An Ultrametric Model of Reactive Programming

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, which provide a natural Cartesian closed generalization of causal stream functions. We define a domain-specific language corresponding to the model. We then show how reactive programs written in this language may be implemented efficiently using an imperatively updated dataflow graph and give a higher-order separation logic proof that this low-level implementation is correct with respect to the high-level semantics.

1. Introduction

There is a broad spectrum of models for reactive programming. Functional reactive programming (FRP), as introduced by Elliott and Hudak [12], is highly expressive and generally shallowly embedded in powerful general-purpose languages. At the other end, synchronous languages such as Esterel [5], Lustre [8] and Lucid Synchrone [17] provide a restricted, domain-specific model of computation supporting specialized compilation strategies and analysis techniques. Synchronous languages have been extremely successful in application areas 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. More recent variants of FRP [15, 18] 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 uses of 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 — and 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 [8] and Lucid Synchrone [17]. 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 those 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, which generalizes previous models based on causal stream functions. Our model supports full Cartesian closed structure, thus giving a natural mathematical notion of higher-order reactive programs.

The use of metric spaces means that we can 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. Since streams distribute over products and form a comonad, the co-Kleisli category of streams is also Cartesian closed, thus giving us *two* natural notions of function for reactive programs. Prior work [7, 19] has focused primarily on the co-Kleisli category, but the interpretation of fixed points is significantly more natural in the base category of ultrametric spaces. By adapting the adjoint calculus presentation of intuitionistic linear logic [3, 4], our language allows one to work in the two categories simultaneously. The idea is to decompose the stream comonad into a pair of adjoint functors, which in the term calculus become modal operators connecting the two lambda calculi. We can then interpret fixed points in the category of ultrametrics (thereby retaining a simple equational theory for them) while still enabling programming implicitly with streams, as is common in dataflow languages. Furthermore, we extend the adjoint calculus with additional judgements to track contractiveness, so that we can use typechecking to ensure that clients can only take fixed points of well-defined, strictly-contractive functions.

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 rather 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.

2. 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

$\texttt{Price}^\omega \to \texttt{Order}^\omega$ 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 output at time n should depend only on the first n inputs. We formalize this as follows, 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 said to be* causal *when, for all 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.

Whilst causality is an intuitive and appealing definition for streams of basic types (such as integers), 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:

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

An operational way of thinking about when such fixed points are well-defined is to observe that the function $\lambda\texttt{xs}.\ \texttt{0} :: \texttt{map succ xs}$ can produce its first output without looking at its input. So we can 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 said to be* guarded *when there exists a $k > 0$ such that for all for all n and streams 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 an intuitive and natural property, but generalizations to higher types seem both useful and unobvious. For example, we may want to write a recursive *function*:

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

So the natural questions to ask are: what does guardedness mean at higher type, and how can we interpret fixed points at higher type? We will answer these questions by moving to metric spaces.

3. 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 rather than (A, d_A).

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 said to be *strictly contractive* when it shrinks the distance between any two points by a nonzero 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 is given by the Cartesian product of the underlying sets, equipped with the pointwise sup-metric:

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

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

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

Any set X can be made into an ultrametric space $D(X)$ by equipping it with the discrete metric that defines $d(x,x')$ to be 0 if $x = x'$ and 1 otherwise.

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*:

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

Furthemore, this is functorial: for any map $f : A \to B$, we define $f^\omega : S(A) \to S(B)$ by mapping f over the input stream, which is easily seen to preserve identity and composition.

The interpretation of the stream metric is easiest to understand in the case of streams of discrete elements. In this case, the 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 hence will be equal streams).

PROPOSITION 2. *(Banach's Fixed Point Theorem) For any nonempty, complete metric space A and strictly contractive endofunction $f : A \to A$, there exists a unique fixed point of f.*

3.1 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 in the previous section.

THEOREM 1. *(Causality is Nonexpansiveness) Suppose A and B are sets. Then 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) Suppose A and B are sets. Then 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. However, the consequences of are quite dramatic! By interpreting our programs in the category of ultrametric spaces:

1. We can interpret tuples and functions (with the full β and η rules) thanks to the Cartesian closure of this category.
2. Since streams are functorial, we can interpret streams of streams.

3. Furthermore, contractiveness gives an analogue of guardedness that makes sense at higher types, and likewise Banach's fixed point theorem gives an interpretation of fixed points that makes sense at higher types.

In an abstract sense, this semantics fulfill the original promise of functional reactive programming 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.

3.2 The Co-Kleisli Category of Streams

Synchronous dataflow languages like Lucid Synchrone [17], have a programming model that differs somewhat from that which the previous section suggests. In these languages time is *implicit*, and streams are only rarely manipulated directly. A definition like $\mathtt{sum} = \lambda(\mathtt{x}, \mathtt{y}).\ \mathtt{x} + \mathtt{y}$ is implicitly *lifted* to operate pointwise over streams. One reason for this choice is brevity, but a more important one is operational. Arbitrary stream functions, even causal ones, can be hard or impossible to implement without *space leaks*. We aim to implement reactive programs using some state such that the current inputs and state determine the current outputs and next state. For hardware compilation or hard real-time programming, one needs that state to be bounded, and even for less constrained applications, it is unacceptable for the state to grow unboundedly. By keeping time implicit, restricting oneself to stream functions that are the the result of the compiler's automatic lifting, one can make it harder (or impossible) to write programs that leak memory by retaining arbitrary amounts of history.

Fortunately, we can capture the essence of this class of restriction in our mathematical semantics by working in the co-Kleisli category of the stream comonad on the category of ultrametric spaces.

Recall that a *comonad* on a category $\mathbb{U}$ is a functor $S : \mathbb{U} \to \mathbb{U}$, equipped with two natural transformations $\epsilon_A : S(A) \to A$ (the counit) and $\delta_A : S(A) \to S(S(A))$ (the comultiplication) satisfying the equations $\delta_A; S(\delta_A) = \delta_A; \delta_{S(A)}$ and $\delta_A; S(\epsilon_A) = id = \delta_A; \epsilon_{T(A)}$. In the case of streams, the counit ϵ is the head function on streams, and the comultiplication takes a stream and returns the a stream containing the successive tails of the input stream.[1] $S(\cdot)$ is a *strong* functor: $S(A \times B) \simeq S(A) \times S(B)$. We write `unzip` and `zip` for the components of this isomorphism.

The co-Kleisli category $\mathbb{U}^S$ of a comonad $S : \mathbb{U} \to \mathbb{U}$ is the category of free S-coalgebras, which may be presented as having the same objects $\mathbb{U}$, but taking maps from A to B to be in $\mathbb{U}^S$ to be maps $f : S(A) \to B$ in $\mathbb{U}$. Amazingly, $\mathbb{U}^S$ is also cartesian closed; the identity, composition, projection, pairing, currying and evaluation maps are defined in Figure 1.

We think of a map $e : A \to B$ in $\mathbb{U}^S$ as the interpretation of a synchronous dataflow program with a free variable of type A in the style of Lucid Synchrone - i.e. something that is really implicitly lifted to work on streams. The dynamic behavior of synchronous terms is understood via the coextension; $e^\dagger$ is a map $e : S(A) \to S(B)$ in $\mathbb{U}$, and is the function we get by feeding e the successive tails of the original input stream. So this says that if the input A takes on the values $as = [a_0, a_1, a_2, \ldots]$, then the output results will be $[e(as), e(tail(as)), e(tail(tail(as))), \ldots]$.

Although $\mathbb{U}^S$ is Cartesian closed, it does not have coproducts. Furthermore the instantaneous interpretation of terms makes it hard to support operations acting on streams of streams. The key difficulty is that there is no map in this category which takes a stream of streams, and whose coextension yields the head of the stream of streams. This, in turn, makes defining fixed point operators very difficult: a variable of type $A \Rightarrow A$ denotes a stream of functions. If all of these functions are contractive, and we take fixed point of coextensions pointwise, then we get a stream of streams — at which point we discover that we can never look at the *second* element of any of the results. This amnesia is the very reason that space leaks become harder to program: this is not an accidental difficulty!

[1] The "Kleisli triple" formulation for monads is perhaps more familiar to functional programmers, which is given in terms of an extension operator sending maps $f : A \to T(B)$ to $\mathtt{bind}(f) : T(A) \to T(B)$. Dually, the *coextension* for comonads sends maps $f : S(A) \to B$ to $f^\dagger : S(A) \to S(B)$, and can be defined as $f^\dagger = \delta; S(f)$.

$$
\begin{array}{lcl}
\mathsf{id} & = & \epsilon \\
\mathsf{f;g} & = & \delta; S(f); g \\
\\
\mathsf{fst} & = & \mathsf{unzip}; \pi_1; \epsilon \\
\mathsf{snd} & = & \mathsf{unzip}; \pi_2; \epsilon \\
\mathsf{pair}(f, g) & = & (f, g) \\
\\
\mathsf{curry}(f) & = & \lambda(\mathsf{zip}; f) \\
\mathsf{eval} & = & \mathsf{unzip}; (\epsilon \times id); eval
\end{array}
$$

Figure 1. Definition of operations in the co-Kleisli category

3.3 Adjoint Logic

At this point, we have two views of reactive programming. One, which we might call "FRP-style", has a very simple semantics in terms of ultrametrics and stream values, and naturally supports fixed points at any type. However, it seems difficult to implement efficiently. On the other hand, we can take a "synchronous dataflow" view to support more efficient implementation techniques, at the price of making it difficult to give good semantics to feedback. So it is natural to ask if there is some way of combining the strengths of the two approaches. We meet this goal by adapting adjoint-style models of intuitionistic linear logic [3, 4]. Originally developed to give a model of linear logic, these turn out to be abstract enough to apply naturally to the setting of dataflow programming as well.

DEFINITION 3. *An* adjoint model *is specified by:*

1. *A cartesian closed category* $(\mathbb{U}, 1, \times, \Rightarrow)$.
2. *A symmetric monoidal closed category* $(\mathbb{U}^S, I, \otimes, \multimap)$.
3. *A symmetric monoidal adjunction* $((-)^\omega, V, \eta, \varepsilon, m, n)$ *from* $\mathbb{U}$ *to* $\mathbb{U}^S$. *Here,* η *and* ε *witness the adjunction, and* m *and* n *are the natural transformations showing that the monoidal structure is preserved.*

In our setting, we take the CCC to be the co-Kleisli category of ultrametric spaces, and take the monoidal closed structure in the definition to be the cartesian closed structure of the category of ultrametric spaces (since Cartesian products are a special case of monoidal products). We decompose the stream comonad into the usual free and forgetful functors that go between the category of ultrametric spaces and the co-Kleisli category of streams over it. Specifically, the functors $(-)^\omega : \mathbb{U}^S \to \mathbb{U}$ and $V(-) : \mathbb{U} \to \mathbb{U}^S$ are defined as:

$$
\begin{array}{lcl}
X^\omega & = & S(X) \\
(f : X \to Y)^\omega & = & \delta_X; S(f) = f^\dagger \\
V(A) & = & A \\
V(f : A \to B) & = & \epsilon_A; f
\end{array}
$$

Intuitively $V(f)$ takes a function f from the general FRP world and embeds it into the synchronous dataflow world by having it act at each instant on the head of the stream. The action of the other half of the adjoint $(g)^\omega$ takes a synchronous dataflow function, and turns it into a general function on streams, via the coextension operation. Verifying that $V(f)^\omega = S(f)$ is immediate from the definitions.

Furthemore, we know that $(X \times Y)^\omega \simeq X^\omega \otimes Y^\omega$, and also $X \Rightarrow V(B) \simeq V(X^\omega \multimap B)$. This latter isomorphism is actu-

$$\frac{x : X \in \Gamma}{\Gamma \vdash x : X} \qquad \frac{\Gamma \vdash e : X \Rightarrow Y \qquad \Gamma \vdash e' : X}{\Gamma \vdash e\, e' : Y}$$

$$\frac{\Gamma, x : X \vdash e : Y}{\Gamma \vdash \lambda x.\, e : X \Rightarrow Y} \qquad \frac{\Gamma; x : X \vdash e : Y}{\Gamma \vdash \lambda x.\, e : X \rightsquigarrow Y} \qquad \frac{}{\Gamma \vdash \langle\rangle : 1}$$

$$\frac{\Gamma \vdash e : A \qquad \Gamma \vdash e' : Y}{\Gamma \vdash (e, e') : X \times Y} \qquad \frac{\Gamma \vdash e : X_1 \times X_2}{\Gamma \vdash \pi_i\, e : X_i} \qquad \frac{x : A \in \Delta}{\Gamma; \Delta \vdash x : A}$$

$$\frac{\Gamma; \Delta \vdash t : A \multimap B \qquad \Gamma; \Delta \vdash t' : A}{\Gamma; \Delta \vdash t\, t' : B} \qquad \frac{\Gamma; \Delta, x : A \vdash t : B}{\Gamma; \Delta \vdash \lambda x.\, t : A \multimap B}$$

$$\frac{\Gamma; \Delta; x : A \vdash t : B}{\Gamma; \Delta \vdash \lambda x.\, t : A \multimapdot B} \qquad \frac{}{\Gamma; \Delta \vdash \langle\rangle : I}$$

$$\frac{\Gamma; \Delta \vdash t : A \qquad \Gamma; \Delta \vdash t' : B}{\Gamma; \Delta \vdash (t, t') : A \otimes B} \qquad \frac{\Gamma; \Delta \vdash t : A_1 \otimes A_2}{\Gamma; \Delta \vdash \pi_i\, t : A_i}$$

$$\frac{\Gamma; \cdot \vdash t : A}{\Gamma \vdash \mathsf{val}\, t : \mathsf{val}\, A} \qquad \frac{\Gamma \vdash e : \mathsf{val}\, A}{\Gamma; \Delta \vdash \mathsf{start}(e) : A} \qquad \frac{\Gamma \vdash e : X}{\Gamma; \Delta \vdash e^\omega : X^\omega}$$

$$\frac{\Gamma; \Delta \vdash t : X^\omega \qquad \Gamma, x : X; \Delta \vdash t' : A}{\Gamma; \Delta \vdash \mathsf{let}\, x^\omega = t\, \mathsf{in}\, t' : A}$$

Figure 2. Syntax for Adjoint Logic

ally the key property which lets us switch our view of a program between the synchronous and the functional reactive views.

For example, under the adjoint view, it becomes easy to resolve the puzzle at the end of the last subsection: the feedback operator is just the fixed point operator in the category of ultrametric spaces, with type $(X^\omega \multimapdot X^\omega) \multimap X^\omega$. Once we have a stream in hand, we can give it to a synchronous program to start computing with whenever we want.

4. A Domain Specific Language

We now give a small domain-specific language corresponding to the semantics of the previous section. As in adjoint logic, we have two sorts of types, writing A, B, C for the types of the lambda calculus interpreted in the base category of ultrametric spaces, and X, Y, Z for the types interpreted in the co-Kleisli category. In addition to the standard function, product and adjoint types, we also introduce types corresponding to *contractive functions*, $X \rightsquigarrow Y$ and $A \multimapdot B$.

There are two judgement forms, $\Gamma \vdash e : Y$ and $\Gamma; \Delta \vdash t : B$, where $\Gamma = x_1 : X_1, x_n : X_N$ and $\Delta = y_1 : A_a, \ldots, y_m : A_n$. We then interpret the first judgement as a map in $\mathbb{U}^S$, of type $X_1 \times \ldots \times X_n \to Y$, and the second judgement as a map in $\mathbb{U}$, of type $(X_1 \times \ldots \times X_n)^\omega \otimes A_1 \otimes \ldots \otimes A_m \to B$. Here, t ranges over "FRP programs" interpreted in the base category of ultrametric spaces, and e ranges over "synchronous dataflow programs" interpreted in the co-Kleisli category. Note the asymmetry in the judgments – the dataflow context appears in the FRP judgement, but not vice-versa.

We give typing rules for these programs in Figure 2. Most of the rules are routine, with most of the interest lying in the rules that permit passing between the two worlds, listed as the last four rules of Figure 2. The val-introduction rule takes an FRP expression with no free non-synchronous variables, and lifts it to a synchronous program in $\mathbb{U}^S$. The corresponding elimination rule says that within an FRP program, we can take a synchronous term $e : \mathsf{val}\ A$, and ask for its current value with $\mathsf{start}(e)$. That is, viewing the $\mathsf{val}\ A$ term as something lifted to a stream, this operation takes the head

$$\frac{\Gamma \vdash e : X}{\Gamma; \hat\Gamma \vdash \langle e\rangle : X} \qquad \frac{\Gamma; \hat\Gamma \vdash c : X \rightsquigarrow Y \qquad \Gamma, \hat\Gamma \vdash e : X}{\Gamma; \hat\Gamma \vdash c\, e : Y}$$

$$\frac{\Gamma; \hat\Gamma \vdash c : X \qquad \Gamma; \hat\Gamma \vdash c' : Y}{\Gamma; \hat\Gamma \vdash (c, c') : X \times Y} \qquad \frac{\Gamma; \hat\Gamma \vdash c : X_1 \times X_2}{\Gamma; \hat\Gamma \vdash \pi_i\, c : X_i}$$

$$\frac{}{\Gamma; \hat\Gamma \vdash () : 1} \qquad \frac{\Gamma; \hat\Gamma \vdash c : X \qquad \Gamma, x : X; \hat\Gamma \vdash c' : Y}{\Gamma; \hat\Gamma \vdash \mathsf{letc}\, x = c\, \mathsf{in}\, c' : Y}$$

$$\frac{\Gamma, x : X; \hat\Gamma \vdash c : b}{\Gamma; \hat\Gamma \vdash \lambda x : X.\, c : X \Rightarrow Y} \qquad \frac{\Gamma; \hat\Gamma \vdash c : X \Rightarrow Y \qquad \Gamma; \hat\Gamma \vdash c' : X}{\Gamma; \hat\Gamma \vdash c\, c' : Y}$$

$$\frac{\Gamma; \hat\Gamma, x : X \vdash c : Y}{\Gamma; \hat\Gamma \vdash \hat\lambda x : X.\, c : X \rightsquigarrow Y}$$

$$\frac{\Gamma; \Delta \vdash t : A}{\Gamma; \Delta; \hat\Delta \vdash \langle t\rangle : A} \qquad \frac{\Gamma; \Delta; \hat\Delta \vdash d : A \multimapdot B \qquad \Gamma; \Delta, \hat\Delta \vdash t : A}{\Gamma; \Delta; \hat\Delta \vdash d\, t : B}$$

$$\frac{\Gamma; \Delta; \hat\Delta \vdash d : A \qquad \Gamma; \Delta; \hat\Delta \vdash d' : B}{\Gamma; \Delta; \hat\Delta \vdash (d, d') : A \otimes B} \qquad \frac{\Gamma; \Delta; \hat\Delta \vdash d : A_1 \otimes A_2}{\Gamma; \Delta; \hat\Delta \vdash \pi_i\, d : A_i}$$

$$\frac{}{\Gamma; \Delta; \hat\Delta \vdash () : I} \qquad \frac{\Gamma; \Delta; \hat\Delta \vdash d : A \qquad \Gamma; \Delta, x : A; \hat\Delta \vdash d' : B}{\Gamma; \Delta; \hat\Delta \vdash \mathsf{letc}\, x = d\, \mathsf{in}\, d' : B}$$

$$\frac{\Gamma; \Delta, x : A; \hat\Delta \vdash d : b}{\Gamma; \Delta; \hat\Delta \vdash \lambda x : A.\, d : A \multimap B} \qquad \frac{\Gamma; \Delta; \hat\Delta, x : A \vdash d : B}{\Gamma; \Delta; \hat\Delta \vdash \hat\lambda x : A.\, d : A \multimapdot B}$$

Figure 3. Syntax for Contractive Terms

— which is precisely the operation difficult to interpret in a purely synchronous way. Conversely, the two final rule says how FRP programs can define embedded stream values and then bind them for use as synchronous dataflow variables.

In addition to these two judgements, we also need a pair of judgements permitting programmers to define contractive functions, which we describe in Figure 3. The first is $\Gamma; \hat\Gamma \vdash c : X$, which asserts that c is a term which is strictly contractive in the variables in $\hat\Gamma$. (It can be merely nonexpansive in the variables in Γ.) Likewise, we have a judgement $\Gamma; \Delta; \hat\Delta \vdash d : A$, with d strictly contractive in the variables in $\hat\Delta$ and nonexpansive in the others. The interpretation of $\Gamma; \hat\Gamma \vdash c : X$ is a morphism with type $\Gamma \to (\hat\Gamma \rightsquigarrow X)$, and the interpretation of $\Gamma; \Delta; \hat\Delta \vdash d : X$ is a morphism of type $\Gamma^\omega \otimes \Delta \to (\hat\Delta \multimapdot A)$.

At first glance, these judgements look similar to the usual rules for introducing and eliminating functions, pairs, and so on. Upon second glance, they look extremely peculiar! We introduce a second (or third) context of variables, into which we move the binders for contractive functions. However, we do so *without* giving a corresponding variable rule that permits directly using these hypothesis. Instead, there are some rules (such as the application rule for contractive functions) that permit moving the variables from the new context into the old. From a semantic point of view, this is only to be expected — the interpretation of the variable rule in categorical proof theory is an identity morphism, and identities *are not contractive*. So we cannot expect to have a normal variable rule for the hypotheses corresponding to arguments of contractive functions. Operationally, this embodies our need to ensure that the variables are only used as arguments to our primitive contractive functions (such as `cons`), which will ensure that the terms they appear in are always guarded.

To define the semantics, we need some maps in the category of ultrametrics which witness the fact that contractiveness is preserved by almost everything. The implementations of these functions are the evident ones given the types, since contractive functions are just a subset of the nonexpansive ones. (Also, we elide the essentially identical semantics of contractive terms in the co-Kleisli category.)

5. Implementation Language and Dataflow Library

Implementation Language. The programming language in which we implement our domain-specific language is a polymorphic 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 $\mathsf{ref}\ \tau$, 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 syntax, typing, and semantics of the implementation language are all standard, and we omit them for reasons of space. The construction of fixed points in the implementation language is restricted to *pointed* types, which are the monadic types, products of pointed types and function types whose codomain is pointed. The other monadic primitives are $\mathtt{new}(e)$, $!e$, and $e := e'$, which allocate, read and write references (inhabiting type $\mathsf{ref}\ \tau$), respectively.

Program Logic. We reason about programs in the implementation language in the program logic whose syntax is shown in Figure 6. 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 conjunction, disjunction and implications, as well as quantifiers ranging over the sorts in ω. This 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 [2]. In addition to the usual connectives of Hoare logic, we add *spatial* connectives to talk about shared state. The separating conjunction $p * q$ is satisfied by states can be split into two *disjoint* parts, one of which satisfies p, and the other of which satisfies q. The disjointness property makes the noninterference of p and q implicit, simplifying specifications greatly. emp, which is true only of the empty heap, is the unit of $*$. The points-to relation $e \mapsto e'$, holds of the one-element heap in which the value of the reference e has contents equal to the value of e'.

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 A, 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.

Dataflow Library. We implement our DSL on top of an imperative dataflow network, which is rather like a generalized spreadsheet. There is a collection of cells, each of which contains 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.

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

Figure 6. Specification Language

We will compile a synchronous dataflow program into a dataflow graph, which is run 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 give the interface to a dataflow library in Figure 7. We have given a correctness proof of this library in prior work [14], but will describe the specification here in detail, since we use it as a component of the present work.

The interface features two abstract data types, cell and code. We actually expose the implementations in the figure, to better discuss them. $\mathsf{cell}\ \alpha$ is the type of cells that compute a value of type α. It is implemented as a record with four fields. The $code$ field contains an expression that will compute both a value of type α and a set of cells that were read in the process. The $(value)$ field is used for memoizing the computed value. $(reads)$ tracks which cells this one has read, whilst (obs) records which cells are observing this one. When the cell is read it returns its memoized value if it has one; otherwise it runs its stored code to compute a value, updates its memo field, transitively invalidates its observers and registers itself as an observer of those cells it read during evaluation.

The $\mathsf{code}\ \alpha$ type is a user-defined monadic type, as is commonly defined in Haskell. It has the responsibility of both computing a value of type α, as well as returning a set of all the cells that it read in the process of computing its return value. The `bind` and `return` operations are the unit and extension operations of the user-level monad, and the operations it supports are given in lines 12-16. There is an operation to `read` a cell, an operation `cell` to create a cell, and `getref`, `setref`, and `newref` operations to create local state in the dataflow network. On line 17, there is also an operation to update a cell, but it does not live within the code monad.

As each cell tracks both who it reads and who observes it, it may seem that there is a global invariant on the whole dataflow graph, which would make local reasoning difficult. This is true, but it is possible to work around this difficulty. The key idea is to introduce an *domain-specific separation logic* tuned to the needs of proving dataflow programs correct. That is, we introduce a predicate $H(\theta)$ describing the global invariant of the whole dataflow graph, but index it by spatial formulas which describe the dataflow graph in a *local* way. Then, we can use these formulas to specify the operations of the library.

We give the syntax of formulas in Figure 8, above the specifications of the library operations. I and $\phi \otimes \psi$ correspond to the emp and separating conjunction of separation logic, denoting empty graphs and two disjoint collections of cells. However, in addition to the $\mathsf{ref}(\mathtt{r}, \mathtt{v})$ predicate (which corresponds to points-to in separation logic), we include a pair of predicates describing cells. The predicate $\mathsf{cell}^{-}(\mathtt{c}, e)$ means that $\mathtt{c}$ is a cell in the dataflow graph containing code e, and that it is not ready — i.e., it needs to be

$$
\begin{array}{lcl}
[\![\Gamma \vdash x_i : X_i]\!] & = & \pi_i \\
[\![\Gamma \vdash e\ e' : Y]\!] & = & ([\![\Gamma \vdash e : X \Rightarrow Y]\!], [\![\Gamma \vdash e' : X]\!]); \mathit{eval} \\
[\![\Gamma \vdash \lambda x.\ e : X \Rightarrow Y]\!] & = & \lambda([\![\Gamma, x : X \vdash e : Y]\!]) \\
[\![\Gamma \vdash () : 1]\!] & = & 1_\Gamma \\
[\![\Gamma \vdash (e, e') : A \times B]\!] & = & ([\![\Gamma \vdash e : A]\!], [\![\Gamma \vdash e' : B]\!]) \\
[\![\Gamma \vdash \pi_i(e) : A_i]\!] & = & [\![\Gamma \vdash e : A_1 \times A_2]\!]; \pi_i \\
[\![\Gamma \vdash \mathsf{val}\ t : \mathsf{val}\ A]\!] & = & \eta_\Gamma; V([\![\Gamma; \cdot \vdash t : A]\!])
\end{array}
$$

$$
\begin{array}{lcl}
[\![\Gamma; \Delta \vdash x_i : A_i]\!] & = & \pi_2; \pi_i \\
[\![\Gamma; \Delta \vdash t\ t' : B]\!] & = & ([\![\Gamma; \Delta \vdash e : A \multimap B]\!], \\
 & & \quad [\![\Gamma; \Delta \vdash e' : A]\!]); \mathit{eval} \\
[\![\Gamma; \Delta \vdash \lambda x.\ t : A \multimap B]\!] & = & \lambda([\![\Gamma; \Delta, x : A \vdash t : B]\!]) \\
[\![\Gamma; \Delta \vdash () : I]\!] & = & 1_\Delta \\
[\![\Gamma; \Delta \vdash (t, t') : A \otimes B]\!] & = & ([\![\Gamma; \Delta \vdash t : A]\!], [\![\Gamma; \Delta \vdash t' : B]\!]) \\
[\![\Gamma; \Delta \vdash \pi_i(t) : A_i]\!] & = & [\![\Gamma; \Delta \vdash t : A_1 \otimes A_2]\!]; \pi_i \\
[\![\Gamma; \Delta \vdash t^\omega : X^\omega]\!] & = & \pi_1; ([\![\Gamma \vdash t : X]\!])^\omega \\
[\![\Gamma; \Delta \vdash \mathsf{let}\ x^\omega = t\ \mathsf{in}\ t' : A]\!] & = & ((\pi_1, [\![\Gamma; \Delta \vdash t : X^\omega]\!]); n, \pi_2); \\
 & & [\![\Gamma, x : X; \Delta \vdash t' : A]\!] \\
[\![\Gamma; \Delta \vdash \mathsf{start}(t) : A]\!] & = & \pi_1; ([\![\Gamma \vdash t : \mathsf{val}\ A]\!])^\omega; \varepsilon_A
\end{array}
$$

Figure 4. Semantics of Adjoint Language

$$
\begin{array}{l}
\mathit{sweak} : X \to (Y \rightsquigarrow Z) \\
\mathit{spair} : (X \rightsquigarrow Y) \times (X \rightsquigarrow Z) \to (X \rightsquigarrow Y \times Z) \\
\mathit{scurry} : (X \times Y \rightsquigarrow Z) \to (X \rightsquigarrow Y \rightsquigarrow Z) \\
\mathit{seval} : (X \rightsquigarrow Y \rightsquigarrow Z) \times (X \Rightarrow Y) \to (X \rightsquigarrow Z) \\
\mathit{swap} : (X \Rightarrow Y \rightsquigarrow Z) \simeq (Y \rightsquigarrow X \Rightarrow Z) \\
\mathit{scomposer}(f : Y \to Z) : (X \rightsquigarrow Y) \to (X \rightsquigarrow Z)
\end{array}
$$

$$
\begin{array}{lcl}
[\![\Gamma; \hat\Gamma \vdash \langle e \rangle : X]\!] & = & [\![\Gamma \vdash e : X]\!]; \mathit{sweak} \\
[\![\Gamma; \hat\Gamma \vdash c\ e : Y]\!] & = & ([\![\Gamma; \hat\Gamma \vdash c : X \rightsquigarrow Y]\!], \lambda^{\hat\Gamma}([\![\Gamma, \hat\Gamma \vdash e : X]\!])); \mathit{seval} \\
[\![\Gamma; \hat\Gamma \vdash \pi_i(c) : X_i]\!] & = & [\![\Gamma; \hat\Gamma \vdash c : X_1 \times X_2]\!]; \mathit{scomposer}(\pi_i) \\
[\![\Gamma; \hat\Gamma \vdash () : 1]\!] & = & 1_\Gamma; \mathit{sweak}_{\hat\Gamma} \\
[\![\Gamma; \hat\Gamma \vdash \mathsf{letc}\ x = c\ \mathsf{in}\ c' : Y]\!] & = & (\lambda([\![\Gamma, x : X; \hat\Gamma \vdash c' : Y]\!]); \mathit{swap}, [\![\Gamma; \hat\Gamma \vdash c : X]\!]); \\
 & & \mathit{spair}; \mathit{scomposer}(\mathit{eval}) \\
[\![\Gamma; \hat\Gamma \vdash \lambda x : X.\ c : X \Rightarrow Y]\!] & = & \lambda([\![\Gamma, x : X; \hat\Gamma \vdash c : Y]\!]); \mathit{swap} \\
[\![\Gamma; \hat\Gamma \vdash \lambda x : X.\ c : X \rightsquigarrow Y]\!] & = & [\![\Gamma; \hat\Gamma, x : X \vdash c : Y]\!]; \mathit{scurry} \\
[\![\Gamma; \hat\Gamma \vdash (c, c') : X \times Y]\!] & = & ([\![\Gamma; \hat\Gamma \vdash c : X]\!], [\![\Gamma; \hat\Gamma \vdash c' : Y]\!]); \mathit{spair}
\end{array}
$$

Figure 5. Semantics of Contractive Terms

evaluated before producing a value. The predicate $\mathsf{cell}^+(\mathtt{c}, e, \mathtt{v}, rs)$ 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. Since establishing this can require us to follow paths in the heap, we introduce two inductively-defined relations $\mathsf{unready}(\theta, \mathtt{c})$, and $\mathsf{ready}(\theta, \mathtt{c}, \mathtt{v})$. These are defined in the obvious way on the syntax of formulas θ, and 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.

Now, we can explain the specifications of the code expressions in Figure 8. First, all of these specifications are parameterized by an extra quantifier $\forall\psi. \ldots$, which lets us manually build in a kind of frame rule into this specification — any formula we can derive will also be quantified, and hence work 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 line 1, `return` leaves its frame untouched, and returns its value v without reading any cells. On lines 2-5, we have a specification for `bind`. It looks complicated, but is actually very straightforward — the specification of `bind e f` is that it takes its state from θ to θ'', assuming that e takes θ to θ', and f (with the return value of e) takes θ' to θ''. The operations `newref`, `getref`, `setref` have the obvious actions on local references, and `cell` simply allocates a new cell, leaving it in an unready state. On lines 10-11, we have a specification for `read`, in the case that its argument is ready. Finally, on lines 12-16, we have the rule that explains what happens when the cell is unready — we need to know what the code in the cell does, and we also need to know that this code does not modify the current cell itself. If so, then the heap is updated to reflect *both* the action of the code, and the effect of setting the cell to a positive state.

1 $\mathsf{code} : \star \to \star$
2 $\mathsf{code}\ \alpha = \bigcirc(\alpha \times \mathsf{set}(\mathsf{ecell}))$

3 $\mathsf{cell} : \star \to \star$
4 $\mathsf{cell}\ \alpha = \{\mathit{code} : \mathsf{ref\ code}\ \alpha;$
5 $\quad \mathit{value} : \mathsf{ref\ option}\ \alpha;$
6 $\quad \mathit{reads} : \mathsf{ref\ set}(\mathsf{ecell}));$
7 $\quad \mathit{obs} : \mathsf{ref\ set}(\mathsf{ecell}));$
8 $\quad \mathit{unique} : \mathbb{N}\}$

9 $\mathsf{ecell} = \exists\alpha : \star.\ \mathsf{cell}\ \alpha$

10 $\mathtt{return} : \forall\alpha : \star.\ \alpha \to \mathsf{code}\ \alpha$
11 $\mathtt{bind} : \forall\alpha, \beta : \star.\ \mathsf{code}\ \alpha \to (\alpha \to \mathsf{code}\ \beta) \to \mathsf{code}\ \beta$
12 $\mathtt{read} : \forall\alpha : \star.\ \mathsf{cell}\ \alpha \to \mathsf{code}\ \alpha$
13 $\mathtt{cell} : \forall\alpha : \star.\ \mathsf{code}\ \alpha \to \mathsf{code\ cell}\ \alpha$
14 $\mathtt{newref} : \forall\alpha : \alpha \to \mathsf{code\ ref}\ \alpha$
15 $\mathtt{getref} : \forall\alpha : \mathsf{ref}\ \alpha \to \mathsf{code}\ \alpha$
16 $\mathtt{setref} : \forall\alpha : \mathsf{ref}\ \alpha \to \alpha \to \mathsf{code\ unit}$

17 $\mathtt{update} : \forall\alpha : \star.\ \mathsf{code}\ \alpha \to \mathsf{cell}\ \alpha \to \bigcirc\mathsf{unit}$

Figure 7. Implementation of Notification Networks

6. The Implementation

We have two cartesian closed categories in play, each of which is represented a bit differently. Below, we give the interpretation of the types into our functional language.

$$
\begin{array}{lcl}
(\!|I|\!)_u & = & \mathsf{unit} \\
(\!|A \otimes B|\!)_u & = & (\!|A|\!)_u \star (\!|B|\!)_u \\
(\!|X^\omega|\!)_u & = & \mathsf{stream}\ (\!|X|\!)_s \\
(\!|A \multimap B|\!)_u & = & \mathsf{code\ option}\ (\!|A|\!)_u \to \mathsf{code\ option}\ (\!|B|\!)_u \\
(\!|A \multimapdot B|\!)_u & = & \mathsf{code\ option}\ (\!|A|\!)_u \to \mathsf{code\ option}\ (\!|B|\!)_u \\
(\!|1|\!)_s & = & \mathsf{unit} \\
(\!|X \times Y|\!)_s & = & (\!|X|\!)_s \star (\!|X|\!)_s \\
(\!|X \to Y|\!)_s & = & \mathsf{stream}\ (\!|X|\!)_s \to \mathsf{code\ stream}\ (\!|Y|\!)_s \\
(\!|X \rightsquigarrow Y|\!)_s & = & \mathsf{stream}\ (\!|X|\!)_s \to \mathsf{code\ stream}\ (\!|Y|\!)_s \\
(\!|\mathsf{val}\ A|\!)_s & = & (\!|X|\!)_u \\
\mathsf{stream}\ \tau & = & \mathsf{cell\ option}\ \tau
\end{array}
$$

$$\phi, \psi, \theta \ ::= \ I \ | \ \phi \otimes \psi \ | \ \mathsf{cell}^{+}(\mathtt{c}, e, v, rs) \ | \ \mathsf{cell}^{-}(\mathtt{c}, e) \ | \ \mathsf{ref}(\mathtt{r}, \mathtt{v})$$

$$\forall \psi.\ \{H(\psi)\}\, \mathtt{return}(\mathtt{v})\, \{(\mathtt{a}, \emptyset).\ H(\psi) \wedge \mathtt{a} = \mathtt{v}\}$$

$$\begin{aligned}
&\forall \psi.\ \{H(\theta \otimes \psi)\}\, \mathtt{e}\, \{(\mathtt{a}, r).\ H(\theta' \otimes \Re(u, \psi)) \wedge \mathtt{a} = \mathtt{v} \wedge r = r_1\} \text{ and} \\
&\forall \psi.\ \{H(\theta' \otimes \psi)\}\, \mathtt{f}\ \mathtt{v}\, \{(\mathtt{a}, r).\ H(\theta'' \otimes \Re(u', \psi)) \wedge \mathtt{a} = \mathtt{v}' \wedge r = r_2\} \\
&\Longrightarrow \forall \psi.\ \{H(\theta \otimes \psi)\}\, \mathtt{bind}\ \mathtt{e}\ \mathtt{f}\{(\mathtt{a}, r).\ H(\theta'' \otimes \Re(u \cup u', \psi)) \\
&\qquad\qquad \wedge\ \mathtt{a} = \mathtt{v}' \wedge r = r_1 \cup r_2\}
\end{aligned}$$

$$\forall \psi. \{H(\psi)\}\, \mathtt{newref}(\mathtt{v})\, \{(\mathtt{a}, \emptyset).\ H(\psi \otimes \mathsf{ref}(\mathtt{a}, \mathtt{v}))\}$$

$$\forall \psi. \{H(\mathsf{ref}(\mathtt{r}, \mathtt{v}) \otimes \psi)\}\, \mathtt{getref}(\mathtt{r})\, \{(\mathtt{a}, \emptyset).\ H(\mathsf{ref}(\mathtt{r}, \mathtt{v}) \otimes \psi) \wedge \mathtt{a} = \mathtt{v}\}$$

$$\forall \psi. \{H(\mathsf{ref}(\mathtt{r}, -) \otimes \psi)\}\, \mathtt{setref}(\mathtt{r}, \mathtt{v})\, \{(\mathtt{a}, \emptyset).\ H(\mathsf{ref}(\mathtt{r}, \mathtt{v}) \otimes \psi)\}$$

$$\forall \psi. \{H(\psi)\}\, \mathtt{cell}(\mathtt{code})\, \big\{(\mathtt{a}, \emptyset).\ H(\mathsf{cell}^{-}(\mathtt{a}, \mathtt{code}) \otimes \psi)\big\}$$

$$\begin{aligned}
&\mathsf{ready}(\theta, \mathtt{c}, \mathtt{v}) \Longrightarrow \forall \psi. \{H(\theta \otimes \psi)\}\, \mathtt{read}(\mathtt{c})\{(\mathtt{a}, r).\ H(\theta \otimes \psi) \\
&\qquad\qquad \wedge\ r = \{\mathtt{c}\} \wedge \mathtt{a} = \mathtt{v}\}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{unready}(\theta, \mathtt{c}) \text{ and } \mathsf{unready}(\theta', \mathtt{c}) \text{ and } \mathsf{code}(\theta, \mathtt{c}, \mathtt{code}) \text{ and} \\
&\forall \psi. \{H(\theta \otimes \psi)\}\, \mathtt{code}\, \{(\mathtt{a}, r).\ H(\theta' \otimes \Re(u, \psi)) \wedge \mathtt{a} = \mathtt{v} \wedge r = rs\} \\
&\Longrightarrow \forall \psi. \{H(\theta \otimes \psi)\} \\
&\qquad\quad \mathtt{read}(\mathtt{c}) \\
&\qquad\quad \{(\mathtt{a}, \{\mathtt{c}\}).\ H([\Re(\{c\}, \theta') | \mathsf{cell}^{+}(\mathtt{c}, \mathtt{code}, \mathtt{v}, rs)]) \wedge \mathtt{a} = \mathtt{v}\}
\end{aligned}$$

Figure 8. Library Specification

Units and products of the ultrametric world can be represented directly using ML types. The first interesting case is at A^{ω}, where we give the representation of streams of values of type A. This clause is interpreted as $(\!|A|\!)_s$, which can be understood as follows. A stream is a cell, which yields values by accessing and modifying the dataflow graph each time it is read. This means we must be very careful about the operations we permit on this type, because otherwise we will break the illusion that this is a pure, time-independent value. Furthermore, we do not realize streams of type τ with a cell τ. Instead, we use a cell option τ. The reason for this decision is that we want to implement recursion via feedback, and so we need to be able to have cells which are unitialized on their first time step.

This drives the lazy representation of the function space $A \multimap B$, as a function from A-computations to computations of B-values. For functions such as head, with type $(\mathsf{val}\ A)^{\omega} \to A$, we need to look at the first element of a stream cell, and so we need to perform a code computation to access it. However, since we implement feedback with unit delay: (1) this access can fail on the first try, so the computation must return an option; and (2) since a contractive operation may do something else on the first timestep before using its input, we need to allow the input to be lazy, to let us use it later.

The interpretation of the co-Kleisli category is one in which time plays a simpler role. The maps here are instantaneous functions of streams, which we implement it via a dataflow graph which we update to get to the next time step. In our implementation, the argument to a function $f : X \Rightarrow Y$ comes as a stream, and the return value is term of type code stream X. The code constructor permits the implementation to read and extend the dataflow graph, doing some initialization to return a stream cell. Surpisingly, this result is a stream, and not a point value, the way that the intepretation of morphisms in the co-Kleisli category might initially suggest. In fact, our implementations actually realize the coextensions of the morphisms of the co-Kleisli category, and the logical relation needs to be adjusted in this clause to make it fit.

Selected parts of this implementation can be seen in Figures 9 through 12. We suppress many of the definitions for products and exponentials, since they are the same as in our programming language, modulo monadic sequencing. (For readability, we use Haskell-style do-notation to write the terms our code type.)

However, the implementations of `cons` and `fix` are two of the most complex functions in the whole library. The `cons` function takes a thunk `xt`, and then returns a contractive function which will appends `xt`'s value to the front of any input stream computation it receives. The complexity arises from the fact that the stream expression may not yield a value on the first tick (since its evaluation may attempt to read an unitialized stream), and even if it does yield a stream, that stream may itself have a delay. So we need to keep evaluating the stream expression until we get a stream, and then use it to get values. Furthermore, if it gets an unitialized stream, `cons` should simply replace the first None element with `x`, and if it is initialized, it should buffer its input for the next time step. The trick is to use two reference cells – one to memoize the stream expression, and the other as a one-place buffer for the stream. Note the call to the `register` auxilliary function, which records a list of all cells that read or write local state in the variable `i`, so that the event loop can ensure that all stateful operations are performed every time step so that these cells do not "lose" ticks.

The `fix` operator implements recursion via feedback. It takes a stream of contractive dataflow functions, and returns a stream realizing the fixed point of the first element of this stream. To do this, we allocate a reference `r` (initially set to None), which the cell `input` reads to produce its values. Then, we call `f` with the input, and then construct a cell `output` which takes the return values and writes them to `r` in order to prepare `input` for the next time step.

The implementation of (part of) the co-Kleisli category, given in Figure 10 is very straightforward. Each operation simply takes in a stream cell, and builds a cell to return its return value. This is because all the difficulties have been encapsulated into a single function: the `zip` operation (defined in Figure 11). Given two stream cells, it returns a cell which pairs the successive elements of its two inputs. This is a tricky function to verify, since the function must work with lagged inputs, and we may receive a pair of inputs in which one component is lagged and the other not. However, our specification does not mention delays at all. So `zip` must pair up the n-th elements irrespective of the possibly-differing delays of its inputs.

To implement this, `zip` tests the two inputs, and introduces an artificial delay, if one cell is delayed and the other is not. Otherwise, it simply returns a cell which performs the pairing. Since implementing a delay uses auxilliary state, we need to register the cell — but we only `register` the cell in the case it needs the state. This reduces the number of cells that will get forced at the end of each trip through the event loop, and so lets the dataflow graph remain lazier. Finally, in Figure 12, we give the implementation of the functorial actions of the two adjoint functors, which are straightforward.

For space reasons, we have suppressed most of the definitions of the combinators of the contractive operations. However, these are all the same as the implementation of the ordinary functions — we simply use them in restricted contexts.

7. The Specification

In Figures 13 and 14, we give three mutually-dependent relations, one for ultrametric values, one for co-Kleisli values, and one for the memory state of the dataflow graph.

A memory state is a pair (θ, R), where θ is a formula describing a dataflow network, and R is a *rely* describing the stream of semantic values each cell in the network is expected to produce. To relate these two, we have a satisfaction relation $(\theta, R)\ \mathrm{sat}\ d$, which can be read as saying, roughly, that θ implements the set of streams in R to distance d. This satisfaction relation is the most complicated part of the definition, since we have to account for all of the issues discussed in the previous section. Our satisfaction relation is given in Figure 14. We specify a rely R as 9-tuple:

```
id = λx. x
compose f g = λx. g(f(x))

cons : A → (val A)^ω ⊸ (val A)^ω
cons = λxt. do x′ ← xt; ofold (return None) f x′
with f x = return(Some(g x))
and g x ys = do r ← newref(Some(x));
                xsr ← newref(None);
                zs ← cell(do old ← getref(r);
                             xs′ ← do v ← getref(xsr);
                                      case v of
                                       Some(xs) → return(v)
                                       None → do xs′ ← xst;
                                                 setref(xsr, xs′);
                                                 return(xs′)
                             new ← do xs′ ←!xsr;
                                      ofold (return None) read xs
                             case old of
                              None → return(new)
                              Some(_) → do setref(r, new);
                                           return(old));
                register(zs);
                return(Some(zs))

fix : (A^ω ⊸ A^ω) ⊸ A^ω
fix ft = do f′ ← ft; ofold (return None) fix′ f
fix′ = λf. do r ← newref(None);
              input ← cell(do _ ← read(clock);
                              v ← getref(r);
                              return(v));
              pre ← f(return(Someinput));
              out ← cell(do _ ← read(clock);
                            _ ← read(input);
                            v ← read(pre);
                            setref(r, v);
                            return(v));
              register(out);
              return(Some(out))

ofold none some None      = none
ofold none some Some(x)   = some(x)

ozip Some(x) Some(y) = Some((x, y))
ozip _       _       = None
```

Figure 9. The Implementation of the Ultrametric Category

```
id = λxs. cell(read xs)
compose f g = λas. do bs ← f(as); cs ← g(bs); return(cs)
one xs = cell(return(Some(⟨⟩)))
pair f g = λas. do bs ← f(as); cs ← g(as); zip(bs, cs)
fst = λabs. cell(do ab′ ← read(abs);
                    case ab′ of
                     None → return(None)
                     Some(a, b) → return(Some(a)))
snd = λabs. cell(do ab′ ← read(abs);
                    case ab′ of
                     None → return(None)
                     Some(a, b) → return(Some(b)))
eval = λfas. do fs ← fst(fas);
                as ← snd(fas);
                cell(do f′ ← read(fs)
                        case f′ of
                         None → return None
                         Some(f) → do bs ← f(as);
                                      read(bs))
curry f = λas. cell(Some(λbs. do abs ← zip(as, bs); f(abs)))
```

Figure 10. The Implementation of the co-Kleisli Category

```
zip(as, bs) =
 do a′ ← read(as);
    b′ ← read(bs);
    case (a′, b′) of
     (None, None)
     (Some(_), Some(_)) →
         cell(do a′ ← read(as);
                 b′ ← read(bs);
                 case (a′, b′) of
                  (Some(a), Some(b)) → return(Some((a, b)))
                  (_, _) → return(None))
     (None, Some(_)) →
         do r ← newref(None);
            abs ← cell(do a ← read(as);
                          new ← read(bs);
                          old ← getref(r);
                          setref(r, new);
                          return(Some(a, old)));
            register(abs);
            return(abs)
     (Some(_), None) →
         do r ← newref(None);
            abs ← cell(do b ← read(bs);
                          new ← read(as);
                          old ← getref(r);
                          setref(r, new);
                          return(Some(old, b)));
            register(abs);
            return(abs)

register(xs) =
 do dummy ← read(xs); lst ← getref(i); setref(i, pack(xs) :: lst)
```

Figure 11. Utility Functions

```
omega f = λat. cell(do as′ ← at
                       case as′ of
                        None → return(None)
                        Some(as) → do bs ← f(as);
                                      return(Some(bs))

value f = λxs. cell(f(read(xs)))

varepsilon xs = read(xs)

eta xs = cell(do x′ ← xs
                 case x′ of
                  None → return(None)
                  Some(a) → return(Some(xs)))
```

Figure 12. Implementing the Adjunction

1. A finite set of stream cells and metric types C.
2. A set of reference cells L.
3. A map giving stream values $V_S : C \to \mathit{Value}^\omega$ to each cell.[2]
4. A "delay flag" $D : C \to \mathbb{D}$ which says for each cell whether its output is delayed d or whether it is undelayed u. (We also order delays so that u $\sqsubseteq$ d and give it the evident lattice structure.)
5. An assignment of a stream of values $V^L : L \to (1 + \mathit{Value})^\omega$ for each local reference. Note in particular that our local state is given as a stream of *options*: this is because we might want to use some state for "only a little while". (The cons function does this when it is applied to a lagged cell. It will use the value

[2] We should write this as a dependent product, with an element of C having type $\Sigma A : \text{type}.\ \text{cell option}\ (\!|A|\!)_s$, and V_S having the type $V_S : \Pi(A, \_) \in C.\ A^\omega$. However, we will suppress these dependencies to reduce notational clutter.

in its reference cell on the first time step, and then never store a value in it again.)

6. A "getter" for each reference cell $G : L \rightharpoonup C$. This is the cell which will read that reference cell. This is a *partial* function, so there can be reference cells which are not yet going to be read by anyone. This lets us build up the dataflow graph incrementally, while still remaining within the rely.
7. A "setter" for each reference cell $S : L \rightharpoonup C$. This is the cell which has responsibility for writing the next value of the reference cell. Like the getter G, the getter S is also partial. However, we require that its domain be a superset of G's — that is, we will always define getters before setters.
8. A function $\Delta : C \to \mathcal{P}(C)$, giving the "static dependencies" of each cell. The idea is that for each cell $\mathtt{c}$, then $\mathtt{c}' \in \Delta(\mathtt{c})$ tells us that $\mathtt{c}'$ is a cell in the *current* heap which the evaluation of $\mathtt{c}$ may update. (So reading $\mathtt{c}$ may create new cells it depends on, but since they are not in the current heap they do not appear in $\Delta(\mathtt{c})$.) Viewed as a relation, the reflexive transitive closure of Δ must be a partial order, to ensure that there will be no nontrivial cycles in the dependency graph. We will write $\Delta^*(\mathtt{c})$ to denote the cells reachable via the reflexive transitive closure of Δ, and $\Delta^+(\mathtt{c})$ for the transitive closure.
9. A function $\Delta^L : L \to \mathcal{P}(C)$, which describes the static dependencies of the contents of each reference cell. The reason we need to track the dependencies of reference contents is that cells may read references and use their contents, and so we need to know what the dependencies for each value may be.

When we need to deal with multiple relies, we will name the appropriate component using the name subscripted with the rely. So if R is a rely, then we will write C_R for its cells, V_R for the values of the cells, and so on.

We equip relies with a partial order as follows. We say that $R \sqsubseteq R'$, when $C_R \subseteq C_{R'}$ and $L_R \subseteq L_{R'}$, and furthermore each function is extended pointwise. That is, if $c \in \mathrm{dom}(R)$, then $V_{R'}(c) = V_R(c)$, and similarly for D_S, L, V_L, G, S, and Δ. (Note in particular that the static dependencies for a given cell do not grow — the extension order for Δ is more stringent than simply extension of the partial order.)

Furthermore, for any rely, we can also define its tail $\mathit{tail}(R)$. First, the footprints of the heap are unchanged $C_{\mathit{tail}(R)} = C_R$ and $L_{\mathit{tail}(R)}$. Second, the reference values all go to their tails $V^L_{\mathit{tail}(R)} = \lambda\mathtt{r}.\ \mathit{tail}(V^L_R(\mathtt{r}))$. Third, the stream values for cells go to their tails if they are not delayed, and remain unchanged if they are. $V_{\mathit{tail}(R)} = \lambda\mathtt{c}.\ \mathsf{if}\ D_R(\mathtt{c}) = \mathsf{u}\ \mathsf{then}\ \mathit{tail}(V_R(\mathtt{c}))\ \mathsf{else}\ V_R(\mathtt{c})$. Fourth, the delay flag becomes u for all the cells, $D_{\mathit{tail}(R)} = \lambda\mathtt{c}.\ \mathsf{u}$. Finally, the other components — Δ, Δ^L, S, G — remain unchanged.

Given this, we can explain the satisfaction relation in Figure 14. We say when a graph ϕ *satisfies* a rely R to distance d, (written "(ϕ, R) sat d") when:

- The cells of the graph are C_R plus the clock, and the references of the graph are equal to L_R plus the imperative list $\mathtt{i}$. This is lines 2-3.
- Each cell in the graph is either ready or unready. (Line 4)
- Each stateful ready cell depends upon the $\mathtt{clock}$. (Lines 5-6)
- The update list $\mathtt{i}$ has some of the state-handling cells. (Line 7)
- References realizes the head of their stream if their setter is unready, and the head of their tail stream otherwise. (Line 9)
- Each cell $\mathtt{c}$ in the graph realizes a stream of values corresponding to $V_R(\mathtt{c})$, out to distance d. (Line 8)

Notice that the satisfaction relation *does not require* the contents of references without setters to realize the expected values for those references. Similarly, the clause of the satisfaction relation for the update list $\mathtt{i}$ is imprecise — it only requires a *subset* of the state-accessing cells. This is a deliberate design decision: the reason we make this choice is to let incomplete networks be extensions of complete ones. This lets us use our logical relation to say something the behavior of programs which run in incomplete dataflow networks. This enables us to write programs which evaluate some cells while building another part of a dataflow network.

The way we will reconcile this with our desire to say that closed programs always build complete, closed networks, is to require that every cell and function in the relation must always "make things better". That is, we never admit *values* into our relation which increase the number of references without setters, or which increase the number of stateful references which do not appear in the update list $\mathtt{i}$. Then, since we start a program in a complete dataflow graph, we can only procede to complete graphs.

Having described the satisfaction relation and the extension ordering for relies, we can now describe the extension ordering for memories (i.e., pairs of a state formula and a rely). This is given in Figure 16. On line 1, we introduce the set $\mathit{Mem}(d)$, which are just pairs of formulas and relies in the satisfaction relation. Then, on lines 2-7, we describe what the ordering for valid memories is. First, the relies must lie in the rely extension ordering, and then the formulas must satisfy a number of extra conditions. First, anything ready in the smaller state must remain ready in the larger one. Second, all of the code in the cells must be the same in the smaller memory and its extension. Finally, any reference which does not have a defined writer must be unchanged (be the same physical program value) in the smaller and larger states.

Then, on lines 8-10, we describe the *temporal* ordering of states. The idea is that the Kripke ordering seen so far describes how a dataflow graph can change *during* a time step, and the temporal ordering describes how it changes *upon* a tick of the clock. The idea is that to advance time one step, we update the clock to make it invalid. This propagates a wave of invalidations throughout the dataflow network, leaving it ready to compute the values of the next time step. Then, two memories are in the relation $\mu' \gg^d_n \mu$ when μ' is a state that could lie n steps in the future of μ. What this means is that if $n = 0$, then μ' and μ merely need to be in the Kripke order (line 9). However, if $n = m + 1$, then there needs to be a state μ_0 which (a) larger is in the Kripke order with respect to μ, and whose tail is n steps away in time from μ'. Here, $\mathit{tail}(\theta, R)$ is $(\Re(\mathtt{clock}, \theta), \mathit{tail}(R))$, in accordance with the idea that the event loop updates the clock to propagate this notification out to the rest of the flow graph. Finally, one additional condition we impose on this order is that we only tick on *complete* memories (i.e., ones in which all the references have getters and setters).

Now, we can finally describe the $\mathit{Stream}^d_X(\mathtt{v}, (\theta, R))$ clause on line 8 of Figure 14. It says that for any n less than the log of $1/(2 \times d)$, reading the stream cell should return the n-th value of the stream. (Or, if the stream is delayed, it should return None on the first timestep, and the $n - 1$st value of the stream at subsequent times.) It does this by appeal to the $\mathit{Head}^d_X(\mathtt{v}, \mu)$ predicate, defined on lines 12-18. This predicate says that if we have a heap implementing the dataflow network in θ, then reading it should return the appropriate value — either the head of the stream, or None, depending on whether the stream is lagged or not. Regardless, the updated cells in the network should either be in $\Delta(\mathtt{v})$ or new cells (this is the meaning of the Update predicate, defined on 18-20 of Figure 15). Furthermore, reading $\mathtt{v}$ should change neither the set of references lacking getter or setters, nor the set of cells touching mutable state but not appearing in $\mathtt{i}$ (this is the Stable predicate, defined on lines 21-29 of Figure 15). These predicates will

appear in the postconditions of the Hoare triples in the $UBuild$ and $VBuild$ relations, also — by ensuring that every imperative modification does not increase the number of dangling references, we can conclude that clients will never write programs that break this invariant.

The relation for ultrametric values, $U_A^d(v, \mathtt{v}, \mu, \sigma)$, is given in lines 1-9 of Figure 13. It relates elements v of a metric space A to a concrete program term $\mathtt{v}$. Intuitively, it can be read as saying "the semantic value v is approximated by the computational value $\mathtt{v}$ to at least distance d, when in memory μ and depending on cells σ". The clauses for unit and products are straightforward, and the cases for the two function spaces are only slightly more complicated — we quantify over all future heaps (both extensions of the current in the Kripke ordering, and over the changes induced by the temporal order) before asserting that applying related values to related functions yield computations producing related results. The temporal quantification enforces the requirement that the function be safe to call "at any time". They both use the $UBuild$ relation (defined on lines 1-7 of Figure 15), which encapsulates safely reading and extending the heap. The guts of this relation (and of the $VBuild$ relation used for the other category) resemble the $Stream$ predicate. The main difference between is that running $UBuild$ is permitted to delay returning a value (i.e., return None), if any of the cells σ it may read are delayed. The exception is the return value of contractive functions, which must always produce an undelayed value. (This is what justifies the implementation of the fixed point using a lagged input.)

On line 12, we give the relation for streams X^ω. It just says that a stream value is a cell in the dataflow graph which will produce the appropriate values without delay, whose static dependencies are bounded by σ cells.

Next, the relation $V_X^d(v, \mathtt{v}, \mu, \sigma)$, defined on lines 18-28 of Figure 13, relates values in the co-Kleisli category of streams to the program terms $\mathtt{v}$. The relation $V_A^d(v, \mathtt{v}, \mu, \sigma)$ has the intuitive reading "the semantic value v is approximated by computational value $\mathtt{v}$ and memory state μ to at least distance d. Furthermore, the use of $\mathtt{v}$ may involve evaluating the cells in σ".

On line 18, the other side of the adjunction, $\mathsf{val}\ A$, is interpreted by deferring to the U-relation. Units and pairs (lines 19-20) continue to be interpreted simply, as before. As a convenience prior to defining functions, on line 21 we add a auxilliary clause in this relation for streams, which are simply cells in the dataflow graph which will produce the correct stream of values, and whose static dependencies are less than what the relation asks for.

Then, on lines 22-25, we give the interpretation of the function space. We first quantify over extensions to the distance, the memory, and the dependencies, and then says that if we have a dataflow cell $\mathtt{v}$ realizing some stream vs, we will $VBuild$ an output stream cell realizing $f^\dagger\ vs$. Furthermore, the static dependencies of the result are bounded by the inputs. Finally, it says that the result will be no more delayed than the input is (i.e., if the input is not lagged, then the output won't be, but if the input is lagged, then the output might or might not be). The definition of the contractive function space $A \multimap\!\!\bullet B$ on lines 26-28 is similar, except that it promises that its output will not be lagged, full stop. One noteworthy point is that the dataflow functions here do *not* need to explicitly quantify over all temporally future heaps, the way which the other function spaces do — this is a consequence of our understanding of lifted functions as instananeous.

7.1 Correctness Proof

First, we establish some basic properties of our logical relation.

PROPOSITION 3. *(Kripke Monotonicity) If $d' \geq d, \mu' \sqsupseteq^{d'} \mu, \sigma' \supseteq \sigma'$, we have that*

1. *If $U_X^d(v, \mathtt{v}, \mu, \sigma)$ then $U_X^{d'}(v, \mathtt{v}, \mu', \sigma')$.*
2. *If $V_A^d(v, \mathtt{v}, \mu, \sigma)$ then $V_A^{d'}(v, \mathtt{v}, \mu', \sigma')$.*
3. *If μ sat d then μ sat d'.*

LEMMA 1. *(Approximation Lemma)*

1. *If $\forall d' > d.\ U_A^{d'}(v, \mathtt{v}, \mu, \sigma)$ then $U_A^d(v, \mathtt{v}, \mu, \sigma)$.*
2. *If $\forall d' > d.\ V_A^{d'}(v, \mathtt{v}, \mu, \sigma)$ then $V_A^d(v, \mathtt{v}, \mu, \sigma)$.*
3. *If $\forall d' > d.\ \mu$ sat d' then μ sat d.*

LEMMA 2. *(Induction Lemma)*

1. *If $\forall d' > 2 \cdot d.\ U_A^{d'}(v, \mathtt{v}, \mu, \sigma, \delta) \Rightarrow U_A^{d'/2}(v, \mathtt{v}, \mu, \sigma, \delta)$ then $U_A^d(v, \mathtt{v}, \mu, \sigma, \delta)$.*
2. *If $\forall d' > 2{\cdot}d.\ V_A^{d'}(v, \mathtt{v}, \mu, \sigma) \Rightarrow V_A^{d'}(v, \mathtt{v}, \mu)$ then $V_A^d(v, \mathtt{v}, \mu, \sigma)$.*
3. *If $\forall d' > 2 \cdot d.\ \mu$ sat $d' \Rightarrow \mu$ sat $d'/2$ then μ sat d.*

We often need to extend the memory in our proofs, which requires us to re-establish the satisfaction relation for each cell in the heap with the extended memory. We encapsulate this pattern of reasoning with the following two lemmas. (The notation $[f|x : v]$ denotes extending f's domain by x and letting it be v there.)

LEMMA 3. *(Reference Allocation) Suppose that (θ, R) sat d, and that $\theta' = \theta \otimes \mathsf{ref}(\mathtt{r}, \mathtt{v})$. Let R' be the same as R, except that $L_{R'} = L_R \cup \{\mathtt{r}\}$, $V_{R'} = [V_R|\mathtt{r} : vs]$ for some vs, and $\Delta_{R'}^L = [\Delta_R^L|\mathtt{r} : \kappa]$ for some $\kappa \subseteq C_R$. Then we have that (θ', R') sat d.*

LEMMA 4. *(Cell Allocation)*
Suppose that (θ, R) sat d and $\mathrm{extsat}((\theta, R), d)$ and let $\theta' = \theta \otimes \mathsf{cell}^-(\mathtt{c}, \mathtt{code})$. Further suppose we have $R' \sqsupseteq R$ such that $C_{R'} = C_R \cup \{\mathtt{c}\}$, $L_{R'} = L_R$, $G_{R'} = [G_R|\mathtt{r_i} : \mathtt{c}]$ for some set of references indexed by I, and $S_{R'} = [G_R|\mathtt{r_j} : \mathtt{c}]$ for some set of references indexed by J.

Then if we can show that for all $d' > 2 \cdot d$, (θ', R') sat d' implies $Stream_A^{d'/2}(\mathtt{c}, (\theta', R'))$, we can conclude that (θ', R') sat d.

These two lemmas are enough to prove the correctness of every operation in our library except for allocating the input cell in the definition of the fixed point operation. The reason for this is that when we write back values into the reference cell, the static dependencies of the values may include the input. However, this is a harmless dependency, since the input cell just dereferences a pointer — it doesn't read any other cells or do any other computation. So we can prove a custom lemma just for this case.

LEMMA 5. *(Feedback Input Cell) Suppose that (θ, R) sat d. Then suppose that $\theta' = \theta \otimes \mathsf{ref}(\mathtt{r}, \mathsf{None}) \otimes \mathsf{cell}^-(\mathtt{c}, \mathtt{code})$, where $\mathtt{code}$ is $\mathtt{do\_} \leftarrow \mathtt{read}(\mathtt{clock}); !\mathtt{r}$. Furthermore, suppose we have R' such that $C_{R'} = C_R \cup \{\mathtt{c}\}$, $V_{R'} = [V_R|\mathtt{c} : vs]$, $D_{R'} = [D_R|\mathtt{c} : \mathtt{d}]$, $\Delta_{R'} = [\Delta|\mathtt{c} : Y]$ where $Y \subseteq C_{R'}$, $L_{R'} = L_R \cup \{\mathtt{r}\}$, $V_{R'} = [V_R|\mathtt{r} : (\mathsf{None} :: (\mathsf{map\ Some}\ vs)]$, $\Delta_{R'}^L = [\Delta_R^L|\mathtt{r} : X]$ where $X \subseteq C_{R'}$, $G_{R'} = [G_R|\mathtt{r} : \mathtt{c}]$, and $S_{R'} = S_R$.*

Then we have that (θ', R') sat d.

Now we have enough machinery to prove the correctness of the library. Let μ_0 be the least memory, with an empty sets of cells and local references (except for the clock and the mutable cell list $\mathtt{i}$). Then, we can show that

THEOREM 3. *Define $Realize_{\mathbb{U}}^{X \to Y}(f, \mathtt{f})$ to be $V_{X \Rightarrow Y}^0(f, \mathtt{f}, \mu_0, \emptyset)$, and define $Realize_{\mathbb{U}^S}^{A \to B}(g, \mathtt{g})$ to be $U_{A \multimap B}^0(g, \mathtt{g}, \mu_0, \emptyset, \mathtt{u})$. Then:*

- *If $\Gamma \vdash e : X$, then $Realize_{\mathbb{U}^S}^{\Gamma \to X}(\llbracket \Gamma \vdash e : X \rrbracket, (\!| \Gamma \vdash e : X |\!)_u)$*
- *If $\Gamma; \Delta \vdash t : A$, then $Realize_{\mathbb{U}}^{\Gamma^\omega \otimes \Delta \to A}(\llbracket \Gamma; \Delta \vdash t : A \rrbracket, (\!| \Gamma; \Delta \vdash t : A |\!)_u)$*

Here, the banana brackets means replacing the categorical combinators of Figure 4 with our implementation combinators.

$$
\begin{array}{l}
U^d_I(\langle\rangle, \langle\rangle, \mu, \sigma) = \text{true} \\
U^d_{A\otimes B}((a,b),(\mathtt{a},\mathtt{b}),\mu,\sigma) = U^d_A(a,\mathtt{a},\mu,\sigma) \wedge U^d_B(b,\mathtt{b},\mu,\sigma) \\
U^d_{A\multimap B}(f,\mathtt{f},\mu,\sigma) = \\
\quad \forall d' > 2\cdot d, n \le \log(1/d'), \sigma' \supseteq \sigma, \mu' \gg^{d'}_n \mu, v, \mathtt{v}. \\
\quad\quad UBuild^{d'\cdot 2^n}_A(v,\mathtt{v},\mu',\sigma') \Rightarrow UBuild^{d'\cdot 2^n}_B(f\ v, \mathtt{f}\ \mathtt{v}, \mu', \sigma') \\
U^d_{A\multimap\!\bullet B}(f,\mathtt{f},\mu,\sigma) = \\
\quad \forall d' > 2\cdot d, n \le \log(1/d'), \sigma' \supseteq \sigma, \mu' \gg^{d'}_n \mu, v, \mathtt{v}. \\
\quad\quad UBuild^{d'\cdot 2^n}_A(v,\mathtt{v},\mu',\sigma') \Rightarrow UBuild^{d'\cdot 2^n}_B(f\ v, \mathtt{f}\ \mathtt{v}, \mu', \sigma', \mathtt{u}) \\
U^d_{X^\omega}(vs,\mathtt{v},(\theta,R),\sigma) = V_R(\mathtt{v}) = vs \wedge \Delta_R(\mathtt{v}) \subseteq \sigma \\
V^d_{\mathsf{val}\ A}(v,\mathtt{v},\mu,\sigma) = U^d_A(v,\mathtt{v},\mu,\sigma) \\
V^d_I(\langle\rangle, \langle\rangle, \mu, \sigma) = \text{true} \\
V^d_{X\times Y}((x,y),(\mathtt{x},\mathtt{y}),\mu,\sigma) = V^d_X(x,\mathtt{x},\mu,\sigma) \wedge V^d_Y(y,\mathtt{y},\mu,\sigma) \\
\hat{V}^d_{S(X)}(vs,\mathtt{v},(\theta,R),\sigma) = V_R(\mathtt{v}) = vs \wedge \Delta_R(\mathtt{v}) \subseteq \sigma \\
V^d_{X\Rightarrow Y}(f,\mathtt{f},\mu,\sigma) = \\
\quad \forall d' > 2\cdot d, \mu' \sqsupseteq^{d'} \mu, \sigma' \supseteq \sigma, v, \mathtt{v} \\
\quad\quad V^{d'}_{S(X)}(vs,\mathtt{v},(\theta',R') \text{ as } \mu',\sigma') \Rightarrow \\
\quad\quad\quad VBuild^{d'}_Y(f^\dagger\ vs, \mathtt{f}\ \mathtt{v}, \theta', \sigma' \cup \{\mathtt{v}\}, \textstyle\bigsqcup_{\mathtt{w}\in\sigma'\cup\{v\}} D_{R'}(\mathtt{w})) \\
V^d_{X\rightsquigarrow Y}(f,\mathtt{f},\mu,\sigma) = \\
\quad \forall d' > 2\cdot d, \mu' \sqsupseteq^{d'} \mu, \sigma' \supseteq \sigma, v, \mathtt{v}. \\
\quad\quad \hat{V}^{d'}_{S(X)}(vs,\mathtt{v},\mu',\sigma') \Rightarrow VBuild^{d'}_Y(f^\dagger\ vs, \mathtt{f}\ \mathtt{v}, \mu', \sigma' \cup \{\mathtt{v}\}, \mathtt{u})
\end{array}
$$

Figure 13. The Logical Relation

$$
\begin{array}{l}
(\theta,R) \text{ sat } d \triangleq \\
\quad \text{cells}(\theta) = C_R \uplus \{\mathtt{clock}\} \text{ and} \\
\quad \text{refs}(\theta) = L_R \uplus \{\mathtt{i}\} \text{ and} \\
\quad \forall \mathtt{c} : X \in C_R.\ \mathsf{unready}(\theta,\mathtt{c}) \vee \exists \mathtt{v}.\ \mathsf{ready}(\theta,\mathtt{c},\mathtt{v}) \text{ and} \\
\quad \forall \mathtt{c} \in G_R(L_R) \cup S_R(L_R). \\
\quad\quad\quad\quad \mathsf{ready}(\theta,\mathtt{c},-) \Rightarrow \mathtt{clock} \in \mathsf{deps}(\theta,\mathtt{c}) \text{ and} \\
\quad \exists I.\ \mathsf{hasref}(\theta,\mathtt{i},I) \wedge I \subseteq (G_R(L_R) \cup S_R(L_R)) \text{ and} \\
\quad \forall \mathtt{c} : X \in C_R.\ \mathit{Stream}^d_X(\mathtt{c},(\theta,R)) \text{ and} \\
\quad \forall \mathtt{r} : X \in L_R.\ \mathit{Local}^d_X(\mathtt{r},(\theta,R)) \\
\mathit{Stream}^d_X(\mathtt{v},\mu) = \\
\quad \forall d' > 2\cdot d, n \le \log(1/d'), \mu' \gg^{d'}_n \mu.\ \mathit{Head}^{d'\cdot 2^n}_X(\mathtt{v},\mu') \\
\mathit{Head}^d_X(\mathtt{v},(\theta,R) \text{ as } \mu) = \\
\quad \{H(\theta) \wedge \text{extsat}(\mu,d)\} \\
\quad \mathtt{read}\ \mathtt{v} \\
\quad \{(\mathtt{a},\_).\ \exists(\theta',R') \text{ as } \mu' \sqsupseteq^d \mu, u.\ H(\theta') \wedge \\
\quad\quad\quad\quad \mathit{Ready}^d_X(\mathtt{v},\mathtt{a},\mu') \wedge \mathit{Update}(u,\mu,\mu',\mathtt{v}) \wedge \mathit{Stable}(\mu,\mu')\} \\
\mathit{Local}^d_A(\mathtt{r},(\theta,R)) = \\
\quad \forall d' > 2\cdot d.\ S_R(\mathtt{r}) \text{ defined} \Rightarrow \\
\quad\quad \mathsf{unready}(\theta,S_R(\mathtt{r})) \Rightarrow \\
\quad\quad\quad \mathit{Ref}^{d'}_X(\mathit{head}(V^R_L(r)),\mathtt{r},\mu,\Delta^+_R(S_R(\mathtt{r}))) \text{ and} \\
\quad\quad \mathsf{ready}(\theta,S_R(\mathtt{r}),-) \Rightarrow \\
\quad\quad\quad \mathit{Ref}^{d'}_X(\mathit{head}(\mathit{tail}(V^R_L(r))),\mathtt{r},\mu,\Delta^+_R(S_R(\mathtt{r}))) \\
\quad\quad\quad \wedge \exists \mathtt{v}.\ \mathsf{ready}(\theta,G_R(\mathtt{r}),\mathtt{v})) \\
\mathit{Ref}^d_X(v,\mathtt{r},(\theta,R),\sigma) = \\
\quad \exists \mathtt{v}.\ \mathsf{hasref}(\theta,\mathtt{r},\mathtt{v}) \wedge \mathit{Opt}^{d'}_X(v,\mathtt{v},(\theta,R),\sigma) \\
\mathit{Ready}^d_X(\mathtt{c},\mathtt{v},(\theta,R)) = \\
\quad \mathit{Opt}^d_X(\text{if } D_R(\mathtt{v}) \text{ then } \mathsf{None} \text{ else } \mathsf{Some}(V_R(\mathtt{v})),\mathtt{a},(\theta,R),\Delta^+_R(\mathtt{c})) \\
\mathit{Opt}^d_X(\mathsf{None},\mathsf{None},\mu,\sigma) = \text{true} \\
\mathit{Opt}^d_X(\mathsf{Some}(v),\mathsf{Some}(\mathtt{v}),\mu,\sigma) = V^d_X(v,\mathtt{v},\mu,\sigma)) \\
\text{extsat}((\theta,R),d) = \\
\quad \forall \mathtt{r} : X \in L_R.\ S_R(\mathtt{r}) \text{ undef} \Rightarrow \mathit{Ref}^d_X(\mathit{head}(V^L_R(\mathtt{r})),\mathtt{r},(\theta,R))
\end{array}
$$

Figure 14. The Satisfaction Relation

$$
\begin{array}{l}
\mathit{UBuild}^d_A(v,\mathtt{code},(\theta,R) \text{ as } \mu,\sigma) = \\
\quad \mathit{UBuild}^d_A(v,\mathtt{code},(\theta,R) \text{ as } \mu,\sigma,\textstyle\bigsqcup_{\mathtt{w}\in\sigma} D_R(\mathtt{w})) = \\
\mathit{UBuild}^d_A(v,\mathtt{code},(\theta,R) \text{ as } \mu,\sigma,L) = \\
\quad \{H(\theta) \wedge \text{extsat}(\mu,d)\} \\
\quad \mathtt{code} \\
\quad \{(\mathtt{a},\_).\ \exists(\theta',R') \text{ as } \mu' \sqsupseteq^d \mu, u.\ H(\theta') \wedge \mathit{UOpt}^d_A(v,\mathtt{a},\mu',\sigma,L) \wedge \\
\quad\quad\quad \mathit{Update}(u,\mu,\mu',\sigma) \wedge \mathit{Stable}(\mu,\mu')\} \\
\mathit{UOpt}^d_A(v,\mathsf{None},\mu',\sigma,\mathtt{d}) = \text{true} \\
\mathit{UOpt}^d_A(v,\mathsf{Some}(\mathtt{v}),\mu',\sigma,L) = U^D_A(v,\mathsf{Some}(\mathtt{v}),\mu',\sigma \\
\mathit{VBuild}^d_X(vs,\mathtt{code},(\theta,R) \text{ as } \mu,\sigma,L) = \\
\quad \{H(\theta) \wedge \text{extsat}((\theta,R),d)\} \\
\quad \mathtt{code} \\
\quad \{(\mathtt{a},\_).\ \exists(\theta,R') \text{ as } \mu' \sqsupseteq^d \mu, u.\ H(\theta') \wedge \mathtt{a} \in C_{R'} - C_R \wedge \\
\quad\quad \mathit{NewStream}^d_A(vs,\mathtt{a},\mu',\sigma,L) \wedge \\
\quad\quad \mathit{Update}(u,\mu,\mu',\mathtt{a}) \wedge \mathit{Stable}(\mu,\mu')\} \\
\mathit{NewStream}^d_X(vs,\mathtt{a},(\theta',R'),\sigma,L) = \\
\quad D_{R'}(\mathtt{a}) \sqsubseteq L \wedge \Delta^*_{R'}(\mathtt{a}) \subseteq \sigma \wedge \hat{V}^d_{S(X)}(vs,\mathtt{a},(\theta',R'),\Delta_{R'}(\mathtt{a})) \\
\mathit{Update}(u,(\theta,R),(\theta',R'),\mathtt{a}) = \\
\quad \forall \mathtt{c}.\ \mathsf{ready}(\theta',c,-) \wedge \mathsf{unready}(\theta,c) \iff c \in u \wedge \\
\quad \forall \mathtt{c} \in u.\ \mathtt{c} \in \Delta^*_R(\mathtt{a}) \vee \mathtt{c} \in (C_{R'} - C_R) \\
\mathit{Stable}(\mu,\mu') = \mathit{StableRefs}(\mu,\mu') \wedge \mathit{StableImps}(\mu,\mu') \\
\mathit{StableRefs}((\theta,R),(\theta' R')) = \\
\quad \forall \mathtt{r} \in L_{R'}.\ S_{R'}(\mathtt{r}) \text{ undef} \iff \mathtt{r} \in L_R \wedge S_R(\mathtt{r}) \text{ undef and} \\
\quad \forall \mathtt{r} \in L_{R'}.\ G_{R'}(\mathtt{r}) \text{ undef} \iff \mathtt{r} \in L_R \wedge G_R(\mathtt{r}) \text{ undef and} \\
\quad \forall \mathtt{r} \in L_R - \text{dom}(G_R).\ \exists \mathtt{v}.\ \mathsf{hasref}(\theta,\mathtt{r},\mathtt{v}) \wedge \mathsf{hasref}(\theta',\mathtt{r},\mathtt{v}) \\
\mathit{StableImps}((\theta,R),(\theta',R'))\} = \\
\quad \forall I, I'.\ \mathsf{hasref}(\theta,\mathtt{i},I) \wedge \mathsf{hasref}(\theta,\mathtt{i},I') \Rightarrow \\
\quad\quad [I' - S^{-1}_{R'}(L_{R'}) - G^{-1}_{R'}(L_{R'})] = [I - S^{-1}_R(L_R) - G^{-1}_R(L_R)] \\
\quad\quad \text{and } \forall \mathtt{c} \in I'.\ \mathsf{unready}(\theta',\mathtt{c}) \Rightarrow \mathtt{c} \in I \wedge \mathsf{unready}(\theta,c)
\end{array}
$$

Figure 15. Auxilliary Relations

$$
\begin{array}{l}
\mathit{Mem}(d) = \{(\theta,R) \mid (\theta,R) \text{ sat } d\} \\
(\sqsupseteq^d) \subseteq \mathit{Mem}(d) \times \mathit{Mem}(d) \\
(\theta',R') \sqsupseteq^d (\theta,R) \text{ iff} \\
\quad R' \sqsupseteq R \text{ and} \\
\quad \forall \mathtt{c} : X \in C_R, \mathtt{v}.\ \mathsf{ready}(\theta,\mathtt{c},\mathtt{v}) \Rightarrow \mathsf{ready}(\theta',\mathtt{c},\mathtt{v}) \text{ and} \\
\quad \forall \mathtt{c} : X \in C_R, \mathtt{code}.\ \mathsf{code}(\theta,\mathtt{c},\mathtt{code}) \Rightarrow \mathsf{code}(\theta',\mathtt{c},\mathtt{code}) \text{ and} \\
\quad \forall \mathtt{r} \in E_R, \mathtt{v}.\ \mathsf{ref}(\theta,\mathtt{r},\mathtt{v}) \iff \mathsf{ref}(\theta',\mathtt{r},\mathtt{v}) \\
\mu' \gg^d_0 \mu \quad = \mu' \sqsupseteq^d \mu \\
\mu' \gg^d_{n+1} \mu = \\
\quad \exists \mu_0 \in \mathit{Mem}(d).\ \mu_0 \sqsupseteq^d \mu \wedge \text{complete}(\mu_0) \wedge \mu' \gg^{2\cdot d}_n \mathit{tail}(\mu_0) \\
\text{complete}(\theta,R) = \\
\quad \forall \mathtt{r} \in L_R. \exists \mathtt{c} \in C_R.\ S_R(\mathtt{r}) = \mathtt{c} \wedge \exists \mathtt{v}.\ \mathsf{ready}(\theta,\mathtt{c},\mathtt{v})
\end{array}
$$

Figure 16. Orderings on Memories

8. Discussion

We have used ultrametric spaces to reinterpret the stream transformers familiar from the semantics of synchronous dataflow. In the special case of functions from streams to streams, causality and nonexpansiveness precisely coincide, but complete ultrametric spaces are Cartesian closed, supporting function spaces at all orders, and a general notion of contractiveness for defining well-founded fixed points. Furthermore, to support efficient implementation, we also need to make use of the co-Kleisli category over the stream comonad, which we connect to the base category via an adjunction.

Pouzet and Caspi [7] extended synchronous dataflow programming to higher order with their co-iterative semantics. They illustrated how that this generated a Cartesian closed category (of size-preserving functions), which they used to interpret functions. Uustalu and Vene [19] subsequently observed that size-preserving functions could be understood more abstractly as the co-Kleisli category of streams. However, in both of these works, feedback was handled in a somewhat *ad hoc* fashion.

The proper treatment of feedback is delicate, and disentangling the two main pieces of it kept us busy for a long time. First, we use ultrametrics to give a semantic criterion for causality, which permits us to avoid explicitly looking at the syntax of a program to identify dependencies. Second, we needed to make explicit use of the adjunction between the base category of ultrametric spaces and the co-Kleisli category of streams, in order to explain the semantics of feedback. By using two categories, we do not need to compromise on our reasoning principles in any way, which is quite remarkable given the low-level, imperative nature of our implementation.

Functional reactive programming was introduced by Elliott and Hudak [12], and was given a semantics in terms of event streams and unrestricted functions over them. In this and subsequent work [9], the semantics of fixed points were given denotationally. This gives semantics to all FRP expressions, including non-well-founded programs (which will go into infinite loops).

A notable feature of FRP is a treatment of continuous time. We believe that our proof framework should extend to proving an sampling theorem as in Wan and Hudak [20]. On the semantic side, continuous streams can be modelled as functions $\mathbb{R} \to A$, and the causal ultrametric extends naturally to this case. On the implementation side, the `clock` can supply time deltas (in contrast to its current delivery of pure ticks).

Due to the problem of space leaks, arrowized FRP [16] was introduced in order to restrict the set of definable stream transformers to the the causal ones. The restriction to arrows is roughly equivalent to first-order functional programming, though Nilsson *et al.* introduced additional combinators to recover higher-order and dynamic behavior. Our semantics gives a way of eliminating these restrictions and admitting higher-order and dynamic behavior in a very uniform way.

Metric methods were entered semantics in the early 1980s, to simplify the denotational semantics of concurrency [11]. The applications to stream programming were recognized early, but not followed up on: in a surprisingly little-cited 1985 paper [10], de Bakker and Kok proposed an ultrametric semantics for a language of first-order stream programs over integers and wrote "We think there are no problems when we allow functions of higher order[. . .]". This is a conjecture which we confirm, a full quarter-century later: it is true, but only if we make it true twice over!

More recently, Birkedal and his coworkers [6] have used ultrametric models to give logics and semantics for sequential programs involving advanced features such as higher-order state and polymorphism, and have shown connections between these ideas and the more operationally-flavoured technique of step-indexed models [1]. Our 'custom' logical relation is very much in the spirit of the recursively-defined predicates and relations used in these metric models, and we additionally exploit metric structure to define recursive *values*.

A fascinating and suggestive paper by Escardo [13], gives a metric model to PCF extended with timeout operators. Since cancels and interrupt operations pervade interactive programs, this suggests we should investigate whether they can be supported without harming the reasoning principles of the language.

Finally, we have actually implemented the API of this paper as a library in Objective Caml. After implementing the combinatory interface as a module interface, we implemented syntax extensions using CamlP4, which compile terms of the adjoint calculus into calls to our Ocaml library. In addition, we have implemented bindings to the GTK GUI toolkit, and are currently investigating proof principles for reasoning about "retained mode" architectures (of which GUI toolkits and the HTML DOM are two examples).

References

[1] A. W. Appel and D. A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, 2001.

[2] L. B. B. Biering and N. Torp-Smith. BI-hyperdoctrines, higher-order separation logic and abstraction. *ACM TOPLAS*, 29(5), 2007.

[3] N. Benton. A mixed linear and non-linear logic: Proofs, terms and models. In *Computer Science Logic*, volume 933 of *LNCS*, 1995.

[4] P. N. Benton and P. Wadler. Linear logic, monads and the lambda calculus. In *LICS*, pages 420–431, 1996.

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

[6] L. Birkedal, K. Støvring, and J. Thamsborg. The category-theoretic solution of recursive metric-space quations. Technical Report ITU-2009-119, IT University of Copenhagen, 2009.

[7] P. Caspi and M. Pouzet. A co-iterative characterization of synchronous stream functions. *Electr. Notes Theor. Comput. Sci.*, 11, 1998.

[8] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE: A declarative language for real-time programming. In *Proceedings of the 14th Symposium on Principles of Programming Languages*, 1987.

[9] A. Courtney. *Modeling User Interfaces in a Functional Language*. PhD thesis, Yale University, 2004.

[10] J. W. de Bakker and J. N. Kok. Towards a uniform topological treatment of streams and functions on streams. In *ICALP*, 1985.

[11] J. W. de Bakker and J. I. Zucker. Denotational semantics of concurrency. In *STOC*, pages 153–158. ACM, 1982.

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

[13] M. Escardó. A metric model of PCF. In *Workshop on Realizability Semantics and Applications*, 1999.

[14] N. Krishnaswami, L. Birkedal, and J. Aldrich. Verifying event-driven programs using ramified frame properties. In *Proceedings of Types in Language Design and Implementation*, 2010.

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

[16] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, page 64. ACM, 2002.

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

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

[19] T. Uustalu and V. Vene. The essence of dataflow programming. In *Central European Functional Programming School*, volume 4164 of *LNCS*, 2006.

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