# Higher-Order Functional Reactive Programming in Bounded Space


## Abstract

Functional reactive programming (FRP) is an elegant and successful approach to programming reactive systems declaratively. The high levels of abstraction and expressivity that make FRP so attractive as a programming model do, however, often lead to the resource usage of functional reactive programs being excessive and/or hard to predict.

In this paper, we show how to solve the problem of space leaks in discrete-time functional reactive programs without giving up on the higher-order abstractions that make FRP so expressive. We give a higher-order functional reactive programming language that statically bounds the size of the dataflow graph a reactive program creates, while still permitting free use of higher-order functions and higher-type streams such as streams of streams. We achieve this with a novel linear type theory that both controls allocation and ensures that all recursive definitions are well-founded.

We also give a new denotational semantics for our language by combining recent work on metric spaces for the interpretation of higher-order causal functions with length-space models of space-bounded computation. The resulting category is doubly closed and hence forms a model of the logic of bunched implications.


## 1. Introduction

Reactive systems engage in an ongoing interaction with their environment, consuming input events and producing corresponding output events. Examples of such systems range from embedded controllers and sensor networks up to complex graphical user interfaces, web applications, games and simulations. Programming reactive systems in a general-purpose imperative language can be unpleasant, as different parts of the program interact not by structured control flow, but by dynamically registering state-manipulating callback functions with one another. The complexity of writing and reasoning about programs written in such a higher-order imperative style, as well as the critical nature and resource requirements of many reactive systems, has inspired extensive research into domain-specific languages (DSL), libraries and analysis techniques for reactive programming.

Synchronous dataflow languages, such as Esterel [2], Lustre [3], and Lucid Synchrone [18], implement a domain-specific computational model deriving from Kahn networks. A program corresponds to a fixed network of stream-processing nodes that communicate with one another, each consuming and producing a statically-known number of primitive values at every clock tick. Synchronous languages have precise, analysable semantics, provide strong guarantees about bounded usage of space and time, and are widely used in applications such as hardware synthesis and embedded control software.

Functional reactive programming (FRP), as introduced by Elliott and Hudak [6], also works with time-varying values (rather than mutable state) as a primitive abstraction, but provides a much richer model than the synchronous languages: signals (behaviours) can vary continuously as well as discretely, values can be higher-order (including both first-class functions and signal-valued signals), and the overall structure of the system can change dynamically. FRP has been applied in problem domains including robotics, animation, games, web applications and GUIs. However, the expressivity and apparently simple semantics of the classic FRP model come at a price. Firstly, the intuitively appealing idea of modelling $A$-valued signals as elements of the stream type $A^\omega$ (or, in the continuous case, $A^{\mathbb{R}}$) and reactive systems as stream functions $\text{Input}^\omega \to \text{Output}^\omega$ does not rule out systems that violate causality (the output today can depend upon the input tomorrow) or reactivity (ill-founded feedback can lead to undefined behaviour). Secondly, as the model is highly expressive and abstracts entirely from resource usage, the time and space behaviour of FRP programs is hard to predict and, even with sophisticated implementation techniques, can often be poor. It is all too easy to write FRP programs with significant space leaks, caused by, for example, inadvertantly accumulating the entire history of a signal.[1]

Subsequent research has attempted to reduce 'junk' and alleviate performance problems by imposing restrictions on the classic FRP model. The Yale Haskell Group's Yampa [9, 16], for example, is an embedded DSL for FRP that constructs signal processing networks using Hughes's arrow abstraction [10]. Signals are no longer first-class, and signal-processing functions must be built from well-behaved casual primitives by causality-preserving combinators. Arrowized FRP allows signals to carry complex values but is essentially first-order (there is no exponential at the level of signal functions), though certain forms of dynamism are allowed via builtin 'switching' combinators. Yampa does not enforce reactivity or provide resource guarantees but, empirically at least, makes certain kinds of leaks less likely.

Krishnaswami and Benton [11] recently described a semantic model for higher-order, discrete-time functional reactive programs based on ultrametric spaces, identifying causal functions with non-expansive maps and interpeting well-founded feedback via Banach's fixpoint theorem. They gave an associated language, featuring a Nakano-style [15] temporal modality for well-founded recursion, and showed the correctness of an implementation using an imperatively-updated dataflow graph. This implementation is much

[1] Closely related are 'time leaks', which occur when sampling a time-dependent value can invoke an arbitrarily lengthy computation to 'catch up' with the current time.

more efficient than directly running the functional semantics, but nothing prevents the dataflow graph from growing unboundedly as a program executes, leading to space leaks that are unacceptable in many applications. In this paper, we solve the problem of such space leaks by extending the ultrametric approach to FRP with linearly-typed resources that represent the permission to perform heap-allocation, following the pattern of Hofmann's work [7, 8] on non-size-increasing computation.

We give a denotational model for bounded higher-order reactive programming in terms of 'complete ultrametric length spaces', which carry both an ultrametric distance measure and a size function. Maps between such spaces must be non-expansive and non-size-increasing. Intuitively, the metric is used to enforce good temporal behaviour (causality and productivity of recursive definitions), whilst the size measure enforces good spatial behaviour, bounding the number of cells in the dataflow graph. The category of complete ultrametric length spaces is doubly-closed, forming a model of the logic of bunched implications [17] and exposing a surprising connection between the type theory of stream programming and separation logic.

We define a typed term language, corresponding to our model, for writing bounded reactive programs. The type theory of our language is rather novel. Judgements are all time-indexed, with the successor operation on times internalized in a modality $\bullet$. Terms are typed in three contexts, one carrying linear (actually affine) resources, of type $\Diamond$, giving permission to allocate; one binding pure, resource-free variables; and one binding potentially resourceful variables. We internalize resource-freedom via a ! modality, in the style of linear logic. We give the language an interesting staged operational semantics, which separates the normalizing reduction that takes place within each time step from the transitions that take place when the clock advances, and show that this is soundly modelled in the denotational semantics. The operational semantics uses terms of the language itself to encode the heap context within which evaluation takes place.

We also give a number of important example definitions in our language that illustrate the practicability of our approach. The examples show that we can work with recursively-defined higher-order functions and streams in a natural way while still ensuring causality, productivity and bounded space usage.

To improve readability, we invert the 'logical' order of presentation: Section 2 gives an informal account of the language that suffices to let us give some of the motivating examples. Section 3 formally presents the language and the type system. We then, in Section 4, define the operational semantics and, in Section 5 and 6, present the details of the denotational model. Finally, in Section 7, we discuss our work and relate it to existing research.

## 2. Programming Language

Our language is essentially a simply-typed $\lambda$-calculus with a type constructor $\mathsf{S}(-)$ for infinite streams, extended with three non-standard notions: delay types, resource types, and pure types. We treat streams as much as possible as mathematical sequences, but want to ensure that we can also interpret them as the successive values of signals generated by implementable clocked systems. Thus a recursive definition (in a Haskell-like syntax for now)

```
1 nats :: ℕ → S(ℕ)
2 nats n = cons(n, nats(n+1))
```

denotes a parameterized infinite stream but can also be understood as a stateful process ticking out successive natural numbers as time advances. It is necessary to restrict recursive definitions, so as to ensure that signals are well-defined at all times. The recursion in the program above is clearly *guarded*: the recursive call to $nats$ only occurs underneath a cons constructor, so successively unfolding the stream at each clock tick is productive: we always discover at least one new cons constructor which we can examine to find the head and the tail of the stream at the current time.

However, simple syntactic guardedness checks (used by languages as varied as Lucid Synchrone, Agda and Coq) do not integrate well with a higher-order style. For example, one might want a higher-order stream functional that abstracts over the constructor:

```
1 higher_order f v = f(v, higher_order f (v + 1))
```

The guardedness of $higher\_order$ now depends on the definition of $f$, which is an unknown parameter. Krishnaswami and Benton [11] use the *next-step* modality $\bullet$ to be more explicit in the type system about the time at which values are available. A value of type $\bullet A$ is a computation which will yield value of type $A$ when executed on the *next* clock tick. The tail function is typed as $tail : S(A) \to \bullet S(A)$, expressing that the tail of a stream only becomes available in the future. Similarly the type of cons is refined to $cons : A \times \bullet S(A) \to S(A)$, capturing that streams are constructed from a value today and a stream tomorrow. By giving the fixed point combinator the type $fix : (\bullet A \to A) \to A$, one can ensure that all recursive definitions are well-founded, without having to restrict their syntactic form.

If we care about space usage, however, Krishnaswami and Benton's use of types to track guardedness still admits too many programs. The problem is to limit the amount of data that must be buffered to carry it from one time tick to the next. In the case of *nats* above, it is clear that the current state of the counting process can always be held in a single natural number. But consider a similar definition at a higher type:

```
1 constantly_leak :: S(A) → S(S(A))
2 constantly_leak xs = cons(xs, constantly_leak xs)
```

So *constantly_leak xs* is a stream of streams that is constantly *xs*. This is a perfectly well-founded functional definition, but as a stateful process, requires the whole accumulated history of *xs* to be buffered so it can be pushed forward on each time step. Running *constantly_leak xs* for $n$ time steps requires $O(n)$ space, which is unreasonable. On the other hand, there are definitions of the same type that *can* be implemented in constant space:

```
1 tails :: S(A) → S(S(A))
2 tails xs = cons(xs, tails (tail xs))
```

Since $tails$ returns the successive tails of its stream argument, we can use the same mutable data structure at each time step, with no buffering.

To account for the memory usage associated with creating and buffering stream data, we adopt the linear (affine) *resource types* of Hofmann's LFPL [7, 8]. The type $\Diamond$ represents a permission to create one new stream; the tensor product $R \otimes S$ is the permission to do both $R$ and $S$; and the linear function space $R \multimap A$ builds an $A$, consuming the resources in $R$. We further refine the construction of streams of type $S(A)$ to take *three* arguments using the syntactic form $\mathsf{cons}(u, e, u'.\ e')$. The term $u : \Diamond$ is a permission to allocate a cons cell and the term $e : A$ is the head of the stream. The tail, $e' : \bullet S(A)$, is defined in scope of the variable $u'$, which re-binds the allocation permission that will be freed up on the next time step.

We still permit sharing of stream values without restriction since dataflow programs gain efficiency precisely from the ability to share. Therefore, we also support a context of unrestricted variables and an intuitionistic[2] function space $A \to B$. (Hofmann's original language featured a strictly linear type discipline.) Function closures can also need buffering if they capture streams in their environment, so it is useful to introduce the type constructor !. The

[2] We do *not* decompose the intuitionistic function space as $!A \multimap B$.

type $!A$ classifies those $A$-values that need no buffering and may be freely carried forward in time.

### 2.1 Examples

Our languages makes the intuition of bounded resource consumption explicit and so enables a natural programming style.

```
nats : ℕ → ◊ ⊸ S(A)
nats = fix loop : ℕ → ◊ ⊸ S(A)
         λ n u. cons(u, n, u'. loop (n+1) u')

fibs : (ℕ×ℕ) → ◊ ⊸ S(A)
fibs = fix loop : (ℕ×ℕ) → ◊ ⊸ S(A)
         λ (n,m) u. cons(u, n, u'. loop (m,n+m) u')
```

The definitions of the classic stream functions $nats$ and $fibs$ in our language are similar to the usual definitions. The types of the functions state that they need one permission to create a cons cell.

```
constantly : !A → ◊ ⊸ S(A)
constantly a' =
  let !a = a in
    fix loop : ◊ ⊸ S(A). λ u. cons(u, a, u'. loop u')

tails : S(A) → ◊ ⊸ S(S(A))
tails =
  fix tails : S(A) → ◊ ⊸ S(S(A)).
    λ xs u.
      let xs' = tail(xs) in
      cons(u, xs, u'. tails xs' u')

cokleisli : !(S(A) → B) → S(A) → ◊ ⊸ S(B)
cokleisli g =
  let !f = g in
  fix loop : S(A) → ◊ ⊸ S(B).
    λ xs u.
      let ys = tail(xs) in
      cons(u, f xs, u'. loop ys u')
```

These three definitions show that we can efficiently build constant streams when it is cheap to move values across time, and that streams of streams can be efficient when the clocks align properly. The function argument to *cokleisli* is at a pure type, since it will be called successively at many different clock ticks.

```
unfold : !(X → A × •X) → X → ◊ ⊸ S(A) =
unfold h =
  let !f = h in
  fix loop : X → ◊ ⊸ S(A).
    λ x u.
      let (a, d) = f(x) in
      let •x' = d in
      cons(u, a, v. loop x' v)

map : !(A → B) → S(A) → ◊ ⊸ S(B)
map h =
  let !f = h in
  fix loop : S(A) → ◊ ⊸ S(B).
    λ xs u.
      let ys = tail xs in
      cons(f (head xs), v. loop ys v)

zip : S(A) × S(B) → ◊ ⊸ S(A × B)
zip =
  fix zip : S(A) × S(B) → ◊ ⊸ S(A × B).
    λ (xs, ys) u.
      let xs' = tail(xs) in
      let ys' = tail(ys) in
      cons(u, (head xs, head ys), v. zip (xs', ys') v)
```

Above, we show the important stream functionals are definable.

```
sum : S(ℕ) × S(ℕ) → (◊ ⊗ ◊) ⊸ S(ℕ) =
  λ (xs, ys) (u, v). map !+ (zip (xs,ys) u) v

double : S(ℕ) → (◊ ⊗ ◊) ⊸ S(ℕ)
double ns (u, v) =
  sum(ns, ns) (u, v)
```

The function $double$ illustrates the importance of the intuitionistic treatment of streams, since it passes $sum$ the *same* stream in both arguments.

```
flip : S(•A) → •(◊ ⊸ S(A))
flip =
  fix flip : S(•A) → •(◊ ⊸ S(A))
    λ xs.
      let •x' = head(xs) in
      let xs' = tail(xs) in
      •(let •f = flip xs' in
          λ u. cons(u, x', u'. f u'))

unflip : •S(A) → ◊ ⊸ S(•A)
unflip =
  fix unflip : •S(A) → ◊ ⊸ S(•A).
    λ xs' u.
      let •ys = xs' in
      cons(u, •(head(ys)), u'.
                let ys' = tail(ys) in
                unflip (• ys') u')
```

These two programs show that we can program the distributivity of delays through streams, by explicitly scheduling delayed computations. Below, we define one of the switching combinators of arrowized FRP:.

```
switch : S(bool) → S(S(A)) → ◊ ⊸ S(A)
switch =
  let loop =
    fix loop : S(bool) → S(S(A)) → S(A) → ◊ ⊸ S(A).
      λ bs xss current u.
        let yss = tail(xss) in
        let bs' = tail(bs) in
        if head(bs) then
          let zs = tail(head xss) in
          cons(u, head(head xss), u'. loop bs' yss zs)
        else
          let zs = tail(current) in
          cons(u, head(current), u'. loop bs' yss zs)
  in
  λ bs xss u. loop bs xss (head xss) u
```

The function $switch$ takes a stream of boolean events and a stream of streams. It then yields the elements of the head of the stream of streams until the boolean stream yields `true`, at which point it starts generating the elements of the current stream from the stream of streams. Note that $switch$ is *definable* in our language, and is not a special primitive. We make essential use of our support of higher-order streams, as well as the ability to define fixed points at arbitrary type.

## 3. Syntax and Typing

We give the types and syntax of the programming language in Figure 1. The types include base types $P$, stream types $S(A)$, the next-step modality $\bullet A$, ordinary functions $A \to B$, the purity modality $!A$, and the linear function space $R \multimap A$. Resources $R$ include the allocation permission $\Diamond$, and the tensor product $R \otimes S$. For space reasons, we do not include products $A \times B$ or sums $A + B$. (Since we require coproducts to define switching combinators, we emphasize that this is purely for space reasons: there are no technical difficulties in supporting sum types.)

The typing rules are defined in Figure 2. The two type judgements of our type theory are $\Theta \vdash t :_i R$ and $\Theta; \Pi; \Gamma \vdash e :_i A$.

Both judgements are time-indexed, in the sense that the type system judges a term to have a type at a particular time $i$. Furthermore, all hypotheses in contexts are also indexed by the time at which they can be used.

The judgement $\Theta \vdash t :_i R$ states that in the context of affine resource variables $\Theta$, the term $t$ has resource type $R$, at time $i$. The resource terms are built from affine pairs of the ground type $\Diamond$ and are permissions to allocate one or more cons cells.

The judgement $\Theta; \Pi; \Gamma \vdash e :_i A$ has three contexts: the affine resource context $\Theta$ contains again permissions to allocate cons cells; the intuitionistic context $\Pi$ contains pure hypotheses (i.e., variables in $\Pi$ bind non-state-dependent values); and the intuitionistic context $\Gamma$ binds arbitrary value types, and permits unrestricted sharing and reuse of variables. Under these three contexts, we judge a term $e$ to be an expression of type $A$ at time $i$.

There are only two rules for the affine resource term calculus in Figure 2. We have the hypothesis rule RHYP, which lets us use a resource at any time after the hypothesis says it is available. Additionally, we have the tensor rule R$\otimes$I, which states that we can form an affine pair $\langle t, t' \rangle$ by dividing the resources in the context between the two contexts. We permit extra hypotheses to be present (and hence go unused) by the hypothesis rule, but require R$\otimes$I to divide its context. Thus, we allow weakening but not contraction.

The rule PHYP lets us use a pure hypothesis at any time after the time index in the variable context. In contrast, the rule EHYP only permits using a variable $x$ at *exactly* its time index in $\Gamma$. This difference is one of the keys to accurately tracking space usage: we may substitute values which require buffering for variables in $\Gamma$, and by disallowing implicit transport of values across time, we ensure that the programmer uses explicit buffering whenever needed.

The rules $\to$I and $\to$E introduce and eliminate intuitionistic functions. The introduction rule does not permit the body to use any resources, since we can call functions multiple times. (The presence of $\Theta$ in the conclusion of $\to$I and the other value forms builds in weakening, so that we do not have to give a separate structural rule.) The elimination rule does allow expressions to use resources, since we will use a call-by-value evaluation strategy that will evaluate the two terms (using up their resource permissions) before substituting a value into a lambda-term.

The rules $\multimap$I and $\multimap$E introduce and eliminate linear functions. The introduction rule only permits the body to use the resources it receives in the argument, since we need to ensure that the function can be safely called multiple times. As a result, our typing rules do not permit currying of linear functions ($R \multimap S \multimap A \simeq R \otimes S \multimap A$), even though our underlying semantic model does permit it. If our type theory had the tree-structured contexts of the logic of bunched implications [17], then currying linear functions would be syntactically expressible.

The rules SI, SE-HEAD and SE-TAIL are the introduction and elimination rules for streams. The syntactic form $\mathsf{cons}(t, e, u.\ e')$ takes three arguments: the expression $t$ is a permission to create a cons cell, the expression $e$ is the head of the stream, and $e'$ is the tail of the stream. The tail subterm $e'$ occurs underneath a binder for the resource variable $u'$. The intuition is that each stream takes up one unit of space at each successive time step, and $u'$ names the permission $t$, after one time step has elapsed. This lets us pass the permission to use $t$ to functions on subsequent time steps in the body of $e'$.

The rule SE-HEAD is straightforward: given a stream of type $S(A)$, we get a value of type $A$, at the same time. The rule SE-TAIL uses the form $\mathsf{let}\ x = \mathsf{tail}(e)\ \mathsf{in}\ e'$ to bind the tail of $e$ to the variable $x$ in $e'$. Other than the head of a stream, the tail of a stream at time $i$ lives at time $i + 1$. So we choose a binding elimination form to maintain the invariant that no term of time $i$ contains any subterms at time $< i$.

The rules $\bullet$I and $\bullet$E introduce and eliminate delay terms. The rule $\bullet$I says that if $e$ is a term of type $A$ at time $i + 1$, then $\bullet e$ is a term of type $\bullet A$ at time $i$.

Like the other value introduction forms, it prevents $e$ from using any resources, so that $e$ can be substituted freely. The rule $\bullet$E gives a binding elimination $\mathsf{let}\ \bullet x = e\ \mathsf{in}\ e'$ for the next-step modality. We use a binding elimination for the same reason as in the rule SE-TAIL.

The rule !I introduces a term of pure type $!A$. It does so by typing the body of a term $!A$ at type $A$ with an empty resource and shared context. Since $e$ can only refer to hypotheses in $\Pi$, which are pure, it follows that $e$ itself must be pure. The rule !E types the elimination form $\mathsf{let}\ !x = e\ \mathsf{in}\ e'$, which binds the value $e$ to the variable $x$ in the pure context $\Pi$.

The rule $\otimes$E is the elimination form for the tensor. Given a term $t$ of type $R \otimes S$, the expression $\mathsf{let}\ \langle u, v \rangle = t\ \mathsf{in}\ e$ binds the components to the variables $u$ and $v$ for the scope of $e$.

The rule LET introduces local let-bindings. The introduced binding $x = e$ must be at the same time $i$ as the overall expression.

The rule FIX types fixed points $\mathsf{fix}\ x : A.\ e$. The body $e$ is typed at time $i$ with the recursive hypothesis $x :_{i+1} A$ one tick later. The one-step delay ensures the guardedness of recursive definitions. Furthermore, the typing of the expression $e$ is derived under an empty resource context and with an empty shared context. Since $e$ may unfold multiple times, giving $e$ resources would violate linearity. Furthermore, the unrollings can happen at different times, which means that using any variables in the shared context might require buffering.

We now state the weakening and substitution principles.

**Lemma 1.** *(Weakening)*

1. *If* $\Theta \vdash t :_i R$ *then* $\Theta, \Theta' \vdash t :_i R$.
2. *If* $\Theta; \Pi; \Gamma \vdash e :_i A$ *then* $\Theta, \Theta'; \Pi; \Gamma \vdash e :_i A$.
3. *If* $\Theta; \Pi; \Gamma \vdash e :_i A$ *then* $\Theta; \Pi, \Pi'; \Gamma \vdash e :_i A$.
4. *If* $\Theta; \Pi; \Gamma \vdash e :_i A$ *then* $\Theta; \Pi; \Gamma, \Gamma' \vdash e :_i A$.

**Theorem 1.** *(Substitution) We have that:*

1. *If* $\Theta \vdash t :_i R$ *and* $\Theta', u :_i R \vdash t' :_j R'$, *then* $\Theta, \Theta' \vdash [t/u]t' :_j R'$.
2. *If* $\Theta \vdash t :_i R$ *and* $\Theta, u :_i R; \Pi; \Gamma \vdash e :_j A$, *then* $\Theta, \Theta'; \Pi; \Gamma \vdash [t/u]e :_j A$.
3. *If* $\cdot; \Pi; \cdot \vdash e :_i A$ *and* $\Theta; \Pi, x :_j A; \Gamma \vdash e' :_k B$ *and* $i \leq j$, *then* $\Theta; \Pi; \Gamma \vdash [e/x]e' :_k B$.
4. *If* $\cdot; \Pi; \Gamma \vdash e :_i A$ *and* $\Theta; \Pi; \Gamma, x :_i A \vdash e' :_j B$, *then* $\Theta; \Pi; \Gamma \vdash [e/x]e' :_j B$.

Lemma 1 and Theorem 1 can be proved by structural inductions on the type derivations in the respective premises.

## 4. Operational Semantics

Theorem 1 formulates the substitution principles so that a general expression $e$ that replaces a variables has to be typed without resource variables. That is, the substitution $[e/x]e'$ is only sound if the substitutee $e$ uses no resource variables. On the other hand, the typing rule for cons cells demands the use of a resource, raising the question: what is the operational semantics of cons cells?

A purely substitution-based operational semantics cannot be correct, because it does not account for the sharing of cons cells. Consider the following expression that is well-typed in our system.

```
1  let xs = cons(u, ...) in // u is the linear allocation permission
2  sum (xs, xs) v
```

Types

| | | | |
|---|---|---|---|
| General | $A$ | $::=$ | $P \mid A \to B \mid S(A) \mid \bullet A$ |
| | | $\mid$ | $!A \mid R \multimap A$ |
| Resource | $R$ | $::=$ | $\Diamond \mid R \otimes R$ |

Terms

| | | | |
|---|---|---|---|
| General | $e$ | $::=$ | $x \mid \lambda x : A.\ e \mid e\ e' \mid \lambda u : R.\ e \mid e\ t$ |
| | | $\mid$ | $\mathsf{cons}(t, e, u.\ e') \mid \mathsf{head}(e)$ |
| | | $\mid$ | $\mathsf{let}\ x = \mathsf{tail}(e)\ \mathsf{in}\ e'$ |
| | | $\mid$ | $\bullet e \mid \mathsf{let}\ \bullet x = e\ \mathsf{in}\ e'$ |
| | | $\mid$ | $!e \mid \mathsf{let}\ !x = e\ \mathsf{in}\ e'$ |
| | | $\mid$ | $\mathsf{let}\ \langle u, v\rangle = t\ \mathsf{in}\ e \mid \mathsf{fix}\ x : A.\ e$ |
| | | $\mid$ | $\mathsf{let}\ x = e\ \mathsf{in}\ e'$ |
| Resource | $t$ | $::=$ | $u \mid \langle t, t'\rangle$ |
| Values | $v$ | $::=$ | $\lambda x : A.\ e \mid \lambda u : R.\ e \mid !v \mid \bullet e \mid x$ |

Contexts

| | | | |
|---|---|---|---|
| General | $\Gamma$ | $::=$ | $\cdot \mid \Gamma, x :_i A$ |
| Pure | $\Pi$ | $::=$ | $\cdot \mid \Pi, x :_i A$ |
| Resource | $\Theta$ | $::=$ | $\cdot \mid \Theta, u :_i R$ |

Evaluation Contexts

| | | | |
|---|---|---|---|
| | $C$ | $::=$ | $\Box \mid \mathsf{let}\ x = \mathsf{tail}(y)\ \mathsf{in}\ C$ |
| | | $\mid$ | $\mathsf{let}\ x = \mathsf{cons}(u, v, u'.\ e)\ \mathsf{in}\ C$ |

**Figure 1.** Syntax

Here, we construct $xs$ once, but use it twice in the call to $sum$. We cannot simply substitute the cons into the body of the let in our system, as that would duplicate the linear variable $u$.

One approach for managing permissions is to introduce a heap for cons cells, and refer to streams indirectly by reference. However, adding a heap moves us away from our functional intuitions and makes it more difficult to connect to our denotational model. Instead, we retain the idea of referring to streams by reference, but use *variables* for the indirect reference, by defining evaluation to put terms into let-normal form, such as:

```
1  let xs = cons(u1, e1, v1. e1') in
2  let ys = cons(u2, e2, v2. e2') in
3  let xs' = tail(xs) in
4  ...
5  let zs = cons(u3, e3, v2. e3') in
6  let ys' = tail(ys) in
7  v
```

Now, the nested let-bindings act as our heap. The value $v$ may contain many references to individual streams (such as $xs$), but since each stream is bound only once, we can respect the linearity constraint on the allocation permissions $u_i$. (Taking the tail of streams, as in the definition $xs'$ and $ys'$, also needs to be in let-normal form, since we cannot cut out the tail of a cons cell until the next tick.) Using variable to denote sharing makes it easy to continue using our denotational model to interpret the resulting terms. The scoping rules for let-binding also restrict us to a DAG dependency structure, an invariant that imperative reactive programming implementations based on dependency graphs must go to some lengths to implement and maintain.

Let-normalizing cons cells has a second benefit: we can advance the global clock by taking the tails of each cons cell in the context.

```
1  let xs = [u1/v1]e1' in
2  let ys = [u2/v2]e2' in
3  let xs' = xs in
4  ...
```

$\boxed{\Theta \vdash t :_i R}$ $\qquad$ $\boxed{\Theta; \Pi; \Gamma \vdash e :_i A}$

$$\frac{u :_i R \in \Theta \qquad i \le j}{\Theta \vdash u :_j R}\ \text{RHYP}$$

$$\frac{\Theta \vdash t :_i A \qquad \Theta' \vdash t' :_i B}{\Theta, \Theta' \vdash \langle t, t'\rangle :_i A \otimes B}\ \text{R}\otimes\text{I}$$

$$\frac{x :_i A \in \Pi \qquad i \le j}{\Theta; \Pi; \Gamma \vdash x :_j A}\ \text{PHYP} \qquad \frac{x :_i A \in \Gamma}{\Theta; \Pi; \Gamma \vdash x :_i A}\ \text{EHYP}$$

$$\frac{\cdot; \Pi; \Gamma, x :_i A \vdash e :_i B}{\Theta; \Pi; \Gamma \vdash \lambda x : A.\ e :_i A \to B}\ \to\text{I}$$

$$\frac{\Theta; \Pi; \Gamma \vdash e :_i A \to B \qquad \Theta'; \Pi; \Gamma \vdash e' :_i A}{\Theta, \Theta'; \Pi; \Gamma \vdash e\ e' :_i B}\ \to\text{E}$$

$$\frac{u :_i R; \Pi; \Gamma \vdash e :_i A}{\Theta; \Pi; \Gamma \vdash \lambda u : R.\ e :_i R \multimap A}\ \multimap\text{I}$$

$$\frac{\Theta; \Pi; \Gamma \vdash e :_i R \multimap A \qquad \Theta' \vdash t :_i R}{\Theta, \Theta'; \Pi; \Gamma \vdash e\ t :_i A}\ \multimap\text{E}$$

$$\frac{\begin{array}{c}\Theta \vdash t :_i \Diamond \\ \Theta'; \Pi; \Gamma \vdash e :_i A \qquad \Theta'', u :_{i+1} \Diamond; \Pi; \Gamma \vdash e' :_{i+1} S(A)\end{array}}{\Theta, \Theta', \Theta''; \Pi; \Gamma \vdash \mathsf{cons}(t, e, u.\ e') :_i S(A)}\ \text{SI}$$

$$\frac{\Theta; \Pi; \Gamma \vdash e :_i S(A)}{\Theta; \Pi; \Gamma \vdash \mathsf{head}(e) :_i A}\ \text{SE-HEAD}$$

$$\frac{\begin{array}{c}\Theta; \Pi; \Gamma \vdash e :_i S(A) \\ \Theta'; \Pi; \Gamma, y :_{i+1} S(A) \vdash e' :_i B\end{array}}{\Theta, \Theta'; \Pi; \Gamma \vdash \mathsf{let}\ y = \mathsf{tail}(e)\ \mathsf{in}\ e' :_i B}\ \text{SE-TAIL}$$

$$\frac{\cdot; \Pi; \Gamma \vdash e :_{i+1} A}{\Theta; \Pi; \Gamma \vdash \bullet e :_i \bullet A}\ \bullet\text{I}$$

$$\frac{\Theta; \Pi; \Gamma \vdash e :_i \bullet A \qquad \Theta'; \Pi; \Gamma, x :_{i+1} A \vdash e' :_i B}{\Theta, \Theta'; \Pi; \Gamma \vdash \mathsf{let}\ \bullet x = e\ \mathsf{in}\ e' :_i B}\ \bullet\text{E}$$

$$\frac{\cdot; \Pi; \cdot \vdash e :_i A}{\Theta; \Pi; \Gamma \vdash !e :_i !A}\ !\text{I}$$

$$\frac{\Theta; \Pi; \Gamma \vdash e :_i !A \qquad \Theta'; \Pi, x :_i A; \Gamma \vdash e' :_i B}{\Theta, \Theta'; \Pi; \Gamma \vdash \mathsf{let}\ !x = e\ \mathsf{in}\ e' :_i B}\ !\text{E}$$

$$\frac{\Theta \vdash t :_i R \otimes S \qquad \Theta', u :_i R, v :_i S; \Pi; \Gamma \vdash e :_i C}{\Theta, \Theta' \vdash \mathsf{let}\ \langle u, v\rangle = t\ \mathsf{in}\ e :_i C}\ \otimes\text{E}$$

$$\frac{\cdot; \Pi, x :_{i+1} A; \cdot \vdash e :_i A}{\Theta; \Pi; \Gamma \vdash \mathsf{fix}\ x : A.\ e :_i A}\ \text{FIX}$$

$$\frac{\Theta; \Pi; \Gamma \vdash e :_i A \qquad \Theta'; \Pi; \Gamma, x :_i A \vdash e' :_i B}{\Theta, \Theta'; \Pi; \Gamma \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' :_i B}\ \text{LET}$$

**Figure 2.** Typing Rules

```
5    let zs = [u3/v3]e3' in
6    let ys' = ys in
7    v
```

Since we know where all of the cons cells are, we can rewrite them to model the passage of time. Advancing the clock for tail expressions simply drops the tail. Intuitively, yesterday they promised a tail stream today, and after the step the binding they refer to contains that tail stream.

Our operational semantics has two phases: the within-step operational semantics, which puts an expression into let-normal form, and the step semantics, which advances the clock by one tick by rewriting the cons cells to be their tails.

### 4.1 Within-Step Operational Semantics

The syntax of values and evaluation contexts is given in Figure 1 and the typing and auxilliary operations are given in Figure 4. We define the reduction relation in Figure 5, in big-step style. We write $\Sigma \rhd_i \Omega_i \vdash C \dashv \Omega_i'$ for the evaluation-context typing judgement. The context $\Sigma$ is a resource context $\Theta$ that consists only of $\Diamond$ hypotheses. Similar, the contexts $\Omega_i$ are restricted forms general contexts $\Gamma$ consisting only of stream variables at time $i$ or $i+1$. Both are defined in Figure 3. The judgement $\Sigma \rhd_i \Omega_i \vdash C \dashv \Omega_i'$ reads as "the evaluation context $C$ creates the bindings in $\Omega_i'$, uses the resources in $\Sigma$ to do so, and may refer to the bindings in $\Omega_i$".

The context-concatenation operation $C \circ C'$ appends two evaluation contexts $C$ and $C'$. It is defined in Figure 4 and satisfies the following properties.

**Lemma 2.** *(Context Concatenation) We have that:*

- $\circ$ *is associative with unit* $\Box$.
- *If* $\Sigma \rhd_i \Omega_i \vdash C \dashv \Omega_i'$ *and* $\Sigma' \rhd_i \Omega_i, \Omega_i' \vdash C' \dashv \Omega_i''$*, then* $\Sigma, \Sigma' \rhd_i \Omega \vdash C \circ C' \dashv \Omega_i', \Omega_i''$.
- *If* $\Sigma \rhd_i \Omega_i \vdash C_1 \circ C_2 \dashv \Omega_i'$*, then there exist* $\Sigma_1, \Sigma_2$ *and* $\Omega_i^1, \Omega_i^2$ *such that* $\Sigma = \Sigma_1, \Sigma_2$ *and* $\Omega_i'' = \Omega_i^1, \Omega_i^2$ *and* $\Sigma_1 \rhd_i \Omega_i \vdash C_1 \dashv \Omega_i^1$ *and* $\Sigma_2 \rhd_i \Omega_i, \Omega_i^1 \vdash C_2 \dashv \Omega_i^2$.

In Figure 5, we give the context semantics, evaluating an expression in a context into a value in a larger context. Note that the value forms for streams are variables, since we need to preserve sharing for them, and we can use variable names as pointers into the evaluation context. For most expression forms, the context semantics works as expected; it evaluates each subexpression in context, building a value in a larger context.

The rule ConsE is one of the two rules that extend the context. It evaluates a cons cell, creates a binding to a fresh variable, and returns the fresh variable as the value for that stream. The other rule that extends the context is TailE. It adds a binding to the context naming the tail it constructs.

The rule HeadE, on the other hand, uses the $C@x \Rightarrow v$ relation that is defined in Figure 4 to find the head of the cons cell bound to the variable $x$.

The rule FixE looks entirely conventional. We simply unfold the fixed point and continue. Surprisingly, we are nevertheless able to prove a normalization result for the within-step operational semantics since the fixed point substitutes for a variable at a future time.

We begin the metatheory with a type preservation proof.

**Theorem 2.** *(Type Preservation) If we have that*

$$\Sigma \rhd_i \cdot \vdash C \dashv \Omega_i, \quad \Sigma'; \cdot; \Omega_i \vdash e :_i A, \quad \textit{and} \quad C[e] \Downarrow C''[v],$$

*then there is an* $\Omega_i'$ *and* $C'$ *such that*

$$C'' = C \circ C', \;\; \Sigma' \rhd_i \Omega_i \vdash C' \dashv \Omega_i', \;\; \textit{and} \;\; \cdot; \cdot; \Omega_i, \Omega_i' \vdash v :_i A.$$

To show soundness, we will prove termination via a Kripke logical relations argument. Since we evaluate terms $e$ in contexts $C$, and return a value $v$ in some larger context $C'$, we take our Kripke worlds to be the closed contexts. That is, worlds are those $C$ such that $\Sigma \rhd_i \cdot \vdash C \dashv \Omega_i$. We define the ordering $C' \sqsubseteq C$ on worlds so that there should be some $C_1$ such that $C' \equiv C \circ C_1$. Thus, a future world is one in which more bindings are available.

In Figure 3, we define the logical relation by induction on types. There is one clause $\mathcal{V}_A(C)$ for each type $A$, defining a subset of the well-typed expressions of type $A$, closed save for the variables bound by $C$. The expression relation $\mathcal{E}_A(\Sigma; C)$ consists of the expressions that use resources in $\Sigma$ and evaluate to a value in $\mathcal{V}_A(C)$ in the context $C$.

The definition of the logical relation for streams states that a variable $x$ is in $\mathcal{V}_{S(A)}(C)$ if $x$ binds a cons cell with $v$ in its head, and $v$ is in the $A$-relation. As expected, the relation for functions consists of lambdas such that in any future world, applying a value in the $A$-relation should result in a term in the expression relation at type $B$. In the case of the delay modality we allow any closed expression, since the within-step evaluation relation does not evaluate any terms at time $i$, and the body of a delay is at time $i+1$. The $!A$ relation consists of values $!v$ such that $v$ is in the $A$-relation in the empty world $\Box$, since we want values of type $!A$ to not depend on the stream values $C$ binds. Last, the relation at $R \multimap A$ consists of those lambda-terms $\lambda u : R.\ e$ such that for any resource $t$ of type $R$ in context $\Sigma$, the expression $[t/u]e$ is in the expression relation for $A$ with resources $\Sigma$.

To prove the fundamental property, we define some auxilliary predicates in Figure 3. The predicate $Good(\Omega_i)$ picks out those contexts in which all of the bindings in $\Omega_i$ at time $i$ contain true streams, according to the stream relation. The $\mathcal{V}(\Gamma_{\geq i}; C)$ and $\mathcal{V}!(\Pi_{\geq i})$ sets extend the value relation to substitutions rather than single values, and $\Sigma \vdash \vartheta : \Theta$ defines linear substitutions. The notations $\Pi_{\geq i}$ and $\Gamma_{\geq i}$ mean contexts where every variable is at time $i$ or later.

**Theorem 3.** *(Fundamental Property) Suppose* $\Theta; \Pi_{\geq i}; \Omega_i, \Gamma_{\geq i} \vdash e :_i A$*. Furthermore, suppose that* $C \in Good(\Omega)$ *and* $\Sigma \vdash \vartheta : \Theta$ *and* $\pi \in \mathcal{V}!(\Pi_i)$ *and* $\gamma \in \mathcal{V}(\Gamma; C)$*. Then* $(\vartheta \circ \pi \circ \gamma)(e) \in \mathcal{E}_A(\Sigma; C)$

The fundamental property suffices to prove normalization, once we observe that typing derivations satisfy the following history independence property:

**Lemma 3.** *(History Independence)*

- *If* $\Theta; \Pi; \Gamma, x :_i A \vdash e :_j B$ *and* $i < j$*, then* $\Theta; \Pi; \Gamma \vdash e :_j B$.
- *If* $\Theta; \Pi, x :_i A; \Gamma \vdash e :_j B$ *and* $i < j$*, then* $\Theta; \Pi, x :_j A; \Gamma \vdash e :_j B$.
- *If* $\Theta, u :_i R; \Pi; \Gamma \vdash e :_j B$ *and* $i < j$*, then* $\Theta, u :_j R; \Pi; \Gamma \vdash e :_j B$.

That is, the syntax ensures expressions at time $j > i$ do not have to depend on a variable of time $i$. As a result, we only need to consider contexts in which $\Pi$ and $\Gamma$ contain variables no younger than the current time. Normalization immediately follows:

**Corollary 1.** *(Normalization) Suppose* $\Sigma; \cdot; \cdot \vdash e :_i A$*. Then* $\Box[e] \Downarrow C[v]$.

Finally note that we are considering normalization of open terms, since we have no constants of type $\Diamond$. The non-existence of such contexts is, of course, what ensures that the languge respects space bounds.

**Theorem 4.** *(Space Bounded Evaluation) Suppose* $\Sigma; \cdot; \cdot \vdash e :_i A$ *and* $\Box[e] \Downarrow C[v]$*. Then the* size *of* $C$ *— the number of cons cells it binds — is bounded by the size of* $\Sigma$.

Since we know type preservation holds, this theorem is trivial, since each cons cell in the context needs a distinct resource variable, which is obviously bounded by the size of $\Sigma$.

Parameter of the logical relation: $i$

Base Contexts $\Sigma ::= \cdot \mid \Sigma, u :_0 \Diamond$
$\Omega_i ::= \cdot \mid \Omega_i, x :_i S(A) \mid \Omega_i, x :_{i+1} S(A)$

$$\boxed{\mathcal{V}_A(\Sigma \rhd_i \cdot \vdash C \dashv \Omega_i) \subseteq \{v \mid \cdot;\cdot;\Omega_i \vdash v :_i A\}}$$

$$\boxed{\mathcal{E}_A(\Sigma;\Sigma' \rhd_i \cdot \vdash C \dashv \Omega_i) \subseteq \{e \mid \Sigma;\cdot;\Omega_i \vdash e :_i A\}}$$

$$\mathcal{V}_{S(A)}(C) = \{x \mid \exists v.\ C@x \Rightarrow v \wedge v \in \mathcal{V}_A(C)\}$$
$$\mathcal{V}_{A\to B}(C) = \{\lambda x : A.\ e \mid \forall C', v \in \mathcal{V}_A(C \circ C').\ [v/x]e \in \mathcal{E}_B(\cdot; C \circ C')\}$$
$$\mathcal{V}_{\bullet A}(\Sigma \rhd_i \cdot \vdash C \dashv \Omega_i) = \{\bullet e \mid \cdot;\Pi;\Omega_i \vdash \bullet e :_i \bullet A\}$$
$$\mathcal{V}_{!A}(C) = \{!v \mid v \in \mathcal{V}_A(\Box)\}$$
$$\mathcal{V}_{R\multimap A}(C) = \{\lambda u : R.\ e \mid \forall C', \Sigma, t.\ \Sigma \vdash t :_i R \Rightarrow [t/u]e \in \mathcal{E}_A(\Sigma; C \circ C')\}$$

$$\mathcal{E}_A(\Sigma; C) = \{e : A \mid \exists C', v.\ C[e] \Downarrow (C \circ C')[v] \wedge v \in \mathcal{V}_A(C \circ C')\}$$

$$\mathit{Good}(\Omega_i) = \{C \mid \forall x :_i S(A) \in \Omega_i.\ x \in \mathcal{V}_{S(A)}(C)\}$$

$$\mathcal{V}(\cdot; C) = \{\cdot\}$$
$$\mathcal{V}(\Gamma, x :_i A; C) = \{(\gamma, [v/x]) \mid \gamma \in \mathcal{V}(\Gamma; C) \wedge v \in \mathcal{V}_A(C)\}$$

$$\mathcal{V}!(\cdot) = \{\cdot\}$$
$$\mathcal{V}!(\Pi, x :_i A) = \{(\gamma, [v/x]) \mid \gamma \in \mathcal{V}!(\Gamma) \wedge v \in \mathcal{V}_A(\Box)\}$$

$$\boxed{\Sigma \vdash \vartheta : \Theta}$$

$$\frac{}{\cdot \vdash \cdot : \cdot} \qquad \frac{\Sigma \vdash t :_i R \qquad \Sigma' \vdash \vartheta : \Theta}{\Sigma, \Sigma' \vdash (\vartheta, [t/u]) : (\Theta, u :_i R)}$$

**Figure 3.** Logical Relation for Termination

$$\boxed{\Sigma \rhd_i \Omega_i \vdash C \dashv \Omega_i'}$$

$$\frac{}{\Sigma \rhd_i \Omega_i \vdash \Box \dashv \Omega_i}\ \textsc{ExtNil}$$

$$\frac{\cdot;\cdot;\Omega_i \vdash v :_i A \qquad \Sigma', u' :_{i+1} \Diamond;\cdot;\Omega_i \vdash e :_{i+1} S(A) \qquad \Sigma \rhd_i \Omega_i, x :_i S(A) \vdash C \dashv \Omega_i'}{\Sigma, u :_i \Diamond, \Sigma' \rhd_i \Omega_i \vdash \mathsf{let}\ x = \mathsf{cons}(u, v, u'.\ e)\ \mathsf{in}\ C \dashv \Omega_i', x :_i S(A)}\ \textsc{ExtCo}$$

$$\frac{y :_i S(A) \in \Omega_i \qquad \Sigma \rhd_i \Omega_i, x :_{i+1} S(A) \vdash C \dashv \Omega_i'}{\Sigma \rhd_i \Omega_i \vdash \mathsf{let}\ x = \mathsf{tail}(y)\ \mathsf{in}\ C \dashv \Omega_i', x :_{i+1} S(A)}\ \textsc{ExtTail}$$

$$\boxed{C \circ C'}$$

$$\Box \circ C' \triangleq C'$$
$$(\mathsf{let}\ x = \mathsf{tail}(y)\ \mathsf{in}\ C) \circ C' \triangleq \mathsf{let}\ x = \mathsf{tail}(y)\ \mathsf{in}\ (C \circ C')$$
$$(\mathsf{let}\ x = \mathsf{cons}(u, v, u'.\ e)\ \mathsf{in}\ C) \circ C' \triangleq \mathsf{let}\ x = \mathsf{cons}(u, v, u'.\ e)\ \mathsf{in}\ (C \circ C')$$

$$\boxed{C@x \Rightarrow v}$$

$$\frac{C \equiv C' \circ \mathsf{let}\ y = \mathsf{tail}(z)\ \mathsf{in}\ \Box \qquad C'@x \Rightarrow v}{C@x \Rightarrow v}\ \textsc{LookupTail}$$

$$\frac{C \equiv C' \circ \mathsf{let}\ y = \mathsf{cons}(u, v', u'.\ e)\ \mathsf{in}\ \Box \qquad C'@x \Rightarrow v}{C@x \Rightarrow v}\ \textsc{LookupNext}$$

$$\frac{C \equiv C' \circ \mathsf{let}\ x = \mathsf{cons}(u, v, u'.\ e)\ \mathsf{in}\ \Box}{C@x \Rightarrow v}\ \textsc{LookupCurr}$$

**Figure 4.** Context Typing and Operations

### 4.2 Next-Step Operational Semantics

Recall that when we advance time, we want to replace the tails of stream variables with just the variable. We define the necessary operations in Figure 6. We have $\mathsf{Step}(C[x])$, which takes a stream in context and constructs a new expression by taking the tails of the streams in $C$ and dropping the tails out of tail expressions in $C$. The $\mathsf{Step}(\Omega_i)$ operation says how to change the typing: it sends all streams at time $i$ to time $i+1$ (and leaves streams at time $i+1$ alone). Then we can prove the following type soundness theorem.

**Theorem 5.** *(Advancing the Clock) If* $\Sigma;\cdot;\Omega_i \vdash C[x] :_i S(A)$, *then* $\Sigma;\cdot;\mathsf{Step}(\Omega_i) \vdash \mathsf{Step}(C[x]) :_{i+1} S(A)$.

While our definition of advancing the clock is intuitively plausible, it is still unclear that it is *correct*. To prove this, we will give a denotational semantics of these stream programs, and then prove that advancing the clock is sound with respect to that semantics, by showing that $\mathsf{Step}(C[x])$ does equal the tail of the denotational semantics of $C[x]$.

## 5. Semantic Intuitions

### 5.1 Causality and Ultrametric Spaces

The intuition underpinning reactive programming is the *stream transformer*, a function which takes a stream of inputs and generates a stream of outputs. But not all functions on streams are implementable reactive programs — in order to be implementable at all, reactive programs must respect the *causality* condition. That is, the first $n$ outputs of a stream function may depend on at most its first $n$ inputs. Writing $\lfloor xs \rfloor_n$ for the $n$-element prefix of the stream $xs$, we formalize causality as follows:

**Definition 1.** *(Causality) A stream function* $f : A^\omega \to B^\omega$ *is* causal, *when for all* $n$ *and all streams* $as$ *and* $as'$, *we have that if* $\lfloor as \rfloor_n = \lfloor as' \rfloor_n$ *then* $\lfloor f\ as \rfloor_n = \lfloor f\ as' \rfloor_n$.

Furthermore, reactive programs often define streams by feedback. If a stream transformer can produce the first value of its output without looking at its input, then we can constructing a fixed point via feedback, taking the $n$-th output and supplying it as the input at time $n+1$. So as long as we can generate *more* than $n$ outputs from the first $n$ inputs, we can find a fixed point. Formalizing this gives us a definition of guardedness for defining fixed points:

**Definition 2.** *(Guardedness) A function* $f : A^\omega \to B^\omega$ *is* guarded, *when there exists a* $k > 0$ *such that for all* $n$ *and all streams* $as$ *and* $as'$, *we have that if* $\lfloor as \rfloor_n = \lfloor as' \rfloor_n$ *then* $\lfloor f\ as \rfloor_{n+k} = \lfloor f\ as' \rfloor_{n+k}$.

However, these definitions apply only to stream functions, and real programs need more types than just the stream type. So we need generalizations of causality which work at other types such as streams of streams and higher-order functions.

$$\boxed{C[e] \Downarrow C'[v]}$$

$$\frac{}{C[v] \Downarrow C[v]}\ \textsc{ValE}$$

$$\frac{C[e] \Downarrow C'[\lambda x : A.\ e''] \quad C'[e'] \Downarrow C''[v] \qquad C''[[v/x]e''] \Downarrow C'''[v]}{C[e\ e'] \Downarrow C'''[v]}\ \textsc{AppE}$$

$$\frac{C[e] \Downarrow C'[\lambda u : R.\ e'] \qquad C'[[t/u]e'] \Downarrow C''[v]}{C[e\ t] \Downarrow C''[v]}\ \textsc{LAppE}$$

$$\frac{C[e] \Downarrow C'[v] \quad x \text{ fresh} \qquad C'' = C' \circ \mathsf{let}\ x = \mathsf{cons}(t, v, u.\ e')\ \mathsf{in}\ \Box}{C[\mathsf{cons}(t, e, u.\ e')] \Downarrow C''[x]}\ \textsc{ConsE}$$

$$\frac{C[e] \Downarrow C'[x] \qquad C'@x \Rightarrow \mathsf{cons}(t, v, u.\ e)}{C[\mathsf{head}(e)] \Downarrow C'[v]}\ \textsc{HeadE}$$

$$\frac{C[e] \Downarrow C'[y] \qquad (C' \circ \mathsf{let}\ x = \mathsf{tail}(y)\ \mathsf{in}\ \Box)[e'] \Downarrow C''[v]}{C[\mathsf{let}\ x = \mathsf{tail}(e)\ \mathsf{in}\ e'] \Downarrow C''[v]}\ \textsc{TailE}$$

$$\frac{C[e] \Downarrow C'[\bullet e_1] \qquad C'[[e_1/x]e'] \Downarrow C''[v'']}{C[\mathsf{let}\ \bullet x = e\ \mathsf{in}\ e'] \Downarrow C''[v'']}\ \bullet\text{E}$$

$$\frac{C[[\mathsf{fix}\ x : A.\ e/x]e] \Downarrow C'[v]}{C[\mathsf{fix}\ x : A.\ e] \Downarrow C'[v]}\ \textsc{FixE}$$

$$\frac{C[e] \Downarrow C'[v] \qquad C'[[v/x]e'] \Downarrow C''[v'']}{C[\mathsf{let}\ x = e\ \mathsf{in}\ e'] \Downarrow C''[v'']}\ \textsc{LetE}$$

$$\frac{C[[t_1/u, t_2/v]e'] \Downarrow C'[v']}{C[\mathsf{let}\ \langle u, v\rangle = \langle t_1, t_2\rangle\ \mathsf{in}\ e'] \Downarrow C'[v']}\ \otimes\text{E} \qquad \frac{C[e] \Downarrow C'[v]}{C[!e] \Downarrow C'[!v]}\ !\text{I}$$

$$\frac{C[e] \Downarrow C'[!v] \qquad C'[[v/x]e'] \Downarrow C''[v'']}{C[\mathsf{let}\ !x = e\ \mathsf{in}\ e'] \Downarrow C''[v'']}\ !\text{E}$$

**Figure 5.** Within-Step Operational Semantics

To systematically answer these questions, we follow Krishnaswami and Benton [11] and make use of the category of metric spaces. A *complete 1-bounded bisected ultrametric space* $A$ (which we will simply call "ultrametric space") is a pair $(|A|, d)$, where $|A|$ is a set and $d \in |A| \times |A| \to [0, 1]$ is a distance function satisfying the folllowing properties:

1. $d(x, y) = 0$ iff $x = y$
2. $d(x, y) = d(y, x)$
3. $d(x, z) \leq \max(d(x, y), d(y, z))$
4. $d(x, y) = 0$ or $2^{-n}$ for some $n$
5. All Cauchy sequences have limits

We take the morphisms between ultrametric spaces to be the *nonexpansive maps* $f : A \to B$. These are the set-theoretic functions $f \in |A| \to |B|$ such that:

- For all $a, a' \in |A|$, we have $d_B(f\ a, f\ a') \leq d_A(a, a')$

$$\boxed{\mathsf{Step}(C[x])}$$

$$\begin{array}{l}\mathsf{Step}(\Box[x]) = x\\ \mathsf{Step}(\mathsf{let}\ x = \mathsf{tail}(y)\ \mathsf{in}\ C[z]) = \mathsf{let}\ x = y\ \mathsf{in}\ \mathsf{Step}(C[z])\\ \mathsf{Step}(\mathsf{let}\ x = \mathsf{cons}(u, v, u'.\ e)\ \mathsf{in}\ C[z]) =\\ \qquad \mathsf{let}\ x = [u/u']e\ \mathsf{in}\ \mathsf{Step}(C[z])\end{array}$$

$$\boxed{\mathsf{Step}(\Omega_i)}$$

$$\begin{array}{lcl}\mathsf{Step}(\cdot) & = & \cdot\\ \mathsf{Step}(\Omega_i, x :_i S(A)) & = & \mathsf{Step}(\Omega_i), x :_{i+1} S(A)\\ \mathsf{Step}(\Omega_i, x :_{i+1} S(A)) & = & \mathsf{Step}(\Omega_i), x :_{i+1} S(A)\end{array}$$

$$\boxed{\mathsf{Step}_{\Omega_i}(\omega) \in [\![\Omega_i]\!]_i \to [\![\mathsf{Step}(\Omega_i)]\!]_i}$$

$$\begin{array}{lcl}\mathsf{Step}_{\cdot}(\langle\rangle) & = & \langle\rangle\\ \mathsf{Step}_{\Omega_i, x:_i S(A)}((\omega, vs)) & = & (\mathsf{Step}_{\Omega_i}(\omega), \mathit{tail}(vs))\\ \mathsf{Step}_{\Omega_i, x:_{i+1} S(A)}((\omega, vs)) & = & (\mathsf{Step}_{\Omega_i}(\omega), vs)\end{array}$$

**Figure 6.** The Next Step Operator

- For all $a \in |A|$, we have $\sigma_B(f\ a) \leq \sigma_A(a)$

That is, a morphism between $A$ and $B$ is a function $f$ such that it takes any two points in $A$ to two points in $B$ that are at least as close — it is a non-distance-increasing function.

The category of ultrametric spaces proved useful for two reasons. First, the causal stream functions are *exactly* the nonexpansive maps between spaces of streams with the Cantor metric (i.e., the distance between two streams is $2^{-n}$, where $n$ is the first position at which they disagree). Since the category of 1-bounded complete ultrametric spaces is Cartesian closed, we have our higher-type generalization of causality — one which would be very difficult to find from purely operational considerations. Second, nonempty metric spaces satisfy Banach's theorem, which lets us define fixed points at arbitrary types:

**Proposition 1.** *(Banach's Contraction Map Theorem) If $A$ is a nonempty complete metric space, and $f : A \to A$ is a strictly contractive function, then $f$ has a unique fixed point.*

### 5.2 Modeling Space Bounds with Length Spaces

As we noted earlier, casuality still permits defining unwanted functions. Requiring a stream function to depend only on its history does not prevent it from depending on its *whole* history.

To account for this problem, we adapt the *length spaces* of Hofmann [8], which give a denotational model of space-bounded computation. The idea behind this model is to start with a partially ordered resource monoid $R$ representing space resources ($\mathbb{N}$ in the original work). Then, we construct the category of length spaces as follows.

A *length space* $A$ is a pair $(|A|, \sigma_A : |A| \to R)$, consisting of a set of elements $|A|$ and a *size function* $\sigma$ which assigns a size $\sigma_A(a)$ to each element $a \in |A|$. A morphism of length spaces $f : A \to B$ is a *non-size-increasing function*. That is, it is a set-theoretic function $f \in |A| \to |B|$ with the property that:

$$\forall a \in |A|.\ \sigma_B(f\ a) \leq \sigma_A(a)$$

The programming language intuition is that a morphism $A \to B$ is a term of type $B$ with a free variable in $A$, and so a term cannot use more memory than it receives from its environment.

To model the permission to allocate, we can define a type $\Diamond \triangleq (1, \lambda\langle\rangle.\ 1)$. The space $\Diamond$ is uninteresting computationally (its set only has the unit in it), but it brings a permission to allocate with it. So we can model computations which do allocation by giving them permission elements, thereby controlling the allocation performed.

# 6. The Denotational Semantics

## 6.1 The Resource Model

In the synchronous dataflow model, there is a global, ambient notion of time. Furthermore, higher-order reactive programs can create a dataflow graph dynamically, by waiting for an event before choosing to build cons cells to do some computation. So we need a resource structure capable of modeling space usage over time.

Therefore we take resources to be the monoidal lattice $(R = \mathrm{Time} \to \mathrm{Space}, \bot, \max, \top, \min, 0, \oplus, \leq)$, where we take $\mathrm{Time} = \mathbb{N}$, and $\mathrm{Space} = \mathbb{N} \uplus \{\infty\}$ (the vertical natural numbers with a topmost element). Intuitively, time is discrete, and measured in ticks. Space counts the number of cons cells used in the program, and may be infinite (obviously, we cannot implement such programs). We define the lattice operations as follows:

1. $\bot = \lambda k.\ 0$
2. $\top = \lambda k.\ \infty$
3. $0 = \lambda k.\ 0$
4. $\max(c, d) = \lambda k.\ \max(c_k, d_k)$
5. $\min(c, d) = \lambda k.\ \min(c_k, d_k)$
6. $c \oplus d = \lambda k.\ c_k + d_k$
7. $c \leq d$ iff $\forall k \in \mathrm{Time}$, we have $c_k \leq d_k$

Essentially, we lift the lattice structure of the vertical natural numbers pointwise across time (with $(0, +)$ as the monoidal structure), so that a resource $c \in R$ describes the number of cons cells that are used at each time step.

We then turn $R$ into an ultrametric space by equipping it with the Cantor metric:

$$d_R(c, d) = 2^{-n} \text{ where } n = \min\{k \in \mathrm{Time} \mid c_k \neq d_k\}$$

## 6.2 The Category of Complete Ultrametric Length Spaces

A *complete 1-bounded bisected ultrametric length space* $A$ (which we will gloss as "metric length space") is a tuple $(|A|, d, \sigma)$, where $(|A|, d)$ is a complete 1-bounded bisected ultrametric space, and $\sigma_A : |A| \to R$ is a size function giving each element of $|A|$ a size drawn from $R$.

Furthermore, the size function $\sigma : |A| \to R$ must be a nonexpansive map between $(|A|, d)$ and $(R, d_R)$. Nonexpansiveness ensures that we cannot tell if the memory usage requirements of two elements of $|A|$ differs until we know that the elements themselves differ. In addition to being intuitively reasonable, this requirement ensures that limits of Cauchy sequences will be well-behaved with respect to size, which is important for ensuring that the our interpretation of $!A$ as the subspace of $A$ with size 0.

The morphisms of this category are the *nonexpansive size-preserving maps* $f : A \to B$, which are the set-theoretic functions $f \in |A| \to |B|$ such that:

- For all $a, a' \in |A|$, we have $d_B(f\ a, f\ a') \leq d_A(a, a')$
- For all $a \in |A|$, we have $\sigma_B(f\ a) \leq \sigma_A(a)$

That is, the morphisms we consider are the functions which are both causal and space-bounded.

## 6.3 Categorical Structure

Metric length spaces and nonexpansive size-preserving maps form a category that we use to interpret our programming language. First, it forms an intuitionistic bicartesian BI category, which is a doubly-closed category with both cartesian and monoidal closed structure, as well as supporting coproduct structure. Second, this category also models the resource types $\Diamond$ of Hofmann [8], as well as a resource-freedom modality $!A$, which is comonadic in the usual fashion of linear logic. Third, it supports a version of the delay modality of Krishnaswami and Benton [11], which lets us interpret guarded recursion via Banach's fixed point theorem.

We give the definitions of all of these objects below. In Figure 8, we define the distance and size functions, and in Figure 7, we dgive the natural transformations associated with the objects.

- $1 = (\{*\}, d_1, \sigma_1)$
- $A + B = (|A| + |B|, d_{A+B}, \sigma_{A+B})$
- $A \times B = (|A| \times |B|, d_{A\times B}, \sigma_{A\times B})$
- $A \Rightarrow B = (|A| \to |B|, d_{A\to B}, \sigma_{A\to B})$
- $A + B = (|A| + |B|, d_{A+B}, \sigma_{A+B}$
- $A \star B = (|A| \times |B|, d_{A\star B}, \sigma_{A\star B})$
- $A \mathrel{-\!\!\star} B = (|A| \to |B|, d_{A\to B}, \sigma_{A\to B})$
- $\bullet A = (|A|, d_{\bullet A}, \sigma_{\bullet A})$
- $S(A) = (|A|^\omega, d_{S(A)}, \sigma_{S(A)})$
- $!A = (\{a \in A \mid \sigma_A(a) = 0\}, \sigma_{!A})$
- $\Diamond = (\{*\}, d_1, \sigma_\Diamond)$
- $\diamond = (\{*\}, d_1, \sigma_\diamond)$

We will merely point out some highlights of these constructions.

The construction of Cartesian and monoidal products closely follows that of Hofmann [8]. The Cartesian product is a "sharing product", in which the associated resources are available to both components (this explains the use of $\max$), and the monoidal product is a "disjoint product", in which the resources are divided between the two components (explaining the use of $\oplus$ in the size function). The best intuition for the closed structure comes from implementing first-class functions as closures: the monoidal exponential $A \mathrel{-\!\!\star} B$ takes an argument which does not share with the captured environment, and the Cartesian exponential $A \Rightarrow B$ which does.

A difference between our work and earlier work on length spaces is our heavy use of the category's Cartesian closed structure. Indeed, dal Lago and Hofmann's work on implicit complexity [13] uses realizability to eliminate this structure, since duplication of variables in lambda-terms makes establishing resource bounds more difficult. As we only want to track the allocation of cells, and want to allow free sharing otherwise, the CCC structure takes on a central role.

Our next-step modality's metric $d_{\bullet A}$ is the same as in Krishnaswami and Benton [11], but the size function $\sigma_{\bullet A}$ (which shifts all sizes 1 timestep into the future relative to $\sigma_A$), means that $\bullet(A \to B) \not\cong \bullet A \to \bullet B$. This breaks with Krishnaswami and Benton [11] and Nakano [15], significantly changing the elimination rules. We also no longer have a general delay operator $\delta_A : A \to \bullet A$; it is only present for certain types such as $!A$ and $\Diamond$.

The size function for streams gives a size at time $k$ of the size of the $k$-th element of the stream at time $k$, plus 1 to account for the size of the stream itself. This is the only place where we increment the size of a value relative to its components, which justifies the idea that sizes measure the number of cons cells.

Our semantic model contains a space $\Diamond$ to interpret the resource type $\Diamond$, which gives 1 unit of space at every time tick. Our model

$$
\begin{array}{lll}
1 & : A \to I & = \lambda a.\ * \\
\pi_1 & : A \times B \to A & = \lambda(a,b).\ a \\
\pi_2 & : A \times B \to B & = \lambda(a,b).\ b \\
\langle f, g\rangle & : A \to B \times C & = \lambda a.\ (f\ a, g\ a) \\
\quad \text{where } f : A \to B, g : A \to C & & \\
\lambda\langle f\rangle & : A \to B \Rightarrow C & = \lambda a.\ \lambda b.\ f(a,b) \\
\quad \text{where } f : A \times B \to C & & \\
\mathit{eval} & : (A \Rightarrow B) \times A \to B & = \lambda(f,a).\ f\ a \\
f \star g & : A \star B \to C \star D & = \lambda(a,b).\ (f\ a, g\ b) \\
\quad \text{where } f : A \to C, g : B \to D & & \\
\alpha & : (A \star B) \star C \to A \star (B \star C) & = \lambda((a,b),c).\ (a,(b,c)) \\
\alpha^{-1} & : A \star (B \star C) \to (A \star B) \star C & = \lambda(a,(b,c)).\ ((a,b),c) \\
\gamma & : A \star B \to B \star A & = \lambda(a,b).\ (b,a) \\
\rho & : A \star I \to A & = \lambda(a,*).\ a \\
\rho^{-1} & : A \star I \to A & = \lambda a.\ (a,*) \\
\hat{\lambda}\langle f\rangle & : A \to B \multimap C & = \lambda a.\ \lambda b.\ f(a,b) \\
\quad \text{where } f : A \star B \to C & & \\
\mathit{eval}_{\multimap} & : (A \multimap B) \star A \to B & = \lambda(f,a).\ f\ a \\
\epsilon & : !A \to A & = \lambda a.\ a \\
f^{\dagger} & : !A \to !B & = \lambda a.\ f\ a \\
\quad \text{where } f : !A \to B & & \\
!f & : !A \to !B & = \lambda a.\ f\ a \\
\quad \text{where } f : A \to B & & \\
\delta & : !A \to \bullet !A & = \lambda a.\ a \\
\bullet f & : \bullet A \to \bullet B & = \lambda a.\ f\ a \\
\quad \text{where } f : A \to B & & \\
\eta & : !\bullet A \to \bullet !A & = \lambda a.\ a \\
\eta^{-1} & : \bullet !A \to \bullet A & = \lambda a.\ a \\
\mathit{head} & : S(A) \to A & = \lambda(x \cdot xs).\ x \\
\mathit{tail} & : S(A) \to \bullet S(A) & = \lambda(x \cdot xs).\ xs \\
\mathit{cons} & : \diamond \star (A \times \bullet S(A) \to S(A) & = \lambda(*,(x,xs)).\ x \cdot xs \\
\mathit{split} & : \Diamond \to \diamond \star \bullet \Diamond & = \lambda *.\ (*,*) \\
\mathit{split}^{-1} & : \diamond \star \bullet \Diamond \to \Diamond & = \lambda(*,*).\ * \\
\mathit{fix} & : !(\bullet !A \Rightarrow !A) \to !A & = \lambda f.\ \mu(f) \\
\iota & : \bullet A \times \bullet B \to \bullet(A \times B) & = \lambda(a,b).\ (a,b) \\
\iota^{-1} & : \bullet(A \times B) \to \bullet A \times \bullet B & = \lambda(a,b).\ (a,b) \\
\iota_{\star} & : \bullet A \star \bullet B \to \bullet(A \star B) & = \lambda(a,b).\ (a,b) \\
\iota_{\star}^{-1} & : \bullet(A \star B) \to \bullet A \star \bullet B & = \lambda(a,b).\ (a,b) \\
\xi & : A \star B \to A \times B & = \lambda(a,b).\ (a,b) \\
\sigma & : A \times !B \to A \star !B & = \lambda(a,b).\ (a,b) \\
\psi & : !(A \times B) \to !A \star !B & = \lambda(a,b).\ (a,b) \\
\psi^{-1} & : !A \star !B \to !(A \times B) & = \lambda(a,b).\ (a,b)
\end{array}
$$

**Figure 7.** Categorical Combinators

uses the additional metric length space $\diamond$, which gives 1 unit of space at time 0, and no units of space at any other time. This lets us give a nice type to $\mathit{cons} : \diamond \star (A \times \bullet S(A)) \to S(A)$. Note that the type $A \times \bullet S(A)$ lacks the space to form a stream — we need 1 unit of space at time 0, which neither the $A$ nor the $\bullet S(A)$ provide.

### 6.4 Denotational Interpretation

We give the interpretation of types and contexts in Figure 9. The interpretation of types offers no surprises, but the interpretation of contexts is relative to the current time. The interpretations of $\Theta$ and $\Delta$ keeps hypotheses at times earlier than the current time, but $\Gamma$ simply drops all earlier hypotheses. This corresponds to the difference between the type rules RHYP and PHYP on the one hand, and the rule EHYP on the other. In all three cases, future hypotheses are interpreted with the delay modality.

In Figure 10, we give a time-indexed interpretation function for expressions, $[\![\Theta; \Pi; \Gamma \vdash e :_i A]\!]_i$ which has the type $[\![\Theta]\!]_i \star ![\![\Pi]\!]_i \star [\![\Gamma]\!]_i \to [\![A]\!]$. The interpretation of $\bullet$I makes use of the functoriality of $\bullet A$ to interpret the body of the delay in the future, and then bring it back to the past, with the necessary action on contexts defined in Figure 9. The other rules are as expected, with the resource context managed in a single-threaded way and the other contexts duplicated freely. We can then show that this semantics is sound with respect to substitution.

$$
\begin{array}{lll}
d_{A+B} & = \lambda(v,v').\ \begin{cases} d_A(x,x') & \text{if } v = \mathsf{inl}\ x \wedge v' = \mathsf{inl}\ x' \\ d_B(y,y') & \text{if } v = \mathsf{inr}\ y \wedge v' = \mathsf{inr}\ y' \\ 1 & \text{otherwise} \end{cases} \\
d_1 & = \lambda(\langle\rangle, \langle\rangle).\ 0 \\
d_{A \times B} & = \lambda((a,b),(a',b')).\ \max(d_A(a,a'), d_B(b,b')) \\
d_{A \star B} & = \lambda((a,b),(a',b')).\ \max(d_A(a,a'), d_B(b,b')) \\
d_{A \Rightarrow B} & = \lambda(f,g).\ \max\{d_B(f\ a, g\ a) \mid a \in |A|\} \\
d_{A \multimap B} & = \lambda(f,g).\ \max\{d_B(f\ a, g\ a) \mid a \in |A|\} \\
d_{\bullet A} & = \lambda(a,a').\ \frac{1}{2} d_A(a,a') \\
d_{S(A)} & = \lambda(xs,ys).\ \max\{2^{-n} \cdot d_A(xs_n, ys_n) \mid n \in \mathbb{N}\} \\
d_{!A} & = \lambda(a,a').\ d_A(a,a') \\
d_{\Diamond} & = \lambda(\langle\rangle, \langle\rangle).\ 0 \\
d_{\diamond} & = \lambda(\langle\rangle, \langle\rangle).\ 0
\end{array}
$$

$$
\begin{array}{lll}
\sigma_{A+B} & = \lambda v.\ \begin{cases} \sigma_A(a) & \text{if } v = \mathsf{inl}\ a \\ \sigma_B(b) & \text{if } v = \mathsf{inr}\ b \end{cases} \\
\sigma_1 & = \lambda\langle\rangle.\ 0 \\
\sigma_{A \times B} & = \lambda(a,b).\ \max(\sigma_A(a), \sigma_B(b)) \\
\sigma_{A \star B} & = \lambda(a,b).\ \sigma_A(a) \oplus \sigma_B(b) \\
\sigma_{A \Rightarrow B} & = \lambda f.\ \min\{k \in R \mid \forall a \in |A|.\ \sigma_B(f\ a) \le \max(k, \sigma_A(a))\} \\
\sigma_{A \multimap B} & = \lambda f.\ \min\{k \in R \mid \forall a \in |A|.\ \sigma_B(f\ a) \le k \oplus \sigma_A(a)\} \\
\sigma_{\bullet A} & = \lambda a.\ \lambda k.\ \text{if } k = 0 \text{ then } 0 \text{ else } \sigma_A(a)(k-1) \\
\sigma_{S(A)} & = \lambda xs.\ \lambda k.\ 1 + \sigma_A(xs(k))(k) \\
\sigma_{\Diamond} & = \lambda\langle\rangle.\ \lambda k.\ 1 \\
\sigma_{\diamond} & = \lambda\langle\rangle.\ \lambda k.\ \text{if } k = 0 \text{ then } 1 \text{ else } 0 \\
\sigma_{!A} & = \lambda a.\ 0
\end{array}
$$

**Figure 8.** The size and distance functions

**Theorem 6.** *(Semantic Substitution) Suppose $\theta \in [\![\Theta]\!]_i$, $\pi \in [\![\Pi]\!]_i$ and $\gamma \in [\![\Gamma]\!]_i$. Then*

1. *If $\Theta \vdash t :_i R$ and $\Theta', u :_i R \vdash t' :_j R'$, then*
   *$[\![\Theta, \Theta' \vdash [t/u]t' :_j R']\!]\ \theta$ is equal to*
   *$[\![\Theta', u :_i R \vdash t' :_j R']\!]\ (\theta, [\![\Theta \vdash t :_i R]\!]\ \theta)$.*
2. *If $\Theta \vdash t :_i R$ and $\Theta, u :_i R; \Pi; \Gamma \vdash e :_j A$, then*
   *$[\![\Theta, \Theta'; \Pi; \Gamma \vdash [t/u]e :_j A]\!]\ (\theta, \pi, \gamma)$ equals*
   *$[\![\Theta, u :_i R; \Pi; \Gamma \vdash e :_j A]\!]\ ((\theta, [\![\Theta \vdash t :_i R]\!]\ \theta), \pi, \gamma)$*
3. *If $\cdot; \Pi; \cdot \vdash e :_i A$ and $\Theta; \Pi, x :_j A; \Gamma \vdash e' :_k B$ and $i \le j$, then*
   *$[\![\Theta; \Pi; \Gamma \vdash [e/x]e' :_k B]\!]\ (\theta, \pi, \gamma)$ equals*
   *$[\![\Theta; \Pi, x :_j A; \Gamma \vdash e' :_k B]\!]\ (\theta, (\pi, [\![\cdot; \Pi; \cdot \vdash e :_i A]\!]\ (\langle\rangle, \pi, \langle\rangle)), \gamma)$*
4. *If $\cdot; \Pi; \Gamma \vdash e :_i A$ and $\Theta; \Pi; \Gamma, x :_i A \vdash e' :_j B$, then*
   *$\Theta; \Pi; \Gamma \vdash [e/x]e' :_j B$ equals*
   *$[\![\Theta; \Pi; \Gamma, x :_i A \vdash e' :_j B]\!]\ (\theta, \pi, (\gamma, [\![\cdot; \Pi; \Gamma \vdash e :_i A]\!]\ (\langle\rangle, \pi, \gamma)))$*

In Figure 11, we give an interpretation of contexts, which uses the expression semantics to define the meanings of each stream bound by the context. Then we can show that this interpretation is sound with respect to the "hole-filling" of terms in contexts:

**Theorem 7.** *(Context Soundness) If $\Sigma \rhd_i \Omega_i \vdash C \dashv \Omega_i'$ and $\Sigma'; \cdot; \Omega_i, \Omega_i' \vdash e :_i A$ and $\sigma \in [\![\Sigma]\!]_i$ and $\sigma' \in [\![\Sigma']\!]_i$ and $\omega \in [\![\Omega_i]\!]_i$, then $[\![\Sigma, \Sigma'; \cdot; \Omega_i \vdash C[e] :_i A]\!]$ is equal to $[\![\Sigma'; \cdot; \Omega_i, \Omega_i' \vdash e :_i A]\!]$ $(\sigma', \langle\rangle, (\omega, [\![\Sigma \rhd_i \Omega_i \vdash C \dashv \Omega_i']\!]\ (\sigma, \omega))$*

Now we can show that the within-step operational semantics is sound with respect to the denotational model:

**Theorem 8.** *(Soundness of Within-Step Semantics) Suppose* $\Sigma; \cdot; \cdot \vdash C[e] :_i A$ *and* $C[e] \Downarrow C'[v]$. *Then* $[\![\Sigma; \cdot; \cdot \vdash C[e] :_i A]\!]$ *equals* $[\![\Sigma; \cdot; \cdot \vdash C'[v] :_i A]\!]$.

Finally, we can show that advancing the clock has the expected semantics:

**Theorem 9.** *(Soundness of Advancing the Clock) Suppose* $\Sigma; \cdot; \Omega_i \vdash C[x] :_i S(A)$, *and* $\sigma \in [\![\Sigma]\!]_i$ *and* $\omega \in [\![\Omega_i]\!]_i$. *Then we have that*

$$\mathsf{tail}([\![\Sigma; \cdot; \Omega_i \vdash C[x] :_i S(A)]\!]\ (\sigma, \langle\rangle, \omega))$$

*is equal to* $\bullet[\![\Sigma; \cdot; (\mathsf{Step}(\Omega_i) \vdash \mathsf{Step}(C[x]) :_{i+1} S(A)]\!]\ (\sigma', \langle\rangle, \omega'))$ *where* $\sigma' = \mathrm{Next}^i_\Sigma\ \sigma$ *and* $\omega' = \mathsf{Step}_{\Omega_i}(\omega)$.

This theorem connects the operation of stepping the heap with the denotational interpretation — each time we advance the clock, each stream in a closed context $C$ will become its tail. So advancing the clock will successively enumerate the elements of the stream.

## 7. Discussion

In this paper, we have introduced an expressive type system for writing stream programs, and given an operational semantics respecting the space-efficiency claims of the type system. Our semantic model is one of the primary contributions of this paper, since it lets us reason about space usage without wholly surrendering the sets-and-functions view of FRP. Also, our model contains many operations which are not currently expressible in our language: for example, in the future we might want richer types in the affine context and function space, so that operations like in-place map can be typed $!(A \to B) \to S(A) \multimap S(B)$.

Wan et al. [20] introduced real-time FRP; a restricted subset of FRP sharing many of the same design choices of synchronous dataflow languages. It is essentially first-order (streams can carry general values of the host language, but these values can not themselves refer to streams), and makes use of a novel continuation typing to ensure that all recursive signals are tail-recursive. As a result, the language requires only constant-stack and constant-time FRP reductions. Event-driven FRP [21] is similar, but relaxed FRP's timing constraints by dropping the global clock.

Liu et al.'s causal commutative arrows [14] are another attempt to control the memory and space usage of reactive programs. This work takes advantage of the fact that pure arrowized FRP (i.e., without switching) builds fixed dataflow graphs to transform programs into single-loop code via a transformation reminiscent of the Bohm-Jacopini theorem. Our language is not amenable to such a transformation, since it fully supports higher-order streams and functions.

Sculthorpe and Nilsson's work [19] on safe functional programming uses Agda's types to ensure productivity, by having dependent types track whether signal processors have delays before permitting feedback. Our guardedness modality is simpler but less flexible, since it cannot depend on the *values* a signal produces. However, modalities work smoothly at higher-order, which is (again) where our work shines.

Cooper and Krishnamurthi [5] described the FrTime system, which embeds FRP into the PLT Scheme (now Racket) implementation. One commonality between FrTime and our work is that switching does not come from special primitives, but from ordinary conditionals and case statements. Unlike our denotational model, Cooper's operational semantics [4] exemplifies the "imperative FRP" tradition, in which reactivity is modeled explicitly as a kind of heap update in the operational semantics. We think the operational semantics in this paper, which is quasi-imperative in flavor, is close enough to both a denotational model and Cooper's semantics that it makes sense to study how to reunify the pure and the imperative flavors of FRP.

$$
\begin{array}{lcl}
[\![A \to B]\!] & = & [\![A]\!] \to [\![B]\!] \\
[\![S(A)]\!] & = & S([\![A]\!]) \\
[\![\bullet A]\!] & = & \bullet[\![A]\!] \\
[\![!A]\!] & = & ![\![A]\!] \\
[\![R \multimap A]\!] & = & [\![R]\!] \multimap\!\star [\![A]\!] \\
[\![\Diamond]\!] & = & \Diamond \\
[\![R \otimes S]\!] & = & [\![R]\!] \star [\![S]\!]
\end{array}
$$

$\boxed{[\![\Theta]\!]_i}$ $\boxed{[\![\Pi]\!]_i}$ $\boxed{[\![\Gamma]\!]_i}$

$$
\begin{array}{lcll}
[\![\Theta]\!]_i & \in & \mathrm{Obj} & \\
[\![\cdot]\!]_i & = & I & \\
[\![\Theta, x :_n R]\!]_i & = & [\![\Theta]\!]_i \star \bullet^{n-i}[\![R]\!] & \text{when } n \geq i \\
[\![\Theta, x :_n R]\!]_i & = & [\![\Theta]\!]_i \star [\![R]\!] & \text{when } n < i
\end{array}
$$

$$
\begin{array}{lcll}
[\![\Pi]\!]_i & \in & \mathrm{Obj} & \\
[\![\cdot]\!]_i & = & I & \\
[\![\Pi, x :_n A]\!]_i & = & [\![\Pi]\!]_i \times \bullet^{n-i}[\![A]\!] & \text{when } n \geq i \\
[\![\Pi, x :_n A]\!]_i & = & [\![\Pi]\!]_i \times [\![A]\!] & \text{when } n < i
\end{array}
$$

$$
\begin{array}{lcll}
[\![\Gamma]\!]_i & \in & \mathrm{Obj} & \\
[\![\cdot]\!]_i & = & I & \\
[\![\Gamma, x :_n A]\!]_i & = & [\![\Gamma]\!]_i \times \bullet^{n-i}[\![A]\!] & \text{when } n \geq i \\
[\![\Gamma, x :_n A]\!]_i & = & [\![\Gamma]\!]_i & \text{when } n < i
\end{array}
$$

$\boxed{\mathrm{Next}^i_\Theta \in [\![\Theta]\!]_i \to \bullet[\![\Theta]\!]_{i+1}}$ $\boxed{\mathrm{Next}^i_\Pi \in [\![\Pi]\!]_i \to \bullet[\![\Pi]\!]_{i+1}}$

$\boxed{\mathrm{Next}^i_\Gamma \in [\![\Gamma]\!]_i \to \bullet[\![\Gamma]\!]_{i+1}}$

$$
\begin{array}{lcll}
\mathrm{Next}^i_\cdot\ \langle\rangle & = & \langle\rangle & \\
\mathrm{Next}^i_{\Theta, x :_n R}\ (\theta, r) & = & (\mathrm{Next}^i_\Theta\ \theta, \delta(r)) & \text{if } n \leq i \\
\mathrm{Next}^i_{\Theta, x :_n R}\ (\theta, r) & = & (\mathrm{Next}^i_\Theta\ \theta, r) & \text{if } n > i
\end{array}
$$

$$
\begin{array}{lcll}
\mathrm{Next}^i_\cdot\ \langle\rangle & = & \langle\rangle & \\
\mathrm{Next}^i_{\Pi, x :_n A}(\pi, v) & = & (\mathrm{Next}^i_\Pi\ \pi, \delta(v)) & \text{if } n \leq i \\
\mathrm{Next}^i_{\Pi, x :_n A}(\pi, v) & = & (\mathrm{Next}^i_\Pi\ \pi, v) & \text{if } n > i
\end{array}
$$

$$
\begin{array}{lcll}
\mathrm{Next}^i_\cdot\ \langle\rangle & = & \langle\rangle & \\
\mathrm{Next}^i_{\Gamma, x :_n A}(\gamma, v) & = & \mathrm{Next}^i_\Gamma & \text{if } n \leq i \\
\mathrm{Next}^i_{\Gamma, x :_n A}(\gamma, v) & = & (\mathrm{Next}^i_\Gamma\ \gamma, v) & \text{if } n > i
\end{array}
$$

**Figure 9.** Interpretation of Types and Contexts

Krishnaswami and Benton [12] have also presented a language for writing reactive GUI programs. This language also makes use of linear types to track the allocation of new GUI widgets, though they prove no space bounds. It is not yet clear how to combine space-boundedness with this kind of dynamism: we may need to add a dynamic allocation monad to our model to integrate this work.

Supporting other dynamic data structures (not necessarily streams) suggests looking at the work of Acar et al. [1], who have studied adding user-controlled incrementalization to self-adjusting computation, which shares many implementation issues with FRP.

## References


[1] U. A. Acar, G. E. Blelloch, R. Ley-Wild, K. Tangwongsan, and D. Türkoglu. Traceable data types for self-adjusting computation. In *PLDI*, pages 483–496, 2010.

$$\boxed{[\![\Theta \vdash t :_i R]\!]_i \in [\![\Theta]\!]_i \to [\![R]\!]}$$

$$\boxed{[\![\Theta; \Pi; \Gamma \vdash e :_i A]\!]_i \in [\![\Theta]\!]_i \star !^{[\![\Pi]\!]_i} \star [\![\Gamma]\!]_i \to [\![A]\!]}$$

$[\![\Theta \vdash u :_j R]\!]_i\ \theta = \theta(u)$
$[\![\Theta, \Theta' \vdash (t, t') :\ R \otimes S(\theta, \theta')]\!]_i = ([\![t]\!]\ \theta, [\![t']\!]\ \theta')$

$[\![\Theta; \Pi; \Gamma \vdash x :_j A]\!]_i\ (\theta, \pi, \gamma) = \pi(x)\ \ (\text{if } x :_i A \in \Pi)$

$[\![\Theta; \Pi; \Gamma \vdash x :_i A]\!]_i\ (\theta, \pi, \gamma) = \gamma(x)\ \ (\text{if } x :_i A \in \Gamma)$

$[\![\Theta; \Pi; \Gamma \vdash \lambda x : A.\ e :_i A \to B]\!]_i\ (\theta, \pi, \gamma) =$
$\quad \lambda v.\ [\![\cdot; \Pi; \Gamma, x :_i A \vdash e :_i B]\!]_i\ (\langle\rangle, \pi, (\gamma, v))$

$[\![\Theta, \Theta'; \Pi; \Gamma \vdash e\ e' :_i B]\!]_i\ ((\theta, \theta'), \pi, \gamma) =$
$\quad \mathsf{let}\ f = [\![e : A \to B]\!]_i\ (\theta, \pi, \gamma)\ \mathsf{in}$
$\quad \mathsf{let}\ v = [\![e' : A]\!]_i\ (\theta', \pi, \gamma)\ \mathsf{in}$
$\quad \mathit{eval}(f, v)$

$[\![\Theta; \Pi; \Gamma \vdash !e :_i !A]\!]_i\ (\theta, \pi, \gamma) = [\![\cdot; \Pi; \cdot \vdash e :_i A]\!]_i^\dagger\ (\langle\rangle, \pi, \langle\rangle)$

$[\![\Theta, \Theta'; \Pi; \Gamma \vdash \mathsf{let}\ !x = e\ \mathsf{in}\ e' :_i B]\!]_i\ (\theta, \pi, \gamma) =$
$\quad [\![e']\!]_i\ (\theta', \psi^{-1}(\pi, [\![e]\!]_i\ (\theta', \pi, \gamma)), \gamma)$

$[\![\Theta, \Theta', \Theta''; \Pi; \Gamma \vdash \mathsf{cons}(t, e, u'.\ e') :_i S(A)]\!]_i\ ((\theta, \theta', \theta''), \pi, \gamma) =$
$\quad \mathsf{let}\ (d, r) = \mathit{split}([\![t]\!]_i\ \theta)\ \mathsf{in}$
$\quad \mathsf{let}\ h = [\![e]\!]_i\ (\theta', \pi, \gamma)\ \mathsf{in}$
$\quad \mathsf{let}\ t = \bullet[\![e']\!]_{i+1}\ ((\mathrm{Next}^i_{\Theta''}\ \theta'', r), \mathrm{Next}^i_\Pi\ \pi, \mathrm{Next}^i_\Gamma\ \gamma)\ \mathsf{in}$
$\quad \mathit{cons}(d, (h, t))$

$[\![\Theta; \Pi; \Gamma \vdash \mathsf{head}(e) :_i A]\!]_i\ (\theta, \pi, \gamma) =$
$\quad \mathit{head}([\![\Theta; \Pi; \Gamma \vdash e :_i S(A)]\!]_i\ (\theta, \pi, \gamma))$

$[\![\Theta, \Theta'; \Pi; \Gamma \vdash \mathsf{let}\ x = \mathsf{tail}(e)\ \mathsf{in}\ e' :_i B]\!]_i\ ((\theta, \theta'), \pi, \gamma) =$
$\quad \mathsf{let}\ vs = [\![\Theta]\!]_i \Pi\Gamma e S(A) i\ \mathsf{in}\ (\theta, \pi, \gamma)$
$\quad [\![\Theta'; \Pi; \Gamma, x :_{i+1} S(A) \vdash e' :_i B]\!]_i\ (\theta', \pi, (\gamma, \mathit{tail}(vs)))$

$[\![\Theta; \Pi; \Gamma \vdash \bullet e :_i \bullet A]\!]_i\ (\theta, \pi, \gamma) =$
$\quad \bullet[\![\cdot; \Pi; \Gamma \vdash e :_{i+1} A]\!]_{i+1}\ (\langle\rangle, \mathrm{Next}^i_\Pi\ \pi, \mathrm{Next}^i_\Gamma\ \gamma)$

$[\![\Theta, \Theta'; \Pi; \Gamma \vdash \mathsf{let}\ \bullet x = e\ \mathsf{in}\ e' :_i B]\!]_i\ ((\theta, \theta'), \pi, \gamma) =$
$\quad \mathsf{let}\ v = [\![\Theta; \Pi; \Gamma \vdash e :_i \bullet A]\!]_i\ (\theta, \pi, \gamma)\ \mathsf{in}$
$\quad [\![\Theta'; \Pi; \Gamma, x :_{i+1} A \vdash e' :_i B]\!]_i\ (\theta', \pi, (\gamma, v))$

$[\![\Theta; \Pi; \Gamma \vdash \lambda u : R.\ e :_i R \multimap A]\!]_i\ (\theta, \pi, \gamma) =$
$\quad \lambda r.\ [\![u :_i R; \Pi; \Gamma \vdash e :_i A]\!]\ (r, \pi, \gamma)$

$[\![\Theta, \Theta'; \Pi; \Gamma \vdash e\ t :_i A]\!]_i\ ((\theta, \theta'), \pi, \gamma) =$
$\quad \mathit{eval}_\star([\![e]\!]\ (\theta, \pi, \gamma), [\![t]\!]\ \theta')$

$[\![\Theta, \Theta'; \Pi; \Gamma \vdash \mathsf{let}\ (u, v) = t\ \mathsf{in}\ e :_i A]\!]_i\ ((\theta, \theta'), \pi, \gamma) =$
$\quad \mathsf{let}\ (r, s) = [\![\Theta \vdash t :_i R \otimes S]\!]\ \theta\ \mathsf{in}$
$\quad [\![\Theta', u :_i R, v :_i S; \Pi; \Gamma \vdash e :_i A]\!]\ ((\theta', r, s), \pi, \gamma)$

$[\![\Theta; \Pi; \Gamma \vdash \mathsf{fix}\ x : A.\ e :_i A]\!]_i\ (\theta, \pi, \gamma) =$
$\quad \mathsf{let}\ f = \lambda v.\ \left(\begin{array}{l}\mathsf{let}\ \pi' = \psi^{-1}(\pi, \eta^{-1}\ v)\ \mathsf{in}\\ [\![\cdot; \Pi, x :_{i+1} A; \cdot \vdash e :_i A]\!]_i\ (\langle\rangle, \pi', \langle\rangle)\end{array}\right)\ \mathsf{in}$
$\quad \mathit{fix}(f)$

$[\![\Theta, \Theta'; \Pi; \Gamma \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' :_i B]\!]_i\ ((\theta, \theta'), \pi, \gamma) =$
$\quad \mathsf{let}\ v = [\![\Theta; \Pi; \Gamma \vdash e :_i A]\!]_i\ (\theta, \pi, \gamma)\ \mathsf{in}$
$\quad [\![\Theta'; \Pi; \Gamma, x :_i A \vdash e' :_i B]\!]_i\ (\theta', (\pi, v), \gamma)$

**Figure 10.** Denotational Semantics of Terms

$$\boxed{[\![\Sigma \rhd_i \Omega_i \vdash C \dashv \Omega_i']\!] \in [\![\Sigma]\!]_i \star [\![\Omega_i]\!] \to [\![\Omega_i']\!]}$$

$[\![\Sigma \rhd_i \Omega_i \vdash \Box \dashv \cdot]\!]\ (\theta, \gamma) = \langle\rangle$
$[\![\Sigma \rhd_i \Omega_i \vdash \mathsf{let}\ y = \mathsf{tail}(x)\ \mathsf{in}\ C \dashv y :_{i+1} S(A), \Omega_i']\!]\ (\theta, \gamma) =$
$\quad \mathsf{let}\ v = \mathit{tail}(\gamma(x))\ \mathsf{in}$
$\quad \mathsf{let}\ \gamma' = [\![\Sigma \rhd_i \Omega_i, y :_{i+1} S(A) \vdash C \dashv \Omega_i']\!]\ (\theta, (\gamma, v))\ \mathsf{in}$
$\quad (\gamma', v)$
$[\![\Sigma, \Sigma', \Sigma'' \rhd_i \Omega_i \vdash \mathsf{let}\ x = \mathsf{cons}(u, v, u'.\ e)\ \mathsf{in}\ C \dashv \Omega_i', x :_{i+1} S(A)]\!]$
$\quad ((\theta, \theta', \theta''), \gamma) =$
$\quad \mathsf{let}\ (r, d) = [\![\Sigma' \vdash u :_i \Diamond]\!]\ \theta'\ \mathsf{in}$
$\quad \mathsf{let}\ h = [\![\cdot; \cdot; \Omega_i \vdash v :_i A]\!]\ (\langle\rangle, \langle\rangle, \gamma)\ \mathsf{in}$
$\quad \mathsf{let}\ \theta_1'' = (\mathrm{Next}^i_{\Sigma''}\ \theta'', d)\ \mathsf{in}$
$\quad \mathsf{let}\ \gamma_1 = \mathrm{Next}^{\Omega_i}_\gamma\ \mathsf{in}$
$\quad \mathsf{let}\ t = \bullet[\![\Sigma'', u' :_{i+1} \Diamond; \cdot; \Omega_i \vdash e :_{i+1} S(A)]\!]\ (\theta_1'', \langle\rangle, \gamma_1')\ \mathsf{in}$
$\quad \mathsf{let}\ vs = \mathit{cons}(r, (h, t))\ \mathsf{in}$
$\quad \mathsf{let}\ \gamma' = [\![\Sigma \rhd_i \Omega_i, x :_i S(A) \vdash C \dashv \Omega_i']\!]\ (\theta, (\gamma, vs))\ \mathsf{in}$
$\quad (\gamma', vs)$

**Figure 11.** Interpretation of Contexts


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

[3] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE: A declarative language for real-time programming. In *POPL*, 1987.

[4] G. Cooper. *Integrating dataflow evaluation into a practical higher-order call-by-value language*. PhD thesis, Brown University, 2008.

[5] G. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. *Programming Languages and Systems*, pages 294–308, 2006.

[6] C. Elliott and P. Hudak. Functional reactive animation. In *ICFP*, 1997.

[7] M. Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4), 2000.

[8] M. Hofmann. Linear types and non-size-increasing polynomial time computation. *Information and Computation*, 183(1), 2003.

[9] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots and functional reactive programming. In *Advanced Functional Programming*, volume 2638 of *LNCS*. Springer, 2003.

[10] J. Hughes. Generalizing monads to arrows. *Sci. Comput. Program.*, 37(1-3), 2000.

[11] N. Krishnaswami and N. Benton. Ultrametric semantics of reactive programs. In *LICS*. IEEE, 2011.

[12] N. Krishnaswami and N. Benton. A semantic model of graphical user interfaces. In *ICFP*, 2011.

[13] U. D. Lago and M. Hofmann. Realizability models and implicit complexity. *Theor. Comput. Sci.*, 412(20):2029–2047, 2011.

[14] H. Liu, E. Cheng, and P. Hudak. Causal commutative arrows and their optimization. In *ICFP*, 2009.

[15] H. Nakano. A modality for recursion. In *LICS*, pages 255–266, 2000.

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

[17] P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2), 1999.

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

[19] N. Sculthorpe and H. Nilsson. Safe functional reactive programming through dependent types. In *ICFP*, 2009.

[20] Z. Wan, W. Taha, and P. Hudak. Real-time frp. In *ICFP*, pages 146–156, 2001.

[21] Z. Wan, W. Taha, and P. Hudak. Event-driven frp. In *PADL*, pages 155–172, 2002.