsf{stable} \vdash e' : C\ \mathsf{now}}{\Gamma \vdash \mathsf{let\ stable}(x) = e\ \mathsf{in}\ e' : C\ \mathsf{now}}\ \Box\mathrm{E}$$

$$\frac{\Gamma \vdash e : A\ \mathsf{now} \qquad A\ \mathsf{stable}}{\Gamma \vdash \mathsf{promote}(e) : \Box A\ \mathsf{now}}\ \textsc{Promote}$$

$$\frac{\Gamma \vdash e : A\ \mathsf{now} \qquad \Gamma \vdash e' : \bullet(S\, A)\ \mathsf{now}}{\Gamma \vdash \mathsf{cons}(e, e') : S\, A\ \mathsf{now}}\ \mathrm{SI}$$

$$\frac{\Gamma \vdash e : S\, A\ \mathsf{now} \qquad \Gamma, x : A\ \mathsf{now}, xs : \bullet(S\, A)\ \mathsf{now} \vdash e' : C\ \mathsf{now}}{\Gamma \vdash \mathsf{let\ cons}(x, xs) = e\ \mathsf{in}\ e' : C\ \mathsf{now}}\ \mathrm{SE}$$

$$\frac{\Gamma^{\Box}, x : A\ \mathsf{later} \vdash e : A\ \mathsf{now}}{\Gamma \vdash \mathsf{fix}\, x.\, e : A\ \mathsf{now}}\ \textsc{Fix}$$

$$\frac{x : A\ q \qquad q \in \{\mathsf{now}, \mathsf{stable}\}}{\Gamma \vdash x : A\ \mathsf{now}}\ \textsc{Hyp}$$

$$\frac{\Gamma^{\Box} \vdash e : A\ \mathsf{now}}{\Gamma \vdash e : A\ \mathsf{stable}}\ \textsc{TStable} \qquad \frac{\Gamma^{\bullet} \vdash e : A\ \mathsf{now}}{\Gamma \vdash e : A\ \mathsf{later}}\ \textsc{TLater}$$

Figure 4. Typing

$\boxed{A\ \mathsf{stable}}$

$$\frac{A\ \mathsf{stable} \qquad B\ \mathsf{stable}}{A \times B\ \mathsf{stable}} \qquad \frac{A\ \mathsf{stable} \qquad B\ \mathsf{stable}}{A + B\ \mathsf{stable}}$$

$$\frac{}{b\ \mathsf{stable}} \qquad \frac{}{\Box A\ \mathsf{stable}}$$

Figure 5. Stability of Types

$$\boxed{\langle \sigma; e\rangle \Downarrow \langle \sigma'; v\rangle}$$

$$\frac{}{\langle \sigma; v\rangle \Downarrow \langle \sigma; v\rangle}$$

$$\frac{\langle \sigma; e_1\rangle \Downarrow \langle \sigma'; v_1\rangle \qquad \langle \sigma'; e_2\rangle \Downarrow \langle \sigma''; v_2\rangle}{\langle \sigma; (e_1, e_2)\rangle \Downarrow \langle \sigma''; (v_1, v_2)\rangle}$$

$$\frac{\langle \sigma; e\rangle \Downarrow \langle \sigma'; (v_1, v_2)\rangle}{\langle \sigma; \mathsf{fst}\, e\rangle \Downarrow \langle \sigma'; v_1\rangle} \qquad \frac{\langle \sigma; e\rangle \Downarrow \langle \sigma'; (v_1, v_2)\rangle}{\langle \sigma; \mathsf{snd}\, e\rangle \Downarrow \langle \sigma'; v_2\rangle}$$

$$\frac{\langle \sigma; e\rangle \Downarrow \langle \sigma'; v\rangle}{\langle \sigma; \mathsf{inl}\, e\rangle \Downarrow \langle \sigma'; \mathsf{inl}\, v\rangle} \qquad \frac{\langle \sigma; e\rangle \Downarrow \langle \sigma'; v\rangle}{\langle \sigma; \mathsf{inr}\, e\rangle \Downarrow \langle \sigma'; \mathsf{inr}\, v\rangle}$$

$$\frac{\langle \sigma; e\rangle \Downarrow \langle \sigma'; \mathsf{inl}\, v\rangle \qquad \langle \sigma'; [v/x]e'\rangle \Downarrow \langle \sigma''; v''\rangle}{\langle \sigma; \mathsf{case}(e, \mathsf{inl}\, x \to e', \mathsf{inr}\, y \to e'')\rangle \Downarrow \langle \sigma''; v''\rangle}$$

$$\frac{\langle \sigma; e\rangle \Downarrow \langle \sigma'; \mathsf{inr}\, v\rangle \qquad \langle \sigma'; [v/y]e''\rangle \Downarrow \langle \sigma''; v''\rangle}{\langle \sigma; \mathsf{case}(e, \mathsf{inl}\, x \to e', \mathsf{inr}\, y \to e'')\rangle \Downarrow \langle \sigma''; v''\rangle}$$

$$\frac{\begin{array}{c}\langle \sigma; e_1\rangle \Downarrow \langle \sigma'; \lambda x.\ e_1'\rangle \\ \langle \sigma'; e_2\rangle \Downarrow \langle \sigma''; v_2\rangle \qquad \langle \sigma''; [v_2/x]e_1'\rangle \Downarrow \langle \sigma'''; v\rangle\end{array}}{\langle \sigma; e_1\ e_2\rangle \Downarrow \langle \sigma'''; v\rangle}$$

$$\frac{\langle \sigma; e'\rangle \Downarrow \langle \sigma'; \diamond\rangle \qquad l \notin \mathrm{dom}(\sigma')}{\langle \sigma; \delta_{e'}(e)\rangle \Downarrow \langle (\sigma', l : e\ \mathsf{later}); l\rangle}$$

$$\frac{\langle \sigma; e\rangle \Downarrow \langle \sigma'; l\rangle \qquad \langle \sigma'; [!l/x]e'\rangle \Downarrow \langle \sigma''; v\rangle}{\langle \sigma; \mathsf{let}\ \delta(x) = e\ \mathsf{in}\ e'\rangle \Downarrow \langle \sigma''; v\rangle} \qquad \frac{l : v\ \mathsf{now} \in \sigma}{\langle \sigma; !l\rangle \Downarrow \langle \sigma; v\rangle}$$

$$\frac{\langle \sigma; e\rangle \Downarrow \langle \sigma'; v\rangle}{\langle \sigma; \mathsf{into}\, e\rangle \Downarrow \langle \sigma'; \mathsf{into}\, v\rangle} \qquad \frac{\langle \sigma; e\rangle \Downarrow \langle \sigma'; \mathsf{into}\, v\rangle}{\langle \sigma; \mathsf{out}\, e\rangle \Downarrow \langle \sigma'; v\rangle}$$

$$\frac{\langle \sigma; e\rangle \Downarrow \langle \sigma'; v\rangle}{\langle \sigma; \mathsf{stable}(e)\rangle \Downarrow \langle \sigma'; \mathsf{stable}(v)\rangle}$$

$$\frac{\langle \sigma; e\rangle \Downarrow \langle \sigma'; v\rangle}{\langle \sigma; \mathsf{promote}(e)\rangle \Downarrow \langle \sigma'; \mathsf{stable}(v)\rangle}$$

$$\frac{\langle \sigma; e\rangle \Downarrow \langle \sigma'; \mathsf{stable}(v)\rangle \qquad \langle \sigma'; [v/x]e'\rangle \Downarrow \langle \sigma''; v''\rangle}{\langle \sigma; \mathsf{let}\ \mathsf{stable}(x) = e\ \mathsf{in}\ e'\rangle \Downarrow \langle \sigma''; v''\rangle}$$

$$\frac{\langle \sigma; e\rangle \Downarrow \langle \sigma'; v\rangle \qquad \langle \sigma'; e'\rangle \Downarrow \langle \sigma''; v'\rangle}{\langle \sigma; \mathsf{cons}(e, e')\rangle \Downarrow \langle \sigma''; \mathsf{cons}(v, v')\rangle}$$

$$\frac{\langle \sigma; e\rangle \Downarrow \langle \sigma'; \mathsf{cons}(v, l)\rangle \qquad \langle \sigma'; [v/x, l/xs]e'\rangle \Downarrow \langle \sigma''; v''\rangle}{\langle \sigma; \mathsf{let}\ \mathsf{cons}(x, xs) = e\ \mathsf{in}\ e'\rangle \Downarrow \langle \sigma''; v''\rangle}$$

$$\frac{\langle \sigma; [\mathsf{fix}\, x.\ e/x]e\rangle \Downarrow \langle \sigma'; v\rangle}{\langle \sigma; \mathsf{fix}\, x.\ e\rangle \Downarrow \langle \sigma'; v\rangle}$$

Figure 6. Expression Semantics

$$\boxed{\sigma \Longrightarrow \sigma'}$$

$$\frac{}{\cdot \Longrightarrow \cdot} \qquad \frac{\sigma \Longrightarrow \sigma' \qquad \langle \sigma'; e\rangle \Downarrow \langle \sigma''; v\rangle \qquad l \notin \mathrm{dom}(\sigma'')}{\sigma, l : e\ \mathsf{later} \Longrightarrow \sigma'', l : v\ \mathsf{now}}$$

$$\frac{\sigma \Longrightarrow \sigma' \qquad l \notin \mathrm{dom}(\sigma')}{\sigma, l : v\ \mathsf{now} \Longrightarrow \sigma', l : \mathsf{null}} \qquad \frac{\sigma \Longrightarrow \sigma' \qquad l \notin \mathrm{dom}(\sigma')}{\sigma, l : \mathsf{null} \Longrightarrow \sigma', l : \mathsf{null}}$$

Figure 7. Tick Semantics

a later variable. On line 4, we make use of the fact that natural numbers are *stable* in order to rebind n to the stable variable x. This lets us refer to x in both the head and tail of the cons-cell on line 5. The head of the cons cell is just x itself, and the tail is a delayed stream, which we may allocate since we have a permission u.

Below, we give a summation function, which takes a stream of numbers and returns a stream containing the cumulative sum of the stream. To implement this, we introduce an auxiliary function sum_acc which stores the sum in an accumulator variable, and then call it with an initial sum of 0.

```
sum_acc : S alloc → S ℕ → ℕ → S ℕ                                  1
sum_acc us ns acc =                                                2
  let cons(u, δ(us′)) = us in                                      3
  let cons(n, δ(ns′)) = ns in                                      4
  let stable(x) = promote(n + acc) in                              5
  cons(x, δ_u(sum_acc us′ ns′ x))                                  6

sum : S alloc → S ℕ → S ℕ                                          7
sum us ns = sum_acc us ns 0                                        8
```

Higher-Order Stream Operations Our language permits the free use of streams of streams, as well as higher-order functions on streams. We illustrate this with a simple function which takes a stream, and returns the stream of successive tails of that stream.

```
tails : S alloc → S A → S (S A)                                    1
tails us xs =                                                      2
  let cons(u, δ(us′)) = us in                                      3
  let cons(x, δ(xs′)) = xs in                                      4
  cons(xs, δ_u(tails us′ xs′))                                     5
```

The higher-order map functional is definable as follows:

```
map : S alloc → □(A → B) → S A → S B                               1
map us h xs =                                                      2
  let cons(u, δ(us′)) = us in                                      3
  let cons(x, δ(xs′)) = xs in                                      4
  let stable(f) = h in                                             5
  cons(f x, δ_u(map us′ stable(f) xs′))                            6
```

Note that the map functional calls its argument function on every element of the input stream. As a result, we need to use the function at many different time steps, and so we need to give the functional argument the stable type $\square(A \to B)$ to ensure that we can safely use it both now and later.

The fact that all computable streams are definable in our language is witnessed by giving the unfold operation, which shows that the universal property of streams is definable within the language.

```
unfold : S alloc → □(X → A × •X) → X → S A                         1
unfold us h x =                                                    2
  let cons(u, δ(us′)) = us in                                      3
  let stable(f) = h in                                             4
  let (a, δ(x′)) = f x in                                          5
  cons(a, δ_u(unfold us′ stable(f) x′))                            6
```

Dynamic Changes of Streams Switching behavior is directly programmable in our language. The swap function below takes a number n, and two streams xs and ys. It yields the same values as xs on the first n time steps, and afterward the same values as ys.

```
swap : S alloc → ℕ → S A → S A → S A                                  1
swap us n xs ys =                                                     2
  if n = 0 then                                                       3
    ys                                                                4
  else                                                                5
    let cons(u, δ(us′)) = us in                                       6
    let cons(x, δ(xs′)) = xs in                                       7
    let cons(y, δ(ys′)) = ys in                                       8
    let stable(m) = promote(n) in                                     9
    cons(x, δ_u(swap us′ (m − 1) xs′ ys′))                            10
```

What is interesting about this example is that there is nothing surprising about it: it is basically an ordinary functional program. Unlike many other approaches to reactive programming, we program conditional behavior with ordinary conditional control flow.

3.2 Type Encodings

Streams In order to make the examples more readable, we included streams as a primitive type in our language, but that is actually unnecessary, since they are encodable using recursive types:

$$\mathsf{S}\,A \triangleq \hat{\mu}\alpha.\ A \times \alpha$$

This encoding looks completely conventional, but because unfolding a recursive type requires guarding the recursive type with a delay modality before substituting, this type is isomorphic to:

$$\mathsf{S}\,A \simeq A \times \bullet(A \times \bullet(A \times \ldots))$$

From a logical perspective, the stream type $\mathsf{S}\,A$ corresponds to the "always" operator of temporal logic, which asserts that at every time step, we always have a witness to the proposition A.

Events In reactive programs, there are often operations which take a length of time observable to the user. For example, downloading a file is an operation which can take many ticks to happen, and the file value is not available for use until the operation is complete.

Such operations, which we call "events", are surprisingly difficult to model in a stream-based paradigm, since nothing happens until the completion of the process, and once the final event happens, no more events occur. Stream-based languages have used a number of tricks to encode behaviours like this, but when we have a temporal recursive type, it is straightforward to define a type of events:

$$\mathsf{E}\,A \triangleq \hat{\mu}\alpha.\ A + \alpha$$

Here, we simply replace the product with a sum, which means that an element of type A is either immediately available, or we have to wait until the next tick to try again. (That is, $\mathsf{E}\,A \simeq A + \bullet(\mathsf{E}\,A)$.)Events make implementing dynamic switching behavior very natural.

```
switch : S alloc → S A → E (S A) → S A                                1
switch us xs e =                                                      2
  let cons(u, δ(us′)) = us in                                         3
  let cons(x, δ(xs′)) = xs in                                         4
  case(out e,                                                         5
       inl ys → ys,                                                   6
       inr t → let δ(e′) = t in                                       7
               cons(x, δ_u(switch us′ xs′e′)))                        8
```

In this example, the call switch us xs e behaves like xs until the event e returns a stream ys, and then it behaves like ys.

Events correspond to the "eventually" operator of temporal logic. Since the eventually operator forms a closure operator on the Kripke structure of times, the event type constructor correspondingly forms a monad, whose monadic operations may be defined as follows:

```
return : A → E A                                                      1
return x = into (inl x)                                               2

bind : S alloc → □(A → E B) → E A → E B                               3
bind us h e =                                                         4
  let cons(u, δ(us′)) = us in                                         5
  let stable(f) = h in                                                6
  case(out e,                                                         7
       inl a → f a,                                                   8
       inr t → let δ(e′) = t in                                       9
               into (inr (δ_u(bind us′ stable(f) e′))))               10
```

Here, we perform sequencing by taking a function that maps elements of A to B events, and waiting until an A-event yields an A to apply the function. Because we do not know when we will need to invoke the function, bind requires it to be stable.

Events are closely related to promises and futures [4, 16], in that they are proxies for computations which have not yet completed constructing a value. However, unlike most implementations of futures, our type $\mathsf{E}\,A$ permits clients to test whether or not the computation has finished. Because we embed our events into a synchronous programming language, this choice does not introduce nondeterminism into our programming language.

Until Just as the always and eventually operators in temporal logic can be merged into the "until" operator, we can also give a combined operation for processes that produce values of type A until they terminate, producing a value of type B.

$$A\,\mathcal{U}\,B \triangleq \hat{\mu}\alpha.\ B + (A \times \alpha)$$

Note that streams correspond to producing elements of A until the empty type, $\mathsf{S}\,A \simeq A\,\mathcal{U}\,0$, and events correspond to yielding units until B, $\mathsf{E}\,B \simeq 1\,\mathcal{U}\,B$.

Resumptions The fact that we have function spaces and arbitrary recursive types enables us to go well beyond the expressive power of temporal logic. In this example, we will illustrate this by showing how resumptions [37], a simple interleaving model of concurrency, may be encoded in our type system.

$$\mathsf{R}_{I,O}\,A \triangleq \hat{\mu}\alpha.\ \alpha \times (A + (I \to O \times \alpha))$$

Elements of the type $\mathsf{R}_{I,O}\,A$ can be thought of as representations of thread values, in an interleaving model of concurrency. The left-hand-side of the pair $\alpha \times (A + (I \to O \times \alpha))$ can be seen as what to do when the thread is not executed on this tick, and the right-hand-side says that either the thread finishes execution with a value of A, or it takes an input message of type I and yields an output message of type O, and continues computing.

Given two threads, we can implement a scheduler which takes two threads and interleaves their execution until one of them finishes.

```
par : S alloc → R_{I,O} A × R_{I,O} A → R_{I,O} A                     1
par us (p₁, p₂) =                                                     2
  let cons(u, δ(us′)) = us in                                         3
  case((snd (out p₁),                                                 4
       inl a → p₁,                                                    5
       inr f → let δ(p₁′) = fst (out p₁) in                           6
               let δ(p₂′) = fst (out p₂) in                           7
               let p′ = δ_u(par us′ p₁′ p₂′) in                       8
               let f′ = λi. let (o, δ(p₁″)) = f i in                  9
                        into (o, δ_u(par us′ p₂′ p₁″)) in             10
               into (p′, inr f′))                                     11
```

One line 5, we see that if the first process p_1 completes on this tick, then the parallel composition completes on this tick. If p_1 does not complete, then we need to return (1) how to defer the parallel composition, and (2) the I/O behavior of the parallel composition. We construct the deferred computation for the parallel composition

by taking (on lines 6-8) the deferred computations for each process individually, and then resuming their parallel composition on the next tick. On line 9-10, we define the function f', which produces the same I/O as p_1 on this tick, defers p_2 to the next tick, and then on the next tick schedules p_2 for execution and defers p_1's next action.

Definability of Fixed Points Our calculus has a term-level fixed point operator $\mathsf{fix}\, x.\ e$. However, fixed points are *definable*, illustrating the high expressiveness that guarded recursive types permit. We will show the inhabitation of the type $\Box(\bullet A \to A) \to \mathsf{S}\,\mathsf{Alloc} \to A$, in three steps. First, we define the recursive type $X \triangleq \hat{\mu}\alpha.\ \Box(\mathsf{S}\,\mathsf{alloc} \to \alpha \to A)$. We use this type to define the operation selfapp, which takes an element of type X and applies it to itself, wrapped around a call to a function $f : \bullet A \to A$:

```
selfapp : (•A → A) → S alloc → X → A                          1
selfapp f us v =                                              2
  let cons(u, δ(us')) = us in                                 3
  let stable(w) = out v in                                    4
  f (δu(w us' (into (stable w))))                             5
```

Next, we can use this self-application function to implement a fixed-point combinator.

```
fixedpoint : □(•A → A) → S alloc → A                          1
fixedpoint h us =                                             2
  let stable(f) = h in                                        3
  selfapp f us (into (stable(selfapp f)))                     4
```

This fixed point operator is essentially a variant of Curry's fixed point combinator [11] $Y \triangleq \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$, with extra noise to deal with the modal operators and iso-recursive types. The most significant difference is that we need to additionally pass in a stream of allocation tokens to ensure that we can construct the necessary delay thunks.

3.3 Blocking Space Leaks Without Ruling Out Buffering

Our operational semantics makes it impossible to implicitly retain values across multiple time ticks, and our type systems statically rejects programs which try. For example, if we try to program a function which takes a stream and returns that stream constantly, then it fails to typecheck, as we desire:

```
scary_const : S alloc → S ℕ → S (S ℕ)                         1
scary_const us ns =                                           2
  let cons(u, δ(us')) = us in                                 3
  let stable(xs) = promote(ns) in  — TYPE ERROR               4
  cons(xs, δu(scary_const us' xs))                            5
```

The reason for this error is that we need to use the argument stream at multiple times, and since streams are not a stable type, we cannot promote them into a stable variable, which we need in order to use the stream value at multiple times. So we get a compile-time error.

We blocked this function definition because implementing it would require potentially unbounded buffering, and we do not want to do that implicitly, since that would mean that variable references could create unexpected memory leaks.

However, there are many legitimate programs which need to retain data across multiple time steps: for example, we may wish to retain data to compute a moving average. Our language does not *prohibit* these kinds of programs; instead, it demands that programmers *explicitly* write all the buffering code.

As an extreme (and somewhat ridiculous) example, we will write the function scary_const, which takes a stream argument and repeats it constantly.

```
buffer : S alloc → ℕ → S ℕ → S ℕ                              1
buffer us n xs =                                              2
  let cons(u, δ(us')) = us in                                 3
  let cons(x, δ(xs')) = xs in                                 4
  let stable(x') = promote(x) in                              5
  cons(n, δu(buffer us' x' xs'))                              6

forward : S alloc → S ℕ → •(S ℕ)                              7
forward us xs =                                               8
  let cons(u, δ(us')) = us in                                 9
  let cons(x, δ(xs')) = xs in                                 10
  let stable(x') = promote(x) in                              11
  δu(buffer us' x' xs')                                       12

scary_const : S alloc → S ℕ → S (S ℕ)                         13
scary_const us xs =                                           14
  let cons(u, δ(us')) = us in                                 15
  let δ(xs') = forward us xs in                               16
  cons(xs, δu(scary_const us' xs'))                           17
```

In this example, we use a function buffer, which appends a natural number to the head of a stream, and then use buffer to define a function forward, which pushes its argument one tick into the future, and then define scary_const, which repeatedly calls forward to keep moving the argument one tick into the future.

This definition makes the memory leak explicit in the source code: our program repeatedly calls forward on the argument stream to scary_const, using more memory each time.

In general, it is possible to define buffering for a recursive type $\hat{\mu}\alpha.\ A$, if the expression A is constructed from the variable α, sums of bufferable types $A + B$, products of bufferable types $A \times B$, any stable type $\Box A$, or delays of bufferable types $\bullet A$. It is not possible to buffer arbitrary members of function types $A \to B$, because the environment of a function closure cannot be examined.

4. Metatheory

Overview Kripke logical relations have a long history in giving semantics to higher-order stateful languages [2, 13, 36]. Since our dynamic dataflow graph can be viewed as a store, they are a natural tool for showing the soundness of our type system.

The basic idea behind the technique of logical relations is to give a family of sets of closed terms $[\![A]\!]$ by induction on the structure of the type A, each of which possesses the soundness property we desire. Then, we prove a theorem (the fundamental property) that shows that every well-typed term $e : A$ lies within the set $[\![A]\!]$, and from that we can conclude that the language is sound.

In a Kripke logical relation, in addition to the type, the relations are also indexed by some contextual information, the *world*, which is used to relativize the interpretation of each type. In our setting, this contextual information will include the store terms are to evaluate under, as well as the permission information telling us whether the term may extend the dataflow graph.

Recursive types make defining relations by induction on the syntax of a type difficult, since semantically we expect a recursive type to be defined in terms of its unfolding, and unfolding a recursive type can make it larger. To resolve this issue, we make use of the idea of *step-indexing*, originally introduced by Appel and McAllester [3], in which we extend the world with a natural number n, and interpret a recursive type only at strictly smaller numbers.

In this section, we give a high-level overview of our definitions, and a brief tour of the soundness proof. We give the full proof in the technical report [25] provided in the supplementary material.

Supportedness and Location Permutations Before we can describe the structure of our logical relation, there is one technical issue we need to discuss. The allocation rule in the expression semantics is non-deterministic – it chooses a location that is not in the current heap. However, the tick semantics $\sigma \Longrightarrow \sigma'$, when given

$$\boxed{e \sqsubseteq \sigma}$$

$$e \sqsubseteq \sigma \triangleq \text{free locations of } e \subseteq \text{dom}(\sigma)$$

$$\boxed{\sigma \text{ supported}}$$

$$\frac{}{\cdot \text{ supported}} \qquad \frac{\sigma \text{ supported} \qquad v \sqsubseteq \sigma}{\sigma, l : v \text{ now supported}}$$

$$\frac{\sigma \text{ supported} \qquad e \sqsubseteq \sigma}{\sigma, l : e \text{ later supported}} \qquad \frac{\sigma \text{ supported}}{\sigma, l : \text{null supported}}$$

Figure 8. Definition of Supportedness

a nonempty store, removes the most recently-allocated location l from the store, updates the older heap, and then updates the removed location. The technical question is: what happens if the update of the older heap allocates l itself?

Intuitively, this is not a serious problem, since our allocator could always have chosen a different location to have allocated. To formalize this intuition, we introduce the idea of *supportedness*, described in Figure 8. We write $e \sqsubseteq \sigma$ if all of the free locations in e are in the domain of σ. We write σ supported, when every pointer containing an expression or value is supported by the heap cells allocated earlier (that is, to the left in the list). Then, we can prove the following three lemmas.

Lemma 1 (Permutability)**.** *We have that:*

1. *If* $\pi \in \text{Perm}$ *and* $\langle \sigma; e \rangle \Downarrow \langle \sigma'; v \rangle$ *then* $\langle \pi(\sigma); \pi(e) \rangle \Downarrow \langle \pi(\sigma'); \pi(v) \rangle$.
2. *If* $\pi \in \text{Perm}$ *and* $\sigma \Longrightarrow \sigma'$ *then* $\pi(\sigma) \Longrightarrow \pi(\sigma')$.

Lemma 2 (Supportedness)**.** *We have that:*

1. *If* σ supported *and* $e \sqsubseteq \sigma$ *and* $\langle \sigma; e \rangle \Downarrow \langle \sigma'; v \rangle$ *then* $v \sqsubseteq \sigma'$ *and* σ' supported.
2. *If* σ supported *and* $\sigma \Longrightarrow \sigma'$ *then* σ' supported.

Lemma 3 (Quasi-determinacy)**.** *We have that:*

1. *If* $\langle \sigma; e \rangle \Downarrow \langle \sigma'; v' \rangle$ *and* $\langle \sigma; e \rangle \Downarrow \langle \sigma''; v'' \rangle$ *and* σ supported *and* $e \sqsubseteq \sigma$*, then there is a* $\pi \in \text{Perm}$ *such that* $\pi(\sigma') = \sigma''$ *and* $\pi(\sigma) = \sigma$.
2. *If* $\sigma \Longrightarrow \sigma'$ *and* $\sigma \Longrightarrow \sigma''$ *and* σ supported*, then there is a* $\pi \in \text{Perm}$ *such that* $\pi(\sigma') = \sigma''$ *and* $\pi(\sigma) = \sigma$.

Here, Perm is the set of finite permutations on locations, and $\pi(e)$ and $\pi(\sigma)$ lift it to expressions and stores in the obvious way. Together, these lemmas imply that we can rename locations however we like, and that the nondeterminism of the allocator can *only* affect how locations are named.

Kripke Worlds We now describe the structure of the worlds we use in our logical relation, which we lay out in Figure 9.

A world w is a triple (n, σ, a). Here, n is a step index, indicating that an element of this relation must be good for at least n ticks into the future. The term a is a *capability*. The capability $\top$ means that we *do not* have the permission to extend the dataflow graph, and $\bot$ means that we *do* have the permission to extend the graph. Finally, the store σ must be an element of Heap_n, which is the set of supported heaps for which the tick relation is defined for at least n steps. That is, if $\sigma \in \text{Heap}_n$, then there are $\sigma_1, \ldots, \sigma_n$ such that $\sigma \Longrightarrow \sigma_1 \Longrightarrow \cdots \Longrightarrow \sigma_n$. As a notational shorthand, we write $w.n$ for the step index of w, $w.\sigma$ for the store component of w, and $w.a$ for the capability of w. We also write $\pi(w)$ to mean the world $(w.n, \pi(w.\sigma), w.a)$.

In most applications, step-indexing has been used purely as a technical device to force the inductive definition of types to be well-founded. In our setting, in contrast, steps have a concrete operational reading, corresponding directly to the passage of time: a step index of n tells us that the tick relation can definitely tick at least n times.

Each of these components has an associated preorder $\leq$. A step index n' is below an index n, if n' is less than or equal to n. A heap σ' is below a heap σ, $\sigma' \leq \sigma$, if σ' is an extension of σ – that is, if σ' has a larger domain than σ, and agrees with it on the overlap. Finally, a capability a' is below a, if either they are the same, or a' is $\bot$ and a is $\top$.

The intuition behind the order on worlds is that $w' \leq w$ when w' is a possible future state of w. In the future of w, we may have extended the dataflow graph, or we can potentially receive a capability to allocate from our environment.

The Logical Relations In Figure 10, we define three logical relations.

The set $\mathcal{V}[\![A]\!]\,\rho\,w$ defines the *value relation*, the set of closed values semantically inhabiting the type A at the world w, with the parameter ρ giving the type interpretation of each of the free variables in A. The set $\mathcal{E}[\![A]\!]\,\rho\,w$ gives the *expression relation*, the set of expressions semantically inhabiting the type A (that is, they will evaluate to a value of type A if they are run on the current tick). Similarly, we also define $\mathcal{L}[\![A]\!]\,\rho\,w$, the *later expression relation*. These are the expressions that will evaluate to a value of type A if they are run on the *next* time step.

Both the expression relation $\mathcal{E}[\![A]\!]\,\rho\,w$ and the later relation $\mathcal{L}[\![A]\!]\,\rho\,w$ are defined in terms of the value relation. The expression relation consists of closed, supported expressions, which evaluate to values in the value relation. Furthermore, the expression evaluation may extend the heap only if the world contains the capability to allocate. If it does not, then the store will be untouched.

The later relation $\mathcal{L}[\![A]\!]\,\rho\,w$ is defined by cases. If the world's step index is 0, then we place no constraints on it – it is simply the set of closed, supported expressions. If the world's step index is $n+1$ and the tick relation sends the current world's store to σ', then terms are in $\mathcal{L}[\![A]\!]\,\rho\,w$ if they are in the expression relation of A, at step n and store σ'. That is, it consists of those expressions that will be in the expression relation *on the next tick*.

The value relation is defined by induction on the syntax of the type A. Matters first become interesting in the function case. First, we require that a lambda term $\lambda x.\ e$ inhabiting $\mathcal{V}[\![A \to B]\!]\,\rho\,w$ be *supported* with respect to the heap component of the world. As is usual, we quantify over all future worlds $w' \leq w$, but we also quantify over *location permutations* $\pi \in \text{Perm}$.

Then, we consider all arguments e' coming from the A expression relation at the permuted world $\pi(w')$, and assert that applying the term to the renamed function should also be in the B expression relation at $\pi(w')$. The extra renaming requirement semantically formalizes the idea that we need to ignore the exact choice of names (and indeed is very similar to the definition of the function space in nominal set theory [17]).

The later type $\bullet A$ is interpreted as the set of pointers l, which point to an expression $l : e$ later where e is in the later relation of A. As with functions, we quantify over future worlds and renamings. The interpretation of the stability modality $\mathcal{V}[\![\Box A]\!]\,\rho\,w$ contains values $\mathsf{stable}(v)$, where $v \in \mathcal{V}[\![A]\!]\,\rho\ (w.n, \cdot, \top)$. That is, these values v are not allowed to depend upon the store, or to assume that they have the capability to extend the heap.

Recursive types $\hat{\mu}\alpha.\ A$ are interpreted by the interpretation of A, where the environment is extended by the interpretation of $\bullet(\hat{\mu}\alpha.\ A)$. Superficially, this looks like a circular definition, except that the next-step modality is defined in terms of the later relation for $\hat{\mu}\alpha.\ A$,

$$\text{Heap}_0 = \{\sigma \in \text{Store} \mid \sigma \text{ supported}\}$$

$$\text{Heap}_{n+1} = \left\{ \sigma \in \text{Store} \;\middle|\; \begin{array}{l} \sigma \text{ supported} \wedge \\ \exists \sigma'.\ \sigma \Longrightarrow \sigma' \wedge \sigma' \in \text{Heap}_n \end{array} \right\}$$

$$\text{Cap} = \{\top, \bot\}$$

$$\text{World} = \{(n, \sigma, a) \mid n \in \mathbb{N} \wedge \sigma \in \text{Heap}_n \wedge a \in \text{Cap}\}$$

$$\sigma' \leq \sigma \iff \exists \sigma_0.\ \sigma \cdot \sigma_0 = \sigma'$$

$$a' \leq a \iff a = a' \vee (a' = \bot \wedge a = \top)$$

$$(n', \sigma', a') \leq (n, \sigma, a) \iff n' \leq n \wedge \sigma' \leq \sigma \wedge a' \leq a$$

Figure 9. Worlds

which lowers the step index, making the definition well-founded. Finally, the semantic interpretation of alloc is the token $\diamond$, if the world has the capability, and is the empty set otherwise.

Soundness The key property we prove is the following:

Theorem 1 (Fundamental Property)**.** *The following properties hold:*

1. *If* $\cdot \vdash e : A$ later*, then* $e \in \mathcal{L}[\![A]\!] \cdot w$.
2. *If* $\cdot \vdash e : A$ stable*, then* $e \in \mathcal{E}[\![A]\!] \cdot (w.n, \cdot, \top)$.
3. *If* $\cdot \vdash e : A$ now*, then* $e \in \mathcal{E}[\![A]\!] \cdot w$.

The proof of this theorem follows the usual pattern for soundness proofs by logical relations. We extend the definition of the expression relation to define environments binding expressions to variables, and then prove by induction on the syntax of the typing derivation that all substitutions of well-typed terms by well-formed environments are in the logical relation. As usual, a fair number of auxiliary lemmas must be proved (such as the monotonicity of the value relation, and the stability of all relations under renaming of locations), and we refer interested readers to the companion technical report [25] for details. Once we have the fundamental property, we get a soundness property as an almost immediate corollary.

Corollary 1. *(Soundness) If* $\cdot \vdash e : A$ *, then* $\langle \cdot; e \rangle \Downarrow \langle \sigma; v \rangle$. *Furthermore, for all* n, $v \in \mathcal{V}[\![A]\!]\,\rho\,(n, \sigma, \bot)$.

Note that expression evaluation always terminates. This shows that our type discipline enforces well-founded use of fixed points $\text{fix}\,x.\ e$. Also, note that $\sigma \in \text{Heap}_n$ for any n, and so we can tick the clock as often as we like. Thus, if we compute a stream value, then each time we tick the clock, the tail pointer will point to a new cons cell, whose head contains the next value of the stream, and whose tail is the pointer to chase on the next tick.

5. Implementation

While the work described here is largely theoretical, that theory arises from an attempt to understand the correctness of a new language, AdjS (the implementation can be downloaded from the author's website), whose type system closely tracks the type system of this paper. (The major extensions are polymorphism and linear types, and a system to infer uses of the promote(e) operation.) The attempt to prove the soundness of the type system was fortuitous: in the course of the formalization we discovered two soundness bugs in our implementation!

The first soundness bug was that the AdjS compiler has both delay types $\bullet A$ and $\mathsf{S}\,A$ as primitive, but only required an allocation token to construct a stream. The second soundness bug was that we had originally treated the allocation type alloc as a promotable stable type. Each of these bugs made it possible to build a value of type $\Box(\bullet A)$, and use it to access a thunk after its lifetime has ended.

There remain a few features of the implementation which we have not completely formalized. First, the operational semantics in this paper forces every thunk when time elapses. Our actual implementation is lazier about this, only forcing thunks when they are read. It should still be the case that no thunk is forced, except on its scheduled tick, but it is additionally possible for thunks to go unforced. This is not difficult to model, but it seemed to complicate the definition of Kripke extension without sufficient expository advantages to compensate.

More seriously, we have not given a correctness proof of our implementation of linear types. While we do have both denotational and type-theoretic models of linear guarded types [26], we do not yet have a soundness proof of our implementation strategy for it, in the same way that this paper shows the soundness of our implementation strategy of the functional part of the language. This is because the linear types are used to model GUI widgets, and we need a plausible, yet tractable, operational semantics for the GUI widgets. This does not seem impossibly out of reach, though: recently Lerner et al. [29] have given a formal model of the HTML DOM, and we are currently investigating using a simplified version of their event model to build the operational semantics we need.

In order to make the synchrony hypothesis (namely, that all computations finish quickly relative to the size of a tick), our implementation runs at a fixed clock speed of 60 Hz. This does mean that the runtime wakes even when nothing is happening. However, we do not foresee many issues in increasing the clock speed, since Haskell implementations demonstrate that it is possible to sustain very high rates of thunk allocation. In contrast, Elliott [14] gives an implementation of streams using futures. In our terms, his stream type $\mathsf{S}\,A$ is $\mu\alpha.\ A \times \mathsf{E}\,\alpha$, where the recursive type is guarded by an event constructor instead of a unit delay. This means that each stream can run at a different rate, permitting the system to quiesce until events become available, at the cost of complicating the merging of streams.

6. Related Work

Implementing Reactive Programming DSLs for reactive programming languages tend to fall into one of two camps. The "purist" camp, such as Yampa [35], typically features fairly simple implementation strategies and limited dynamic behaviour, which fairly closely track the semantic model of stream programming. The "pragmatist" camp, such the Froc library for Ocaml [12], FrTime [10], and Scala.React [31], feature more sophisticated implementations based on dataflow engines and change propagation, and better support for dynamic behavior.

Now, it has long been recognized that there are connections between lazy evaluation and the synchronous dataflow paradigm [6], but the precise relationship between the two semantics has been unclear to date. Our semantics clarifies this connection, by decomposing streams into a recursive type over the next-step modality of temporal logic. We then show that thunking and lazy evaluation can be used to give *realizers* for the next-step modality, and that the synchrony assumption (that is, a global notion of time) enables *scheduling* when these thunks are forced. In other words, we show that two standard functional programming evaluation strategies – call-by-value and call-by-need – jointly supply all the computational primitives well-behaved reactive programs need.

We can also clarify what parts of the sophisticated implementations used by the pragmatist languages are necessary, and which parts are optimizations. These languages typically work by building a graph of dataflow nodes (which might be thought of as a kind of generalized spreadsheet), and incrementally recomputing values when node values change. The recomputations are guided by the dependency structure of the dataflow graph, often using quite so-

$$
\begin{array}{lcl}
\mathcal{V}\,[\![\alpha]\!]\,\rho\,w & = & \rho(\alpha)\,w \\
\mathcal{V}\,[\![A+B]\!]\,\rho\,w & = & \{\mathsf{inl}\,v \mid v \in \mathcal{V}\,[\![A]\!]\,\rho\,w\} \cup \{\mathsf{inr}\,v \mid v \in \mathcal{V}\,[\![B]\!]\,\rho\,w\} \\
\mathcal{V}\,[\![A\times B]\!]\,\rho\,w & = & \{(v_1,v_2) \mid v_1 \in \mathcal{V}\,[\![A]\!]\,\rho\,w \wedge v_2 \in \mathcal{V}\,[\![B]\!]\,\rho\,w\} \\
\mathcal{V}\,[\![A\to B]\!]\,\rho\,w & = & \left\{ \lambda x.\, e \;\middle|\; \begin{array}{l} \lambda x.\, e \sqsubseteq w.\sigma \wedge \\ \forall \pi \in \mathrm{Perm}, w' \le w, e' \in \mathcal{E}\,[\![A]\!]\,\rho\,\pi(w').\ [e'/x]\pi(e) \in \mathcal{E}\,[\![B]\!]\,\rho\,\pi(w') \end{array} \right\} \\
\mathcal{V}\,[\![\bullet A]\!]\,\rho\,w & = & \{l \mid w.\sigma = (\sigma_0, l : e\ \mathsf{later}, \sigma_1) \wedge \forall \pi \in \mathrm{Perm}, w' \le (w.n, \sigma_0, w.a).\ \pi(e) \in \mathcal{L}\,[\![A]\!]\,\rho\,\pi(w')\} \\
\mathcal{V}\,[\![\Box A]\!]\,\rho\,w & = & \{\mathsf{stable}(v) \mid v \in \mathcal{V}\,[\![A]\!]\,\rho\,(w.n, \cdot, \top)\} \\
\mathcal{V}\,[\![\hat{\mu}\alpha.\ A]\!]\,\rho\,w & = & \{\mathsf{into}\,v \mid v \in \mathcal{V}\,[\![A]\!]\,(\rho, \mathcal{V}\,[\![\bullet(\hat{\mu}\alpha.\ A)]\!]\,\rho\,w/\alpha)\,w\} \\
\mathcal{V}\,[\![\mathsf{S}\,A]\!]\,\rho\,w & = & \{\mathsf{cons}(v,v') \mid v \in \mathcal{V}\,[\![A]\!]\,\rho\,w \wedge v' \in \mathcal{V}\,[\![\bullet\mathsf{S}\,A]\!]\,\rho\,w\} \\
\mathcal{V}\,[\![\mathsf{alloc}]\!]\,\rho\,w & = & \{\diamond \mid w.a = \bot\} \\
\\
\mathcal{E}\,[\![A]\!]\,\rho\,(n,\sigma,a) & = & \left\{ e \;\middle|\; \begin{array}{l} e \sqsubseteq \sigma \wedge \\ \forall \sigma' \le \sigma.\ \exists \sigma'' \le \sigma', v \in \mathcal{V}\,[\![A]\!]\,\rho\,(n,\sigma'',a). \\ \quad \langle \sigma'; e\rangle \Downarrow \langle \sigma''; v\rangle \wedge (a = \top \implies \sigma'' = \sigma') \end{array} \right\} \\
\mathcal{L}\,[\![A]\!]\,\rho\,(0,\sigma,a) & = & \{e \in \mathrm{Expr} \mid e \sqsubseteq \sigma\} \\
\mathcal{L}\,[\![A]\!]\,\rho\,(n+1,\sigma,a) & = & \{e \in \mathrm{Expr} \mid e \sqsubseteq \sigma \wedge \sigma \Longrightarrow \sigma' \wedge \forall w' \le (n,\sigma',a).\ e \in \mathcal{E}\,[\![A]\!]\,\rho\,w'\}
\end{array}
$$

Figure 10. The Logical Relation

phisticated techniques: for example, Froc is explicitly based on the implementation strategies in self-adjusting computation [1].

Our decomposition also clarifies the semantics of imperative implementation techniques of FRP, by showing how imperative state supports a purely functional surface language. Again, the decomposition of streams via recursive types and the later modality is crucial. *Imperative state* is essential, since it lets us implement the later modality with memoization, and thereby avoid redundant multiple recomputations when we reference a later variable multiple times. However, *persistent state with identity* is not necessary to implement higher-order FRP, and abandoning it enables significant simplifications over the dataflow graph strategy. In a traditional dataflow graph, streams are primitives, and represented by dataflow nodes with persistent identity. This greatly complicates correctness proofs: in [27], we gave such a representation Modeling pure streams with dataflow nodes, and were forced to give a vastly more complex logical relation to account for the persistence of dataflow nodes.

Logics of Time and Space To our knowledge, Sculthorpe and Nilsson [41] first suggested using temporal logic to verify FRP programs. There, they gave a standard (albeit dependently-typed) arrowized FRP system, and temporal logic was used to describe the behavior of these programs. Jeffrey [19] and Jeltsch [22] were the first to suggest using LTL directly as a type system for FRP.

Jeltsch [21] proposes a design for a Haskell FRP library with some similarities with our approach. As in our approach, streams are viewed as a kind of generator which incrementally produce values. However, instead of using temporal modalities to control when streams are used, the types of streams are indexed with polymorphic type parameters ("era parameters") in the style of Haskell's `runST` operation. Though no correctness proof is given, the use of polymorphism suggests that the techniques of Jeffrey [20] may be applicable.

The Modal μ-calculus Our use of recursive types to model temporal operations naturally invites comparisons to the modal μ-calculus [24]. The μ-calculus takes the propositional calculus, adds a next-step modality, and adds constructors for inductive and coinductive formulas.

The biggest difference is that the μ-calculus restricts recursive variables to occur in strictly positive position, but places no guardedness condition on those variables. In contrast, our notion of recursive type is drawn from Nakano [34], who uses a guardedness condition instead of a positivity condition, allowing variables to occur negatively as long as they occur underneath a delay operator.

Permitting negative occurrences is extremely powerful: as we have seen, term-level fixed points can be defined using recursive types with negative occurrences. The definability of fixed points in turn means that guarded recursive types enjoy a form of limit-colimit coincidence similar to the same property in domain theory. In type-theoretic terms, inductive and coinductive interpretations of a guarded recursive type coincide. This means that it is not possible to distinguish may (coinductive) and must (inductive) properties: for example, our event type constructor says that an event may occur, but does not say it will necessarily occur. (Recently, Jeltsch [23] has investigated potential applications of must-operators to reactive programming.)

Our choice to use guarded recursive types was guided by the nature of interactive programs: if we begin to download a file, there is no way to be *sure* that the download necessarily completes (such as the wifi may go down). As a result, placing must-properties in types is perhaps philosophically arguable. (On a pragmatic note, expressing fixed points with guarded recursion leads to very natural and idiomatic code.)

Our stable type $\Box A$ is not found in the modal μ-calculus, but may be understood as a constructivization of the always modality. When passing from a model-theoretic semantics to a proof theory, it is often the case that a single model-theoretic concept bifurcates into two or more proof-theoretic concepts. The always modality exemplifies this: under a computational interpretation,"always A" can either be interpreted as a single value of type A which is always available (that is, stability $\Box A$), or as a different A on each tick (that is, streams $\mathsf{S}\,A$).

Very recently, Cave et al. [8] proposed a type-theoretic constructivization of the modal μ-calculus (without the $\Box A$ modality). In particular, they use inductive and coinductive types instead of a Nakano-style recursive type, and use it to express fairness properties (for example, of schedulers) with types. While it is too early to make a detailed comparison (larger examples are needed), having calculi with both kinds of type recursion available seems like a valuable tool for understanding the design space.

Stability and Permissions Our stability judgment and promotion rule are inspired by the mobility judgment in the distributed language

ML5 [33]. We wanted to identify values we could use at multiple times, and they wanted to identity values they could use at multiple locations. Promotion is actually definable in our calculus, as a kind of eta-expansion. However, operationally these coercions traverse the entire data structure, and so for efficiency's sake we added it as a primitive. Allocation permissions were introduced by Hofmann [18], to control memory allocation in a linearly-typed language. Our observation that in an intuitionistic setting, these tokens correspond to an object capability style [32] seems to be new, and potentially has applications beyond reactive programming. However, we do not yet have a full Curry-Howard explanation of allocation permissions.

Acknowledgments

I would like to thank Derek Dreyer, Bob Harper, and the anonymous referees for helpful discussions and suggestions.

References

[1] U. A. Acar. *Self-Adjusting Computation*. PhD thesis, Carnegie Mellon University, 2005.

[2] A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *Principles of Programming Languages (POPL)*, pages 340–353, 2009.

[3] A. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(5):657–683, 2001.

[4] H. Baker and C. Hewitt. The incremental garbage collection of processes. In *Symposium on Artificial Intelligence Programming Languages*, August 1977.

[5] G. Berry and L. Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In *Seminar on Concurrency*, pages 389–448. Springer, 1985.

[6] P. Caspi and M. Pouzet. Synchronous Kahn networks. In *Proceedings of the first ACM SIGPLAN international conference on Functional programming*, ICFP '96, pages 226–238, New York, NY, USA, 1996. ACM. ISBN 0-89791-770-7. .

[7] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE: A declarative language for real-time programming. In *Principles of Programming Languages (POPL)*, 1987.

[8] A. Cave, F. Ferreira, P. Panangaden, and B. Pientka. Fair reactive programming. Technical report, McGill University, 2013.

[9] G. H. Cooper. *Integrating Dataflow Evaluation into a Practical Higher-Order Call-by-Value Language*. PhD thesis, Brown University, 2008.

[10] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *European Symposium on Programming (ESOP)*, pages 294–308, 2006.

[11] H. B. Curry. The Paradox of Kleene and Rosser. *Transactions of the American Mathematical Society*, 50(3):454–516, Nov. 1941.

[12] J. Donham. Froc: a library for functional reactive programming in OCaml. `http://jaked.github.com/froc/`, 2010.

[13] D. Dreyer, G. Neis, and L. Birkedal. The impact of higher-order state and control effects on local relational reasoning. In *International Conference on Functional Programming (ICFP)*, pages 143–156, 2010.

[14] C. Elliott. Push-pull functional reactive programming. In *Haskell Symposium*, 2009. URL `http://conal.net/papers/push-pull-frp`.

[15] C. Elliott and P. Hudak. Functional reactive animation. In *International Conference on Functional Programming (ICFP)*, 1997.

[16] D. Friedman and D. Wise. The impact of applicative programming on multiprocessing. In *International Conference on Parallel Processing*, pages 263–272, 1976.

[17] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2001.

[18] M. Hofmann. Linear types and non-size-increasing polynomial time computation. In *Logic in Computer Science (LICS)*, 1999.

[19] A. Jeffrey. LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. In *Programming Languages Meets Program Verification (PLPV)*, pages 49–60, 2012.

[20] A. Jeffrey. Causality for free!: parametricity implies causality for functional reactive programs. In *Programming Languages Meets Program Verification (PLPV)*, pages 57–68, 2013.

[21] W. Jeltsch. Signals, not generators! *Trends in Functional Programming*, 10:145–160, 2009.

[22] W. Jeltsch. Towards a common categorical semantics for linear-time temporal logic and functional reactive programming. *Electronic Notes in Theoretical Computer Science*, 286:229–242, Sept. 2012.

[23] W. Jeltsch. Temporal logic with "until", functional reactive programming with processes, and concrete process categories. In *Programming Languages Meets Program Verification (PLPV)*, pages 69–78, 2013.

[24] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333 – 354, 1983.

[25] N. R. Krishnaswami. Proofs for higher-order reactive programming without spacetime leaks (supplementary material). Technical report, Max Planck Institute for Software Systems (MPI-SWS), 2013.

[26] N. R. Krishnaswami and N. Benton. A semantic model for graphical user interfaces. In *International Conference on Functional programming (ICFP)*, pages 45–57, 2011.

[27] N. R. Krishnaswami and N. Benton. Ultrametric semantics of reactive programs. In *Logic in Computer Science (LICS)*, pages 257–266, 2011.

[28] N. R. Krishnaswami, N. Benton, and J. Hoffmann. Higher-order functional reactive programming in bounded space. In *Principles of Programming Languages (POPL)*, pages 45–58, 2012.

[29] B. S. Lerner, M. J. Carroll, D. P. Kimmel, H. Q.-D. La Vallee, and S. Krishnamurthi. Modeling and reasoning about DOM events. In *Conference on Web Application Development (WebApps)*, 2012.

[30] H. Liu, E. Cheng, and P. Hudak. Causal commutative arrows and their optimization. In *International Conference on Functional Programming (ICFP)*, 2009.

[31] I. Maier and M. Odersky. Deprecating the Observer Pattern with Scala.React. Technical report, EPFL, 2012.

[32] M. Miller. *Robust composition: Towards a unified approach to access control and concurrency control*. PhD thesis, Johns Hopkins University, 2006.

[33] T. Murphy VII., K. Crary, and R. Harper. Type-safe distributed programming with ML5. In *Conference on Trustworthy Global Computing (TGC)*, pages 108–123, 2008.

[34] H. Nakano. A modality for recursion. In *Logic in Computer Science (LICS)*, pages 255–266, 2000.

[35] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *ACM Haskell Workshop*, page 64, 2002.

[36] A. Pitts and I. Stark. Operational reasoning for functions with local state. *Higher Order Operational Techniques in Semantics*, 1998.

[37] G. D. Plotkin. A powerdomain construction. *SIAM Journal of Computation*, 5(3):452–487, 1976.

[38] A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science (FOCS)*, pages 46 –57, 1977.

[39] M. Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, 2006.

[40] N. Sculthorpe and H. Nilsson. Safe functional reactive programming through dependent types. In *International Conference on Functional Programming (ICFP)*, 2009.

[41] N. Sculthorpe and H. Nilsson. Keeping calm in the face of change. *Higher Order Symbolic Computation*, 23(2):227–271, June 2010.

[42] P. Wadler, W. Taha, and D. MacQueen. How to add laziness to a strict language, without even being odd. In *Workshop on Standard ML*, 1998.

[43] Z. Wan, W. Taha, and P. Hudak. Event-driven FRP. In *Practical Applications of Declarative Languages (PADL)*, pages 155–172, 2002.