A Semantic Model for Graphical User Interfaces

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

Nick Benton
Microsoft Research
<nick@microsoft.com>

Abstract

We give a denotational model for graphical user interface (GUI) programming using the Cartesian closed category of ultrametric spaces. The ultrametric structure enforces causality restrictions on reactive systems and allows well-founded recursive definitions by a generalization of guardedness. We capture the arbitrariness of user input (e.g., a user gets to *decide* the stream of clicks she sends to a program) by making use of the fact that the closed subsets of an ultrametric space themselves form an ultrametric space, allowing us to interpret nondeterminism with a "powerspace" monad.

Algebras for the powerspace monad yield a model of intuitionistic linear logic, which we exploit in the definition of a mixed linear/non-linear domain-specific language for writing GUI programs. The non-linear part of the language is used for writing reactive stream-processing functions whilst the linear sublanguage naturally captures the generativity and usage constraints on the various linear objects in GUIs, such as the elements of a DOM or scene graph.

We have implemented this DSL as an extension to OCaml, and give examples demonstrating that programs in this style can be short and readable.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages, Theory, Design

Keywords denotational semantics, ultrametric spaces, functional reactive programming, guarded recursion, linear logic

1. Introduction

Graphical user interfaces (GUI) libraries are one of the most widely-used examples of higher-order programming; even languages such as Java or C#, in which programmers normally eschew higher-order style, offer user interface toolkits which expose higher-order interfaces. Unfortunately, these libraries are a poor advertisement for higher-order style; they are extremely imperative, difficult to extend, and understanding the behavior of client code requires deep familiarity with the internal implementation of the toolkit.

Since its introduction, functional reactive programming [12] has held great promise for simplifying the specification and interface to graphical user interface libraries. However, a persistent difficulty with modeling user interface toolkits with *functional* reactive programs is that certain basic abstractions, such as widgets, seem inherently effectful.

For example, creating two buttons differs from creating one button and accessing it twice, since creating two buttons creates two *distinct* streams of button clicks. That is, since the button abstraction uses its access to the outside world to generate a potentially arbitrary stream of clicks – indeed, this is the essence of the abstraction – it is difficult to explain it as a pure value.

In this paper, we extend our earlier work [17] on the semantics of functional reactive programming to account for this phenomenon. In that work, we used the category of ultrametric spaces to interpret higher-order functional reactive programs, and used Banach's theorem to interpret (guarded) feedback. This offers a simple, general semantics for functional reactive programming with an associated language that is essentially the simply-typed lambda calculus with a type constructor for streams and a 'next' modality.

Our primary observation is that the basic abstraction of reactive programming, the event stream, allows us to decouple the state changes arising from a stream taking on new values as time passes from the effects associated with opening new channels of communication with the user. We can model the creation of a new button, for example, as a nondeterministic choice over all possible streams of clicks that the user might generate.

The powerspace monad we use to interpret the nondeterminism associated with user interaction is rather well-behaved, allowing us to design a cleaner term language for GUI-manipulating code than would be possible for general imperative code. In particular, algebras for the powerspace monad form a model of intuitionistic linear logic, so we can use a linear lambda calculus syntax for GUI programming. Viewing the monadic effect as the composition of the free and forgetful functors, our DSL splits into two parts, a conventional nonlinear functional language (with an unconventional treatment of fixed points) in which programmers can write programs to control the *behavior* of their interface, and a linear one in which they can *construct* and compose widgets and other interactive components.

The linear treatment of widgets is a natural fit for interfacing our DSL to existing GUI toolkits. Most of these feature some global piece of state (variously called the DOM, the scene graph, the canvas, the widget hierarchy, etc.) representing the graphical interface. Modifications to this data structure change the graphical layout of the interface, and the components of this data structure typically[1] have a strong linearity constraint — a given node has a unique path from the root of the scene graph. By placing widget-manipulating operations in the linear part of our API, we can conveniently use this global mutable data structure, while still maintaining the relevant invariants and offering a purely functional specification.

To summarize, our contributions are:

ICFP '11 September 19-21, Tokyo.

[1] At the implementation level, this is typically a consequence of the fact that most GUI toolkits — including GTK, Win32, and the HTML/XML DOM — require that each node contain a pointer to its (unique) parent. The tree-shaped containment hierarchy is used for, amongst other things, routing events efficiently.

1. We build upon our earlier work on semantics of reactive systems to give a simple denotational semantics of GUI programs, including strikingly simple semantics for the arbitrariness of user interaction.
2. We give a type theory for this semantics, which integrates our recent work on type systems for guarded recursion with an adjoint calculus [1, 2] for mixed linear/nonlinear logic. Despite the presence of a fixed point operator fix $x : A.\ e$, this type theory has excellent proof-theoretic properties, including a simple normalization proof.
3. We illustrate our theoretical work with an implementation, demonstrating that it can lead to clean user code for GUI programs.

2. Programming Language and Model

In this section, we will describe the type structure and equational theory of our DSL. Our focus will be on conveying intuition about the programming model, as preparation for exploring an extended example. The denotational semantics and proof theory will come after that. In slogan form, we use an intuitionistic lambda calculus to give a language for higher-order reactive programming, and combine it with a linear lambda calculus to model the stateful nature of the display and user input.

Reactive programs are usually interpreted as *stream transformers*. A time-varying value of type X is a stream of Xs, and so a program that takes a time-varying X and produces a time-varying Y is then a function that takes a stream of Xs and produces a stream of Ys. In their original paper on functional reactive programming, Elliott and Hudak [12] proposed using general stream-processing functional programs to manipulate these dynamic values.

While stream programming is flexible, it suffers from a major difficulty: there are definable stream-processing programs which do not correspond to physically realistic stream transformers. If a stream of values represents a time-varying signal, then it follows that stream processors should be *causal*: the first n outputs of a program should only depend on the first n inputs. But there are common stream functions which are not causal; the simplest example of which is $\mathsf{tail}\ xs$, whose n-th value is the $n+1$-st value of xs.

There have been several attempts to resolve this difficulty [18, 20], all of which build on the basic idea of using data abstraction to restrict the definable functions to the causal ones. In recent work [17], we have described a new solution to this problem which still blocks non-causal definitions, but does not surrender the flexibility of the original FRP model — in particular, its support for making free use of higher-order functions for building abstractions.

Our basic idea was to work in a lambda calculus with types not only for data, but also indexed by *time*. We introduced a type constructor $\bullet X$, pronounced "next X". If X is a type of values, then $\bullet X$ represents X at the next time step. Then, we can type infinite streams as follows:

$$\begin{array}{lcl} \mathsf{head} & : & S(X) \to X \\ \mathsf{tail} & : & S(X) \to \bullet S(X) \\ \mathsf{cons} & : & X \times \bullet S(X) \to S(X) \end{array}$$

Here, $S(X)$ is the type of streams of elements of X. The head operation has the usual type — it takes a stream and returns the first element, a value of type X. We diverge from the standard with the type of tail, which returns a value of type $\bullet S(X)$. So tail does not return a stream right away — it only returns a value of "stream tomorrow". As a result, it is impossible to take the head of the tail of a stream and perform a non-causal operation — typing rules out this possibility. Of course, given a value v of type X, and a stream tomorrow vs of type $\bullet S(X)$, we can construct a stream today, which will return v today, and then start producing the elements of vs tomorrow.

In Figure 1, we give the types of a small lambda typed calculus corresponding to this idea. The "nonlinear types" are the types just discussed, and are typed with a judgement $\Gamma \vdash e :_i X$. This can be read as saying "e is an expression of type X at time i, in context Γ". The context Γ also indexes all hypotheses with times, which controls when we can use variables — the UHYP rule says that a variable at time i can only be used at time i or later. The rules for the $\bullet X$ type internalize these time indices. The UDELAY rule tells us an expression of type X at time $i+1$ can be turned into a $\bullet X$ at time i, and conversely the UAWAIT rule tells us that a $\bullet X$ at time i can be used as an X at time $i+1$.

The last novelty in the nonlinear fragment is the UFIX rule. At first glance, it seems like an unrestricted fixed point, which is surprising in a calculus intended for defining well-founded stream programs. However, when we look at the time indices, it says that if we can create a value of type X at time i with a hypothesis at time $i+1$, then we can create a value at time i. The underlying intuition is best conveyed at the stream type. Suppose that we can build a stream, if only we had one tomorrow. Since we can't look at a stream tomorrow, this means that we can at least produce the head today — and so by turning today's output into tomorrow's input, we can tie the knot and construct a stream. (However, this works at *all* types, including functions and nested streams!)

The nonlinear language is sufficient for writing pure reactive programs, but cracks start to show if we try to bind it to GUI toolkits. The problem is best illustrated with an example. Suppose we had a function $\mathsf{button} : 1 \to S(\mathsf{bool})$, which created a button and returned a stream of booleans representing clicks. Now, consider the following two programs:

```
1 let bs = button() in  |  let bs = button() in
2 let bs' = button() in |  map xor (zip bs bs)
3 map xor (zip bs bs')  |
```

On the left, we have a program which creates two buttons, and then returns the xor of their click streams. On the right, we have a program which creates a single button, and then a stream resulting from xor'ing the stream with itself. The program in the right will return a constantly-false stream, but there is no reason to expect that the same should happen for two *different* buttons. Were we to stay within our pure functional language, which satisfies all the usual CCC equations, we would have to identify these two programs, which is clearly wrong.

We will deal with the side-effects associated with changes to the widget hierarchy by assigning linear types to GUI expressions. (Though the behaviour associated with clicks and changes to, for example, the text displayed in widgets, are still dealt with functionally.) From a purely syntactic point of view, the linearity constraint means we cannot coalesce common subexpressions, and so typing blocks problematic examples like the above. We still need to say precisely what such a syntax should mean — a question whose answer we defer to the denotational semantics — but for now we keep in mind that any solution will have to account for the fact that users are free to click a button arbitrarily, whenever they choose.

So we add a linear sub-language, in the style of the LNL calculus [1], also known as the adjoint calculus [2]. We write A, B, C for the linear types, instead of the X, Y, Z we use for nonlinear types. Linear types include the tensor $A \otimes B$, the linear function space $A \multimap B$, and a type Window of GUI windows. We can give a small but representative set of constants for building GUIs with the following types:

$$\begin{array}{lcl} \mathsf{label} & : & F(S(\mathsf{string})) \multimap \mathsf{Window} \\ \mathsf{vstack} & : & \mathsf{Window} \otimes \mathsf{Window} \multimap \mathsf{Window} \\ \mathsf{hstack} & : & \mathsf{Window} \otimes \mathsf{Window} \multimap \mathsf{Window} \\ \mathsf{button} & : & F(S(\mathsf{string})) \multimap \mathsf{Window} \otimes F(S(\mathsf{bool})) \end{array}$$

The type constructor $F(X)$ embeds a *F*unctional type into the linear GUI sub-language (and dually the type $G(A)$ embeds a *G*UI type into the functional sub-language), and so the type of label says that if we supply a stream of strings (the time-varying message to display) we will receive a window as a return value. The vstack and hstack operations stack two windows vertically or horizontally, with the linearity constraint ensuring that a window cannot be packed on top of itself. The button function takes a stream of label messages, and then returns both a (linear) window, and a (nonlinear) stream of booleans representing the clicks from this button.

The typing judgment $\Gamma; \Delta \vdash t :_i A$ for linear types is a little more complicated than for nonlinear ones. There are two contexts, one for nonlinear variables, and a second for linear ones. The $\Gamma; \Delta \vdash t :_i A$ can be read as, "t is a term of type A at time i, which may freely use the variables in Γ, but must use the variables in Δ exactly once". The presence of the linear variables in Δ permits us to reason about imperative data structures such as windows in an apparently-functional way.

The $G(A)$ and $F(X)$ types serve to embed each language in the other. The UG rule says that we can treat a term of type A as a duplicable computation, when it uses no resources from the context. This rule is in the nonlinear part of the calculus, but its elimination rule LG lives in the linear sub-language, and says that any nonlinear $G(A)$ expression e can be run with the $\mathsf{runG}(e)$ expression to produce an A.

Conversely, the LFI rule says that we can embed any nonlinear term t into the linear language $F(t)$, without using any resources. Its elimination form $\mathsf{let}\ F(x) = t\ \mathsf{in}\ t'$ takes a term of type $F(X)$, and binds its result to the nonlinear variable x in the scope of t'. Now t' may use the result bound to x as often as it likes, even if constructing t originally needed some resources.

Finally, in Figure 2, we give the equational theory of our little language. Since our DSL is a total language, it supports all the β and η laws of the lambda calculus, for both the linear and nonlinear parts of the DSL. We also have $\beta\eta$-equalities for the adjoint type constructors and the delay type.

3. Example: A Stack-based Calculator

In this section, we illustrate our language by developing a small stack-based calculator program.

3.1 Implementation Description

Conceptually, the calculator consists of two parts. First, we implement the semantics of a calculator, making use of all the standard facilities of the host language of our DSL (in this case, Objective Caml). In the second part, we will use our DSL to turn the semantics into a reactive event processor, and to connect the event processor to a small GUI which lets a user interact with it.

3.1.1 Objective Caml Calculator Interpreter

In Figure 3, we give the Objective Caml code which implements the calculator's functionality. An RPN calculator acts on a stack, so on line 1 we define a type of stacks simply as a list of integers. On line 3-4 we define the logical events to which our calculator will react, including receiving a digit, an arithmetic operation, a push or pop instruction, or a no-op. Our type of events has no connection to the internal event-loop API of our GUI toolkit — it is just an ordinary Caml datatype representing the semantic events in terms of which we wish to program our calculator.

On lines 6-16, we give a step function which takes an event and a stack, and returns a new event. We process a digit event by adding it as the least significant digit of the topmost element of the stack (creating it if necessary). For example, if the topmost element of the stack is 7, then the sequence of digit operations `Digit 1`, `Digit 2`, `Digit 3` will take the stack top from 7 to the number 7123.

$$
\begin{array}{llll}
\text{Nonlinear} & X, Y & ::= & 1 \mid X \times Y \mid X \Rightarrow Y \mid S(X) \mid \bullet X \mid G(A) \\
\text{Linear} & A, B & ::= & \mathrm{I} \mid A \otimes B \mid A \multimap B \mid \mathsf{Window} \mid F(X)
\end{array}
$$

$$
\begin{array}{llll}
\text{Terms} & e & ::= & () \mid (e, e) \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid \lambda x.\ t \mid t\ t' \mid G(t) \\
 & & \mid & \bullet(e) \mid \mathsf{await}(e) \mid \mathsf{cons}(e, e') \mid \mathsf{head}\ e \mid \mathsf{tail}\ e \\
 & & \mid & \mathsf{fix}\ x : A.\ e \mid x \\
 & t & ::= & () \mid \mathsf{let}\ () = t\ \mathsf{in}\ t' \mid (t, t) \mid \mathsf{let}\ (u, v) = t\ \mathsf{in}\ t' \\
 & & \mid & \lambda x.\ t \mid t\ t' \mid F(e) \mid \mathsf{let}\ F(x) = t\ \mathsf{in}\ t' \mid \mathsf{runG}(e) \mid x \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x :_i X \\
 & \Delta & ::= & \cdot \mid \Delta, x :_i A
\end{array}
$$

$$\boxed{\Gamma \vdash e :_i X} \qquad \boxed{\Gamma; \Delta \vdash t :_i A}$$

$$\frac{i \le j \qquad x :_i A \in \Gamma}{\Gamma \vdash x :_j A}\ \textsc{UHyp} \qquad \frac{}{\Gamma \vdash () :_i 1}\ \textsc{UUnit}$$

$$\frac{\Gamma \vdash e :_i X \qquad \Gamma \vdash e' :_i Y}{\Gamma \vdash (e, e') :_i X \times Y}\ \textsc{UPair} \qquad \frac{\Gamma \vdash e :_i X \times Y}{\Gamma \vdash \mathsf{fst}\ e :_i X}\ \textsc{UFst}$$

$$\frac{\Gamma \vdash e :_i X \times Y}{\Gamma \vdash \mathsf{snd}\ e :_i Y}\ \textsc{USnd} \qquad \frac{\Gamma, x :_i A \vdash e :_i B}{\Gamma \vdash \lambda x.\ e :_i A \Rightarrow B}\ \textsc{UFun}$$

$$\frac{\Gamma \vdash e :_i A \Rightarrow B \qquad \Gamma \vdash e' :_i A}{\Gamma \vdash t\ t' :_i B}\ \textsc{UApp} \qquad \frac{\Gamma \vdash e :_{i+1} X}{\Gamma \vdash \bullet e :_i \bullet X}\ \textsc{UDelay}$$

$$\frac{\Gamma \vdash e :_i \bullet X}{\Gamma \vdash \mathsf{await}(e) :_{i+1} X}\ \textsc{UAwait}$$

$$\frac{\Gamma \vdash e :_i X \qquad \Gamma \vdash e' :_{i+1} S(X)}{\Gamma \vdash \mathsf{cons}(e, e') :_i S(X)}\ \textsc{UCons}$$

$$\frac{\Gamma \vdash e :_i S(X)}{\Gamma \vdash \mathsf{head}\ e :_i X}\ \textsc{UHead} \qquad \frac{\Gamma \vdash e :_i S(X)}{\Gamma \vdash \mathsf{tail}\ e :_{i+1} S(X)}\ \textsc{UTail}$$

$$\frac{\Gamma, x :_{i+1} A \vdash e :_i A}{\Gamma \vdash \mathsf{fix}\ x : A.\ e :_i A}\ \textsc{UFix} \qquad \frac{\Gamma; \cdot \vdash t :_i A}{\Gamma \vdash G(t) :_i G(A)}\ \textsc{UG}$$

$$\frac{}{\Gamma; x :_i A \vdash x :_i A}\ \textsc{LHyp} \qquad \frac{}{\Gamma; \cdot \vdash () :_i \mathrm{I}}\ \textsc{LUnit}$$

$$\frac{\Gamma; \Delta \vdash t :_i \mathrm{I} \qquad \Gamma; \Delta' \vdash t' :_i A}{\Gamma; \Delta, \Delta' \vdash \mathsf{let}\ () = t\ \mathsf{in}\ t' :_i A}\ \textsc{LUnitE}$$

$$\frac{\Gamma; \Delta \vdash t :_i A \qquad \Gamma; \Delta' \vdash t' :_i B}{\Gamma; \Delta, \Delta' \vdash (t, t') :_i A \otimes B}\ \textsc{LPair}$$

$$\frac{\Gamma; \Delta \vdash t :_i A \otimes B \qquad \Gamma; \Delta', x :_i A, y :_i B \vdash t' :_i C}{\Gamma; \Delta, \Delta' \vdash \mathsf{let}\ (x, y) = t\ \mathsf{in}\ t' :_i C}\ \textsc{LPairE}$$

$$\frac{\Gamma; \Delta, x :_i A \vdash t :_i B}{\Gamma; \Delta \vdash \lambda x.\ t :_i A \multimap B}\ \textsc{LFun}$$

$$\frac{\Gamma; \Delta \vdash t :_i A \multimap B \qquad \Gamma; \Delta' \vdash t' :_i A}{\Gamma; \Delta, \Delta' \vdash t\ t' :_i B}\ \textsc{LApp}$$

$$\frac{\Gamma \vdash e :_i G(A)}{\Gamma; \cdot \vdash \mathsf{runG}(e) :_i A}\ \textsc{LG} \qquad \frac{\Gamma \vdash e :_i X}{\Gamma; \cdot \vdash F(e) :_i F(X)}\ \textsc{LFI}$$

$$\frac{\Gamma; \Delta \vdash t :_i F(X) \qquad \Gamma, x :_i X; \Delta' \vdash t' :_i B}{\Gamma; \Delta, \Delta' \vdash \mathsf{let}\ F(x) = t\ \mathsf{in}\ t' :_i B}\ \textsc{LFE}$$

Figure 1. Types and Syntax

Type	Equation		
I	$\mathsf{let}\ () = ()\ \mathsf{in}\ t$	$=_\beta$	t
	$[t/a]t'$	$=_\eta$	$\mathsf{let}\ () = t\ \mathsf{in}\ [()/a]t'$
$A \otimes B$	$\mathsf{let}\ (a,b) = (t_1,t_2)\ \mathsf{in}\ t'$	$=_\beta$	$[t_1/a][t_2/b]t'$
	$[t/c]t'$	$=_\eta$	$\mathsf{let}\ (a,b) = t\ \mathsf{in}\ [(a,b)/c]t'$
$A \multimap B$	$(\lambda a.\ t')\ t$	$=_\beta$	$[t/a]t'$
	t	$=_\beta$	$\lambda a.\ t\ a$
$F(X)$	$\mathsf{let}\ F(x) = F(e)\ \mathsf{in}\ t$	$=_\beta$	$[e/x]t$
	$[t/a]t'$	$=_\eta$	$\mathsf{let}\ F(x) = t\ \mathsf{in}\ [F(x)/a]t'$
1	e	$=_\eta$	$()$
$X \times Y$	$\mathsf{fst}\ (e,e')$	$=_\beta$	e
	$\mathsf{snd}\ (e,e')$	$=_\beta$	e'
	e	$=_\eta$	$(\mathsf{fst}\ e, \mathsf{snd}\ e)$
$X \Rightarrow Y$	$(\lambda x.\ e')\ e$	$=_\beta$	$[e/x]e'$
	e	$=_\eta$	$\lambda x.\ e\ x$
$S(X)$	$\mathsf{head}\ \mathsf{cons}(e,e')$	$=_\beta$	e
	$\mathsf{tail}\ \mathsf{cons}(e,e')$	$=_\beta$	e'
	e	$=_\beta$	$\mathsf{cons}(\mathsf{head}\ e, \mathsf{tail}\ e)$
$\bullet A$	$\mathsf{await}(\bullet e)$	$=_\beta$	e
	$\bullet(\mathsf{await}\ e)$	$=_\eta$	e
$G(A)$	$\mathsf{runG}(G(e))$	$=_\beta$	e
	t	$=_\eta$	$G(\mathsf{runG}(t))$
X	$\mathsf{fix}\ x : X.\ e$	$=$	$[\mathsf{fix}\ x : X.\ e/x]e$

Figure 2. Equational Theory

As in other RPN-style calculators, we need a command to mark the end of a numeric input, which the `Push` event accomplishes by pushing 0 onto the top of the stack. Since digit operations only affect the topmost element, pushing 0 completes the entry of a number by making it the second element of the stack. The `Pop` operation deletes the topmost element from the stack, discarding it. The `Plus`, `Minus`, and `Times` operations each take the top two elements of the stack, and replace them with the result of performing the appropriate arithmetic operation on them (reducing the height of the stack by a net of one). The `Clear` operation simply sets the topmost element of the stack to 0.

Finally, we have a catchall clause which does nothing to the stack, which handles the `Nothing` event and also ensures that invalid operations (for instance `Plus` on a stack with 1 or 0 elements) do nothing.

On lines 18-22, we define a `display` function which takes a stack and returns a string of a comma-separated list of values. On lines 24-27, we define the `merge` function, which we will use to multiplex multiple streams of operations into a single stream. Since fair merge is inherently nondeterministic, we implement a simple biased choice operation. If its first argument is `Nothing`, then it returns its second argument — otherwise it returns its first argument. Finally, on lines 29-30 we define a function `bool_to_op op b` which returns `op` if `b` is true, and `Nothing` otherwise.

We reiterate that the code in Figure 3 is ordinary Ocaml code. Interpreters and symbolic computation is natural for functional languages, and we do not need any special additional features to implement this part. It is only when we need to write interactive code that programs in standard functional languages (and for that matter, OO languages) start to lose their appeal.

3.1.2 DSL Code for the User Interface

In Figure 4 we give the actual implementation of the user interface, as a program written in our domain-specific language. The ambient context of the code in this listing is on the functional side of the wall, making use of the $G(-)$ constructor to shift into the linear sublanguage as needed.

On lines 1-4, we have a function `stacks`, which takes an initial stack and a stream of operations, and turns it into a stream of stacks. It does this by taking the current operation and applies it to the current stack, to get the next stack, and recursively calls `stacks` on the next state and the tail of the stream of operations. Note that as a recursive definition (at a function type), the UFIX rule requires that any recursive calls to `stacks` must occur at a later time. In this definition, the only recursive call we make is inside the tail argument to cons, which according to the `UCons` rule occurs at a later time than the head argument. As a result, this is a safe recursive call which will pass typechecking.

Figure 4 contains *all* of the code we need to plug our state machine into the GUI event loop — there are no callbacks or anything like that. We just do familiar-looking functional programming with streams, with a type system that warns the user of any ill-founded definitions.

On lines 7-51, we actually build a GUI. On line 7, as an abbreviation we define the type of input builders `input` $= G(\mathsf{Window} \otimes F(S(\mathsf{op})))$. We can read this type as saying it is a GUI command which when executed builds a window yielding a stream of calculator operations. So this is the type we will use for user input GUI widgets in our calculator application.

On lines 9-12, we define the `calculator_button` widget. This function takes a string and an event, and returns a button labeled with the string argument, and which returns the event whenever the button is clicked. On line 11, we create a button, calling the `button` function with a constant stream of the message argument. This returns a window and a stream of clicks, and we map `bool_to_op` over that stream of clicks to generate a stream of events. (As an aside, we use a nested pattern matching syntax `let (w, F(bs)) = ...in ...` rather than splitting the term into the two bindings `let (w, fbs) = ...in let F(bs) = fbs in ...`. This is an easily desugared notational convenience.)

On lines 14-16, we use the `calculator_button` function to define the `numeric` function, which is the function we will use to create the number buttons of our calculator. It is just a call to `calculator_button` which converts its numeric argument to a string and a digit operation before calling `calculator_button`.

Despite its simplicity, this function illustrates a key feature of our library: defining new widgets is *easy*. We construct a new widget for no reason other than to avoid having to write calls like `calculator_button (string_of int 5) (Digit 5)`. This avoids a miniscule amount of redundancy, and yet defining new widget operations is lightweight enough that it is worth doing.

On lines 18-31, we define a function `pack`, which is a combinator for building new input widgets out of old ones. As arguments, it takes a stacking function `stack` to merge windows (which in this case will be `hstack` or `vstack`), and four inputs `(g1, g2, g3, g4)`, which it will coalesce into a new input widget. It does this by executing each of the `g`'s, and merging the returned windows `w1, w2, w3, w4` with `stack`, and merging the returned operation streams `es1, es2, es3, es4` with the `merge` operation.

The `pack` function shows off two strengths and one minor weakness of our DSL. The weakness is simply that we have not yet added lists to the type constructors of our DSL, and so `pack` is hard-coded to take a 4-tuple. The first strength is that `pack` is a higher-order function, parameterized in the packing order. This is something that a first-order embedded DSL, which did not supply its own interpretation of the function space, could not conveniently do, since the function space in question is a *linear* function space.

The second strength is the compositionality of functional programming. We have taken the type $G(\mathsf{Window} \otimes F(S(\mathsf{op})))$ to be the type of input widgets, and we are at liberty to construct new inhabitants of this type. Not only are individual buttons input widgets, but entire input panels are also input widgets. So in the definition of `input_panel` on lines 33-45, we are able to use the same `pack` function to lay out both the individual rows of buttons, and the stack

```
type stack = int list

type op = Push  | Pop   | Digit of int
        | Plus  | Minus | Times
        | Clear | Delete | Nothing

val step : op * stack → stack
let step = function
  | Digit n, m :: s     → (10 * m + n) :: s
  | Digit n, []         → [n]
  | Push,   s           → 0 :: s
  | Pop,    _ :: s      → s
  | Plus,   n :: m :: s → (n + m) :: s
  | Minus,  n :: m :: s → (n - m) :: s
  | Times,  n :: m :: s → (n * m) :: s
  | Clear,  n :: s      → 0 :: s
  | Delete, n :: s      → s
  | _,      s           → s

val display : stack → string
let rec display = function
  | []     → "0"
  | [n]    → string_of_int n
  | n :: s → (string_of_int_n) ^ ", " ^ (display s)

val merge : op * op → op
let merge = function
  | Nothing, e →e
  | e, _       → e

val bool_to_op : op → bool → op
let bool_to_op op b = if b then op else Nothing
```

Figure 3. RPN Calculator Internals (in OCaml)

of rows, simply by varying the stacking parameter. Furthermore, since pack can do computation, it is able to act as a "smart constructor" which appropriately multiplexes the input signals of all its components. As a result, none of the wiring code is explicit in the code building the input panel — the code just follows the tree structure of the visual layout, like an HTML document, even though it also produces an output signal.

3.2 Comparison With Traditional Approaches

It is difficult to give a crisp, precise comparison with the traditional approach to GUI programming based on imperative callbacks and event loops, for two reasons. On the one hand, existing GUI libraries have few algebraic or equational properties, which makes theoretical comparisons difficult, and on the other, we have not yet written any large programs in our language, which makes practical comparisons difficult. Nevertheless, it is still possible to draw distinctions at the architectural level.

Traditional GUI toolkits (based on the model-view-controller pattern [15]) are logically structured as a collection of asynchronous event processors. Each GUI widget listens to events generated by some other widgets (and the main application/event loop), and generates events itself. The primary feature of this design is that widgets can generate events at different rates, and there is no guarantee of the order in which events are delivered to other widgets. (In practice, the order of delivery depends on the specific order in which callbacks were registered with a widget object.)

In contrast, our system is based on a *synchronous* model of time, such as used in languages such as Lustre [6] and Lucid Synchrone [21]. There is a global clock, which is respected by every event stream in the program — every stream generates one event each tick. Our primary motivation was that with a simpler deterministic semantics, it was possible to interpret higher-order

```
val stacks : stack × S(op) ⇒  S(stack)
let rec stacks (stack, ops) =
  let next_stack = step (head ops, stack) in
  cons(stack, stacks(next_stack, tail ops))


type input = G(Window ⊗ F(S(op)))

val calculator_button : string ⇒ op ⇒ input
let calculator_button msg op =
  G(let (window, F(bs)) = button F(constant msg) in
    (window, F(map (bool_to_op op) bs))

val numeric : int ⇒ input
let numeric n =
  calculator_button (string_of_int n) (Num n)

val pack : G(Window ⊗ Window ⊸ Window) ⇒
           (input × input × input × input) ⇒
           input
let pack stack (g1, g2, g3, g4) =
  G(let (w1, F(es1)) = runG g1 in
    let (w2, F(es2)) = runG g2 in
    let (w3, F(es3)) = runG g3 in
    let (w4, F(es4)) = runG g4 in
    let w = runG stack (runG stack (w1, w2),
                        runG stack (w2, w3)) in
    let es = F(merge(es1,
               merge(es2,
               merge(es3, es4)))) in
    (w, es))

val input_panel : input
let input_panel =
  pack G(vstack)
    (pack G(hstack) (numeric 7, numeric 8, numeric 9,
                     calculator_button "+" Plus),
     pack G(hstack) (numeric 4, numeric 5, numeric 6,
                     calculator_button "-" Minus),
     pack G(hstack) (numeric 1, numeric 2, numeric 3,
                     calculator_button "x" Times),
     pack G(hstack) (calculator_button "D" Delete,
                     numeric 0,
                     calculator_button "," Push,
                     calculator_button "C" Clear))

val calculator : G(Window)
let calculator =
  G(let (input, F(es)) = runG input_panel in
    let msgs = F(map display (stacks([], es))) in
    vstack(label msgs, input))
```

Figure 4. Code for Calculator (in embedded GUI DSL)

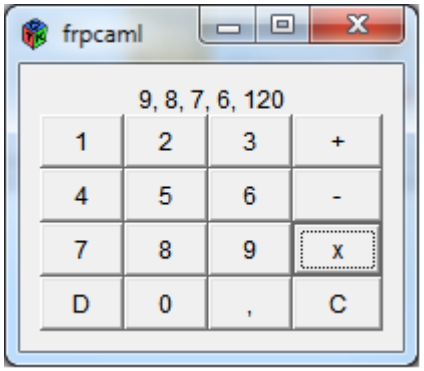

Figure 5. Screenshot of the RPN Calculator

constructions such as functions and streams of streams. We think that these constructions offer the same simplifications of reasoning that the lambda calculus offers over process calculi.

Under a strictly deterministic semantics, it is impossible to write certain programs, such as a fair merge of streams. In our calculator example, we have to explicitly write a function to multiplex events, as can be seen in the definition of `merge` on line 26-28 of Figure 3, and in its use in lines 28-31 of Figure 4. Here, we had to define a *biased* choice, which favors the left argument over the right. Even though fair merges are useful for specifying the behavior of GUIs, we think that it is a bad choice for implementation, since it makes debugging and reasoning about programs much harder than under a deterministic semantics.

Another tradeoff comes when trying to correlate events. As can be seen in the definition of the `op` datatype on line 3-5 of Figure 3, we need to explicitly include a `Nothing` constructor to indicate that no event happened on this tick. Under an asynchronous event semantics, no-ops do not need to be part of the event datatype, since the event source is never obligated to send a message.

However, this advantage is negated by the problems that arise when we need to correlate events from multiple sources. Since neither simultaneity nor order are guaranteed for event deliveries, clients have to build their own state machines to reconstruct event correlations (such as a view attempting to correlate events generated by the model and the controller). Building these state machines can get tricky, and programmers are often tempted to exploit details of the subject-observer implementation and register callbacks in a particular order to guarantee a certain order of event delivery. These tricks are fragile, and any errors lead to bugs which can be fiendishly difficult to diagnose.

4. Denotational Semantics

4.1 Intuition

Reactive programs are typically interpreted as *stream transformers*: functions which take a stream of inputs and generate a stream of outputs. However, not all functions on streams correspond to realistic reactive programs. For example, a stock trading program accepts a stream of price quotes and emits a sequence of buy and sell orders, but the set-theoretic function space $\mathsf{Price}^\omega \to \mathsf{Order}^\omega$ contains functions which produces orders that are a function of the *future* stock prices. These functions are, alas, not implementable.

The condition governing which stream functions are implementable is *causality*: the first n outputs may only depend on the first n inputs. Writing $\lfloor xs \rfloor_n$ for the n-element prefix of the stream xs, we can express causality as follows:

Definitition 1. *(Causality) A stream function $f : A^\omega \to B^\omega$ is* causal*, when for all n and all streams as and as', we have*

$$\lfloor as \rfloor_n = \lfloor as' \rfloor_n \implies \lfloor f\ as \rfloor_n = \lfloor f\ as' \rfloor_n$$

Causality is a very important idea, but it is defined in terms of stream functions, and for building programs we will need many more types than just the stream type. So we need to generalize causality to work at other types such as streams of streams, or even higher-order functions.

Furthermore, it is very common in reactive programming to define streams by feedback and recursion, such as in the following definition of the increasing sequence of naturals:

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

Intuitively, this fixed point is well-defined because the lambda-term can produce the first value of its output (that is, 0) before looking at its input. So we can imagine constructing the fixed point via a feedback process, in which we take the n-the output and feed it back to as the input at time $n+1$. So as long as we can generate *more* than n outputs from the first n inputs, we can find a fixed point.

Definitition 2. *(Guardedness) A function $f : A^\omega \to B^\omega$ is* guarded*, when there exists a $k > 0$ such that for all n and all streams as and as', we have*

$$\lfloor as \rfloor_n = \lfloor as' \rfloor_n \implies \lfloor f\ as \rfloor_{n+k} = \lfloor f\ as' \rfloor_{n+k}$$

Lemma 1. *(Fixed Points of Guarded Functions) Every guarded endofunction $g : A^\omega \to A^\omega$ has a unique fixed point.*

Just as with causality, the generalization to higher types seems both useful and unobvious. For example, we may which to define a recursive function to generate the Fibonacci numbers:

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

The above definition of fib seems intuitively plausible, but goes beyond the stream-based definition of guardedness, since it involves taking a fixed point at function type.

To systematically answer these questions, we make use of the mathematics of metric spaces.

4.2 1-Bounded Complete Ultrametric Spaces

Definitition 3. *(Complete 1-bounded Ultrametric Spaces) A* complete 1-bounded ultrametric space *is a pair (X, d_X), where X is a set and $d_X \in X \times X \to [0,1]$ is a distance function, satisfying the following axioms:*

- $d_X(x,y) = 0$ *if and only if* $x = y$
- $d_X(x,y) = d_X(y,x)$
- $d_X(x,y) \leq \max(d_X(x,z), d_X(z,y))$
- *Every Cauchy sequence in X 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 $i > n$, $d(x, x_i) \leq \epsilon$.

The first three conditions axiomatize the properties of 1-bounded distance functions — that the distance between a point and itself is zero; that distances are symmetric, and the triangle inequality. Ultrametric spaces satisfy a stronger version of the triangle inequality than ordinary metric spaces, which only ask that $d(x,y)$ be less than or equal to $d_X(x,z) + d_X(z,y)$, rather than $\max(d_X(x,y), d_X(y,x'))$. We often just write X for (X, d_X). All the metric spaces we consider are *bisected*, meaning that the distance between any two points is 2^{-n} for some $n \in \mathbb{N}$.

We will often simply write X for a metric space, and d_X for its corresponding metric (or even d, if the expression is unambiguous).

Definitition 4. *(Nonexpansive maps) A function $f : X \to Y$ is* nonexpansive *when for all $a, a' \in X$, we have $d_Y(f\ a, f\ a') \leq d_X(a, a')$.*

Complete 1-bounded ultrametric spaces and nonexpansive maps form a category (which we will write Ult). The category Ult is bicartesian closed (ie, has sums, products and exponentials), and supports a number of other functors on it which we will use in our semantics. (Our semantics will only use nonempty spaces, even though we will perform our semantic constructions in the full category.)

1. Any set X (such as the natural numbers $\mathbb{N}$) forms a metric space under the discrete metric (i.e., $d(x,y) = 0$ if $x = y$, and 1 otherwise).
2. If (X, d_X) is an object of Ult, we can form causal streams $S(X)$ of the elements as follows:
   - $S(X) = (\{as \mid as \in \mathbb{N} \to X\}, d_{S(X)})$, where
   - $d_{S(X)}(xs, ys) = \sup\left\{2^{-n} \times d_X(xs_n, ys_n) \mid n \in N\right\}$

The intuition behind the stream metric is best conveyed by thinking of $S(\mathbb{N})$, the streams of natural numbers (a discrete space). In this case, the distance between two streams xs and ys is 2^{-n}, where n is the first position at which they disagree. So if xs and ys disagree immediately, at time 0, then their distance is 1. If they disagree at time 32, then their distance is 2^{-32}. If they never disagree, then their distance is $2^{-\infty} = 0$ — that is, they are the same stream.

It is a matter of unwinding definitions to establish that a nonexpansive map $S(X) \to S(Y)$ (for discrete spaces X and Y) is equivalent to the definition of *causality* — that the first k outputs are determined by at most the first k inputs. In this sense, ultrametric spaces give a natural setting in which to interpret higher-order reactive programs [17], since we can interpret functions with the Cartesian closure and streams of arbitrary type with the stream functor, and nevertheless we can still rule out the definition of non-causal functions.

3. If (A, d_A) and (B, d_B) are objects of Ult, we have a Cartesian product
   - $(A, d_A) \times (B, d_B) \equiv (A \times B, d_{A\times B})$, where
   - $d_{A\times B}((a, b), (a', b')) = \max(d_A(a, a'), d_B(b, b'))$

   The elements of the product space are pairs of elements of the two underlying spaces, and the distance between two pairs is the maximum of the distances between the individual components.
4. If (A, d_A) and (B, d_B) are objects of Ult, we have an exponential defined as follows:
   - $(A, d_A) \Rightarrow (B, d_B) \equiv (\mathrm{Ult}(A, B), d_{A\Rightarrow B})$, where
   - $d_{A\Rightarrow B}(f, g) = \max\{d_B(f\ a, g\ a) \mid a \in A\}$

   The set of points for the function space is the set of nonexpansive functions from A to B (i.e., the hom-set for A and B), and the distance between two functions is the maximum distance between the results ranging over all arguments a. Intuitively, a function can be thought of as an A-indexed tuple of B's, and so the metric for the function space can be seen as a generalization of the metric for tuples.
5. If (A, d_A) and (B, d_B) are objects of Ult, we have a coproduct defined as follows:
   - $(A, d_A) + (B, d_B) \equiv (A + B, d_{A+B})$, where
   - $d_{A+B}(x, y) = \begin{cases} d_A(a, a') & \text{if } x = \mathsf{inl}(a), y = \mathsf{inl}(a') \\ d_B(b, b') & \text{if } x = \mathsf{inr}(b), y = \mathsf{inr}(b') \\ 1 & \text{otherwise} \end{cases}$

   The presence of coproducts in our semantic model means that it is possible to implement FRP-style "switching combinators" simply by performing a case analysis and returning different stream results, in the ordinary style of functional programming. (For space reasons, we did not include rules for sums in our calculus, though they can be added in the standard way.)
6. For each metric space, there is a contraction $\bullet(X)$ (pronounced "next-X") defined as follows:
   - $\bullet(X, d) = (X, d')$, where
   - $d'_{\bullet(X)}(a, a') = \frac{1}{2} d_X(a, a')$

   This functor is Cartesian closed. That is, there are isomorphisms $zip_\bullet : \bullet X \times \bullet Y \simeq \bullet(X \times Y)$ and $\epsilon : \bullet(X \Rightarrow Y) \simeq \bullet X \Rightarrow \bullet Y$.

   The $zip_\bullet$ isomorphism says that a delayed pair is the same as a pair of delays. A little more surprisingly, the ϵ isomorphism says that a delayed function is equivalent to a function which takes a delayed argument and returns a delayed result. All of these maps are just identities on the points; their real content lies in the fact that these maps are nonexpansive.

   We will write ϵ for the left-to-right direction of the isomorphism, and write ϵ^{-1} for the right-to-left direction, and similarly for the other isomorphisms we make use of.

   We can also iterate these constructions, and will write ϵ_i for the isomorphism of type $\bullet^i(X \Rightarrow Y) \simeq \bullet^i X \Rightarrow \bullet^i Y$ (and again, similarly for the other isomorphisms). We give the explicit definitions of these isomorphisms in Figure 6.

   In addition, we have a natural transformation $\delta_X : X \to \bullet X$ (pronounced "delay") which embeds each space X into $\bullet X$ via the identity on points. Intuitively, this corresponds to taking a value and delaying using it until the tick of the clock.

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

   The utility of the delay functor is that it lets us concisely state Banach's contraction map theorem for any nonempty, bisected metric space X: Banach's theorem is equivalent[2] to the assertion that there is a fixed point operator $fix : (\bullet X \Rightarrow X) \to X$.
7. If X is an object of M, then we can construct the power space $\mathcal{P}_C(X)$ of its closed, nonempty subsets (a subset $S \subseteq X$ is closed if every Cauchy sequence of elements in S has a limit also in S):
   - $\mathcal{P}_C(X) = \{U \subseteq X \mid U \text{ is closed and nonempty}\}$
   - $d_{\mathcal{P}_C(X)}(U, V) = \max\left(\begin{array}{l} \sup_{x\in U} \inf_{y\in V} d_X(x, y), \\ \sup_{y\in V} \inf_{x\in U} d_X(x, y)) \end{array}\right)$

   The metric $d_{\mathcal{P}_C X}$ is known as the *Hausdorff metric*. The functorial action of $\mathcal{P}_C(f) : \mathcal{P}_C(X) \to \mathcal{P}_C(Y)$ applies f pointwise to the elements of its argument, and takes the metric closure of that image. Explicitly, $\mathcal{P}_C(f) = \lambda X.\ \mathsf{cl}(\{f(x) \mid x \in X\})$.

   The functor $\mathcal{P}_C(-)$ gives rise to a strong, commutative monad [25]. We will write $\eta_X : X \to \mathcal{P}_C(X)$ and $\mu_X : \mathcal{P}_C^2(X) \to \mathcal{P}_C(X)$ for its unit and multiplication, and $\sigma_X : X \times \mathcal{P}_C(Y) \to \mathcal{P}_C(X \times Y)$ for its strength. Explicitly, the definition of the unit is $\eta(x) = \{x\}$, and the multiplication is $\mu(U) = \mathsf{cl}(\bigcup U)$.

   Furthermore, the contraction functor distributes over the powerspace; we have an isomorphism $\mathcal{P}_C^\bullet : \bullet\mathcal{P}_C(X) \simeq \mathcal{P}_C(\bullet\ X)$. (Again, this isomorphism is the identity on points, and following the pattern of Figure 6 we can iterate it to construct $\mathcal{P}_C^{\bullet i} : \bullet^i\mathcal{P}_C(X) \simeq \mathcal{P}_C(\bullet^i\ X)$.)
8. To interpret the Window type, we first take Pic to be the set of trees inductively generated from the following grammar (with s ranging over strings):

   $$p \quad ::= \quad \mathsf{Button}(s) \mid \mathsf{Label}(s) \mid \mathsf{Vert}(p, p) \mid \mathsf{Hor}(p, p)$$

   We turn this into a space by equipping it with the discrete metric, and then take Window to be streams of Pic.

4.3 The Eilenberg-Moore Category of $\mathcal{P}_C$

From Ult, we can construct its Eilenberg-Moore category $\mathrm{Ult}^{\mathcal{P}}$, whose objects are algebras $(A, \alpha : \mathcal{P}_C(A) \to A)$, where A is a nonempty 1-bounded ultrametric space, and α satisfies $\alpha \circ \mathcal{P}_C(\alpha) = \alpha \circ \mu$ and $\alpha \circ \eta = id_A$.

The morphisms from (A, α) to (B, β) are maps $A \to B$ in the category of ultrametric spaces which commute with the action (so

[2] We have arranged our grammar of types so that every definable type is nonempty, ensuring that the non-emptiness side-condition always holds.

$$
\begin{array}{lcl}
zip^{i}_{\bullet} & : & \bullet^i(X) \times \bullet^i(Y) \to \bullet^i(X \times Y) \\
zip^{0}_{\bullet} & = & id_{X \times Y} \\
zip^{n+1}_{\bullet} & = & zip_{\bullet} \circ \bullet(zip^{n}_{\bullet}) \\
\\
zip^{i}_{\bullet} & : & \bullet^i(X) \times \bullet^i(Y) \to \bullet^i(X \times Y) \\
zip^{0}_{\bullet} & = & id_{X \times Y} \\
zip^{n+1}_{\bullet} & = & \bullet(zip^{n}_{\bullet}) \circ zip_{\bullet} \\
\\
\epsilon_i & : & \bullet^i(X \Rightarrow Y) \to \bullet^i(X) \Rightarrow \bullet^i(Y) \\
\epsilon_0 & = & id_{X \to Y} \\
\epsilon_{n+1} & = & \epsilon \circ \bullet(\epsilon_n) \\
\\
\epsilon_i^{-1} & : & \bullet^i(X) \Rightarrow \bullet^i(Y) \to \bullet^i(X \Rightarrow Y) \\
\epsilon_0^{-1} & = & id_{X \Rightarrow Y} \\
\epsilon_{n+1}^{-1} & = & \bullet(\epsilon_n) \circ \epsilon^{-1} \\
\\
\delta_i & : & A \to \bullet^i(X) \\
\delta_0 & = & id_A \\
\delta_{n+1} & = & \delta \circ \delta_n
\end{array}
$$

Figure 6. Definition of Iterated Morphisms

that $f \circ \alpha = \beta \circ \mathcal{P}_C(f)$). Identity and composition are inherited from the underlying category Ult. (As a notational aid to keeping the categories distinct, we will use functional composition $f \circ g$ when composing maps in Ult, and diagrammatic order $f; g$ when composing maps in $\mathrm{Ult}^{\mathcal{P}}$.)

Because $\mathcal{P}_C$ is a commutative strong monad, Ult has equalizers and coequalizers [22], and $\mathrm{Ult}^{\mathcal{P}}$ has coequalizers of reflexive pairs (since $\mathcal{P}_C$ preserves coequalizers of reflexive pairs), $\mathrm{Ult}^{\mathcal{P}}$ is symmetric monoidal closed [2, 14, 16]. The tensor product is defined using a coequalizer and the exponential by an equalizer.

We can lift the distribution of the next modality through tensor products and $\hat{zip}_{\bullet} : \bullet A \otimes \bullet B \simeq \bullet(A \otimes B)$ the monoidal exponential $\hat{\epsilon} : \bullet(A \multimap B) \simeq \bullet A \multimap \bullet B$, as well as the delay operator $\hat{\delta} : A \to \bullet A$. These operations simply inherit their structure from Ult, but for the operations to respect the algebra structure we need the isomorphism $\mathcal{P}_C^{\bullet} : \bullet \mathcal{P}_C(X) \simeq \mathcal{P}_C(\bullet\ X)$. This explains why we restricted $\mathcal{P}_C(X)$ to *nonempty* closed subsets of interpretation X — one leg of this isomorphism would fail to be nonexpansive due in the presence of the empty set, which is distance 1 from any other subset.

We also iterate all of these morphisms following the pattern of Figure 6.

The free $F(X) = (\mathcal{P}_C(X), \mu)$ and forgetful $G(A, \alpha) = A$ functors give rise to an adjunction between Ult and $\mathrm{Ult}^{\mathcal{P}}$, whose unit and counit we write $\eta_X : X \to G(F(X))$ and $\varepsilon_A : F(G(A, \alpha)) \to A$. The action of η as the unit of this adjunction is the same natural transformation as the unit of the monad $\mathcal{P}_C$, so no notational confusion can arise. The components of the counit ε are the algebra maps α of (A, α). This is a monoidal adjunction with a natural isomorphism $m : F(X) \otimes F(Y) \simeq F(X \times Y)$. This isomorphism lets us duplicate and discard F-typed values; we have maps $drop : F(X) \to \mathrm{I}$ and $dup : F(X) \to F(X) \otimes F(X)$.

4.4 Interpretation of the Programming Language

In Figure 7, we give the interpretation of the syntactic types in terms of metric spaces. The interpretation follows the categorical constructions – in each category, products go to products, exponentials go to exponentials, and the F and G adjunctions interpret the modal operators connecting the *F*unctional and *G*raphical sub-languages.

In Figure 8, we give the semantic interpretation of the syntax of our language. A context $\Gamma = x :_i X, \ldots, z :_k Z$ is interpreted as a product $\bullet^i[\![X]\!] \times \ldots \bullet^k [\![Z]\!]$, and a term-in-context $\Gamma \vdash e :_i X$ is interpreted as a morphism $[\![\Gamma]\!] \to \bullet^i[\![X]\!]$. A context $\Delta = a :_i A, \ldots, b :_j B$ is interpreted by the tensor product $\bullet^i[\![A]\!] \otimes \ldots \bullet^j [\![B]\!]$. The pair of contexts $\Gamma; \Delta$ is interpreted by $F([\![\Gamma]\!]) \otimes [\![\Delta]\!]$, and a linear term $\Gamma; \Delta \vdash t :_i A$ is interpreted as a morphism $F([\![\Gamma]\!]) \otimes [\![\Delta]\!] \to \bullet^i[\![A]\!]$ in $\mathrm{Ult}^{\mathcal{P}}$.

As an abuse of notation, we abbreviate $[\![\Gamma \vdash e :_i X]\!]$ as $[\![e]\!]$ when Γ, X and i are clear from context, and similarly we abbreviate $[\![\Gamma; \Delta \vdash t :_i A]\!]$ as $[\![t]\!]$.

Our semantics combines the interpretation of adjoint logic given in [1] with the guarded calculus given by Krishnaswami and Benton [17]. The main technical subtlety arose with the interpretation of the time indices. Concretely, consider the interpretation of $\Gamma \vdash G(t) :_i G(A)$, which we expect to be a map $[\![\Gamma]\!] \to \bullet^i[\![G(A)]\!]$, and contrast it with its subderivation $\Gamma; \cdot \vdash t :_i A$, whose interpretation (viewed as a map in Ult) is in $[\![\Gamma]\!] \to G(\bullet^i(A))$. To swap these constructors, we use the $\mathcal{P}_C^{\bullet} : \bullet \mathcal{P}_C(X) \simeq \mathcal{P}_C(\bullet\ X)$ isomorphism.

For readability, the semantics suppresses the associativity and commutativity maps needed to permute the context.

Theorem 2. *(Soundness of Substitution) Suppose $\Gamma \vdash e :_i X$ and $\Gamma; \Delta \vdash t :_i A$, and $i \leq j$. Then:*

- *If $\Gamma, x :_j X \vdash e' :_k Y$, then $[\![e']\!] \circ \langle \Gamma, \delta_X^{j-i} \circ [\![e]\!] \rangle = [\![[e/x]e']\!]$.*
- *If $\Gamma, x :_j X; \Delta \vdash t' :_k B$, then $F(\langle \Gamma, \delta_X^{j-i} \circ [\![e]\!] \rangle); [\![t]\!] = [\![[e/x]t']\!]$.*
- *If $\Gamma; \Delta', a :_j A \vdash t' :_k B$, then $dup_{F(\Gamma)}; (F(\Gamma) \otimes \Delta' \otimes [\![t]\!]); [\![t']\!] = [\![[t/a]t']\!]$.*

Proof. The proof is a routine induction. □

Theorem 3. *(Soundness of Equality Rules) For well-typed terms, the equational rules in Figure 2 are sound with respect to the denotational semantics.*

Proof. The proof is a routine verification. □

4.5 Denotational Semantics of GUI Operations

We have left out the constructors from the syntax to keep from cluttering the proof theory. We give the semantics of a representative collection of GUI operations (including all of the ones used in the calculator example) below:

$$
\begin{array}{lcl}
\mathsf{label} & : & F(S(\mathsf{string})) \to \mathsf{Window} \\
\mathsf{label}\ X & = & \{\lambda n.\ \mathsf{Label}(xs_n) \mid xs \in X\} \\
\\
\mathsf{vstack} & : & \mathsf{Window} \otimes \mathsf{Window} \to \mathsf{Window} \\
\mathsf{vstack}(W, W') & = & \mathsf{cl}(\{\lambda n.\ \mathsf{Vert}(w_n, w'_n) \mid w \in W, w' \in W'\}) \\
\\
\mathsf{hstack} & : & \mathsf{Window} \otimes \mathsf{Window} \to \mathsf{Window} \\
\mathsf{hstack}(W, W') & = & \mathsf{cl}(\{\lambda n.\ \mathsf{Hor}(w_n, w'_n) \mid w \in W, w' \in W'\}) \\
\\
\mathsf{button} & : & F(S(\mathsf{string})) \to \mathsf{Window} \otimes F(S(\mathsf{bool})) \\
\mathsf{button}\ X & = & (\{\lambda n.\ \mathsf{Button}(xs_n) \mid xs \in X\}, S(\mathsf{bool})) \\
\\
\mathsf{delete} & : & \mathsf{Window} \to I \\
\mathsf{delete}\ W & = & () \\
\end{array}
$$

Since the free functor F yields sets of possible values, we must define these constants so that they define appropriate outputs for all their possible inputs. The first label operation essentially just maps an operator over its input, and the two stacking operations take a Cartesian product to define the result for each pair of possibilities.

On the other hand, the button operation, which is an operation representing user input, does something a little more interesting. It returns a set of pairs of window values (constructed by mapping its input over the Button constructor), and a set of streams of booleans (denoting clicks). We define the windows values by mapping over

$$
\begin{array}{lcl}
1 & = & 1 \\
[\![X \times Y]\!] & = & [\![X]\!] \times [\![Y]\!] \\
[\![X \Rightarrow Y]\!] & = & [\![X]\!] \Rightarrow [\![Y]\!] \\
[\![S(X)]\!] & = & S([\![A]\!]) \\
[\![\bullet(X)]\!] & = & \bullet[\![X]\!] \\
[\![G(A)]\!] & = & G([\![A]\!]) \\
 & & \\
[\![I]\!] & = & 1 \\
[\![A \otimes B]\!] & = & [\![A]\!] \otimes [\![B]\!] \\
[\![A \multimap B]\!] & = & [\![A]\!] \multimap [\![B]\!] \\
[\![F(X)]\!] & = & F([\![X]\!]) \\
[\![\mathsf{Window}]\!] & = & F(S(\mathrm{Pic})) \\
 & & \\
F(X) & = & (\mathcal{P}_C(X), \mu) \\
G(A, \alpha) & = & A
\end{array}
$$

Figure 7. Interpretation of Types

the input, as with the other operations, but we return the full space of boolean streams as its second argument.

Therefore, if we receive a fixed stream of string inputs xs, we may receive *any* stream of clicks as a possible return value. So the semantics of button captures the inherent nondeterminism of user actions in a simple way — we simply say that the user can click a button whenever he or she likes, and our semantics has to account for all of these possibilities.

Finally, the delete operation simply throws away its argument. Using the semantics of this command, it is possible to prove program equalities such as the following:

```
1 let (F(x), v) = button() in |  let (F(y), w) = button()
2 let (F(y), w) = button() in |  in w
3 let () = delete v           |
4 in w                        |
```

In this program, we create two buttons and throw the first of them away, which is equivalent to not creating it at all. The proof follows as easily from the semantics of programs as it ought.

One fact worth noting is that our semantics does not rule out the possibility of two different buttons both yielding a click event (ie, returning true) on the same time step. We see essentially two possibilities for ruling out such behavior, both of which we rejected for this work.

If we wish to retain a high-level semantics, where we specify the semantics of the API in terms of events like clicks, rather than primitive events such as mouse movements and keystrokes, then positing a collection of constraints (such as "two buttons are never simultaneously clicked") seems to require explicitly modeling a store of input channels and maintaining these constraints as an invariant on the store. This is a bit ugly, and more than a little technically complicated.

A better approach, in our view, would be to model layout and event synthesis explicitly. In our semantics, we model the display as a time-varying tree, much like an HTML document. If the display model actually specified what the graphic layout of a GUI program was, and supplied the primitive mouse movements, keystrokes, and clicks as an input, then the semantics could explicitly give the functions explaining how to interpret primitive events as high-level ones — for example, we might define a "click" to occur when the mouse is pressed down inside the button area, and subsequently released, while still inside the button area.

We think this level of precision is likely the right approach to take when building new GUI toolkits. However, if we wish to bind to existing toolkits, it is better to give a semantics imprecise enough to tolerate wide variation in precisely how it is implemented.

$$
\begin{array}{lcl}
[\![\Gamma \vdash \bullet e :_i \bullet A]\!] & = & [\![e]\!] \\
[\![\Gamma \vdash \mathsf{cons}(e, e') :_i S(A)]\!] & = & \bullet^i(cons) \circ zip^i_\bullet \circ \langle [\![e]\!], [\![e']\!] \rangle \\
[\![\Gamma \vdash \lambda x.\ e :_i A \to B]\!] & = & \epsilon^i \circ \lambda([\![\Gamma, x :_i A \vdash e :_i B]\!]) \\
[\![\Gamma \vdash x_n :_j A]\!] & = & \delta_A^{j-i} \circ \pi_n, \text{if } x_n :_i A \in \Gamma \\
[\![\Gamma \vdash \mathsf{await}(e) :_{i+1} A]\!] & = & [\![e]\!] \\
[\![\Gamma \vdash \mathsf{head}\ e :_i A]\!] & = & \bullet^i(hd) \circ [\![t]\!] \\
[\![\Gamma \vdash \mathsf{tail}\ e :_{i+1} S(A)]\!] & = & \bullet^i(tl) \circ [\![t]\!] \\
[\![\Gamma \vdash e\ e' :_i B]\!] & = & \bullet^i(eval) \circ zip^i_\bullet \circ \langle [\![e]\!], [\![e']\!] \rangle \\
[\![\Gamma \vdash G(t) :_i G(A)]\!] & = & \mathcal{P}^{\bullet i}_C \circ G([\![t]\!]) \circ \eta_\Gamma \\
[\![\Gamma \vdash \mathsf{fix}\ x : A.\ e :_i A]\!] & = & \bullet^i(fix_A) \circ \epsilon^i \circ \lambda([\![e]\!]) \\
 & & \\
[\![\Gamma; a :_i A \vdash a :_j A]\!] & = & drop; \hat{\delta}_{j-i} \\
[\![\Gamma; \cdot \vdash () :_i \mathrm{I}]\!] & = & drop; \hat{\delta}_i \\
[\![\Gamma; \Delta, \Delta' \vdash \mathsf{let}\ () = t\ \mathsf{in}\ t' :_i B]\!] & = & dup_{F(\Gamma)} \otimes \Delta \otimes \Delta'; [\![t]\!]; [\![t']\!] \\
[\![\Gamma; \Delta, \Delta' \vdash (t, t') :_i A \otimes B]\!] & = & dup_{F(\Gamma)} \otimes \Delta \otimes \Delta'; \\
 & & ([\![t]\!] \otimes [\![t']\!]); \hat{zip}^i_\bullet \\
[\![\Gamma; \Delta, \Delta' \vdash \mathsf{let}\ (a, b) = t\ \mathsf{in}\ t' :_i C]\!] & = & dup_{F(\Gamma)} \otimes \Delta \otimes \Delta'; \\
 & & (F(\Gamma) \otimes \Delta' \otimes [\![t]\!]); [\![t']\!] \\
[\![\Gamma; \Delta \vdash \lambda a.\ t :_i A \multimap B]\!] & = & \lambda([\![\Gamma; \Delta, a :_i A \vdash t :_i B]\!]); \hat{\epsilon}^i \\
[\![\Gamma; \Delta, \Delta' \vdash t\ t' :_i B]\!] & = & dup_{F(\Gamma)} \otimes \Delta \otimes \Delta'; \\
 & & [\![t]\!] \otimes [\![t']\!]; \hat{zip}^i_\bullet; eval \\
[\![\Gamma; \cdot \vdash \mathsf{runG}(e) :_i A]\!] & = & F(\mathcal{P}^{\bullet i}_G \circ [\![\Gamma \vdash e :_i G(A)]\!]); \varepsilon \\
[\![\Gamma; \cdot \vdash F(e) :_i F(X)]\!] & = & F([\![e]\!]) \\
[\![\Gamma; \Delta, \Delta' \vdash \mathsf{let}\ F(x) = t\ \mathsf{in}\ t' :_i C]\!] & = & dup_{F(\Gamma)} \otimes \Delta \otimes \Delta'; \\
 & & F(\Gamma) \otimes \Delta' \otimes [\![t]\!]; \\
 & & m \otimes \Delta'; [\![t']\!]
\end{array}
$$

Figure 8. Denotational Interpretation of Terms

5. Proof Theory

We have established denotationally that the β and η equalities are valid, but this does not make clear just how extraordinarily well-behaved this calculus is. We can prove a stratified normalization result, despite the fact that this language contains a recursion operator of the form $\mathsf{fix}\ x : A.\ e$.

To state the normalization theorem precisely, we must first say what the normal forms are. To see that this is not an entirely trivial question, consider the term $e \triangleq \mathsf{fix}\ xs : S(\mathbb{N}).\ \mathsf{cons}(0, x)$. The term e is equivalent to $\mathsf{cons}(0, e)$, and to $\mathsf{cons}(0, \mathsf{cons}(0, e))$ and so on for infinitely many unfoldings. So we have to ask, which of these unfoldings count as normal forms, and which do not?

In Figure 9, we give normal and atomic forms of the calculus. The normal and atomic forms are a sub-syntax of the full language with the property that only beta-normal terms are grammatically valid. In this syntax, we count fixed points as atomic forms, which means we still need to fix a policy for when to allow fixed points, and when not.

Next, we introduce the idea of a n-normal form, which are normal forms in which fixed point subterms $\mathsf{fix}\ x : A.\ e$ only occur at times of n or greater. So $e \triangleq \mathsf{fix}\ xs : S(\mathbb{N}).\ \mathsf{cons}(0, x)$ is itself a 0-normal form, and $\mathsf{cons}(0, e)$ is a 1-normal form, and $\mathsf{cons}(0, \mathsf{cons}(0, e))$ is a 2-normal form, and so on. To specify n-normal forms, we use the judgments $\Gamma \vdash_n e :_i X$ and $\Gamma; \Delta \vdash_n t :_i A$. These typing rules exactly mirror the existing typing rules, with the exception of the UFIX' rule (given in Figure 9), where we add a side-condition that a fixed point is permitted only when the time index i at which it is typechecked is at least as large as n. Hence an n-normal form is a normal form which passes the n-restricted typechecking rule for fixed points.

So our example $e \triangleq \mathsf{fix}\ xs : S(\mathbb{N}).\ \mathsf{cons}(0, x)$ is a 0-normal form but not a 1-normal form, since it has no redexes, but does have a fixed point at time 0. However, its unfolding $\mathsf{cons}(0, e)$ is a 1-normal form, since e occurs in the tail position of a cons, which is checked at a time index 1 greater than the cons expression.

Now, we can show that every term has a normal form (i.e., a weak normalization result) in the following manner. Given two n-normal terms, we can define a *hereditary substitution* [27] which combines substitution and computing a normal form. For our type system, an appropriate functions are given in Figures 10, 11, and 12.

In each of these figures, we define two of six mutually recursive procedures, one substituting a normal form into a normal form, and the other substituting a normal form into a atomic term. The normal-normal substitution returns a new normal form, and the normal-atomic substitution returns a pair of a term and optionally a type. If the result of the substitution is still an atomic term, then no type is returned, and if it is a normal form, the type of the substitutand is returned. Below, we write $A \sqsubseteq X$ to say that the type A is a subterm of the type X.

Theorem 4. *(Hereditary Substitution) Suppose* $\Gamma \vdash_n \hat{e} :_i X$ *and* $\Gamma; \Delta \vdash_n \hat{t} :_i A$ *and* $i \leq j$*. Then*

- *If* $\Gamma, x :_j X \vdash_n \hat{e}' :_k Y$ *then* $\Gamma \vdash_n \langle \hat{e}/x \rangle_X \hat{e}' :_k Y$
- *If* $\Gamma, x :_j X; \Delta \vdash_n \hat{t}' :_k B$ *then* $\Gamma; \Delta \vdash_n \langle \hat{e}/x \rangle_X \hat{t}' :_k B$
- *If* $\Gamma; \Delta', a :_j A \vdash_n \hat{t}' :_k B$ *then* $\Gamma; \Delta, \Delta' \vdash_n \left\langle \hat{t}/a \right\rangle_A \hat{t}' :_k B$
- *If* $\Gamma, x :_j X \vdash_n g :_k Y$ *then either*
  - $\langle \hat{e}/x \rangle_X g = (\hat{e}', Y)$ *and* $Y \sqsubseteq X$ *and* $\Gamma \vdash_n \hat{e}' :_k Y$
  - *or* $\langle \hat{e}/x \rangle_X g = (g', \mathsf{n/a})$ *and* $\Gamma \vdash_n g' :_k Y$
- *If* $\Gamma, x :_j X; \Delta \vdash_n u :_k B$ *then either*
  - $\langle \hat{e}/x \rangle_X u = (\hat{t}', B)$ *and* $B \sqsubseteq X$ *and* $\Gamma; \Delta \vdash_n \hat{t}' :_k B$
  - *or* $\langle \hat{e}/x \rangle_X u = (u', \mathsf{n/a})$ *and* $\Gamma; \Delta \vdash_n u' :_k B$
- *If* $\Gamma; \Delta', a :_j A \vdash_n u :_k B$ *then either*
  - $\left\langle \hat{t}/a \right\rangle_A u = (\hat{t}', B)$ *and* $B \sqsubseteq A$ *and* $\Gamma; \Delta, \Delta' \vdash_n \hat{t}' :_k B$
  - *or* $\left\langle \hat{t}/a \right\rangle_A u = (u', \mathsf{n/a})$ *and* $\Gamma; \Delta, \Delta' \vdash_n u' :_k B$

Furthermore, hereditary substitution is $\beta\eta$*-equivalent to ordinary substitution.*

The statement of the theorem is a bit complicated, since we have 6 cases, depending on which judgement and context we are substituting into, and whether we are substituting into a normal or atomic form. Luckily, the induction is straightforward, and the algorithm offers no surprises. The induction is lexicographically on the type of the term being substituted, together with the unordered product of the sizes of the two derivations of the substitutand and subtitutee.

Note that hereditary substitution never needs to unroll fixed points. By hypothesis, neither the substitutee nor the substitutand contain fixed point terms at an index less than n, and as a result, the process of substitution and normalization will never create a fixed point at the head of a term at time less than n.

As a result, we can prove an unrolling lemma separately, *without* any mutual recursion with the normalization algorithm. In Figure 13, we define an unfolding operation $\lceil e \rceil$ which unrolls all the fixed points in a term by one step, and prove the following theorem:

Theorem 5. *(Unrolling)*

- *If* $\Gamma \vdash_n e :_i X$*, then* $\Gamma \vdash_{n+1} \lceil e \rceil :_i X$
- *If* $\Gamma; \Delta \vdash_n t :_i X$*, then* $\Gamma; \Delta \vdash_{n+1} \lceil t \rceil :_i X$

Proof. The result follows by a routine induction on the derivation. □

Now we can combine these two theorems to prove a normalization theorem.

Theorem 6. *(Weak Normalization)*

- *Suppose* $\Gamma \vdash e :_i X$*. Then for any* m*, there is an* m*-normal form* e'*, such that* $\Gamma \vdash_m e' :_i X$*, which is* $\beta\eta$*-equivalent to* e*.*

$$
\begin{array}{lrcl}
\text{Normal Nonlinear} & \hat{e} & ::= & () \mid (\hat{e}, \hat{e}') \mid \lambda x.\ \hat{e} \\
& & \mid & \mathsf{cons}(\hat{e}, \hat{e}') \mid G(\hat{t}) \mid \bullet\,(\hat{e}) \mid g \\
\text{Atomic Nonlinear} & g & ::= & x \mid g\ \hat{e} \mid \mathsf{fst}\ g \mid \mathsf{snd}\ g \mid \mathsf{await}(g) \\
& & \mid & \mathsf{head}\ g \mid \mathsf{tail}\ g \mid \mathsf{fix}\ x : X.\ \hat{e} \\
\text{Normal Linear} & \hat{t} & ::= & () \mid (\hat{t}, \hat{t}') \mid \lambda x.\ \hat{t} \\
& & \mid & \mathsf{let}\ () = u\ \mathsf{in}\ \hat{t} \mid \mathsf{let}\ (a, b) = u\ \mathsf{in}\ \hat{t} \\
& & \mid & F(\hat{e}) \mid \mathsf{let}\ F(x) = u\ \mathsf{in}\ \hat{t} \mid u \\
\text{Atomic Linear} & u & ::= & a \mid u\ \hat{t} \mid \mathsf{runG}(g)
\end{array}
$$

$$\boxed{\Gamma \vdash_n e :_i X} \qquad \boxed{\Gamma; \Delta \vdash_n t :_i A}$$

$$\frac{\Gamma, x :_{i+1} X \vdash_n e :_i X \qquad i \geq n}{\Gamma \vdash_n \mathsf{fix}\ x : A.\ e :_i X}\ \textsc{UFix'}$$

(all other rules unchanged except for the appearance of n)

Figure 9. Normal and Atomic Forms, and Leveled Fixed Points

$$
\begin{array}{lcl}
\langle \hat{e}/x \rangle_X\ () & = & () \\
\langle \hat{e}/x \rangle_X\ (\hat{e}', \hat{e}'') & = & (\langle \hat{e}/x \rangle_X\ \hat{e}', \langle \hat{e}/x \rangle_X\ \hat{e}'') \\
\langle \hat{e}/x \rangle_X\ \lambda y.\ \hat{e}' & = & \lambda y.\ \langle \hat{e}/x \rangle_X\ \hat{e}' \\
\langle \hat{e}/x \rangle_X\ \bullet \hat{e}' & = & \bullet(\langle \hat{e}/x \rangle_X\ \hat{e}') \\
\langle \hat{e}/x \rangle_X\ \mathsf{cons}(\hat{e}', \hat{e}'') & = & \mathsf{cons}(\langle \hat{e}/x \rangle_X\ \hat{e}', \langle \hat{e}/x \rangle_X\ \hat{e}'') \\
\langle \hat{e}/x \rangle_X\ G(t) & = & G(\langle \hat{e}/x \rangle_X\ t) \\
\langle \hat{e}/x \rangle_X\ g & = & \begin{cases} g' & \text{when } \langle \hat{e}/x \rangle_X\ g = (g', \mathsf{n/a}) \\ \hat{e}' & \text{when } \langle \hat{e}/x \rangle_X\ g = (\hat{e}', Y) \end{cases}
\end{array}
$$

$$
\begin{array}{lcl}
\langle \hat{e}/x \rangle_X\ x & = & (x, X) \\
\langle \hat{e}/x \rangle_X\ y & = & (y, \mathsf{n/a}) \\
\langle \hat{e}/x \rangle_X\ \mathsf{fix}\ y : Y.\ \hat{e}' & = & (\mathsf{fix}\ y : Y.\ (\langle \hat{e}/x \rangle_X\ \hat{e}'), Y) \\
\langle \hat{e}/x \rangle_X\ g_1\ \hat{e}_2 & = & \begin{array}{l} \mathsf{let}\ \hat{e}_2' = \langle \hat{e}/x \rangle_X\ \hat{e}_2\ \mathsf{in} \\ \mathsf{case}\ \langle \hat{e}/x \rangle_X\ g\ \mathsf{of} \\ \quad (g', \mathsf{n/a}) \rightarrow (g'\ \hat{e}_2', \mathsf{n/a}) \\ \quad (\lambda y.\ \hat{e}_1', Y \Rightarrow Z) \rightarrow \langle \hat{e}_2'/y \rangle_Y\ \hat{e}_1' \end{array} \\
\langle \hat{e}/x \rangle_X\ \mathsf{head}\ g & = & \begin{array}{l} \mathsf{case}\ \langle \hat{e}/x \rangle_X\ g\ \mathsf{of} \\ \quad (g', \mathsf{n/a}) \rightarrow (\mathsf{head}\ g', \mathsf{n/a}) \\ \quad (\mathsf{cons}(\hat{e}_1, \hat{e}_2), S(Y)) \rightarrow (\hat{e}_1, Y) \end{array} \\
\langle \hat{e}/x \rangle_X\ \mathsf{tail}\ g & = & \begin{array}{l} \mathsf{case}\ \langle \hat{e}/x \rangle_X\ g\ \mathsf{of} \\ \quad (g', \mathsf{n/a}) \rightarrow (\mathsf{tail}\ g', \mathsf{n/a}) \\ \quad (\mathsf{cons}(\hat{e}_1, \hat{e}_2), S(Y)) \rightarrow (\hat{e}_2, S(Y)) \end{array} \\
\langle \hat{e}/x \rangle_X\ \mathsf{fst}\ g & = & \begin{array}{l} \mathsf{case}\ \langle \hat{e}/x \rangle_X\ g\ \mathsf{of} \\ \quad (g', \mathsf{n/a}) \rightarrow (\mathsf{fst}\ g', \mathsf{n/a}) \\ \quad ((\hat{e}_1, \hat{e}_2), Y \times Z) \rightarrow (\hat{e}_2, Y) \end{array} \\
\langle \hat{e}/x \rangle_X\ \mathsf{snd}\ g & = & \begin{array}{l} \mathsf{case}\ \langle \hat{e}/x \rangle_X\ g\ \mathsf{of} \\ \quad (g', \mathsf{n/a}) \rightarrow (\mathsf{snd}\ g', \mathsf{n/a}) \\ \quad ((\hat{e}_1, \hat{e}_2), Y \times Z) \rightarrow (\hat{e}_2, Z) \end{array} \\
\langle \hat{e}/x \rangle_X\ \mathsf{await}(g) & = & \begin{array}{l} \mathsf{case}\ \langle \hat{e}/x \rangle_X\ g\ \mathsf{of} \\ \quad (g', \mathsf{n/a}) \rightarrow (\mathsf{await}(g'), \mathsf{n/a}) \\ \quad (\bullet \hat{e}', \bullet Y) \rightarrow (\hat{e}', Y) \end{array}
\end{array}
$$

Figure 10. Substituting Nonlinear Terms into Nonlinear Terms

- *Suppose* $\Gamma; \Delta \vdash t :_i A$*. Then for any* n*, there is an* n*-normal form* t'*, such that* $\Gamma; \Delta \vdash_n t' :_i A$*, which is* $\beta\eta$*-equivalent to* t*.*

To prove this, note than any well-typed term satisfies the UFix' rule for $n = 0$. Hence we can use the unfolding lemma n times to get a term which has no fixed points at times earlier than n. Then we can find a normal form, by inductively walking down the syntax and using hereditary substitution to eliminate reducible expressions.

$$
\begin{array}{lcl}
\langle \hat{e}/x \rangle_X\ () & = & () \\
\langle \hat{e}/x \rangle_X\ (t', t'') & = & (\langle \hat{e}/x \rangle_X\ t', \langle \hat{e}/x \rangle_X\ t'') \\
\langle \hat{e}/x \rangle_X\ \lambda y.\ t' & = & \lambda y.\ \langle \hat{e}/x \rangle_X\ t' \\
\langle \hat{e}/x \rangle_X\ F(\hat{e}') & = & F(\langle \hat{e}/x \rangle_X\ \hat{e}') \\
\langle \hat{e}/x \rangle_X\ \mathsf{let}\ () = u\ \mathsf{in}\ t & = & \begin{array}{l} \mathsf{case}\ \langle \hat{e}/x \rangle_X\ u\ \mathsf{of} \\ \quad (u', \mathsf{n/a}) \ \to\ \mathsf{let}\ () = u'\ \mathsf{in}\ \langle \hat{e}/x \rangle_X\ t \\ \quad ((), I) \ \to\ \langle \hat{e}/x \rangle_X\ t \end{array} \\
\langle \hat{e}/x \rangle_X\ \mathsf{let}\ (a, b) = u\ \mathsf{in}\ t & = & \begin{array}{l} \mathsf{case}\ \langle \hat{e}/x \rangle_X\ u\ \mathsf{of} \\ \quad (u', \mathsf{n/a}) \ \to \\ \qquad \mathsf{let}\ (x, y) = u'\ \mathsf{in}\ \langle \hat{e}/x \rangle_X\ t \\ \quad ((t_1, t_2), A \otimes B) \ \to \\ \qquad \langle t_2/b \rangle_B\ (\langle t_1/a \rangle_A\ (\langle \hat{e}/x \rangle_X\ t)) \end{array} \\
\langle \hat{e}/x \rangle_X\ \mathsf{let}\ F(x) = u\ \mathsf{in}\ t & = & \begin{array}{l} \mathsf{case}\ \langle \hat{e}/x \rangle_X\ u\ \mathsf{of} \\ \quad (u', \mathsf{n/a}) \ \to \\ \qquad \mathsf{let}\ () = u'\ \mathsf{in}\ \langle \hat{e}/x \rangle_X\ t \\ \quad (F(\hat{e}'), F(Y)) \ \to \\ \qquad \langle \hat{e}'/y \rangle_Y\ (\langle \hat{e}/x \rangle_X\ t) \end{array} \\
\langle \hat{e}/x \rangle_X\ u & = & \begin{cases} u' & \text{when } \langle \hat{e}/x \rangle_X\ u = (u', \mathsf{n/a}) \\ t' & \text{when } \langle \hat{e}/x \rangle_X\ u = (t', A) \end{cases}
\end{array}
$$

$$
\begin{array}{lcl}
\langle \hat{e}/x \rangle_X\ a & = & (a, \mathsf{n/a}) \\
\langle \hat{e}/x \rangle_X\ \mathsf{runG}(g) & = & \begin{array}{l} \mathsf{case}\ \langle \hat{e}/x \rangle_X\ g\ \mathsf{of} \\ \quad (g', \mathsf{n/a}) \ \to\ (\mathsf{runG}(g'), \mathsf{n/a}) \\ \quad (G(t), G(A)) \ \to\ (t, A) \end{array} \\
\langle \hat{e}/x \rangle_X\ u_1\ t_2 & = & \begin{array}{l} \mathsf{let}\ t_2' = \langle \hat{e}/x \rangle_X\ t_2\ \mathsf{in} \\ \mathsf{case}\ \langle \hat{e}/x \rangle_X\ u_1\ \mathsf{of} \\ \quad (u_1', \mathsf{n/a}) \ \to\ (u_1'\ \hat{e}_2', \mathsf{n/a}) \\ \quad (\lambda a.\ t_1', A \multimap B) \ \to\ \langle t_2'/a \rangle_A\ t_1' \end{array}
\end{array}
$$

Figure 11. Substituting Nonlinear Terms into Linear Terms

$$
\begin{array}{lcl}
\langle t/a \rangle_A\ () & = & () \\
\langle t/a \rangle_A\ (t', t'') & = & (\langle t/a \rangle_A\ t', \langle t/a \rangle_A\ t'') \\
\langle t/a \rangle_A\ \lambda y.\ t' & = & \lambda y.\ \langle t/a \rangle_A\ t' \\
\langle t/a \rangle_A\ F(\hat{e}') & = & F(\hat{e}') \\
\langle t/a \rangle_A\ \mathsf{let}\ () = u_1\ \mathsf{in}\ t_2 & = & \begin{array}{l} \mathsf{case}\ \langle t/a \rangle_A\ u_1\ \mathsf{of} \\ \quad (u_1', \mathsf{n/a}) \to \\ \qquad \mathsf{let}\ () = u_1'\ \mathsf{in}\ \langle t/a \rangle_A\ t_2 \\ \quad ((), I) \to \langle t/a \rangle_A\ t_2 \end{array} \\
\langle t/a \rangle_A\ \mathsf{let}\ (b, c) = u_1\ \mathsf{in}\ t_2 & = & \begin{array}{l} \mathsf{case}\ \langle t/a \rangle_A\ u_1\ \mathsf{of} \\ \quad (u_1', \mathsf{n/a}) \ \to \\ \qquad \mathsf{let}\ (x, y) = u_1'\ \mathsf{in}\ \langle t/a \rangle_A\ t_2 \\ \quad ((t_3, t_4), B \otimes C) \ \to \\ \qquad \langle t_4/c \rangle_C\ (\langle t_3/b \rangle_B\ (\langle t/a \rangle_A\ t_2)) \end{array} \\
\langle t/a \rangle_A\ \mathsf{let}\ F(x) = u_1\ \mathsf{in}\ t_2 & = & \begin{array}{l} \mathsf{case}\ \langle t/a \rangle_A\ u_1\ \mathsf{of} \\ \quad (u_1', \mathsf{n/a}) \ \to \\ \qquad \mathsf{let}\ () = u_1'\ \mathsf{in}\ \langle t/a \rangle_A\ t_2 \\ \quad (F(\hat{e}_1'), F(Y)) \ \to \\ \qquad \langle \hat{e}_1'/y \rangle_Y\ (\langle t/a \rangle_A\ t_2) \end{array} \\
\langle t/a \rangle_A\ u_1 & = & \begin{cases} u_1' & \text{when } \langle t/a \rangle_A\ u_1 = u_1' \\ t_1' & \text{when } \langle t/a \rangle_A\ u_1 = (t_1', B) \end{cases}
\end{array}
$$

$$
\begin{array}{lcl}
\langle t/a \rangle_A\ a & = & (t, A) \\
\langle t/a \rangle_A\ b & = & (b, \mathsf{n/a}) \\
\langle t/a \rangle_A\ \mathsf{runG}(g) & = & (\mathsf{runG}(g), \mathsf{n/a}) \\
\langle t/a \rangle_A\ u_1\ t_2 & = & \begin{array}{l} \mathsf{let}\ t_2' = \langle t/a \rangle_A\ t_2\ \mathsf{in} \\ \mathsf{case}\ \langle t/a \rangle_A\ u_1\ \mathsf{of} \\ \quad (u_1', \mathsf{n/a}) \ \to\ (u_1'\ \hat{e}_2', \mathsf{n/a}) \\ \quad (\lambda b.\ t_1', B \multimap C) \ \to\ \langle t_2'/b \rangle_B\ t_1' \end{array}
\end{array}
$$

Figure 12. Substituting Linear Terms into Linear Terms

$$
\begin{array}{lcl}
\lceil () \rceil & = & \lceil () \rceil \\
\lceil (e, e') \rceil & = & (\lceil e \rceil, \lceil e' \rceil) \\
\lceil \lambda x.\ e \rceil & = & \lambda x.\ \lceil e \rceil \\
\lceil \mathsf{cons}(e, e') \rceil & = & \mathsf{cons}(\lceil e \rceil, \lceil e' \rceil) \\
\lceil \bullet(e) \rceil & = & \bullet(\lceil e \rceil) \\
\lceil G(t) \rceil & = & G(\lceil t \rceil) \\
\lceil x \rceil & = & x \\
\lceil e\ e' \rceil & = & \lceil e \rceil\ \lceil e' \rceil \\
\lceil \mathsf{fst}\ e \rceil & = & \mathsf{fst}\ \lceil e \rceil \\
\lceil \mathsf{snd}\ e \rceil & = & \mathsf{snd}\ \lceil e \rceil \\
\lceil \mathsf{head}\ e \rceil & = & \mathsf{head}\ \lceil e \rceil \\
\lceil \mathsf{tail}\ e \rceil & = & \mathsf{tail}\ \lceil e \rceil \\
\lceil \mathsf{await}(e) \rceil & = & \mathsf{await}(\lceil e \rceil) \\
\lceil \mathsf{fix}\ x : A.\ e \rceil & = & [\mathsf{fix}\ x : A.\ \lceil e \rceil / x]\ \lceil e \rceil
\end{array}
$$

$$
\begin{array}{lcl}
\lceil () \rceil & = & \lceil () \rceil \\
\lceil (t, t') \rceil & = & (\lceil t \rceil, \lceil t' \rceil) \\
\lceil \lambda a.\ t \rceil & = & \lambda a.\ \lceil t \rceil \\
\lceil F(e) \rceil & = & F(\lceil e \rceil) \\
\lceil a \rceil & = & a \\
\lceil t\ t' \rceil & = & \lceil t \rceil\ \lceil t' \rceil \\
\lceil \mathsf{let}\ () = t\ \mathsf{in}\ t' \rceil & = & \mathsf{let}\ () = \lceil t \rceil\ \mathsf{in}\ \lceil t' \rceil \\
\lceil \mathsf{let}\ (a, b) = t\ \mathsf{in}\ t' \rceil & = & \mathsf{let}\ (a, b) = \lceil t \rceil\ \mathsf{in}\ \lceil t' \rceil \\
\lceil \mathsf{let}\ F(x) = t\ \mathsf{in}\ t' \rceil & = & \mathsf{let}\ F(x) = \lceil t \rceil\ \mathsf{in}\ \lceil t' \rceil \\
\lceil \mathsf{runG}(e) \rceil & = & \mathsf{runG}(\lceil e \rceil)
\end{array}
$$

Figure 13. Unfolding

6. Implementation

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

Then, we implement reactive programs with a dataflow graph, which runs inside an event loop. Instead of representing streams as lazy sequences of elements, we represent our streams with mutable dataflow cells, which enumerate the values of the stream as they evolve over time. The event loop updates a clock cell to notify the cells in the graph that they may need to recompute themselves, so that each one reads the cells it depends on, doing the minimal amount of computation needed at each time step. The details of our implementation strategy are in our earlier paper [17], including a correctness proof for the purely functional fragment.

In this paper, we content ourselves with a brief informal discussion of how we extended that implementation to handle linear types. In our earlier paper, we implemented a family of abstract types representing the morphisms of the category of ultrametric spaces, and then wrote an Ocaml syntax extension which translated our DSL's lambda-calculus into calls to the procedures which called the combinators in question. We extend this implementation strategy continues to the adjoint calculus: now we have two families of abstract types, with one each for the nonlinear side and the linear side.

We represent the type of windows with widget objects from the GUI toolkit. As we mentioned in the introduction, we are exploiting linearity to double effect. At a low level, we use it to enforce the linear use of widgets in the scene graph, even though the high-level denotational semantics never mentions state, generativity or mutation at all – the semantics of windows is just a pure set of stream of Pic.

We have not yet done a correctness proof of the implementation, though this is an obvious next step. In our earlier correctness proof, we proved the adequacy of our denotational semantics with respect to an imperative implementation, by means of a Kripke logical relation which related the state of the dataflow graph to the meaning of the streams in the program. Our implementation was highly imperative to start with, and we expect that the Kripke structure should extend naturally to let us use imperative widget state to model the meanings of the widgets. Essentially, since we only ask the equations of the linear lambda calculus to hold, we ought to be able to realize Window values with mutable state.

On the other hand, specifying our contract with the GUI toolkit will be more challenging. The basic issue is to find a specification of the behavior of the toolkit precise enough to let us prove the correctness of the library, while still being abstract enough that we are not forced to verify large parts of the implementation of the underlying toolkit. A usable Hoare-style specification for an imperative toolkit API could well be a result of independent interest! It may involve ideas from process calculi, since low-level APIs such as Win32 or GTK are often designed with logically concurrent and asynchronous abstractions in mind (even though the actual implementations are usually sequential).

7. Discussion

In our earlier paper [17], we 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 non-expansiveness 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.

Metric methods entered semantics in the early 1980s, to simplify the denotational semantics of concurrency [10]. The applications to stream programming were recognized early, but not followed up on: in a surprisingly little-cited 1985 paper [9], 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[...]".

Birkedal *et al.* [3] have recently given an ultrametric model of Nakano's [19] calculus for guarded recursion. Nakano's types are very similar to ours (though he supports full recursive types), making use of a delay type to guard recursive definition. However, his system includes both subtyping and rules (such as $(\bullet)$) whose application does not affect the subject term, but which we wished to record for operational reasons. His correctness proof relied on a logical relation, whereas in this paper we have been able to prove normalization for a language including the recursion operator $\mathsf{fix}\ x : X.\ e$ using only ordinary induction — as the proof of Banach's theorem hints we ought, since it only uses induction up to ω to find fixed points.

A suggestive paper of Escardo's [13] gives a metric model to PCF. He noticed that ultrametric spaces support a version of the lift monad from domain theory, and that the extra structure of metrics relative to domains means that the lift monad can be used to interpret timeouts. Since cancels and interrupt operations pervade interactive programs, this suggests a direction of investigation for learning how to support these operations without disturbing the reasoning principles of the language.

Recently, Birkedal et al. [4] have given an alternative model of guarded recursion, based on the topos of trees. Their construction forms a topos, and so many of the semantic constructions we performed with metric spaces can be replayed as set-theoretic operations within the internal language of their topos. In particular, we found the calculations involving powerspaces rather difficult, and so the possibility of working with ordinary powersets (albeit in a slightly different mathematical universe) seems quite attractive.

Pouzet and Caspi [5] 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 [24] 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 problem is that these categories contain too few global points (maps $S(1) \to S(\mathbb{N})$) to denote very many streams, including such basic ones such as $(1, 2, 3, 6, 24, \ldots)$.) Still, implicit lifting is a very attractive notation for simplifying writing stream programs, and there is an adjunction between the category of ultrametric spaces and the co-Kleisli category of the stream functor, which would makes it easy to return to Ult to take fixed points.

The original work on functional reactive programming [12] was based on writing reactive programs as unrestricted stream programs, but due to problems with causality, variations such as *arrowized FRP* [20] were introduced to give combinators restricting the definable stream transformers to the causal ones. The restriction to arrows is roughly equivalent to first-order functional programming, though Sculthorpe and Nilsson [23] introduced additional combinators to recover higher-order and dynamic behavior while using dependent types to retain causality. In contrast, Liu, Cheng and Hudak [18] showed how to exploit this first-order structure to develop a very efficient compilation scheme (reminiscent of the Bohm-Jacopini theorem) from arrow programs to single-loop code.

A distinctive feature of the original FRP work is its focus on continuous time. Though it does not yet do so, we hope our proof framework can extend to proving a sampling theorem, as in Wan and Hudak [26]. On the semantic side, we can easily model continuous behaviors as functions of type $\mathbb{R} \to A$, but relating it to an implementation delivering time deltas (instead of ticks, as presently) seems much more challenging.

Cooper and Krishnamurthi [7, 8], describe FrTime, an embedding of functional reactive programming into the PLT Scheme (now Racket) implementation. They integrate a language with full (i.e., non-commutative) effects, which rules out an adjoint-logic style as we have done here. However, it might be possible to use Egger *et al.*'s enriched effect calculus [11] to support both general effects and strong reasoning principles.

References

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

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

[3] L. Birkedal, J. Schwinghammer, and K. Støvring. A metric model of guarded recursion. In *FICS*, 2010.

[4] L. Birkedal, R. E. Møgelberg, J. Schwinghammer, and K. Støvring. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. In *LICS*, 2011.

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

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

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

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

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

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

[11] J. Egger, R. E. Møgelberg, and A. Simpson. Enriching an effect calculus with linear types. In E. Grädel and R. Kahle, editors, *CSL*, volume 5771 of *Lecture Notes in Computer Science*, pages 240–254. Springer, 2009. ISBN 978-3-642-04026-9.

[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] M. P. Fiore and G. D. Plotkin. An extension of models of axiomatic domain theory to models of synthetic domain theory. In *Proceedings of Computer Science Logic (CSL)*, volume 1258 of *Lecture Notes in Computer Science*. Springer, 1996.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

[16] A. Kock. Closed categories generated by commutative monads. *Journal of the Australian Mathematical Society*, 12, 1971.

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

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

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

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

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

[22] J. J. M. M. Rutten. Elements of generalized ultrametric domain theory. *Theor. Comput. Sci.*, 170(1-2):349–381, 1996.

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

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

[25] S. Vickers. Localic completion of generalized metric spaces II: Powerlocales. *Theory and Applications of Categories*, 14(15):328–356, 2005. ISSN 1201-561X.

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

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