Ultrametric Semantics of Reactive Programs

Neelakantan R. Krishnaswami
Microsoft Research
<neelk@microsoft.com>

Nick Benton
Microsoft Research
<nick@microsoft.com>

***Abstract*—We describe a denotational model of higher-order functional reactive programming using ultrametric spaces and nonexpansive maps, which provide a natural Cartesian closed generalization of causal stream functions and guarded recursive definitions. We define a type theory corresponding to this semantics and show that it satisfies normalization. Finally, we show how reactive programs written in this language may be implemented efficiently using an imperatively updated dataflow graph, and give a separation logic proof that this low-level implementation is correct with respect to the high-level semantics.**

I. Introduction

There is a broad spectrum of models for reactive programming. Functional reactive programming (FRP), as introduced by Elliott and Hudak [1], is highly expressive and generally shallowly embedded in powerful general-purpose languages. At the other end, synchronous languages such as Esterel [2], Lustre [3] and Lucid Synchrone [4] provide a restricted, domain-specific model of computation supporting specialized compilation strategies and analysis techniques. Synchronous languages have been extremely successful in applications such as hardware synthesis and embedded control software, and provide strong guarantees about bounded usage of space and time. FRP was initially aimed at dynamic interactive applications running is less resource-constrained environments, such as desktop GUIs, games and web applications. Even in such environments, however, naive versions of FRP are too unconstrained to be implemented efficiently (the implementations are far from naive, but it is still all too easy to introduce significant space and time leaks), but also too unconstrained from the point of view of the programmer, allowing unimplementable programs (e.g. ones that violate causality) to be written. Some recent variants of FRP [5], [6] restrict the model to rule out non-causal functions and ill-formed feedback.

In practice, of course, interactive GUIs and the like are usually implemented in general-purpose languages in a very imperative style. A program implements dynamic behavior by modifying state, and accepting callbacks to modify its own state. These programs exhibit complex aliasing, tricky control flow through callback functions living in the heap, and in general are very difficult to reason about. Part of the difficulty is the inherent complexity of verifying programs using such powerful features, but an even more fundamental problem is that it is not immediately clear even what the semantics of such programs should be; even the most powerful verification techniques are useless without a specification for a program to meet.

The starting point for the work described here is the *synchronous dataflow* paradigm of, for example, Lustre [3] and Lucid Synchrone [4]. We wish to be able to write complex dynamic reactive applications in a high-level declarative style, without abandoning the efficient stateful execution model that synchronous languages provide, at least for the first-order parts of our programs. To this end, we first present a new semantic model for reactive programs in terms of ultrametric spaces, generalizing previous models based on causal stream functions. Our model is Cartesian closed and so yields a mathematically natural semantics for higher-order reactive programs.

Using (ultra)metric spaces lets us use Banach's contraction map theorem to interpret feedback. Unlike earlier semantics based on domain models of streams, we can thus restrict our semantics to *total, well-founded* stream programs. Furthermore, by using an abstract notion of contractiveness instead of an explicit notion of guardedness, our semantics lifts easily to model higher-type streams (e.g., streams of streams) and recursion at higher type.

Next, we give a domain specific language for writing reactive programs. The key idea is to introduce a type constructor for delays, interpreted as an endofunctor that shrinks distances by a factor of one-half. This lets us track contractiveness by types, in much the same spirit as Nakano's calculus for guarded recursion [7].

In the second part of the paper, we give a reasonably efficient implementation of our language in terms of an imperative dataflow graph and prove the correctness of the implementation with respect to the semantics. The correctness proof uses a non-trivial Kripke logical relation, built using ideas from separation logic, rely-guarantee reasoning and step-indexed models, but ensures that clients can reason about programs as well-behaved mathematical objects, satisfying the full range of β, η and fixpoint equations, with all the complexities of the higher-order imperative implementation hidden behind an abstraction barrier.

II. Ultrametric Semantics

A. Reactive Programs and Stream Transformers

Reactive programs are usually interpreted as *stream transformers*. A time-varying value of type A can be viewed as a stream of As, and so a program that takes a time-varying A and produces a time-varying B is then a function that takes a stream of As and produces a stream of Bs.

However, the full function space on streams is too generous: many functions on streams do not have sensible interpretations as reactive processes. For example, a stock trading program receives a stream of prices and emits a stream of orders, but the type $\mathsf{Price}^\omega \to \mathsf{Order}^\omega$ includes functions that produce orders today that are a function of the price tomorrow; such functions are (much to our regret) unrealizable.

The semantic condition that expresses which functions do correspond to implementable processes is *causality*: the n^{th} output should depend only on the first n inputs. More formally, writing $\lfloor xs \rfloor_n$ for the n-element prefix of the stream xs:

Definition 1: (Causality) A stream function $f : S(A) \to S(B)$ is be *causal* when, for all n and streams as and as', if $\lfloor as \rfloor_n = \lfloor as' \rfloor_n$ then $\lfloor f(as) \rfloor_n = \lfloor f(as') \rfloor_n$.
This definition rules out, for example, the tail function, for which the first n outputs depend upon the first $n+1$ inputs.

Causality is an intuitive and appealing definition for streams of base types but it is not immediately clear how to generalize it. What might causality mean over a stream of *streams*, or even a stream of stream functions?

We also want to define streams by feedback or recursion, as in this definition of the increasing sequence of naturals:

$$\mathsf{nats} = \mathsf{fix}(\lambda \mathsf{xs}.\ 0 :: \mathsf{map\ succ\ xs})$$

Thinking operationally about when such fixed points are well-defined, we observe that the function $\lambda \mathsf{xs}.\ 0 :: \mathsf{map\ succ\ xs}$ can produce its first output without looking at its input. We imagine implementing the fixed point by feeding the output at time n back in as the input at time $n+1$, exploiting the fact that at time 0 the input value does not matter. This leads us to define:

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

Proposition 1: (Fixed Points of Guarded Functions) Every guarded endofunction $f : S(A) \to S(A)$ (where A is a nonempty set) has a unique fixed point.

As with causality, guardedness is intuitive and natural, but generalizations to higher types seem both useful and unobvious. For example, we may want to write a recursive *function*:

$$\mathsf{fib} = \mathsf{fix}(\lambda \mathsf{f}\ \lambda(\mathsf{j},\mathsf{k}).\ \mathsf{j} :: \mathsf{f}(\mathsf{k}, \mathsf{j}+\mathsf{k}))$$

What does guardedness mean, and how can we interpret fixed points, at higher types? We will answer these questions by moving to metric spaces.

B. An Ultrametric Model of Reactive Programs

A *complete 1-bounded ultrametric space* is a pair (A, d_A), where A is a set and $d_A \in A \times A \to [0,1]$ is a distance function, satisfying the following axioms:

- $d_A(x,y) = 0$ if and only if $x = y$
- $d_A(x,x') = d_A(x',x)$
- $d_A(x,x') \leq \max(d_A(x,y), d_A(y,x'))$
- Every Cauchy sequence in A has a limit

A sequence $\langle x_i \rangle$ is Cauchy if for any $\epsilon \in [0,1]$, there is an n such that for all $i > n$, $j > n$, $d(x_i, x_j) \leq \epsilon$. A limit is an x such that for all ϵ, there is an n such that for all $i > n$, $d(x, x_i) \leq \epsilon$. Ultrametric spaces satisfy a stronger version of the triangle inequality than ordinary metric spaces, which only ask that $d(x,x')$ be less than or equal to $d_A(x,y) + d_A(y,x')$, rather than $\max(d_A(x,y), d_A(y,x'))$. We often just write A for (A, d_A). All the metric spaces we consider are *bisected*, meaning that the distance between any two points is 2^{-n} for some $n \in \mathbb{N}$.

A map $f : A \to B$ between ultrametric spaces is *nonexpansive* when it is non-distance-increasing:

$$\forall x\, x', d_B(f\ x, f\ x') \leq d_A(x, x')$$

A map $f : A \to B$ between ultrametric spaces is *(strictly) contractive* when it shrinks the distance between any two points by a non-unit factor:

$$\exists q \in [0,1),\ \forall x\, x',\ d_B(f\ x, f\ x') \leq q \cdot d_A(x,x')$$

Complete 1-bounded ultrametric spaces and nonexpansive maps form a Cartesian closed category. The product is given by equipping the set product with the pointwise sup-metric:

$$d_{A \times B}((a,b),(a',b')) = \max\{d_A(a,a'), d_B(b,b')\}$$

Exponentials give the set of nonexpansive maps a sup-metric over all inputs:

$$d_{A \Rightarrow B}(f,f') = \sup\{d_B(f\ a, f'\ a) \mid a \in A\}$$

The discrete ultrametric space $D(X)$ on a set X is given by defining $d(x,x')$ to be 0 if $x = x'$ and 1 otherwise. The category also has coproducts, with $(A, d_A) + (B, d_B)$ being defined as $(A + B, d_{A+B})$, where

$$d_{A+B}(x,y) = \begin{cases} d_A(a,a') & \text{if } x = \mathsf{inl}\, a, y = \mathsf{inl}\, a' \\ d_B(b,b') & \text{if } x = \mathsf{inr}\, b, y = \mathsf{inr}\, b' \\ 1 & \text{otherwise} \end{cases}$$

The shrinking functor $\frac{1}{2}(A, d_A) = (A, d_{\frac{1}{2}A})$ halves all distances:

$$d_{\frac{1}{2}A}(a,a') = \frac{1}{2} d_A(a,a')$$

The $\frac{1}{2}$ functor is Cartesian closed: there are natural isomorphisms $\mathit{unzip}_{\frac{1}{2}}, \mathit{zip}_{\frac{1}{2}} : \frac{1}{2}(A \times B) \simeq \frac{1}{2}A \times \frac{1}{2}B$ and $\epsilon, \epsilon^{-1} : \frac{1}{2}(A \to B) \simeq \frac{1}{2}A \to \frac{1}{2}B$. There is also a natural transformation $\delta_A : A \to \frac{1}{2}A$ (pronounced "delay"), all of which are implemented with the obvious identity embedding on points. In general, however, $\frac{1}{2}(A + B) \not\simeq \frac{1}{2}A + \frac{1}{2}B$.

For an ultrametric space A, the ultrametric space of streams on A is defined by equipping the set $S(A)$ with the *causal metric of streams*:

$$d_{S(A)}(as, as') = \sup\left\{2^{-n} \cdot d_A(as_n, as'_n) \mid n \in \mathbb{N}\right\}$$

This is functorial: for $f : A \to B$, $f^\omega : S(A) \to S(B)$ just maps f over the input stream, which clearly preserves identity and composition.

In the case of streams of discrete elements, the stream metric says that two streams are closer, the later the time at which they first disagree. So two streams which have differing values at time 0 are at a distance of 1, whereas two streams which never disagree will have a distance of 0, and will thus be equal. (The stream type can also be understood as $\mu\alpha.\ A \times \frac{1}{2}\alpha$, but we do not develop the general theory of recursive types here.)

Proposition 2: (Banach's Contraction Map Theorem) If A is a nonempty, complete (ultra)metric space, any contractive $f : A \to A$ has a unique fixed point. Equivalently (as strict contractiveness is uniform), any nonexpansive map $g : \frac{1}{2}A \to A$ has a unique fixed point.

C. From Ultrametrics to Functional Reactive Programs

For streams of base type, the properties of maps in the category of ultrametric spaces correspond exactly to the properties of first-order reactive programs discussed previously.

Theorem 1: (Causality is Nonexpansiveness) For sets A and B, a function $f : S(A) \to S(B)$ is causal if and only if it is a nonexpansive function under the causal metric of streams of elements of the discrete spaces $D(A)$ and $D(B)$.

Theorem 2: (Guardedness is Contractiveness) For sets A and B, a function $f : S(A) \to S(B)$ is guarded if and only if it is a strictly contractive function under the causal metric of streams of elements of the discrete spaces $D(A)$ and $D(B)$.

The proof of these two theorems is nothing more than the unwinding of a few definitions. But the consequences of moving to ultrametric spaces are quite dramatic!:

1) Cartesian closure means we can interpret tuples and functions (with full β and η laws); we also have sums, which let clients implement the "switching" combinators of FRP.
2) Since streams are functorial, we can interpret streams of streams.
3) Contractiveness and Banach's theorem generalize the stream-centric notions of guardedness and guarded recursion to give a notion of well-founded recursion that also works at higher types. Further, the explicit delay functor lets us express contractiveness via types, rather than making it a property of functions.

In an abstract sense, this semantics fulfill the original promise of FRP in a 'no-compromise' way: one can freely and naturally write higher-order programs with stream values, and the properties of ultrametric spaces ensure that all functions are causal and all recursions well-founded.

III. A Language for Stream Programs

We now need a term calculus in which to write reactive programs. Our semantic category is not inherently tailored to reactivity, but the language (building in streams with the causal metric and making particular use of the delay modality) does reflect the synchronous operational semantics and implementation techniques we have in mind. Birkedal *et al.* [8] have recently given an ultrametric model of Nakano's [7] calculus for guarded recursion. Once we learned of this, we chose our syntax of types to be the same though our calculus is different, being more in the spirit of 'standard' natural deduction and Curry-Howard. Nakano's calculus includes both subtyping and rules (such as ($\bullet$)) whose application does not affect the subject term, but which we wish to record for operational reasons. By contrast with the impressively sophisticated metatheory of Nakano, we have a straightforward normalization proof and algorithmic presentation of the system. We discuss the relation with Nakano's system further in the appendix.

A. Syntax and Type Theory

In Figure 1, we give the syntax and typing rules for our calculus. The types are functions, streams, and delays, with sums and products omitted for space reasons. Each of the types appearing in a judgement is annotated with a 'time'. Intuitively, time 0 means 'now', whilst time k means 'k steps into the future'. Time indices are used in typing stream terms. The introduction form $\mathsf{cons}(e, e')$ takes a head of type A at time i, and a tail of type $S(A)$ at time $i+1$. The two elimination forms $\mathsf{hd}\ e$ and $\mathsf{tl}\ e$ take a stream at i, and return the head and tail of a stream, with the head coming at i and the tail one step later, at $i+1$.

The Hyp rule for variables includes subsumption: a hypothesis $x : A_i$ can be used to conclude $x : A_j$ for any time $j \geq i$. The intuition is that values can be maintained for use at later times. The 'later' modality $\bullet A$ partly internalizes this notion of time. The introduction rule $\bullet$I produces a term of type $\bullet A$ at time i, given a premise of type A at time $i+1$, and dually the elimination rule $\bullet$E yields a term of type A at time $i+1$ from a term of type $\bullet A$ at time i.

We remark that the calculus only deals with relative times, and there is no way to define a type valid only at a single moment in time (as might be possible in a hybrid logic [9]). For example, the type of values A, k steps in the future, would be $\bullet^k(A)$, but there is no type corresponding to being valid exactly at time $k = 17$. This property can be formalized in the following theorem, where the notation Γ_{+n} means that we add n to the time index of every hypothesis in Γ.

Lemma 1: (Time Adjustment) If $\Gamma, \Gamma' \vdash e : A_i$, then $\Gamma, \Gamma'_{+n} \vdash e \Leftarrow A_{i+n}$. As a partial converse, if $\Gamma_{+n} \vdash e : A_{i+n}$, then $\Gamma \vdash e : A_i$.

Theorem 3: (Normalization) If $\Gamma \vdash e : A_i$, then there exists a normal form n such that $e \overset{\beta,\eta}{=} n$ and $\Gamma \vdash n : A_i$.
The normalization proof is sketched in the appendix. We give a bidirectional (algorithmic) type system in canonical forms style, which types only normal forms of this calculus. We then define a hereditary substitution [10] which preserves typing and is compatible with our equational theory.

B. Denotational Semantics

In Figure 2, we give a denotational semantics to our DSL, interpreting types as ultrametric spaces and interpreting terms as non-expansive maps.

The $\bullet A$ type is interpreted as $\frac{1}{2}[\![A]\!]$, and a type indexed with a time index A_i is interpreted as $\frac{1}{2}^i[\![A]\!]$. As a result, $[\![A_{i+1}]\!] = [\![\bullet A_i]\!]$, and the interpretation of the rules for $\bullet A$ are identities. The real action of the $\frac{1}{2}(A)$ functor occurs in the interpretation of the hypothesis rule Hyp, in which values

$$\begin{array}{lcll} A & ::= & P \mid A \to B \mid \bullet A \mid S(A) & \text{Types} \\ \Gamma & ::= & \cdot \mid \Gamma, x : A_i & \text{Contexts} \\ e & ::= & \lambda x : A.\ e \mid \bullet e \mid \mathsf{cons}(e,e) & \text{Terms} \\ & \mid & x \mid e\ e \mid \mathsf{await}(e) \mid \mathsf{hd}\ e \mid \mathsf{tl}\ e & \end{array}$$

$$\boxed{\Gamma \vdash e : A_i}$$

$$\frac{\Gamma, x : A_i \vdash e : B_i}{\Gamma \vdash \lambda x.\ e : A \to B_i} \to\mathrm{I} \qquad \frac{\Gamma \vdash e : A_{i+1}}{\Gamma \vdash \bullet e : \bullet A_i} \bullet\mathrm{I}$$

$$\frac{\Gamma \vdash e : A_i \qquad \Gamma \vdash e' : S(A)_{i+1}}{\Gamma \vdash \mathsf{cons}(e, e') : S(A)_i} S\mathrm{I} \qquad \frac{x : A_i \in \Gamma \qquad i \leq j}{\Gamma \vdash x : A_j} \textsc{Hyp}$$

$$\frac{\Gamma \vdash t : A \to B_i \qquad \Gamma \vdash e : A_i}{\Gamma \vdash t\ e : B_i} \to\mathrm{E} \qquad \frac{\Gamma \vdash t : \bullet A_i}{\Gamma \vdash \mathsf{await}(t) : A_{i+1}} \bullet\mathrm{E}$$

$$\frac{\Gamma \vdash t : S(A)_i}{\Gamma \vdash \mathsf{hd}\ t : A_i} S\mathsf{hd} \qquad \frac{\Gamma \vdash t : S(A)_i}{\Gamma \vdash \mathsf{tl}\ t : S(A)_{i+1}} S\mathsf{tl}$$

$$\begin{array}{l|ll} A \to B & (\lambda x : A.\ e)\ e' \overset{\beta}{=} [e'/x]e & e \overset{\eta}{=} \lambda x : A.\ e\ x \\ \bullet A & \mathsf{await}(\bullet e) \overset{\beta}{=} e & e \overset{\eta}{=} \bullet(\mathsf{await}(e)) \\ S(A) & \mathsf{hd}\ (\mathsf{cons}(e, e')) \overset{\beta}{=} e & e \overset{\eta}{=} \mathsf{cons}(\mathsf{hd}\ e, \mathsf{tl}\ e) \\ & \mathsf{tl}\ (\mathsf{cons}(e, e')) \overset{\beta}{=} e' & \end{array}$$

Fig. 1. Syntax and Typing

are moved from $\frac{1}{2}^i [\![A]\!]$ to $\frac{1}{2}^j [\![A]\!]$ by iterating the delay natural transformation $j - i$ times.

The interpretations of other types, such as functions, look entirely standard, with occasional appearances of the isomorphisms mediating between $\frac{1}{2}^n A \to \frac{1}{2}^n B$ and $\frac{1}{2}^n (A \to B)$.

Theorem 4: (Soundness)

- If $\Gamma \vdash e : A_i$ and $\Gamma, x : A_i \vdash e' : B_n$ then $[\![\Gamma \vdash [e/x]e' : B_n]\!] = [\![\Gamma, x : A_i \vdash e' : B_n]\!] \circ \langle id_\Gamma, [\![\Gamma \vdash e : A_i]\!]\rangle$.
- If $\Gamma \vdash e : A_i$ and $\Gamma \vdash e : A_i$ and $e =^{\beta\eta} e'$, then $[\![\Gamma \vdash e : A_i]\!] = [\![\Gamma \vdash e' : A_i]\!]$.

C. Fixed Points

The normalization result establishes a sense in which the core of our language has a well-behaved proof theory and is proved for the language without recursion. Having done this, we add well-founded recursion to the language simply by adding constants fix : $(\bullet A \to A) \to A$ for each (non-empty) type A, interpreted using Banach's theorem. It would be interesting to investigate the proof theory of a language including fixed points, but even defining what the normal forms of proofs involving infinite types (e.g. streams) should be involves many subtleties [11], [12], so we defer this to future work.

D. Examples

Some reactive languages make everything a stream (or stream transformer), implicitly lifting other constructs so that, for example, a syntactic application is semantically a map operation. Primitives are carefully chosen to preserve causality (so head and 'followed by' are OK, but tail is not). We program more explicitly with streams and can implement higher-order operations such as mapping straightforwardly:

```
map = λf:ℕ→ℕ.fix (λ g:•(S(ℕ)→S(ℕ)).
                      λ xs:S(ℕ).
                        let g' = await(g) in
                        let t' = tl (xs) in
                         cons(f (hd xs), g' t'))
```

Note that the fact that the function being recursively defined is only needed at the next time step, justifying that the definition is well-founded, is made explicit, but that the definition is otherwise close to the familiar one. Having defined *map* (and *zip*), we can neatly program the stream of Fibonacci numbers as follows:

```
fibs = fix (λ xs': •(S(ℕ)).
            let xs = await(xs') in
            let ys = tl (xs) in
            cons(1, cons(1, map (+) (zip (xs, ys )))))
```

There is some further subtlety here: in the subexpression zip (xs, ys), we zip together xs and ys, which are streams at *different* times. So xs needs to be "pushed into the future" in order to be used at the same time as ys. If we think of streams as sequences of events emitted over time, this means that we need to buffer xs in order to be able to use it with ys.

On the other hand, the typing rules block an ill-founded definition such as:

$$\mathsf{BAD} = \mathsf{fix}(\lambda xs : \bullet S(A).\ \mathsf{await}(xs))$$

Here, the $\mathsf{await}(xs)$ term gives a term one step in the future of the expected time, which means that the program fails to type check.

Stream processors are often conveniently specified as finite state machines. One can easily program the translation from one representation to another using higher-order functions:

```
unfold = λ f:T→A ×•(T). fix (
  λ loop:•(T→S(A)). λ t:T.
     let (a,t') = f t
     in cons(a, (await(loop) await (t '))))
```

Here *unfold* : $(T \to A \times \bullet T) \to T \to S(A)$ takes a transition function mapping a state (of type T) to an output (of type A) and and a next state, together with an initial state, and produces the resulting stream of outputs.

Making well-foundedness a semantic property that can be verified using typechecking contrasts with the work on synchronous dataflow languages [4], in which guardedness is established via syntactic checks, and gives us a stronger equational theory. These syntactic checks can use dataflow analysis to allow some definitions we do not, but on the other hand our approach scales more naturally to higher-order programs.

IV. IMPLEMENTATION

A. Idealized ML and Program Logic

Implementation Language. The programming language in which we implement our domain-specific language is a poly-

$$
\begin{array}{lcl}
\llbracket A \to B \rrbracket & = & \llbracket A \rrbracket \Rightarrow \llbracket B \rrbracket \\
\llbracket \bullet A \rrbracket & = & \frac{1}{2} \llbracket A \rrbracket \\
\llbracket S(A) \rrbracket & = & S(\llbracket A \rrbracket) \\
\llbracket A_i \rrbracket & = & \frac{1}{2}^i \llbracket A \rrbracket
\end{array}
\qquad
\begin{array}{lcl}
\llbracket \cdot \rrbracket & = & 1 \\
\llbracket \Gamma, x : A_i \rrbracket & = & \llbracket \Gamma \rrbracket \times \llbracket A_i \rrbracket \\
\delta_A^n & \in & A \to \frac{1}{2}^n A \\
\delta_A^0 & = & id_A \\
\delta_A^{n+1} & = & \frac{1}{2}(\delta_A^n) \circ \delta_A
\end{array}
$$

$$
\begin{array}{lcl}
zip^i_{\frac{1}{2}} & \in & \frac{1}{2}^i A \times \frac{1}{2}^i B \to \frac{1}{2}^i (A \times B) \\
zip^{n+1}_{\frac{1}{2}} & = & \frac{1}{2}(zip^n_{\frac{1}{2}}) \circ zip_{\frac{1}{2}} \\
zip^0_{\frac{1}{2}} & = & id_{A \times B} \\
\epsilon^i & \in & (\frac{1}{2}^i A \Rightarrow \frac{1}{2}^i B) \to \frac{1}{2}^i (A \Rightarrow B) \\
\epsilon^0 & = & id_{A \Rightarrow B} \\
\epsilon^{n+1} & = & \frac{1}{2}(\epsilon^n) \circ \epsilon \\
\epsilon_i^{-1} & \in & \frac{1}{2}^i (A \Rightarrow B) \to (\frac{1}{2}^i A \Rightarrow \frac{1}{2}^i B) \\
\epsilon_0^{-1} & = & id \\
\epsilon_{n+1}^{-1} & = & \epsilon_n^{-1} \circ \epsilon^{-1}
\end{array}
$$

$$
\boxed{\llbracket \Gamma \vdash e : A_i \rrbracket \in \llbracket \Gamma \rrbracket \to \frac{1}{2}^i \llbracket A \rrbracket}
$$

$$
\begin{array}{lcl}
\llbracket \Gamma \vdash \bullet e : \bullet A_i \rrbracket & = & \llbracket \Gamma \vdash e : A_{i+1} \rrbracket \\
\llbracket \Gamma \vdash \mathsf{cons}(e, e') : S(A)_i \rrbracket & = & \text{let } h = \llbracket \Gamma \vdash e : A_i \rrbracket \text{ in} \\
 & & \text{let } t = \llbracket \Gamma \vdash e' : S(A)_{i+1} \rrbracket \text{ in} \\
 & & \frac{1}{2}^i (cons) \circ zip^i_{\frac{1}{2}} \circ \langle h, t \rangle \\
\llbracket \Gamma \vdash \lambda x.\, e : A \to B_i \rrbracket & = & \epsilon^i \circ \lambda(\llbracket \Gamma, x : A_i \vdash e : B_i \rrbracket) \\
\llbracket \Gamma \vdash x_n : A_j \rrbracket & = & \delta_A^{j-i} \circ \pi_n, \text{if } x_n : A_i \in \Gamma \\
\llbracket \Gamma \vdash \mathsf{await}(e) : A_{i+1} \rrbracket & = & \llbracket \Gamma \vdash e : \bullet A_i \rrbracket \\
\llbracket \Gamma \vdash \mathsf{hd}\ e : A_i \rrbracket & = & \frac{1}{2}^i (hd) \circ \llbracket \Gamma \vdash t : S(A)_i \rrbracket \\
\llbracket \Gamma \vdash \mathsf{tl}\ e : S(A)_{i+1} \rrbracket & = & \frac{1}{2}^i (tl) \circ \llbracket \Gamma \vdash t : S(A)_i \rrbracket \\
\llbracket \Gamma \vdash e\ e' : B_i \rrbracket & = & \text{let } f = \llbracket \Gamma \vdash e : A \to B_i \rrbracket \text{ in} \\
 & & \text{let } v = \llbracket \Gamma \vdash e' : A_i \rrbracket \text{ in} \\
 & & \frac{1}{2}^i (eval) \circ zip^i_{\frac{1}{2}} \circ \langle f, v \rangle
\end{array}
$$

Fig. 2. Semantics

morphic lambda calculus with monadically typed side-effects. The types are the unit type 1, the function space $\tau \to \sigma$, sums $\tau + \sigma$, products $\tau \star \sigma$, inductive types like the natural number type $\mathbb{N}$, the general reference type ref τ, as well as (higher-kinded but still predicative) universal and existential types $\forall \alpha : \kappa.\ \tau$ and $\exists \alpha : \kappa.\ \tau$. In addition, we have the monadic type $\bigcirc \tau$ for side-effecting computations producing values of type τ. The side effects we consider are heap effects (such as reading, writing, or allocating references) and nontermination. The implementation language is standard, and we omit the details for reasons of space.

Program Logic. We reason about programs in the implementation language in the program logic whose syntax is shown in Figure 3. The Hoare triple $\{p\}\, c\, \{a : \tau.\ q\}$ is used to specify computations, and is satisfied when running the computation c in any heap satisfying the predicate p either diverges or yields a heap satisfying q; note that the value returned by terminating executions of c is bound (by $a : \tau$) in the postcondition. These atomic specifications can then be combined with the usual logical connectives of intuitionistic logic including quantifiers ranging over the sorts in ω. This

$$
\begin{array}{llcl}
\text{Assertion Sorts} & \omega & ::= & \tau \mid \kappa \mid \omega \Rightarrow \omega \mid \mathsf{prop} \\
\text{Assertion} & p & ::= & e \mid \tau \mid x \mid \lambda x : \omega.\ p \mid p\ q \\
\text{Constructors} & & \mid & \top \mid p \wedge q \mid p \Rightarrow q \mid \bot \mid p \vee q \\
 & & \mid & \mathsf{emp} \mid p * q \mid e \mapsto e' \\
 & & \mid & \forall x : \omega.\ p \mid \exists x : \omega.\ p \mid S \\
\text{Specifications} & S & ::= & \{p\}\, c\, \{a : \tau.\ q\} \mid \{p\} \\
 & & \mid & S \text{ and } S' \mid S \Longrightarrow S' \mid S \text{ or } S' \\
 & & \mid & \forall x : \omega.\ S \mid \exists x : \omega.\ S
\end{array}
$$

Fig. 3. Specification Language

permits us to give abstract specifications to modules using existential quantifiers to hide program implementations and predicates.

The assertions in the pre- and post-conditions are drawn from higher-order separation logic [13], including spatial connectives like the separating conjunction $p * q$. The universal and existential quantifiers $\forall x : \omega.\ p$ and $\exists x : \omega.\ p$ are higher-order quantifiers ranging over all sorts ω. The sorts include the language types τ, kinds κ, the sort of propositions prop, and function spaces over sorts $\omega \Rightarrow \omega'$. For the function space, we include lambda-abstraction and application. Because our assertion language contains within it the classical higher-order logic of sets, we will freely make use of features like subsets, indexed sums, and indexed products, exploiting their definability.

Details can be found in the first author's dissertation [14].

B. The Implementation and its Correctness Proof

The basic idea underlying our implementation is the idea of representing a collection of streams with a dataflow graph. Instead of representing streams as (possibly-lazy) sequences of elements, we represent our streams with mutable data structures, which enumerate the values of the stream as they evolve over time. We implement reactive programs with a dataflow graph, which runs inside an event loop. The event loop updates a clock cell to notify the cells in the graph that they may need to recompute themselves, and then it reads the cells it is interested in, doing (hopefully) the minimal amount of computation needed at each time step. We describe our program invariant precisely, illustrating it with examples drawn from the implementation, with the full implementation in the appendix.

Dataflow Graphs. An imperative dataflow network is rather like a generalized spreadsheet. It has a collection of cells, each containing some code whose evaluation may read other cells. When a cell is read, the expression within the cell is evaluated, recursively triggering the evaluation of other cells as they are read by the program expression. Furthermore, each cell memoizes its expression, so that repeated reads of the same cell will not trigger re-evaluation.

We give the interface to a dataflow library in Figure 4. We have implemented and given a correctness proof of this library in prior work [15], [14], but briefly describe the specification here, since we use it as a component of the present work.

The interface features two abstract data types, cell and code. The type cell α are cells that compute a value of type α, and are the nodes of dataflow graph. The expressions within each cell are of type code α, which is a user-defined monadic type, as is commonly defined in Haskell. It has the responsibility of both computing a value of type α, and returning the set of cells it read while computing its return value.

Since the dataflow graph maintains many internal invariants, we give a "domain-specific separation logic" as an abstract interface to the graph's state. I and $\phi \otimes \psi$ correspond to the emp and separating conjunction of separation logic, denoting empty graphs and two disjoint collections of cells. In addition to the $\mathsf{ref}(\mathsf{r}, \mathsf{v})$ predicate (which corresponds to points-to in separation logic), we include a pair of predicates describing cells. The predicate $\mathsf{cell}^-(\mathsf{c}, e)$ means that c is a cell in the dataflow graph containing code e, and that it is not ready — i.e., it needs to be evaluated before producing a value. The predicate $\mathsf{cell}^+(\mathsf{c}, e, \mathsf{v}, rs)$ almost means the opposite: it means that c is ready (i.e., has a memoized value), *conditional* on all its dependencies in rs being ready themselves, which we describe with two relations $\mathsf{unready}(\theta, \mathsf{c})$, and $\mathsf{ready}(\theta, \mathsf{c}, \mathsf{v})$. These establish respectively that the cell c is unready — either it or one of its ancestors are a negative cell — or that c and all of its ancestors are positive cells and it presently contains v.

We now explain the specifications of the code expressions in Figure 4. First, all of these specifications are parameterized by an extra quantifier $\forall\psi.\ldots$, letting us manually build in a kind of frame rule into this specification — any formula we derive will also be quantified, and hence works in larger dataflow graphs. However, one oddity of these rules is that the framed formula ψ is asymmetric; in the postcondition, we frame on a formula like $\Re(u, \psi)$. This is a "ramification operator", whose purpose is to look at the dependencies of cells in ψ and ensure that they are not falsely marked as ready due to other updates.

On lines 1-5 of Figure 4, we see the specifications for the return and bind operation of this monad. One lines 6-8, we give specifications for creating, reading and writing local state, as well as (on line 9) creating a new cell with code in it. On lines 10-11, we give the specification for reading from a ready cell, which merely returns the value in the cell without modifying the heap. On lines 12-17, we specify the behavior of reading an unready cell — if we know that executing its body takes us from θ to θ', then reading the cell will do the same, as well as setting the cell to a positive state.

The Logical Relation. However, we need not only streams, but *programs* to manipulate them. Therefore, our implementation is *imperative* functions which manipulate the dataflow graph, but our denotational model of causal stream functions has no notion of dataflow graphs or mutable state within it. So we need to relate our semantic view of stream programs with our implementation in terms of mutable dataflow graphs. To connect them, we will use a *step-indexed Kripke logical relation* [16].

Our relation, a family relations $V_A^k(v, \mathsf{v}, M)$, is essentially a realizability model, in which a semantic term v in the space

```
1  $\forall\psi.\ \{H(\psi)\}\ \mathsf{return}(\mathsf{v})\ \{(\mathsf{a}, \emptyset).\ H(\psi) \wedge \mathsf{a} = \mathsf{v}\}$
2  $\forall\psi.\ \{H(\theta \otimes \psi)\}\ \mathsf{e}\ \{(\mathsf{a}, r).\ H(\theta' \otimes \Re(u, \psi)) \wedge (\mathsf{a}, r) = (v, r_1)\}$ and
3  $\forall\psi.\ \{H(\theta' \otimes \psi)\}\ \mathsf{f}\ \mathsf{v}\ \{(\mathsf{a}, r).\ H(\theta'' \otimes \Re(u', \psi)) \wedge (\mathsf{a}, r) = (\mathsf{v}', r_2)\}$
4  $\Longrightarrow \forall\psi.\ \{H(\theta \otimes \psi)\}\ \mathsf{bind}\ \mathsf{e}\ \mathsf{f}\{(\mathsf{a}, r).\ H(\theta'' \otimes \Re(u \cup u', \psi))$
5                                                $\wedge\ (\mathsf{a}, r) = (\mathsf{v}', r_1 \cup r_2)\}$
6  $\forall\psi.\{H(\psi)\}\ \mathsf{newref}(\mathsf{v})\ \{(\mathsf{a}, \emptyset).\ H(\psi \otimes \mathsf{ref}(\mathsf{a}, \mathsf{v}))\}$
7  $\forall\psi.\{H(\mathsf{ref}(\mathsf{r}, \mathsf{v}) \otimes \psi)\}\ \mathsf{get}(\mathsf{r})\ \{(\mathsf{a}, \emptyset).\ H(\mathsf{ref}(\mathsf{r}, \mathsf{v}) \otimes \psi) \wedge \mathsf{a} = \mathsf{v}\}$
8  $\forall\psi.\{H(\mathsf{ref}(\mathsf{r}, -) \otimes \psi)\}\ \mathsf{set}(\mathsf{r}, \mathsf{v})\ \{(\mathsf{a}, \emptyset).\ H(\mathsf{ref}(\mathsf{r}, \mathsf{v}) \otimes \psi)\}$
9  $\forall\psi.\{H(\psi)\}\ \mathsf{cell}(\mathsf{code})\ \big\{(\mathsf{a}, \emptyset).\ H(\mathsf{cell}^-(\mathsf{a}, \mathsf{code}) \otimes \psi)\big\}$
10 $\mathsf{ready}(\theta, \mathsf{c}, \mathsf{v})$
11 $\Longrightarrow \forall\psi.\{H(\theta \otimes \psi)\}\ \mathsf{read}(\mathsf{c})\{(\mathsf{a}, r).\ H(\theta \otimes \psi) \wedge (\mathsf{a}, r) = (\mathsf{v}, \{\mathsf{c}\})\}$
12 $\mathsf{unready}(\theta \otimes \mathsf{cell}^{\pm}(c, e), \mathsf{code})$ and
13 $\forall\psi.\{H(\theta \otimes \psi)\}\ \mathsf{e}\ \{(\mathsf{a}, r).\ H(\theta' \otimes \Re(u, \psi)) \wedge (\mathsf{a}, r) = (\mathsf{v}, rs)\}$
14 $\Longrightarrow$
15 $\forall\psi.\{H(\theta \otimes \psi)\}$
16     $\mathsf{read}(\mathsf{c})$
17     $\{(\mathsf{a}, \{\mathsf{c}\}).\ H(\Re(\{c\}, \theta' \otimes \Re(\{c\}, \psi)) \otimes \mathsf{cell}^+(\mathsf{c}, \mathsf{e}, \mathsf{v}, rs)) \wedge \mathsf{a} = \mathsf{v}\}$
```

Fig. 4. Dataflow Library Specification

$[\![A]\!]$ is realized by an ML term v of type $(\!|A|\!)$. The intuitive reading of this relation is that v is an ML program which, given a memory state M, implements the denotational value v of type A for at least k trips through the event loop.

The Kripke structure arises from the fact that the program state evolves over time. This actually induces two preorders on memories. First, states change *within* the progress of a single timestep, for which we use the preorder $M' \sqsubseteq M$. Second, the global clock can advance, and so the ordering $M' \ll^n M$ asserts that M' is a "possible future" of M. That is, it is a state which may be reached n time steps in the future.

The interpretation of DSL types as ML types is given in Figure 5, and the logical relation is given in Figure 6.

Terminals and products are implemented with ML units and pairs, and realized in the obvious fashion. Streams of type A are implemented with type cell $(\!|A|\!)$, the type of imperative nodes in our dataflow graph library. A stream vs is realized by a dataflow cell v, when it is in the abstract memory M, and the memory asserts that this cell yields the elements vs.

The function space $A \to B$ is sent to $(\!|A|\!) \to \mathsf{code}\ (\!|B|\!)$, realizing functions with ML procedures that take values and perform computations in our user-defined dataflow monad. Our implementation must take a value in any state, now or in the future, and return a computation of the result. No matter *when* we call a function, it must return the same value.

The next modality $\bullet A$ is realized by code $(\!|A|\!)$. Unlike our denotational semantics, in which A and $\frac{1}{2}A$ share the same underlying set, a realizer c for a next value v is a computation yielding v, when run from any memory one step in the future.

Finally, the relation $T_A^k(v, \mathsf{c}, M)$ asserts that if we run c in any concrete heap realizing the memory M, it computes a value realizing v, taking us to a future memory state in the same time step. We also reduce the step count to ensure the well-foundedness of the relation.

Abstract Memory. An abstract memory $M \in \mathit{Mem}$ is an

$$
\begin{array}{lcl}
(\!|A \to B|\!) & = & (\!|A|\!) \to \mathsf{code}\ (\!|B|\!) \\
(\!|1|\!) & = & \mathsf{unit} \\
(\!|A \times B|\!) & = & (\!|A|\!) \star (\!|B|\!) \\
(\!|S(A)|\!) & = & \mathsf{cell}\ (\!|A|\!) \\
(\!|\mathbb{N}|\!) & = & \mathsf{int} \\
(\!|\bullet A|\!) & = & \mathsf{code}\ (\!|A|\!)
\end{array}
$$

Fig. 5. Implementation of Types

$$
\begin{array}{ll}
1 & V_1^k((), (), M) \quad = \top \\
2 & V_{A\times B}^k((a,b), (\mathsf{a},\mathsf{b}), M) = V_A^k(a,\mathsf{a},M) \wedge V_B^k(b,\mathsf{b},M) \\
3 & V_{S(A)}^k(vs, \mathsf{v}, M) \quad = \mathsf{v} \in S_M \wedge \pi_1(\sigma_M^S\ \mathsf{v}) = vs \\
4 & V_{\bullet A}^k(v, \mathsf{c}, M) \quad = \forall j \le k, M' \ll^1 M.\ T_A^{j-1}(v, \mathsf{c}, M') \\
5 & V_{A\to B}^k(f, \mathsf{f}, M) = \\
6 & \quad \forall j \le k, n \le j, M' \ll^n M, v, \mathsf{v}. \\
7 & \qquad V_A^{j-n}(v, \mathsf{v}, M') \Rightarrow T_B^{j-n}(f\ v, \mathsf{f}\ \mathsf{v}', M'') \\
8 & T_A^k(v, \mathsf{c}, M) = \\
9 & \quad \forall j < k, \psi. \\
10 & \qquad \{\mathit{Heap}_j(M, \psi)\} \\
11 & \qquad \mathsf{c} \\
12 & \qquad \{(\mathsf{a}, \mathsf{U}).\ \exists M' \sqsubseteq M.\ \mathit{Heap}_j(M', \Re(\mathsf{U}, \psi)) \wedge M' \overset{!}{\equiv} M \wedge \\
13 & \qquad\qquad V_A^k(v, \mathsf{a}, M') \wedge \mathsf{U} \subseteq S_{M'}\} \\
14 & \mathrm{Hom}(A,B) = \big\{(f, \mathsf{f}) \mid \forall k.\ V_{A\to B}^k(f, \mathsf{f}, M_\perp)\big\}
\end{array}
$$

Fig. 6. The Logical Relation

13-tuple(!) describing the state of a dataflow graph. Despite its size, each part arises for natural operational reasons. We give the definition of the set of abstract memories Mem in Figure 7, in lines 1-19.

The first four components S, I, D, and B categorize the four uses of state in our implementation. The set S is a collection of dataflow cells, each of which represents a stream value. It has an associated function σ sending each cell $c \in S$ to a pair of the stream c realizes, and the current implementation state of the cell.

The second component, D, is a set of reference cells used to forward values across time steps. Since our semantics is pure, but represents values with mutable data structures, we need to fix up cross-temporal values whenever the clock advances. Each $r \in D$ stores a computation, which is either a delayed thunk scheduled to run on the next time step, or a thunk received from the previous time step, ready to run now to yield a value. The function ρ^D sends each reference to a pair of the stream of values it yields, and the current computation value it holds.

The third component, I, are the reference cells used to implement cons. The function's type $A \times \bullet S(A) \to S(A)$ tells us it receives a value and a thunk, and constructs a cell yielding the cons'd stream. Our implementation (below) first stores a reference to the head value. After the cons cell returns the head, it saves the thunk to compute the tail. On the next step after that, it uses the thunk to build the tail stream, which it stores in the reference to generate all subsequent values.

```
cons : Hom(A × •S(A), S(A))
cons (x, dxs) =
  do r ← ref ( Init x);
     ys ← cell(do () ← read(clock);
                  x' ← get rx;
                  case x' of
                    Init x   → do set r Make(dxs); return x
                   Make d → do xs ← d; set r Done(xs); hd(xs)
                   Done xs → hd(xs));
    register (ys)
```

For each reference r in I, the function ρ^I says which semantic stream r represents, and which of the three possible states the reference is in. ($register$ marks cells using state.)

The fourth component B are the references used to implement higher-type recursion via backpatching. Its use is illustrated below:

```
fix_{B→C}: Hom(A ×•(B →C), B →C) →Hom(A, B →C)
fix f = λ a. do r ← ref None;
                d ← return (do f' ←get r; return (valOf f'));
                g ← f(a,d);
                set r Some g;
                return g
```

Here, we store a dummy value in r, and pass a thunk dereferencing r to f. We "tie Landin's knot" by assigning r to the return value of the call to f. So r points to the fixed point on every successive timestep after the first, and ρ^B sends each reference to a pair of the intended function, and an option saying what, if anything, the ref cell has been set to. (As an aside, note that the correctness of this routine also depends on the fact that functional values are safe to call in future states.)

Our cells maintain a dynamic dependency graph to memoize their computations, and so we also specify how dependencies evolve over time. The $Deps$ component is an irreflexive partial order overapproximating the true dependency graph (called R) — one cell may read another only if it is below the other in this ordering, ensuring that all dependencies are acyclic.

These dataflow cells use auxiliary state. To specify how that state is used, we use the components $reader : L \to S$ and $writer : L \rightharpoonup S$ (here $L = I \cup D$) to specify the ownership of the references in L. $reader$ tells us the unique reading cell in S for each reference, and $writer$ identifies the unique writing cell in S. To ensure that writers do not trample on readers, we also require the reading cell of a location to lie in the true dependencies of the location's writing cell. Finally, $writer$ is a partial function, letting us defer defining a location's writer. E.g., consider the definition of fixed points for streams:

```
fix  : Hom(A ×•S(B), S(B))  →  Hom(A, S(B))
fix f = λ a. do r ← ref None;
                preinput ← cell (get r);
                // preinput is r's reader, but r has no writer
                input ← return (do vs' ← preinput;
                                   cell(do v' ← head(vs');
                                           valOf v'));
                // call f, using preinput
                preoutput ← f(a, input);
                // out is r's writer, but can only be defined
                // after the call to f
                out ← cell(do () ← read(clock);
                              _ ← hd(preinput);
                              v ← hd(preoutput);
```

```
                    d ← delay_B(v);
                    set r d;
                    return v)
        register (out)
```

We call a function f in a memory state where r has no defined writer, in order to create the cell whose outputs become inputs.

The Heap Relation. In our prose, we have described the intended behavior of the cells, but the definition in lines 1-19 of Figure 7 makes no reference to the logical relation, and *Mem* does not relate any semantic values to implementation values. The heap relation $\mathit{Heap}_k(M, \psi)$, defined on lines 26-32, does this. The first two lines (27-28) translate the mathematical assertions in M into the assertions of separation logic we used to specify the dataflow library. The extra argument ψ just names the cells not in M, and we also explicitly require the existence of a cell for the *clock*, and a registry i of all the cells that need to write state. For each cell in S, D, and I, we assert that there is a cell or reference containing the implementation value promised in σ, ρ^D, and ρ^I, using the $\mathrm{cells}()$ and $\mathrm{refs}()$ auxiliary functions (defined on lines 45-48).

On lines 29-32, we relate the implementation of the graph with its specification. On line 29, each stream cell $c \in S$ is required to satisfy the *Stream* predicate (lines 33-34), which asserts that reading it for the next n time steps will yield the first n elements of the stream. On line 30, we assert that every reference $r \in D$ is a computation either scheduled to be run tomorrow (if its writer has already run and updated it), or good to run today (in case its writer is not ready, and has not yet used its contents). On line 31 we assert each element of I is in an appropriate state depending on how the cons cell has been used, using the *Consref* predicate defined on lines 39-46. On line 32, we assert that each element of the backpatch set B realizes its function, if its contents are not None.

Memory Orderings. As mentioned earlier, there are two ways in which the abstract memory may evolve. The order $M' \sqsubseteq M$ asserts describes how M can become M' within a single time step.

Under $M' \sqsubseteq M$, the respective S, D, I, B sets may grow, reflecting new allocation, as can the allowed dependency graph *Deps* and actual dependency graph R. Both *reader* and *writer* partial can be extended, to reflect the readers and writers of newly-allocated cells (or in the case or writers, to reflect a cell taking on the responsibility of updating a reference). Each of $\sigma, \rho^D, \rho^I, \rho^B$ can also be extended, but has its own preorder, to reflect their differing state update protocols.

On lines 6 and 7, we see that $\sigma_{M'}$ extends σ_M just in case that all ready cells were preserved in M were preserved. This allows the future state to evaluate unready cells. On lines 8-10, we describe how the ρ^I can change. If a reference's writer has not run, then the state must be unchanged — otherwise it may change. On lines 11-14, we see how ρ^D can evolve. As before, we states cannot change before a write. However, in this case, if there is a write, the expected semantic value must become the tail of the stream, since a new value has been written for use in the next time step. Finally, the ρ^B backpatch set has the simple invariant that anything that has been set remains unchanged, on lines 15-16.

The $M'' \ll^n M$ preorder says M'' is n steps in the future of M. This is defined inductively on lines 17-18, making use of two auxiliary relations. If M'' is zero steps in the future, then this is just the same as saying $M'' \sqsubseteq M$. If it is $n+1$ steps in the future, then there is some intermediate state M' which is within the timestep ordering of M and *complete*. That is, all references in $I_{M'}$ and $D_{M'}$ have writers, all of which have performed their writes, and every reference in $B_{M'}$ has been set. Then, M'' must be n steps in the future of the next state $\mathit{Next}(M')$ of M'. The function $\mathit{Next}(M')$, defined on lines 24-34, describes what happens when the clock ticks. All cells which depend on the clock become unready, and the values of the streams in S (and the references in I) become their tails.

The empty memory $M_\perp$ is minimal for both preorders.

Adequacy. If we could implement and prove correct an implementation of each combinator used in the denotational semantics in Figure 2, then we could prove correctness by replacing square brackets $[\![\Gamma \vdash e : A_i]\!]$ with banana brackets $(\!|\Gamma \vdash e : A_i|\!)$ to swap semantic combinators with the implementation combinators.

Unfortunately there is a slight wrinkle in this story. One direction of the isomorphism which distributes the later modality through functions, $\epsilon : (\frac{1}{2}A \Rightarrow \frac{1}{2}B) \to \frac{1}{2}(A \Rightarrow B)$, does not seem to be implementable with our current implementation strategy! Operationally, a value $f : \bullet A \to \bullet B$ is represented by a function that takes an A-computation scheduled for tomorrow, does some work, and then returns a B-computation scheduled for tomorrow. But $\bullet(A \to B)$ cannot do any work today — it contains computations scheduled for tomorrow. Since we use ϵ to interpret the abstraction rule at times greater than 0, a simple translation cannot work.

So we will show that we can translate any term to an isomorphic one using the lambda-abstraction rule only at time 0. The idea is to use the isomorphism $(\frac{1}{2}A \Rightarrow \frac{1}{2}B) \simeq \frac{1}{2}(A \Rightarrow B)$ to push all uses of the next modality $\bullet A$ through the function space, ensuring that all lambdas and variable hypotheses occur at time 0, letting us interpret programs without using ϵ.

In Figure 9, we show how we translate the types into a form in which there are no types of the form $\bullet(A \to B)$. Each type A at time i is translated to $\langle A \rangle^i$, which is isomorphic. Also, when HYP does subsumption, we should not use the delay δ; we need a customized version $\hat{\delta}^j_{\langle A \rangle^i} : [\![\langle A \rangle^i]\!] \to [\![\langle A \rangle^{i+j}]\!]$ which keeps us in the image of our translation. We define it in Figure 9, and note it never uses ϵ. It only uses the inverse ϵ^{-1}, which we *can* implement. We can then show the translation function on derivations preserves semantics.

Lemma 2: (Isomorphism) There exists an isomorphism $\mathit{iso}_{A_i}, \mathit{iso}^{-1}_{A_i} : [\![A_i]\!] \simeq \langle A \rangle^i$, which is additionally the identity on elements of the underlying sets.

Theorem 5: (Translation Preserves Semantics) Suppose $\Gamma \vdash e : A_i$ and $e' = \langle \Gamma \vdash e : A_i \rangle$. Then $\langle \Gamma \rangle \vdash e' : \langle A \rangle^i$, and $[\![\Gamma \vdash e : A_i]\!] = \mathit{iso}^{-1}_{A_i} \circ [\![\langle \Gamma \rangle \vdash e' : \langle A \rangle^i]\!] \circ \mathit{iso}_{\langle \Gamma \rangle}$.

This proof is easy since all of the operations we are considering (δ, $\hat{\delta}$, *iso*, and ϵ^{-1}) are identities on the underlying sets. In particular, that for closed terms of type $S(N)$, the

$$
\begin{array}{l}
\mathit{Mem} = \\
\Sigma S \in \mathit{Cell}, D, I, B \in \mathit{Loc}. \\
\quad \text{let } L = I \cup D \text{ and } C = S \cup \{\mathit{clock}\} \text{ in} \\
\quad \sigma : \Pi(A, \mathsf{c}) \in S.\ S(\llbracket A \rrbracket) \times \mathsf{code}\ (\!| A |\!) \times (1 + (\!| A |\!) \times \mathcal{P}(C)), \\
\quad \rho^D : \Pi(A, r) \in D.\ S(\tfrac{1}{2}\llbracket A \rrbracket) \times (\!| \bullet A |\!), \\
\quad \rho^I : \Pi(A, r) \in I.\ \llbracket S(A) \rrbracket \times ((\!| A |\!) + (\!| \bullet S(A) |\!) + (\!| S(A) |\!)), \\
\quad \rho^B : \Pi(A \to B, r) \in B.\ \llbracket A \to B \rrbracket \times (1 + (\!| A \to B |\!)) \\
\quad \mathit{Deps} \subseteq C \times C, \\
\quad \mathit{reader} : L \to S, \\
\quad \mathit{writer} : L \rightharpoonup S, \\
\quad I, D, \{i\} \text{ are mutually disjoint and } \mathit{clock} \notin S \\
\quad \mathit{reader}, \mathit{writer} \text{ are injective} \\
\quad \mathit{Deps} \text{ strict partial order} \\
\quad \text{let } \mathit{ready} = \lambda c \in S.\ \sigma(c) = (\_, \mathsf{inr}(\_, \_)) \text{ in} \\
\quad \text{let } V = \{c \in S \mid \mathit{ready}(c)\} \text{ in} \\
\quad \text{let } R = \{(c, d) \in C \times C \mid d \in \sigma(c)\} \\
\quad \forall c \in V \cap \mathit{writer}(L).\ (c, \mathit{clock}) \in R^+ \\
\quad \forall c \in V \cap \mathit{writer}(L).\ (c, \mathit{reader}(L)) \in R^* \\
\quad R^+ \subseteq \mathit{Deps}|_{V \times V} \\[1ex]
M' \overset{!}{\equiv} M = \\
\quad \rho^B_M = \rho^B_{M'} \wedge \\
\quad [(L_{M'} - \mathrm{dom}(\mathit{writer}_{M'})) = (L_M - \mathrm{dom}(\mathit{writer}_M))] \\
\mathit{Heap}_k(M, \psi) = \\
\quad H(\psi \otimes \mathsf{cell}^+(\mathit{clock}, (), \mathsf{return}(), \emptyset) \otimes \mathsf{ref}(\mathsf{i}, \mathrm{dom}(\mathit{writer})) \otimes \\
\qquad \mathrm{cells}(\sigma_M) \otimes \mathrm{refs}(\rho^I_M) \otimes \mathrm{refs}(\rho^D_M) \otimes \mathrm{refs}(\rho^B_M)) \wedge \\
\quad \forall (A, c) \in S_M.\ \mathit{Stream}^k_A(\pi_1(\sigma_M(c)), c, M) \wedge \\
\quad \forall (A, r) \in D_M.\ \mathit{Delay}^k_A(r, \mathit{head}(\pi_1(\rho^I_M(r))), \pi_2(\rho^I(r)), M) \wedge \\
\quad \forall (A, r) \in I_M.\ \mathit{Consref}^k_A(\pi_1(\rho^I_M(r)), \pi_2(\rho^I(r)), M) \wedge \\
\quad \forall (A \to B, r) \in B_M.\ \mathit{Function}^k_{A \to B}(\pi_1(\rho^B_M), \pi_2(\rho^B_M(r)), M) \\[1ex]
\mathit{Stream}^k_A(vs, c, M) = \\
\quad \forall j \le k, n \le j, M' \ll^n M.\ T^{j-n}_A(vs_n, \mathsf{hd}\ c, M') \\[1ex]
\mathit{Delay}^k_A(r, v, c, M) = \\
\quad \text{if } r \in \mathrm{dom}(\mathit{writer}_M) \wedge \mathit{ready}_M(\mathit{writer}_M(r)) \\
\quad \text{then } \forall j \le k.\ V^j_{\bullet A}(v, c, M) \\
\quad \text{else } \forall j \le k.\ T^j_A(v, c, M) \\[1ex]
\mathit{Consref}^k_A(r, xs, \mathsf{Init}(\mathsf{v}), M) = \\
\quad \forall j \le k.\ V^j_A(\mathit{head}(xs), \mathsf{v}, M) \\
\mathit{Consref}^k_A(r, xs, \mathsf{Make}(\mathsf{c}), M) = \\
\quad \text{if } r \in \mathrm{dom}(\mathit{writer}_M) \wedge \mathit{ready}_M(\mathit{writer}_M(r)) \\
\quad \text{then } \forall j \le k.\ V^j_{\bullet S(A)}(\mathit{tail}(xs), c, M) \\
\quad \text{else } \forall j \le k.\ T^j_{S(A)}(xs, c, M) \\
\mathit{Consref}^k_A(r, xs, \mathsf{Done}(\mathsf{xs}), M) = \\
\quad \forall j \le k.\ V^j_{S(A)}(xs, \mathsf{xs}, M) \\[1ex]
\mathit{Function}^k_{A \to B}(f, \mathsf{None}, M) = \top \\
\mathit{Function}^k_{A \to B}(f, \mathsf{Some}(\mathsf{f}), M) = V^k_{A \to B}(f, \mathsf{f}, M) \\[1ex]
\mathrm{cells}(\sigma_M) = \bigotimes_{c \in S_M} \mathit{cell}(c, \sigma_M(c)) \\[1ex]
\mathit{cell}((A, c), (\_, \mathit{code}, \mathsf{Some}(\mathsf{v}, \mathit{ds}))) = \mathsf{cell}^+(c, \mathit{code}, \mathsf{v}, \mathit{ds}) \\
\mathit{cell}((A, c), (\_, \mathit{code}, \mathsf{None})) \qquad\quad\ = \mathsf{cell}^-(c, \mathit{code}) \\[1ex]
\mathrm{refs}(\rho) = \bigotimes_{r \in \mathrm{dom}(\rho)} \mathsf{ref}(\mathsf{r}, \pi_2(\rho(\mathsf{r})))
\end{array}
$$

Fig. 7. Specification of Memories

$$
\begin{array}{l}
M' \sqsubseteq M = \\
\quad S_{M'} \supseteq S_M \wedge I_{M'} \supseteq I_M \wedge D_{M'} \supseteq D_M \wedge B_{M'} \supseteq B_M \wedge \\
\quad \mathit{reader}_{M'} \supseteq \mathit{reader}_M \wedge \mathit{writer}_{M'} \supseteq \mathit{writer}_M \wedge \\
\quad \mathit{Deps}_{M'} \supseteq \mathit{Deps}_M \wedge R_{M'} \supseteq R_M \wedge \\
\quad \rho^I_{M'} \sqsupseteq \rho^I_M \wedge \rho^D_{M'} \sqsupseteq \rho^D_M \wedge \rho^B_{M'} \sqsupseteq \rho^B_M \wedge \sigma_{M'} \sqsupseteq \sigma_M \\[1ex]
\sigma_{M'} \sqsupseteq \sigma_M = \\
\quad \forall c \in S_M.\ \mathit{ready}_M(c) \Rightarrow \sigma_M(c) = \sigma_{M'}(c) \\[1ex]
\rho^I_{M'} \sqsupseteq \rho^I_M = \\
\quad \forall r \in I_M, c \in \mathit{writer}_M(c). \\
\qquad \mathit{ready}_M(c) \vee \neg\mathit{ready}_{M'}(c) \Rightarrow \rho^I_M(r) = \rho^I_{M'}(r) \\[1ex]
\rho^D_{M'} \sqsupseteq \rho^D_M = \\
\quad \forall r \in D_M, c \in \mathit{writer}_M(c). \\
\qquad \mathit{ready}_M(c) \vee \neg\mathit{ready}_{M'}(c) \Rightarrow \rho^D_M(r) = \rho^D_{M'}(r) \wedge \\
\qquad \neg\mathit{ready}_M(c) \wedge \mathit{ready}_{M'}(c) \Rightarrow \pi_1(\rho^D_{M'}) = \mathit{tail}((\pi_1(\rho^D_M))) \\[1ex]
\rho^B_{M'} \sqsupseteq \rho^B_M = \\
\quad \forall r \in B_M.\ \rho^B_M(r) \neq \mathsf{None} \Rightarrow \rho^B_M(r) = \rho^B_{M'}(r) \\[2ex]
M' \ll^0 M \quad = M' \sqsubseteq M \\
M'' \ll^{n+1} M = \\
\quad \exists M'. M' \sqsubseteq M \wedge \mathrm{complete}(M') \wedge M'' \ll^n \mathit{Next}(M') \\
\mathrm{complete}(M) = \\
\quad \forall r \in L.\ r \in \mathrm{dom}(\mathit{writer}) \wedge \mathit{ready}_M(\mathit{writer}(c)) \wedge \\
\quad \forall r \in B.\ \pi_2(\rho^B_M(r)) \neq \mathsf{None} \\
\mathit{Next}(M \in \mathit{Mem}) = M' \in \mathit{Mem}, \text{ where} \\
\quad S_{M'} = S_M, \\
\quad D_{M'} = D_M, \\
\quad I_{M'} = I_M, \\
\quad B_{M'} = B_M, \\
\quad \mathit{Deps}_{M'} = \mathit{Deps}_M, \\
\quad \mathit{reader}_{M'} = \mathit{writer}_M, \\
\quad \mathit{reader}_{M'} = \mathit{writer}_M, \\
\quad \sigma_{M'} = \lambda c \in S_{M'}.\ (\mathit{tail}(\pi_1(\sigma_M(c))), \pi_2(\sigma_M(c)), \mathit{update}_M(c)) \\
\quad \rho^D_{M'} = \rho^D_M \\
\quad \rho^I_{M'} = \lambda c \in S_{M'}.\ (\mathit{tail}(\pi_1(\rho^I_M(c))), \pi_2(\rho^I_M(c))) \\
\quad \rho^B_{M'} = \rho^B_M \\
\mathit{update}_M(c) = \text{if } (\mathit{clock}, c) \in R_M \text{ then } \mathsf{None} \text{ else } \pi_3(\sigma_M(c))
\end{array}
$$

Fig. 8. Orderings on Memories

$$
\begin{array}{lcllcl}
\langle 1 \rangle^i & = & \bullet^i 1 & \hat{\delta}^j_{\langle A \rangle^i} & \in & \llbracket \langle A \rangle^i \rrbracket \to \llbracket \langle A \rangle^{i+j} \rrbracket \\
\langle A \times B \rangle^i & = & \bullet^i(\langle A \rangle^0 \star \langle B \rangle^0) & \hat{\delta}^j_{\langle P \rangle^i} & = & \delta^j_{\llbracket \bullet^i P \rrbracket} \\
\langle A \to B \rangle^i & = & \langle A \rangle^i \to \langle B \rangle^i & \hat{\delta}^j_{\langle \bullet A \rangle^i} & = & \hat{\delta}^{j+1}_{\langle A \rangle^i} \\
\langle S(A) \rangle^i & = & \bullet^i(S(\langle A \rangle^0)) & \hat{\delta}^j_{\langle A \to B \rangle^i} & = & (\epsilon^{-1} \circ \delta)^i \\
\langle \mathbb{N} \rangle^i & = & \bullet^i \mathbb{N} & \hat{\delta}^j_{\langle S(A) \rangle^i} & = & \tfrac{1}{2}^i(\delta^i_{\llbracket S(\langle A \rangle^0) \rrbracket}) \\
\langle \bullet A \rangle^i & = & \langle A \rangle^{i+1} & & &
\end{array}
$$

$$
\begin{array}{lcl}
\langle \Gamma, x : A_i \vdash x : A_j \rangle & = & \hat{\delta}^{j-i}_{A_i}(x) \\
\langle \Gamma \vdash \lambda x : A.\ e : A \to B_i \rangle & = & \lambda x : \langle A \rangle^i.\ \langle \Gamma, x : A_i \vdash e : B_i \rangle \\
\langle \Gamma \vdash e_1\ e_2 : B_i \rangle & = & \langle \Gamma \vdash e_1 : A \to B \rangle\ \langle \Gamma \vdash e_2 : A \rangle \\
\langle \Gamma \vdash \bullet e : \bullet A_i \rangle & = & \langle \Gamma \vdash e : A_{i+1} \rangle \\
\langle \Gamma \vdash \mathsf{await}(e) : A_{i+1} \rangle & = & \langle \Gamma \vdash e : \bullet A_i \rangle^i \\
\langle \Gamma \vdash \mathsf{cons}(e_1, e_2) : S(A)_i \rangle & = & \text{let } e_1' = \langle \Gamma \vdash e_1 : A_i \rangle \text{ in} \\
 & & \text{let } e_2' = \langle \Gamma \vdash e_2 : S(A)_{i+1} \rangle \text{ in} \\
 & & \bullet^i(\mathsf{cons}(\mathsf{await}^i(e_1'), \mathsf{await}^{i+1}(e_2'))) \\
\langle \Gamma \vdash \mathsf{hd}\ e : A_i \rangle & = & \bullet^i(\mathsf{hd}\ (\mathsf{await}^i(\langle e \rangle\ S(A)_i))) \\
\langle \Gamma \vdash \mathsf{tl}\ e : S(A)_{i+1} \rangle & = & \bullet^{i+1}(\mathsf{tl}\ (\mathsf{await}^i(\langle \Gamma \vdash e : S(A)_i \rangle)))
\end{array}
$$

$$
\langle \Gamma = x : A_i, \ldots, x' : A'_k \rangle = \langle A \rangle^i_0, \ldots, \langle A \rangle^k_0
$$

Fig. 9. Coercions and Translation of Types

translated term has exactly the same meaning as the original. We now give our correctness result.

Theorem 6: (Correctness of Compilation) Suppose $\Gamma \vdash e : A_i$ and $e' = \langle \Gamma \vdash e : A_i \rangle$. Then for each categorical combinator f used to interpret $[\![\langle\Gamma\rangle \vdash e' : \langle A \rangle^i]\!]$, there is a corresponding implementation f such that $(f, \mathsf{f}) \in \mathrm{Hom}(A, B)$.

More specifically, we implement all the expected operations of products, sums, exponentials, streams, and the later modality, with the sole exception of ϵ. We can also implement fixed points at all (nonempty) types, with an optimized implementation for fixed points of streams.

This lets us prove the following adequacy theorem.

Theorem 7: (Adequacy) First, suppose $M \in \mathit{Mem}$, and let step be the expression

$\mathsf{do}\ \mathit{ws} \leftarrow !\,i\,;\ \ \mathit{iter}\ \ \mathit{ws};\ \mathit{update}\ \ \mathit{clock}\ \ ()\,;\ \_ \leftarrow \mathsf{read}\ \mathit{clock}$

Then for all k, we can show the following Hoare triple:

```
$\{\mathit{Heap}_{k+1}(M, I)\}$
$\mathit{step}$
$\{\exists M' \ll^1 M.\ \mathit{Heap}_k(M, I)\}$
```

Now choose $\cdot \vdash vs : S(\mathbb{N})_0$, and let $\mathsf{cmd} = (\!|\langle \cdot \vdash vs : S(\mathbb{N})_0 \rangle|\!)$. For all k and $i \leq k$,

```
$\{\mathit{Heap}_k(M_\perp, I)\}$
do $\mathit{vs} \leftarrow \mathit{cmd}$;
    $\mathit{repeat}\ \ \mathit{step}\ \ i\,$;
    hd$(\mathit{vs})$
$\{(a, \_).\ \exists M' \ll^n M.\ \mathit{Heap}_k(M', I) \wedge a = vs_i\}$
```

Here, we first show that step advances the clock (thus showing how to implement the event loops), and then show that if vs is a closed term of type $S(\mathbb{N})$, then building the stream it computes and advancing the event loop i steps will give us the i-th element of the stream.

V. Discussion

A. Related Work

Escardo described a metric semantics of PCF [17], in which he interpreted PCF with metric spaces. The construction of the lift monad he gave translates quite naturally to our setting, with $A_\perp = (A \times \mathbb{N}) + 1$, with a metric resembling the recursive type $\mu\alpha.\ 1 + \bullet\alpha$. This has obvious applications for modelling features of interactive programs such as timeouts, which we plan to investigate in the future.

Uustalu and Vene [18] observed that streams have a comonad structure whose co-Kleisli category is Cartesian closed, elegantly extending implicit lifting from the first-order setting to the higher-order case. Unfortunately, it is difficult to interpret fixed points in this category. The category of free coalgebras has too few global points (maps $S(1) \to S(\mathbb{N})$) to denote very many streams, including such basic ones such as $(0, 1, 4, 9, \ldots)$.[1]

The original work on functional reactive program [1] was based on writing reactive programs as unrestricted stream programs, but due to problems with causality, variations such as *arrowized FRP* [19] were introduced to give combinators restricting the definable stream transformers to the causal ones, corresponding roughly to the first-order stream programs, with some special operators for dynamic behavior.

One notable feature of this work is its focus on continuous time. We hope our proof framework can extend to proving a sampling theorem, as in Wan and Hudak [20]. On the semantic side, we can easily model continuous behaviors as function $\mathbb{R} \to A$, but relating it to an implementation delivering time deltas (instead of ticks, as presently) will be more challenging.

There has been much recent work on clarifying the foundations of step-indexed logical relations [16]. Our relation demonstrates a very intricate use of higher-order state, and it would be particularly interesting to see whether it has a natural expression in Dreyer *et al.*'s [21] transition-system indexed relations.

References

[1] C. Elliott and P. Hudak, "Functional reactive animation," in *ICFP*, 1997.

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

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

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

[5] H. Liu, E. Cheng, and P. Hudak, "Causal commutative arrows and their optimization," in *ACM International Conference on Functional Programming*, 2009.

[6] N. Sculthorpe and H. Nilsson, "Safe functional reactive programming through dependent types," in *ICFP*, 2009.

[7] H. Nakano, "A modality for recursion," in *LICS*, 2000, pp. 255–266.

[8] L. Birkedal, J. Schwinghammer, and K. Stvring, "A metric model of guarded recursion," in *FICS*, 2010.

[9] T. Brauener, *Hybrid Logic and its Proof Theory*. Springer, 2011.

[10] K. Watkins, I. Cervesato, F. Pfenning, , and D. Walker, "A concurrent logical framework i: Judgments and properties," Carnegie Mellon University, Tech. Rep. CMU-CS-02-101, 2002.

[11] N. Zeilberger, "Focusing and higher-order abstract syntax," in *POPL*, G. C. Necula and P. Wadler, Eds. ACM, 2008, pp. 359–369.

[12] W. Buchholz, S. Feferman, W. Pohlers, , and W. Sieg, *Iterated Inductive Definitions and Subsystems of Analysis: Recent Proof-Theoretical Studies*. Springer-Verlag, 1981.

[13] B. Biering, L. Birkedal, and N. Torp-Smith, "BI-hyperdoctrines, higher-order separation logic and abstraction," *ACM TOPLAS*, vol. 29, no. 5, 2007.

[14] N. R. Krishnaswami, "Verifying higher-order imperative programs with higher-order separation logic," Ph.D. dissertation, Carnegie Mellon University, forthcoming. [Online]. Available: http://www.cs.cmu.edu/~neelk/thesis.pdf

[15] N. Krishnaswami, L. Birkedal, and J. Aldrich, "Verifying event-driven programs using ramified frame properties," in *TLDI*, 2010.

[16] D. Dreyer, A. Ahmed, and L. Birkedal, "Logical step-indexed logical relations," in *LICS*. IEEE, 2009, pp. 71–80.

[17] M. Escardó, "A metric model of PCF," in *Workshop on Realizability Semantics and Applications*, 1999.

[18] T. Uustalu and V. Vene, "The essence of dataflow programming," in *Central European Functional Programming School*, ser. LNCS, vol. 4164, 2006.

[19] H. Nilsson, A. Courtney, and J. Peterson, "Functional reactive programming, continued," in *ACM Haskell Workshop*. ACM, 2002, p. 64.

[20] Z. Wan and P. Hudak, "Functional reactive programming from first principles," in *PLDI*, 2000, pp. 242–252.

[21] D. Dreyer, G. Neis, and L. Birkedal, "The impact of higher-order state and control effects on local relational reasoning," in *ICFP*. ACM, 2010, pp. 143–156.

[1] In earlier (unpublished) work we looked at interpreting both categories, and mediating between them with an adjunction. An ML implementation of this work, including GUI programs, can be found on our website. We will soon port this to our new language.

Appendix

Relation to Nakano's Calculus.

Recently, Birkedal *et al.* [8] gave a metric model of Nakano's calculus of guarded recursion [7], using a model very similar to ours — in fact, once we learned of this, we chose our syntax of types to be identical. Below , we give a semantics-preserving interderivability theorem for the fragments of the two systems without recursive types.

Since Nakano gave a general syntax for contractive, equirecursive types, we will follow the syntax in Birkedal *et al.* [8], which is the fragment of Nakano's calculus corresponding more closely to our language. We will write Δ for a sequence of hypotheses $x : A, \ldots, x' : A'$, and write $\bullet^n \Delta$ for $x : \bullet^n A, \ldots, x' : \bullet^n A'$. We can send these contexts to our annotated contexts by $\langle x : A, \ldots, x' : A' \rangle^{+n} = x : A_n, \ldots, x' : A'_n$. Then the following two theorems hold for the $1, \times, \to, \bullet$ fragment of the two systems.

Lemma 3: (Derivability of Subtyping) If $A \leq B$ in Nakano's calculus, then there exists a term e such that $\cdot \vdash e : A \to B$ in our calculus. Furthermore, their denotational interpretation of the coercion equals our interpretation of e.

Theorem 8: (Interderivability) If $\bullet^n \Delta \vdash t : \bullet^n A$ in Nakano's calculus, then there exists a term e such that $\langle \Delta \rangle^{+n} \vdash e : A_n$, such that $[\![\bullet^n \Delta \vdash t : \bullet^n A]\!]$ under Birkedal *et al.*'s semantics equals $[\![\langle \Delta \rangle^{+n} \vdash e : A_n]\!]$ under our semantics. Furthermore, the converse also holds.

The proof is a routine structural induction, which makes use of the time adjustment lemma.

Unfortunately, this correspondence does not extend to recursive types, as Nakano's rules for recursive types (and hence Birkedal *et al.*'s semantics) differ from ours.

They interpret recursive types so that $[\![\mu\alpha.\, \tau(\alpha)]\!] \simeq \frac{1}{2}[\![\tau(\mu\alpha.\, \tau(\alpha))]\!]$. This means that under their interpretation of streams $\mu\alpha.\, A \times \alpha$, the first element of the stream is an element of $\frac{1}{2}(A)$, whereas in our semantics the head is an element of A.

This difference arises from the differing operational idea underlying each semantics. The intuition for guarded recursion is that $\frac{1}{2}$ corresponds to a value lying underneath a constructor, whereas we interpret it as a computation scheduled to run on a future trip through an event loop. Both of these give rise to perfectly sensible metric structures, though ultrametric spaces are more abstract than either operational model.

It would be interesting to look for models which reflect more of the operational content, though of course part of the purpose of abstraction is to suppress such details!

Normalization Proof. We prove normalization giving a bidirectional type system for the normal forms, and then defining a hereditary substitution operation for it.

In Figure 10, we give the syntax and typing of normal forms, with two judgments $\Gamma \vdash n \Leftarrow A_i$ for checking that n is a normal term with type A, and a judgment $\Gamma \vdash t \Rightarrow A_i$, which takes an atomic form and synthesizes a type A for it. Note that beta-redexes are not typeable in this type system – applications and projections are always to a head variable. Also note that

$$
\begin{array}{rcll}
n & ::= & \lambda x : A.\, n \mid \bullet\, n \mid \mathsf{cons}(n, n) \mid t & \text{Normal} \\
t & ::= & x \mid t\, n \mid \mathsf{await}(t) \mid \mathsf{hd}\; t \mid \mathsf{tl}\; t & \text{Atomic}
\end{array}
$$

$$\boxed{\Gamma \vdash n \Leftarrow A_i} \qquad \boxed{\Gamma \vdash t \Rightarrow A_i}$$

$$\frac{\Gamma, x : A_i \vdash n \Leftarrow B_i}{\Gamma \vdash \lambda x.\, n \Leftarrow A \to B_i} \to\!\text{I} \qquad \frac{\Gamma \vdash n \Leftarrow A_{i+1}}{\Gamma \vdash \bullet n \Leftarrow \bullet A_i} \bullet\text{I}$$

$$\frac{\Gamma \vdash n \Leftarrow A_i \qquad \Gamma \vdash e' \Leftarrow S(A)_{i+1}}{\Gamma \vdash \mathsf{cons}(n, e') \Leftarrow S(A)_i} S\text{I}$$

$$\frac{\Gamma \vdash t \Rightarrow A_i}{\Gamma \vdash t \Leftarrow A_i} \textsc{Shift} \qquad \frac{x : A_i \in \Gamma \qquad i \leq j}{\Gamma \vdash x \Rightarrow A_j} \textsc{Hyp}$$

$$\frac{\Gamma \vdash t \Rightarrow A \to B_i \qquad \Gamma \vdash n \Leftarrow A_i}{\Gamma \vdash t\, n \Rightarrow B_i} \to\!\text{E}$$

$$\frac{\Gamma \vdash t \Rightarrow \bullet A_i}{\Gamma \vdash \mathsf{await}(t) \Rightarrow A_{i+1}} \bullet\text{E} \qquad \frac{\Gamma \vdash t \Rightarrow S(A)_i}{\Gamma \vdash \mathsf{hd}\; t \Rightarrow A_i} S\mathsf{hd}$$

$$\frac{\Gamma \vdash t \Rightarrow S(A)_i}{\Gamma \vdash \mathsf{tl}\; t \Rightarrow S(A)_{i+1}} S\mathsf{tl}$$

Fig. 10. Syntax and Typing

by changing the arrow $\Rightarrow$ or $\Leftarrow$ into a colon :, we have a derivation in our original type system.

As a result, the ordinary substitution theorem does not go through, since that can introduce redexes. Instead, in Figure 11, we define a pair of mutually-recursive procedures for substituting a normal form n for a variable x. The procedure $\langle\!| n/x |\!\rangle_A\, n'$ substitutes the normal form n (of type A) for the free variable x in n'. The procedure $\langle n/x \rangle_A\, t$ performs the same substitution, only into an atomic term. This procedure can return either a normal form or an atomic term, and in the case that it returns an atomic term, it also computes the type of the expression t.

We can then prove the following theorem:

Theorem 9: (Hereditary Substitution) Suppose $\Gamma \vdash n \Leftarrow A_i$. Then

- If $\Gamma, x : A_i \vdash n' \Leftarrow C_n$, then $\Gamma \vdash \langle\!| n/x |\!\rangle_A\, n' \Leftarrow C_n$
- If $\Gamma, x : A_i \vdash t \Rightarrow C_n$, then either
  - $\langle n/x \rangle_A\, t = (n', C)$ and C is a subterm of A and $\Gamma \vdash n' \Leftarrow C_n$, or
  - $\langle n/x \rangle_A\, t = (t', \bot)$ and $\Gamma \vdash t' \Rightarrow C_n$

Furthermore, in all cases the result of the substitution is $\beta\eta$ equal to $[n/x]n'$.

The proof of this theorem relies on an induction on the size of A and the derivations of the two subterms. It is lexicographic between A and the unordered pair of the sizes of the derivations of the subterms. This establishes a weak normalization result for our calculus, in a very simple way.

Implementation.

$$\begin{array}{lcl}
\llparenthesis n/x \rrparenthesis_A\, \lambda y.\, e' & = & \lambda y.\, \llparenthesis n/x \rrparenthesis_A\, e' \\
\llparenthesis n/x \rrparenthesis_A\, \bullet e' & = & \bullet \llparenthesis n/x \rrparenthesis_A\, e' \\
\llparenthesis n/x \rrparenthesis_A\, \mathsf{cons}(n_1, n_2) & = & \mathsf{cons}(\llparenthesis n/x \rrparenthesis_A\, n_1, \llparenthesis n/x \rrparenthesis_A\, n_2) \\
\llparenthesis n/x \rrparenthesis_A\, t & = & \mathsf{let}\ (e', \_) = \langle n/x \rangle_A\, t\ \mathsf{in}\ e'
\end{array}$$

$$\begin{array}{lcl}
\langle n/x \rangle_A\, y & = & (y, \bot) \\
\langle n/x \rangle_A\, x & = & (n, A) \\
\langle n/x \rangle_A\, t_1\ n_2 & = & \mathsf{let}\ e_2' = \llparenthesis n/x \rrparenthesis_A\, n_2\ \mathsf{in} \\
 & & \mathsf{case}\ \langle n/x \rangle_A\, t_1\ \mathsf{of} \\
 & & \quad (\lambda y.\, e_1', B \to C) \Rightarrow \llparenthesis e_2'/y \rrparenthesis_B\, e_1' \\
 & & \quad (t_1', \_) \Rightarrow (t_1'\ e_2', \bot) \\
\langle n/x \rangle_A\, \mathsf{await}(t) & = & \mathsf{case}\ \langle n/x \rangle_A\, t\ \mathsf{of} \\
 & & \quad (\bullet e', \bullet A') \Rightarrow (e', A') \\
 & & \quad (t', \_) \Rightarrow (\mathsf{await}(t'), \bot) \\
\langle n/x \rangle_A\, \mathsf{hd}\ t & = & \mathsf{case}\ \langle n/x \rangle_A\, t\ \mathsf{of} \\
 & & \quad (\mathsf{cons}(e', e''), S(A')) \Rightarrow (e', A') \\
 & & \quad (t', \_) \Rightarrow (\mathsf{hd}\ t', \bot) \\
\langle n/x \rangle_A\, \mathsf{tl}\ t & = & \mathsf{case}\ \langle n/x \rangle_A\, t\ \mathsf{of} \\
 & & \quad (\mathsf{cons}(e', e''), S(A')) \Rightarrow (e'', S(A')) \\
 & & \quad (t', \_) \Rightarrow (\mathsf{tl}\ t', \bot)
\end{array}$$

Fig. 11. Hereditary Substitution

In what follows, the general pattern is that when we give a type foo : $\mathrm{Hom}(A, B)$, where foo is the name of a categorical combinator *foo*, we mean the following code satisfies the spec $(\mathit{foo}, \mathsf{foo}) \in \mathrm{Hom}(A, B)$. Some operations (such as pairing and currying) are higher-order, and we give their specifications more explicitly.

```
// Basic operations on categories
id : Hom(A, A)
id x = return x

compose : Hom(A, B) →Hom(B, C) →Hom(A, C)
compose f g = λa. do b ← f a;
                     g b

//
// Units and pairs
//

one : Hom(A, 1)
one x = return ()

fst : Hom(A × B, A)
fst (a,b) = return a

snd : Hom(A × B, A)
snd (a,b) = return b

// The spec of pairing is:
// ∀ (f,f') ∈ Hom(A,B), (g,g') ∈ Hom(A,C).
//   (⟨f,g⟩, pair f' g') in Hom(A, B ×C)
//
pair : Hom(A, B) → Hom(A, C) → Hom(A, B × C)
pair f g = λ a. do b ← f a;
                   c ← g a;
                   return (b,c)

//
// Sums
//

inl : Hom(A, A + B)
inl x = return (inl x)

inr : Hom(B, A + B)
inr x = return (inr x)

// The spec of sum is:
// ∀ (f,f') ∈ Hom(A,C), (g,g') ∈ Hom(B,C).
//   ([f, g], sum f' g') in Hom(A + B, C)
//
sum : Hom(A, C) → Hom(B, C) → Hom(A + B, C)
sum f g =
  λ v. case v of
         Inl x → f x
         Inr y → g y

//
// Stream operations
//

// The spec of mapS is:
// ∀ (f,f') ∈ Hom(A,B),
//   (S(f), mapS f') in Hom(S(A), S(B))

mapS : Hom(A,B) →Hom(S(A), S(B))
mapS f = λxs. cell (do x ← hd(xs); f x)

hd : Hom(S(A), A)
hd xs = read xs

tl : Hom(S(A), •S(A))
tl xs = return (return xs)

zip : Hom(S(A) × S(B), S(A × B))
zip (xs, ys) = cell (do x ← hd(xs);
                        y ← hd(ys);
                        return (x,y))

cons : Hom(A × •S(A), S(A))
cons (x, dxs) = do r ← ref (Init x);
                   ys ← cell (do () ← read(clock);
                                 x' ← !r;
                                 case x' of
                                  Init xs →
                                    do r := Make(dxs); hd(xs)
                                  Make dxs →
                                    do xs ← dxs;
                                       r := Done xs;
                                       hd(xs)
                                  Done xs →
                                    hd(xs));
                   register (ys)

tails : Hom(S(A), S(S(A)))
tails xs = cell (return xs)

unzip : Hom(S(A × B), S(A) × S(B))
unzip = pair (mapS fst) (mapS snd)

//
// Operations for the later modality
//

// The spec of mapS is:
```

```
// ∀ (f, f') ∈ Hom(A,B),
//    (½(f), mapN f') in Hom(•A, •B)
//

mapN : Hom(A,B) →Hom(•A, •B)
mapN f = λd. return (do a ← d; f a)

// We define delay_A as an inductive family
// following the type structure . In ML this
// would be written as some a collection of
// combinators

delay_ℕ : Hom(ℕ, •ℕ)
delay n = return (return n)


delay_1 : Hom(1, •1)
delay () = return (return ())

delay_{A→B} : Hom(A → B, •(A → B))
delay f = return (λ d. return (do a ← get_A d;
                                  f(a)))

delay_{A×B} : Hom(A × B, •(A × B))
delay (a,b) = do d ← delay_A(a);
                 e ← delay_B(b);
                 return (do a ← d;
                            b ← e;
                            return (d, e))

delay_{•A} : Hom(• A, •(• A))
delay d = return (do a ← d;        // at the following
                     delay_A(a))   // step get a and delay it

delay_{S(A)} : Hom(S(A), •(S(A)))
delay xs = do x  ← hd(xs);
              xt ← delay_A(xs);
              return
                (do r ← ref xt;
                    c  ← cell (do () ← read(clock);
                                  oldt ← !r;
                                  old ← oldt;
                                  new ←hd(xs);
                                  newt ← delay_A(new);
                                  r := newt;
                                  return old);
                    register (c))


// Now we give the Cartesian closed structure for delays

ziphalf : Hom(•A ×•B, •(A ×B))
ziphalf (da, db) = return (do a ← da;
                              b ← db;
                              return (a,b))

unziphalf : Hom(•(A ×B), •A ×•B)
unziphalf dab = do da ←return (do (a,b) ← dab;
                                  return a);
                   db ← return (do (a,b) ← dab;
                                  return b);
                   return (da,db)

onehalf : Hom(1, •1)
onhalf () = return (return ())

epsilon_inv : Hom(•(A →B), •A → •B)
epsilon_inv d = return (λ e. return (do f ← d;
                                        v ← e;
                                        f(v)))

epsilon : Hom(•A →•B, •(A → B))
// There is no implementation for this ..!

//
// Exponentials
//

apply : Hom((A → B) × A, B)
apply (f, v) = f v

// To implement currying, we need to be able to save
// the the environment so that it remains
// the same in future time steps. This can get expensive
// when you capture a stream as a free variable, since there
// will be unbounded buffering!
//
constant : Hom(A, S(A))
constant x = do r ← ref (return x)
                xs ← cell (do () ← read(clock);
                              dx ← !r;
                              x ← dx;
                              dx' ← delay_A(x);
                              r := dx';
                              return(x));
                register (xs)

// The spec of curry is:
// ∀ (f, f') ∈ Hom(A ×B, C),
//    (λ(f), curry f') in Hom(A, B →C)
//

curry : Hom(A × B, C) → Hom(A, B →C)
curry(f) = λ a. do as ← constant a;
                   return (λ b. do a ← head(as); f(a,b))

//
// Fixed points
//
// These are morally trace operators rather than fixed points,
// which are nicer when translating a lambda calculus into
// categorical operations

fix : Hom(A ×•S(B), S(B)) → Hom(A, S(B))
fix f = λ a. do r ← ref None;
                preinput ← cell (!r);
// None, Some fix(f)_0, ...
                input ← return (do vs' ← preinput;
                                   cell (do v' ← head(vs');
                                            valOf v'));
                preoutput ← f(a, input);
                out ← cell (do () ← read(clock);
                               _ ← head(preinput);
                               v ← head(preoutput);
                               d ← delay_B(v);
                               r := d;
                               return v)
                register (out)

fix : Hom(A ×•(B →C), B → C) → Hom(A, B →C)
fix f = λ a. do r ← ref None;
                d ← return (do f' ←!r; test f');
                g ← f(d);
```

```
                        $r$ := $g$;
                        return $g$

// This is another version, which works better with the
// translation in the paper.
//
fix : $\mathrm{Hom}(A \times \bullet B \to \bullet C, B \to C) \to \mathrm{Hom}(A, B \to C)$
fix $f$ = $\lambda\, a.$ do $r \leftarrow$ ref None;
                        $d \leftarrow$ return (do $f' \leftarrow$
!$r$; valof $f'$); // $\bullet(B \to C)$
                        $g \leftarrow$ now $d$; // $\bullet B \to \bullet C$
                        $h \leftarrow f\ g$;    // $B \to C$
                        $r$ := Some $h$; // tie the knot
                        return $h$

// Utility operations

valOf $x'$ = case $x'$ of
                Some $x \to$ return $x$
                None $\to$ $n/a$

register ($c$) = do $i$ := $c$ :: $xs$;
                       return $c$
```