Linear Temporal Type Theory for Event-based Reactive Programming

Jennifer Paykin
University of Pennsylvania
jpaykin@seas.upenn.edu

Neelakantan R. Krishnaswami
University of Birmingham
n.krishnaswami@cs.bham.ac.uk

Steve Zdancewic
University of Pennsylvania
stevez@cis.upenn.edu

Abstract

Inspired by the Curry-Howard correspondence between Functional Reactive Programming (FRP) and Linear-time Temporal Logic (LTL), this paper applies temporal types to describe a functional, event-driven language with an underlying event queue. The result is a compelling abstraction for linear computations with temporal types. We present a toy surface language for event-based GUIs and give some examples of how to program in temporal style. We then define a double-negation translation to adjoint tensor logic, which turns computations into event handlers. Using the structure of the logic we define a step-indexed Kripke logical relation on machine configurations, and prove soundness guarantees about the resulting states.

1. Introduction

Event-driven reactive programming uses two fundamental abstractions: the *event* and the *callback*. In GUI programming, for example, events are generated by user behavior when a button is clicked or a key is pressed. When such an event occurs, the *event loop* triggers *callbacks*, or event handlers, that execute some code in response to the event. The event-based style of programming is efficient, flexible, and pervasive in real-world GUI programs—see, for instance, the Promises/A+ library for JavaScript or the ReactiveX platform that has been ported to many mainstream languages.

However, the unfettered use of callbacks can lead to erroneous programs. Callback-driven code can be obtuse because event handlers only communicate with the rest of the system using state. Moreover, because implementations need to synchronize these stateful event handlers when processing events in parallel, they can suffer from concurrency errors, including dropped events, data races, and starvation.

Functional Reactive Programming (FRP) is an alternative approach to the event-driven style. FRP programs are purely functional, which alleviates the problems of synchronizing state with behavior. In addition, FRP is known to arise as an instance of Linear-time Temporal Logic (LTL) through the Curry-Howard isomorphism [6, 7]. This strong connection to logic has suggested refinements of the FRP type structure that eliminate common errors such as space leaks and causality violations [8].

In this paper, we analogously extend LTL into a type theory for safer event-driven reactive programming. It is already well-known that event-driven reactive programs can be expressed in a monadic style, which alleviates some of the problems associated with synchronization. The "deferred" type from OCaml's Async library [11, Chapter 18], the implementation of futures in Scala [5], and the yield operator in Narrative JavaScript [12] are all examples of monadic event-driven programming abstractions. Still, due to the underlying use of callbacks, these libraries are themselves quite difficult to implement and reason about. We present a monadic style of functional programming augmented with a temporal and linear type system, show how it admits an efficient implementation using asynchronous event queues, and prove that this implementation satisfies safety properties. The observations in this work uncover elegant parallels between the event-driven and the functional reactive styles.

After briefly recounting the relevant background about linear temporal logic and its connection to monadic programming, we move on to the main contributions:

Monadic programming. Section 3 presents examples of the programming model. Though the ideas in this paper apply to any event-driven system, we make them concrete via a toy language with primitives for GUI programming. We describe the temporal type system in Section 4.

Synchronization. We address the question of how to interpret the "linear time" aspect of LTL in a computational setting. We show that it corresponds directly to the ability for programs to safely observe the total ordering on events by "selecting" the first event to occur among several possibilities. We can think of this as allowing "safe" races in a concurrent program, the results of which are resolved nondeterministically, but whose computations are still guaranteed to be well-behaved. While most monadic event libraries support some form of this `select` primitive, our contribution is to show how it can be derived from the proof-theoretic interpretation of LTL. Section 5 describes our implementation of `select` in detail.

Continuation passing style. Section 5 describes how the surface language event primitives could be implemented by giving a double negation/CPS translation into a stateful target language. We ascribe logical structure to the CPS translation by introducing a novel intermediate *adjoint tensor logic*. The logic explicitly equates logical negation with continuations, and distinguishes temporal judgments (which are valid at any point in the future) from linear judgments (which are only valid at the current point in time). The relationships among the linear, temporal, and persistent fragments are explained via adjunctions that give rise to the monadic structure. We prove

a cut elimination property for then logic, which in a strong sense says that our `select` operator correctly and completely captures the reasoning principles justified by LTL.

Safety. Using the structure of the adjoint tensor logic, we define a step-indexed Kripke logical relation in Section 7. The target is a stateful, untyped lambda-calculus with an explicit event queue. The behavior of program states is asynchronous, in that events can be handled in any order, but single-threaded, which eliminates issues with race conditions. The realizability proofs imply type safety, which establishes in particular that the proposed implementation of `select` is free from synchronization bugs.

2. Background

Modal and temporal logic. When viewed through the lens of the Curry-Howard correspondence, modal logics give rise to type systems for computations inside of different "worlds." The interpretation of "world" depends on the setting at hand [15], including stages of compilation [3], security levels [2], and nodes in distributed systems [13], among others.

Linear-time temporal logic (LTL) [16] is a classical modal logic for reasoning over time. The set of worlds can be viewed as snapshots in time arranged in a total (linear) order. The modalities of the logic capture the time-dependent nature of assertions—just because a proposition is true today doesn't mean it will be true tomorrow. The proposition $\Box\, A$, read "always A," indicates that A is true both now and at every point in the future. The proposition $\diamond\, A$, read "eventually A," says that there is some point, now or in the future, at which A is true. In classical temporal logic, $\Box$ and $\diamond$ are De Morgan duals, which means that $\diamond\, A \equiv \neg\, \Box\, \neg\, A$.

From a computational point of view, there is some leeway in how to ascribe semantics to these temporal modal operators. One sensible way, adopted in functional reactive programming, is to think of the "always" modality as providing a stream of values over time. This makes "always A" isomorphic to a function of type $t \to A$, where t ranges over times (that is, worlds). This design leads to the abstraction **Signal** A of time-varying values of type A, and an emphasis on programming by stream transducers. However, in this setting the dual modality, "eventually A" is somewhat inefficient: a computation $\diamond\, A$ must be "polled" on every clock tick to see whether it is ready to yield a value.

Classical linear logic, callbacks, and continuations. In this paper, we show that another sensible way to ascribe semantics to the modal operators leads to a clean interpretation of event-driven programming, in two parts. First, rather than interpreting "always A" as a stream of values, we instead interpret it as a (non time-varying) value that is stable over time—it is available for use at any point in the future. Second, we interpret it as a *resource* in the sense of Girard's *linear logic* [4].[1] Linearity means that a temporal object must be used exactly once and cannot be duplicated. The intuition is that each event is a unique occurrence of some stateful action (such as a user's mouse click), for which duplication makes no sense.[2] Heraclitus said, "You can never step in the same river twice"; in the same way, you can never click on the same button twice.

Linear logic provides other advantages besides managing the inherently stateful nature of events. By using linear logic as the substructural basis for LTL, we can take negation to be a *constructive* connective that gives types to continuations. This is important because event-driven libraries are usually structured in continuation-passing style (CPS). The connection between events and callbacks (a.k.a. continuations) is usually expressed in monadic libraries via operations like:

$$\texttt{button.onClick} : (\texttt{clickData} \to \texttt{unit}) \to \texttt{unit}.$$

This `onClick` function takes a callback and registers it with the environment: when a button click event occurs, the corresponding callback will be executed. In temporal logic, the callback must be available at any point in the future, so a more precise type is:

$$\texttt{button.onClick} : \Box(\texttt{clickData} \to \texttt{unit}) \to \texttt{unit}.$$

The CPS translation can be parametric in the return type [19], so abstracting away the parametric argument gives the type

$$\texttt{button.onClick} : \neg\Box\neg\texttt{clickData}.$$

By the De Morgan duality that $\diamond\, A \equiv \neg\, \Box\, \neg\, A$, we have

$$\texttt{button.onClick} : \diamond\texttt{clickData}.$$

Rather than thinking of `button.onClick` as a function that installs a callback (via side effects), we instead think of it as an event that produces some `clickData`. The $\diamond$ modality is a monad in temporal logic, and it is this monad that forms the basis of the event-driven monadic style. Intuitively, the bind operator `bind x = t1 in t2` waits for the event `t1` to occur, binds the value of the underlying event data to `x`, and then continues with the computation `t2`.

Asynchronous computation and `select`**.** The bind operator waits for a single event and only then proceeds with the result. This paradigm is a good way to structure event-driven programs, but it is not expressive enough to capture concurrent events. Event-driven libraries often provide a `select` operator, which waits for two or more events in parallel and branches to different alternatives depending on which one happens first.

We show that, remarkably, `select` arises naturally from a proof-theoretic view of LTL. Consider the following axiom, which defines the total (linear) ordering on timesteps, the worlds of LTL:

$$\forall ab, a \neq b \Rightarrow (a < b) \vee (a > b)$$

The `select` operator corresponds to the propositional variant of the axiom, which encodes the total order on the events of LTL:

$$\diamond\, A \to \diamond\, B \to \diamond\, ((A \wedge \diamond\, B) \vee (\diamond\, A \wedge B))$$

The axiom says that if A is eventually true and B is eventually true, then either A comes before B (meaning that B is in A's future), or B comes before A. The disjunction provides an opportunity for `select` to branch based on which event comes first.

However, adding such an axiom to a proof theory of linear logic is not so straightforward. For one thing, the axiom involves several logical connectives, which means that its semantics is not "modular" with respect to the other types. For another, we must show that it can be effectively implemented in the semantics. Our approach to solving these problems is to structure the semantics in the form of an adjoint tensor logic, extended to account for `select`.

Before diving into those technical details, however, we first show how monadic-style features (including `select`) can be conveniently packaged in a surface language that treats $\diamond\, A$ as the central abstraction. The type theory is supported by a linear sublanguage of "event-driven" computations, with a full "ordinary" intuitionistic lambda calculus for writing pure expressions as well.

[1] There is an unfortunate terminological clash of the word "linear" which refers in linear-time temporal logic to the total ordering on times, and refers in linear logic to the constraint that a resource be used exactly once in a computation. We emphasize the distinction when necessary, but usually the meaning should be clear from context.

[2] Linearity principles have similarly been shown to be useful in the FRP interpretation of LTL, where it helps to avoid pitfalls of temporality such as space leaks and causality violations [8].

$$\frac{}{\Gamma; \cdot \vdash () : 1}\ 1\text{-I} \qquad \frac{\Gamma; \Delta_1 \vdash t_1 : 1 \quad \Gamma; \Delta_2 \vdash t_2 : B}{\Gamma; \Delta_1, \Delta_2 \vdash \mathbf{let}\,() = t_1\,\mathbf{in}\,t_2 : B}\ 1\text{-E}$$

$$\frac{\Gamma; \Delta_1 \vdash t_1 : A_1 \quad \Gamma; \Delta_2 \vdash t_2 : A_2}{\Gamma; \Delta_1, \Delta_2 \vdash (t_1, t_2) : A_1 \otimes A_2}\ \otimes\text{-I} \qquad \frac{\Gamma; \Delta_1 \vdash t_1 : A_1 \otimes A_2 \quad \Gamma; \Delta_2, x_1 : A_1, x_2 : A_2 \vdash t_2 : B}{\Gamma; \Delta_1, \Delta_2 \vdash \mathbf{let}\,(x_1, x_2) = t_1\,\mathbf{in}\,t_2 : B}\ \otimes\text{-E}$$

(no 0-I rule) $$\qquad \frac{\Gamma; \Delta \vdash t : 0}{\Gamma; \Delta \vdash \mathbf{case}\,t\,\mathbf{of}\,() : B}\ 0\text{-E}$$

$$\frac{\Gamma; \Delta \vdash t : A_i}{\Gamma; \Delta \vdash \mathbf{in}_i\,t : A_1 \oplus A_2}\ \oplus\text{-I} \qquad \frac{\Gamma; \Delta_1 \vdash t : A_1 \oplus A_2 \quad \Gamma; \Delta_2, x_1 : A_1 \vdash t_1 : B \quad \Gamma; \Delta_2, x_2 : A_2 \vdash t_2 : B}{\Gamma; \Delta_1, \Delta_2 \vdash \mathbf{case}\,t\,\mathbf{of}\,(\mathbf{in}_1\,x_1 \to t_1 \mid \mathbf{in}_2\,x_2 \to t_2) : B}\ \oplus\text{-E}$$

$$\frac{\Gamma; \Delta, x : A \vdash t : B}{\Gamma; \Delta \vdash \lambda x.t : A \multimap B}\ \multimap\text{-I} \qquad \frac{\Gamma; \Delta_1 \vdash t_1 : A \multimap B \quad \Gamma; \Delta_2 \vdash t_2 : A}{\Gamma; \Delta_1, \Delta_2 \vdash t_1\ t_2 : B}\ \multimap\text{-E}$$

$$\frac{\Gamma; \Delta \vdash t : A}{\Gamma; \Delta \vdash \mathbf{return}\,t : \diamond A}\ \diamond\text{-I} \qquad \frac{\forall i : \quad \Gamma; \Delta_i \vdash t_i : \diamond A_i \quad \Gamma; \Delta, x_i : \diamond A_i, \overrightarrow{x_j : \diamond A_j}^{j \neq i} \vdash t'_i : \diamond B}{\Gamma; \Delta, \overrightarrow{\diamond A_i}^{i} \vdash \mathbf{select}\,(t_1 \mid .. \mid t_n)\,\mathbf{of}\,(x_1, .., x_n) \to (t'_1 \mid .. \mid t'_n) : \diamond B}\ \diamond\text{-E}$$

Figure 1. Typing rules for linear types.

3. Examples

To make the ideas of this event-driven programming model more concrete, we explore them in the context of GUI programming. Here, the linear sublanguage is tailored for manipulating GUI objects, though the unrestricted part is a standard simply typed lambda calculus. This toy language would need to be extended in various ways to make it into a full-fledged GUI toolkit, but it suffices to convey the key features of our design.

The components of GUI programs, such as buttons, labels, and text boxes, are called widgets. The type of a widget is written `W`, and it must be treated linearly—every widget must be used at least and no more than once.

The intuition behind linearity in the surface language is that variables of type widget are pure references to impure code. When a variable is consumed, the underlying state changes, so a new variable pointing to the updated state must be provided. The old variable can never again be used because its state no longer exists in the program.

Consider the following "hello world" program:

```
helloWorld : ◇(W ⊗ W)
helloWorld =
  let  b = newButton () in
  bind b = onClick b in
  let  l = newLabel |"Hello World!"| in
  return (b,l)
```

The application starts out with a single button of type `W`. The event `onClick b` has type `◇ W` and will return a handle to the widget only after the button has been clicked. The linearity restriction means that the old variable `b` can never be accessed again, so we write over it with the new reference.

Once the button has been clicked, a label will appear with the text "Hello World!" on it. The `return` construct turns a pure value (a pair of widgets) into an event that immediately returns its value.

A note about the argument to `newLabel`: Strings are not linear data, and they can be used as many times as necessary. We enclose nonlinear data in bars to indicate that it should be used as an argument to a linear function.

Arguably the hardest part of event-based programming is synchronization—waiting for two events in parallel that depend on each other in some way. The following program uses the `select` primitive to race to see which button gets clicked first.

```
race : ◇(W ⊗ W ⊗ W)
race =
  let b1 = newButton () in
  let b2 = newButton () in
  select onClick b1 | onClick b2 of
  | ◇b1, e2 ->

    let l = newLabel |"Button 1 won the race!"| in
    bind b2 = e2 in
    return (b1,b2,l)
  | e1, ◇b2 ->
    let l = newLabel |"Button 2 won the race!"| in
    bind b1 = e1 in
    return (b1,b2,l)
  end
```

The `race` program constructs two buttons and waits for a click on either one. As soon as the first button is clicked (say, `b1`), a label will appear with the text "Button 1 won the race!." The program then waits for the second button to be clicked before terminating.

Select can be thought of as the n-ary form of `bind`. The general form of `select` is as follows:

```
select c1 | ... | cn as
| ◇x1,e2,...,en -> t1
| ...
| e1,...,en-1,◇xn -> tn
end
```

Operationally, `select` runs the computations `c1` through `cn` in parallel. For whichever computation finishes first (say, `ei`) the result of that computation gets bound to `xi`, and the program continues at `ti`. For the sake of linearity, the variables `ej` are fresh handlers to the still ongoing computations.

The `helloWorld` and `race` programs only last for a finite amount of time. In general we will want to write nonterminating programs, for which we use a fixpoint specifically for events. The following program increments a counter every time its button is clicked.

```
counter : ◇1
counter =
  let b = newButton () in
  let l = newLabel |toString 0| in
  letrec loop b l |n| =
    bind b = onClick b in
    let l = updateLabel l |toString (n+1)| in
    force loop b l |n+1|
  in loop b l |0|
```

The `counter` program initializes two widgets: a button and a label. It then calls `loop` with the buttons and an initial value for the counter. The recursive function `loop` waits for a click on the button, updates the label, and then passes the updated value `n+1` to the recursive call. The recursive call to `loop` is actually a thunk of an event, so we invoke it using `force`.

The `letrec` construct does *not* provide general recursion because it applies only to functions whose result is a monadic event. As a consequence, only linear computations, and not pure code, can be nonterminating. This is an important restriction that ensures that the monadic code is *live*—the event queue will continue to process events even if there is a nonterminating computation being executed at the same time.

To write a program that processes concurrent events in parallel, we will need to use the `select` construct inside the loop. Consider a GUI program with two buttons and a counter, where clicking one button increments the counter, and clicking the other button decrements it.

```
counter' : ◇1
counter' =
  let b1 = newButton () in
  let b2 = newButton () in
  let l  = newLabel |toString 0| in
  letrec loop e1 e2 l |n| =
    select e1 | e2 as
    | ◇b1, e2 ->
      let l = updateLabel l |toString (n+1)| in
      force loop (onClick b1) e2 l |n+1|
    | e1, ◇b2 ->
      let l = updateLabel l |toString (n+1)| in
      force loop e1 (onClick b2) l |n-1|
    end
  in loop (onClick b1) (onClick b2) l |0|
```

Each pass of the step function processes *either* a click on `b1` *or* a click on `b2`. As a result, the other click is still in the process of waiting, which means that the step function must act on the events `(onClick b1)` and `(onClick b2)` respectively, instead of on the buttons themselves.

4. Surface Language

In this section we present a linear/non-linear type system for event-based monadic programming. In the linear/non-linear paradigm [1] there are two kinds of types: linear and unrestricted. Linear types are denoted A, and unrestricted types, also called persistent, are denoted τ.

$$A ::= 1 \mid A_1 \otimes A_2 \mid 0 \mid A_1 \oplus A_2 \mid A_1 \multimap A_2 \mid \diamond A \mid |\tau|$$
$$\tau ::= \mathsf{Unit} \mid \tau_1 \times \tau_2 \mid \mathsf{Void} \mid \tau_1 + \tau_2 \mid \tau_1 \to \tau_2 \mid !\, A$$

Similarly, there are two kinds of typing judgments—one for linear types and one for persistent types. Linear typing judgments can use variables both from persistent contexts Γ, as well as from linear contexts Δ. Persistent typing judgments can only use variables from persistent contexts, because they themselves might be treated non-linearly. The typing judgments have the form

$$\Gamma; \Delta \vdash t : A \qquad \text{and} \qquad \Gamma \vdash t : \tau,$$

where

$$\Delta ::= \cdot \mid \Delta, x : A \qquad \text{and} \qquad \Gamma ::= \cdot \mid \Gamma, x : \tau$$

$$\frac{}{\Gamma \vdash () : \mathsf{Unit}}\ \textsf{Unit-I} \qquad \frac{\Gamma \vdash t : \mathsf{Void}}{\Gamma \vdash \mathbf{case}\, t\, \mathbf{of}\, () : \sigma}\ \textsf{Void-E}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (t_1, t_2) : \tau_1 \times \tau_2}\ \times\text{-I} \qquad \frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i\, t : \tau_i}\ \times\text{-E}$$

$$\frac{\Gamma \vdash t : \tau_1}{\Gamma \vdash \mathbf{in}_1\, t : \tau_1 + \tau_2}\ +\text{-I}_1 \qquad \frac{\Gamma \vdash t : \tau_2}{\Gamma \vdash \mathbf{in}_2\, t : \tau_1 + \tau_2}\ +\text{-I}_2$$

$$\frac{\Gamma \vdash t : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash t_1 : \sigma \quad \Gamma, x_2 : \tau_2 \vdash t_2 : \sigma}{\Gamma \vdash \mathbf{case}\, t\, \mathbf{of}\, (\mathbf{in}_1\, x_1 \to t_1 \mid \mathbf{in}_2\, x_2 \to t_2) : \sigma}\ +\text{-E}$$

$$\frac{\Gamma, x : \tau \vdash t : \sigma}{\Gamma \vdash \lambda x.t : \tau \to \sigma}\ \to\text{-I} \qquad \frac{\Gamma \vdash t_1 : \tau \to \sigma \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1\, t_2 : \sigma}\ \to\text{-E}$$

Figure 2. Typing rules for unrestricted types.

$$\frac{\Gamma; \cdot \vdash t : A}{\Gamma \vdash \mathbf{suspend}\, t : !\, A}\ !\text{-I} \qquad \frac{\Gamma \vdash t : !\, A}{\Gamma; \cdot \vdash \mathbf{force}\, t : A}\ !\text{-E}$$

$$\frac{\Gamma \vdash t : \tau}{\Gamma; \cdot \vdash |t| : |\tau|}\ |-|\text{-I}$$

$$\frac{\Gamma; \Delta_1 \vdash t_1 : |\tau| \quad \Gamma, x : \tau; \Delta_2 \vdash t_2 : B}{\Gamma; \Delta_1, \Delta_2 \vdash \mathbf{let}\, |x| = t_1\, \mathbf{in}\, t_2 : B}\ |-|\text{-E}$$

Figure 3. Typing rules for moving between the linear and unrestricted fragments.

The syntax of terms is as follows:

$$\begin{aligned} t ::= {} & x \mid \lambda x.t \mid t_1\, t_2 \\ & \mid () \mid \mathbf{let}\, () = t_1\, \mathbf{in}\, t_2 \mid (t_1, t_2) \mid \mathbf{let}\, (x_1, x_2) = t_1\, \mathbf{in}\, t_2 \\ & \mid \mathbf{case}\, t\, \mathbf{of}\, () \mid \mathbf{in}_1\, t \mid \mathbf{in}_2\, t \mid \mathbf{case}\, t_0\, \mathbf{of}\, (\mathbf{in}_1\, x_1 \to t_1 \mid \mathbf{in}_2\, x_2 \to t_2) \\ & \mid \mathbf{return}\, t \mid \mathbf{select}\, (t_1 \mid \ldots \mid t_n)\, \mathbf{of}\, (x_1, \ldots, x_n) \to (t_1' \mid \ldots \mid t_n') \\ & \mid \mathbf{suspend}\, t \mid \mathbf{force}\, t \mid |t| \mid \mathbf{let}\, |x| = t_1\, \mathbf{in}\, t_2 \end{aligned}$$

There are two typing rules for variables, one for each fragment:

$$\frac{}{\Gamma; x : A \vdash x : A}\ \textsc{var}\text{-}A \qquad \frac{}{\Gamma, x : \tau \vdash x : \tau}\ \textsc{var}\text{-}\tau$$

Figure 1 shows the rules for linear types. The linearity constraints are enforced by how the linear contexts are managed. In the 1-introduction rule for example, the unit $()$ only has type 1 if there are no other linear variables in scope. This ensures that all linear resources get used. The $\otimes$-introduction rule, on the other hand, enforces the fact that linear variables cannot be duplicated. In order to construct a pair (t_1, t_2), we must be able to partition the context into two non-overlapping contexts of resources, Δ_1 and Δ_2, such that t_1 only uses resources from Δ_1 and t_2 only uses resources from Δ_2.

The rules are fairly standard for intuitionistic linear type systems, with the exception of the monadic $\diamond$ operator from LTL. The `return` operator constructs an event that immediately yields the pure term t. The `select` operator runs n computations, t_1 through t_n, in parallel. Depending on which computation finishes first, it proceeds with *one* of the results, t_1' through t_n'. In particular, if t_i finishes first then the result of that computation will be bound to the variable x_i, and the program will proceed as t_i'. The variables x_j for $j \neq i$ will be bound to the ongoing events t_j.

The unary form of `select` is just the monadic bind operator. It is distinguished with its own syntactic sugar:

$$\frac{\Gamma; \Delta_1 \vdash t : \diamond A \quad \Gamma; \Delta, x : A \vdash t' : \diamond B}{\Gamma; \Delta_1, \Delta \vdash \mathbf{bind}\, x = t\, \mathbf{in}\, t' : \diamond B}\ \text{BIND}$$

The persistent, or unrestricted, fragment of the surface language is just the simply-typed lambda calculus. Its typing rules are shown in Figure 2.

The remaining two types are $!\,A$ and $|\tau|$, whose typing rules are presented in Figure 3. A linear term that doesn't use any linear variables can be suspended into a persistent thunk of type $!\,A$. This thunked event can then be instantiated at any time using `force`. The $|-|$ rules say that we can treat persistent terms of type τ as if they were linear with type $|\tau|$.

This set of typing rules are sometimes called *adjoint*, which stems from the categorical understanding of ! as a comonad in linear logic. In the linear/non-linear configuration we decompose ! into two parts, ! and $|-|$, corresponding to the decomposition of a monad into two adjoint functors. Notice that, like $\diamond$, $!\,|\tau|$ forms a monad in the persistent fragment. In Section 6 we show that these two monads can be seen as parallel structures in a three-tiered system with linear, temporal, and persistent types.

β-equality. The operational semantics of the surface language is outside the scope of this paper, where we instead focus on the Kripke semantics defined in Section 7. However, β-equality is still a sound way of understanding the behavior of surface language programs, including select functions. In particular, the following equivalences hold in the semantics:

$$\mathbf{select}\,(\mathbf{return}\, v \mid t_2)\,\mathbf{of}\,(x_1, x_2) \to (t_1' \mid t_2') =_\beta t_1'\{v/x_1, t_2/x_2\}$$
$$\mathbf{select}\,(t_1 \mid \mathbf{return}\, v)\,\mathbf{of}\,(x_1, x_2) \to (t_1' \mid t_2') =_\beta t_2'\{t_1/x_1, v/x_2\}$$

Fix. The language supports a restricted form of fixpoints that only applies to computations of the form $\diamond A$. For any function $A \multimap \diamond B$, there is an operator

$$\texttt{sfix} : !\,(!\,(A \multimap \diamond B) \multimap (A \multimap \diamond B)) \multimap (A \multimap \diamond B).$$

The operator is special in that it is non-blocking—it allows other events not in the (possibly non-terminating) loop to execute in parallel.

The `letrec` construct from the previous section is encoded using `sfix`:

```
letrec f = t1 in t2 :=
  let f = sfix (suspend (fun f -> t1)) in t2
```

GUI primitives. We extend the language with three event-specific types:

$$A ::= \cdots \mid \mathbf{W}$$
$$\tau ::= \cdots \mid \mathbf{WData} \mid \mathbf{EData}$$

These can be summarized as follows:

- **W** is the type of widgets;
- **WData** is the type of data that describes widgets, such as size, positioning, or text to display; and
- **EData** is the type of data that describes events, such as the kind of event (a button click or a key press) and potentially extra data (which key was pressed).

In this paper we leave **WData** and **EData** abstract because they are not the focus of the semantics. The important point is that they describe data in the unrestricted fragment, whereas **W** is linear.

The `onEvent` primitive takes in a widget and produces a computation that will trigger once an event occurs. Its type is

$$\texttt{onEvent} : \mathbf{W} \multimap (\mathbf{W} \otimes \diamond\, |\mathbf{EData}|).$$

The other primitives don't interact with events, but modify the data associated with widgets.

$$\texttt{newWidget} : |\mathbf{WData}| \multimap \mathbf{W}$$
$$\texttt{setWData} : \mathbf{W} \multimap |\mathbf{WData}| \multimap \mathbf{W}$$
$$\texttt{getWData} : \mathbf{W} \multimap \mathbf{W} \otimes |\mathbf{WData}|$$

Variants on `onEvent`, like the function `onClick` in Section 3, can be defined by recursively waiting for events until one of them is a click event.

```
onClick : W —o ◇W
onClick =
  letrec onClick' w =
    let (w,e) = onEvent w in
    bind |d|  = e in
    if isClick d then return w
                 else force onClick' w
  in onClick'
```

5. Target Language

The surface language computes with monadic events at the type $\diamond A$; this section will describe in broad strokes how these monadic events might be implemented. The semantics for which we prove safety is described in Section 7. Here, we simply aim to give an intuition about how terms in the surface language behave. We give sketches for implementations of the key constructs of the surface language: the GUI primitives, recursion in the form of `sfix`, and the monadic operators, `return` and `select`.

The key observation is to implement the $\diamond A$ modality as the continuation monad $\neg\neg A$. The result is code in continuation-passing style (CPS): functions that take in continuations of type $\neg A$ but do not return a meaningful value. Instead, the code simulates returning a value of type A by passing that value to its continuation.

From the perspective of LTL, we treat these continuations as having type $\neg\Box\neg A$, but operationally $\Box$ is ignored, at least for the purposes of this section.

5.1 Operational Behavior

The target of the semantics is an untyped, impure lambda calculus with a mutable heap and an explicit event queue. A store σ is made up of these two parts: a heap and a queue. A heap is a partial function from locations to data, and a queue is a (total) function from locations to multisets of event handlers.

$$\text{(Heaps)} \qquad h \in \mathrm{Loc} \rightharpoonup \mathrm{Val}$$
$$\text{(Queues)} \qquad q \in \mathrm{Loc} \to \mathcal{M}(\mathrm{Val})$$
$$\text{(Stores)} \qquad \sigma ::= (h, q)$$

Expressions include primitives for manipulating the store.

$$\begin{aligned} e ::= {} & x \mid \lambda x.e \mid e\ e \\ & \mid \mathbf{in}_1\ e \mid \mathbf{in}_2\ e \mid \mathbf{case}\ e\ \mathbf{of}\ (\mathbf{in}_1\ x \to e \mid \mathbf{in}_2\ y \to e) \\ & \mid (e, e) \mid \pi_1\ e \mid \pi_2\ e \\ & \mid l \mid \mathsf{new}(e) \mid e := e \mid !e \\ & \mid E \mid \mathsf{attach}(e, e) \\ v ::= {} & \lambda x.e \mid \mathbf{in}_1\ v \mid \mathbf{in}_2\ v \mid (v, v) \mid l \mid E \end{aligned}$$

The expression $\mathsf{new}(e)$ evaluates to a fresh location and initializes that location in the heap with the value of e. Heap locations can be assigned with $e := e'$ and dereferenced with $!e$. The operation $\mathsf{attach}(e, e')$ registers e' as an event handler in the queue at location e. Events E are passed to these event handlers when triggered by the environment.

$$\frac{}{\langle\sigma; v\rangle \Downarrow \langle\sigma; v\rangle}\ \text{VAL} \qquad \frac{\langle\sigma; e\rangle \Downarrow \langle(h, q); l\rangle \quad h(l) = v}{\langle\sigma; !e\rangle \Downarrow \langle(h, q); v\rangle}\ !$$

$$\frac{\langle\sigma; e\rangle \Downarrow \langle(h, q); v\rangle \quad l \notin \mathrm{dom}(h)}{\langle\sigma; \mathsf{new}(e)\rangle \Downarrow \langle([h \mid l : v], q); l\rangle}\ \text{NEW}$$

$$\frac{\langle\sigma; e\rangle \Downarrow \langle\sigma'; l\rangle \quad \langle\sigma'; e'\rangle \Downarrow \langle(h'', q''); v\rangle}{\langle\sigma; e := e'\rangle \Downarrow \langle([h'' \mid l : v], q''); ()\rangle}\ \text{ASSIGN}$$

$$\frac{\langle\sigma; e\rangle \Downarrow \langle\sigma'; l\rangle \quad \langle\sigma'; e'\rangle \Downarrow \langle(h'', q''); v\rangle}{\langle\sigma; \mathsf{attach}(e, e')\rangle \Downarrow \langle(h'', [q'' \mid l : v \cdot q''(l)]); ()\rangle}\ \text{ATTACH}$$

Figure 4. Operational semantics for the impure lambda calculus.

$$\frac{q(l) \vdash_E (h, [q \mid l : \emptyset]) \rightsquigarrow \sigma}{(h, q) \Longrightarrow \sigma} \qquad \frac{}{\emptyset \vdash_E \sigma \rightsquigarrow \sigma}$$

$$\frac{\langle\sigma; k\, E\rangle \Downarrow \langle\sigma'; ()\rangle \quad \overrightarrow{k} \vdash_E \sigma' \rightsquigarrow \sigma''}{k \cdot \overrightarrow{k} \vdash_E \sigma \rightsquigarrow \sigma''}$$

Figure 5. Operational behavior of stores.

Figure 4 shows the operational semantics of these expressions. Evaluation is single-threaded, which means that no two expressions will ever be evaluated at the same time. After an expression completes evaluation, control returns to the scheduler and the environment is free to process incoming events. Every such event E, associated with a location l, is deployed by passing E to each of the callbacks associated with l in the event queue. The operational behavior of stores is shown in Figure 5. If $\sigma \Longrightarrow \sigma'$ by the event E at location l, we will sometimes write $\sigma \overset{(l,E)}{\Longrightarrow} \sigma'$.

5.2 Implementation Sketch

Next we will give the rough sketches of how to interpret the operators from the surface language. We suggestively use type annotations to give a specification to untyped functions; these annotations have no semantic meaning yet. In Section 7 we will formalize the annotations as semantic types and show that there exist implementations that inhabit those types.

GUI primitives. The type of `onEvent` in the surface language is

$$\mathbf{W} \multimap \mathbf{W} \otimes \diamond |\mathbf{EData}|.$$

Through the CPS translation we think of $\diamond|\mathbf{EData}|$ as $\neg\,\Box\,\neg\,\mathbf{EData}$, and its behavior should be to register a callback, which is just a continuation, at the location indicated by the widget. Thus we interpret `onEvent` as the following expression:

```
onEvent := λl.(l, λk.attach(l,k))
```

Fixpoints. Next we give an implementation sketch for the `sfix` operator, which has type

$$\texttt{sfix} : !\,(!\,\diamond\, A \multimap \diamond A) \multimap \diamond A$$

to a first approximation.[3] The target language contains the untyped lambda calculus, so we can encode the Y-combinator to perform recursion:

```
Y := λf.(λx.f (x x))(λx.f (x x))
```

But this is unsatisfactory in terms of efficiency. The Y-combinator is what is called a *blocking* computation because while it is being evaluated, it prevents other events from being handled. The `sfix` function addresses this by only allowing recursion on computations. Because computations of type $\diamond A$ can be executed in parallel, it stands to reason that even non-terminating computations can be executed concurrently with other events.

The implementation of `sfix` works by registering its recursive call in the event queue, at a special location called CLOCK that is triggered at every time step. Therefore `sfix f` always returns control to the scheduler before getting to invoke its recursive call. The definition of `sfix` is

```
sfix := λ (f : ¬□¬A ⊸ ¬□¬A).
  Y (λ (rec : ¬□¬A). λ (k : □¬A).
    attach(CLOCK, λ _. f rec k))
```

In the remaining parts of the paper, we put `sfix` aside. We leave its semantic soundness to future work, as it introduces significant complications to the structure of the logical relation. However the issue of non-blocking recursive computations is an important consideration for a future implementation.

Synchronization primitives. Consider the introduction rule for $\diamond$ in the surface language:

$$\frac{\Gamma; \Delta \vdash t : A}{\Gamma; \Delta \vdash \mathbf{return}\, t : \diamond A}\ \diamond\text{-I}$$

For an expression `e` that interprets t, define

```
return e := λk.k e
```

So `return` takes in a continuation `k` and immediately passes its value (given by `e`) to the continuation.

We will build up the `select` rule in stages, starting with the unary version, `bind`. But `bind` is a special case of the more general `select` operator, and its implementation should reflect that fact.

$$\frac{\Gamma; \Delta_1 \vdash t : \diamond A \quad \Gamma; \Delta, x : A \vdash t' : \diamond B}{\Gamma; \Delta_1, \Delta \vdash \mathbf{bind}\, x = t\, \mathbf{in}\, t' : \diamond B}\ \text{BIND}$$

Suppose `e` interprets t and `e'` interprets `t'`. Define

```
bind x = e in e' := λk.e (λx.e' k).
```

The expression first takes in a continuation k. It then evaluates `e` by passing it a continuation. The one it chooses is the continuation that uses the result of `e` to compute `e'` in the original continuation `k`.

Notice that the time at which `e'` is evaluated depends on how `e` uses its continuation. For example, if `e` registers its continuation in the heap using `onEvent`, then `e'` will not evaluate until an event triggers the callback.

Next we consider the binary select rule—the n-ary version can be generalized from this case.

$$\frac{\begin{array}{c}\Gamma; \Delta_1 \vdash t_1 : \diamond A \qquad \Gamma; \Delta_2 \vdash t_2 : \diamond B \\ \Gamma; \Delta, x : A, y : \diamond B \vdash t_1' : \diamond C \quad \Gamma; \Delta, x : \diamond A, y : B \vdash t_2' : \diamond C\end{array}}{\Gamma; \Delta_1, \Delta_2, \Delta \vdash \mathbf{select}\, (t_1 \mid t_2)\, \mathbf{of}\, (x_1, x_2) \to (t_1' \mid t_2') : \diamond C}$$

The intended behavior of the select rule is to wait for the events t_1 and t_2 in parallel. The implementation therefore defines two complementary continuations, giving one to t_1 and the other to t_2.

As an example, suppose t_1 is triggered before t_2. In this case the continuation being fed to t_1 should evaluate t_1', and the continuation being fed to t_2 should do no extra evaluation. In order to distinguish which of the two events came first, it is necessary to synchronize with a mutable reference variable, which we will call `c`. Over the course of the program, this *channel*, which is just a reference cell following a particular protocol, may be in any of the following

[3] Actually, `sfix` can range over arbitrary functions whose result type is an event, as in $A_1 \multimap \cdots \multimap A_n \multimap \diamond A$. In this discussion we restrict the type to just the events themselves, for the sake of conciseness.

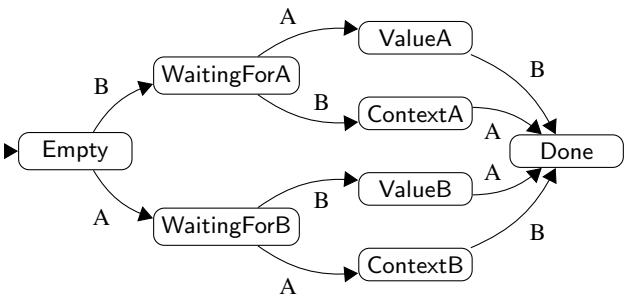

Figure 6. State transition diagram for synchronization on the channel `c` in the select operator. The label A refers to actions taken by the context passed to t_1, and the label B refers to actions taken by the context passed to t_2.

states:

$$
\begin{array}{ll}
\mid \mathsf{Empty} & \\
\mid \mathsf{WaitingForA} & \mid \mathsf{WaitingForB} \\
\mid \mathsf{ValueA}\, v & \mid \mathsf{ValueB}\, v \\
\mid \mathsf{ContextA}\, k & \mid \mathsf{ContextB}\, k \\
\mid \mathsf{Done} &
\end{array}
$$

The intention is that when t_1 returns its value first, the state transitions to WaitingForB, which means that the channel should record the eventual return value of t_2. When t_2 returns, it will record its value as ValueB v. Meanwhile, the context for t_1 will evaluate t_1', which expects an argument of type $\neg\,\Box\,\neg\,B$. The argument we give it is an argument that itself takes a continuation $\mathtt{k} \in \neg\, B$. To synchronize with the value returned by t_2, we also record the continuation in the reference cell as ContextB k. Once both ValueB v and ContextB k are set, the state transitions to Done and we evaluate $k\, v$.

The program, shown in Figure 7, is also expressed as a state transition diagram in Figure 6. The diagram can be thought of as a protocol which manages synchronization through the restricted use of a shared channel. This protocol will be discussed further in Section 7, where we formalize the idea using a concept of resources and permissions.

5.3 Soundness

Now that we have an understanding of how we expect the implementation to behave, we can consider the properties that characterize a well-behaved program.

Expressions should be type-safe in the traditional sense. That is, if t is well-typed in the surface language, then its interpretation should not go wrong in the target language. But this definition must be strengthened to take the event queue into account. Consider the expression $\mathsf{attach}(l, \lambda x.(\,)\,x)$. The expression does evaluate to a value (unit), but the updated store is not well-behaved. When an event triggers the callback on l, the program will crash. Thus the expression generated a safe value, but an unsafe store.

So what does it mean for a store to be safe? Intuitively it should mean that *any* callback in the queue can be triggered and will result in another safe store. As in typical constructions of type safety, this key property can be summed up as:

Preservation: If σ is a safe store and $\sigma \Longrightarrow \sigma'$ then σ' is also safe.

The surface language should therefore satisfy the following soundness property, characterized by the fact that the resulting store is safe:

Soundness: If $\vdash t : A$ in the surface language and $\langle \cdot; [t] \rangle \Downarrow \langle \sigma; v \rangle$, then σ is safe.

```
// e1 : ¬□¬A
// e2 : ¬□¬B
// x:A,y:¬□¬B |- e1' : ¬□¬C
// x:¬□¬A,y:B |- e2' : ¬□¬C

select (e1 | e2) of (x,y) -> (e1' | e2') :=
  λ (kc : □¬C).
  let c = new(Empty) in
  e1 (λ (a : A).
     case !c of
     | Empty       → c := WaitingForB;
                     let z = waitForB c in
                     (e1'{a/x,z/y}) kc
     | WaitingForA → c := ValueA a
     | ContextA k  → c := Done;
                     k a
     | _           → undefined
     end);
  e2 (λ (b : B).
     case !c of
     | Empty       → c := WaitingForA;
                     let z = waitForA c in
                     (e2'{z/x,b/y}) kc
     | WaitingForB → c := ValueB b
     | ContextB k  → c := Done;
                     k b
     | _           → undefined
     end))

where
  waitForA c = λ (ka : □¬A).
    case !c of
    | WaitingForA → c := ContextA ka
    | ValueA a    → c := Done; ka a
    |_            → undefined
    end
  waitForB c = λ (ka : □¬B).
    case !c of
    | WaitingForB → c := ContextB ka
    | ValueB b    → c := Done; ka b
    | _           → undefined
    end
```

Figure 7. Interpretation of `select` operator.

In order to prove this property, we will need to define a logical relation that classifies valid machine configurations (pairs of a store and a value) for any particular type. Unfortunately, the structure of surface language types is not rich enough to define the necessary logical relation. Consider the bind rule as an example:

$$\frac{\Gamma; \Delta_1 \vdash t : \diamond A \quad \Gamma; \Delta, x : A \vdash t' : \diamond B}{\Gamma; \Delta_1, \Delta \vdash \mathbf{bind}\, x = t\, \mathbf{in}\, t' : \diamond B}\ \text{BIND}$$

The linear variables in Δ_1 and Δ may refer to events, which means they rely on the store being well-behaved. But the continuation t' will not be executed until some arbitrary time has passed—time in which the store may have advanced via the event loop. If σ is valid for Δ and $\sigma \Longrightarrow^* \sigma'$, then how do we know that σ' is valid for Δ?

This property of sets of machine configurations, which we will call *temporality*, is exactly characterized by the $\Box$ modality from temporal logic. For a set A of machine configurations, we expect $\Box\, A$ to characterize those stores σ such that whenever $\sigma \Longrightarrow^* \sigma'$, we have σ' valid for A. This is indeed the intended meaning of $\Box$ in $\diamond\, A \equiv \neg\,\Box\,\neg\, A$, but $\Box$ itself never occurs as the principle type in the surface language.

The logic defined in the next section solves this problem by explicitly separating temporal types from purely linear ones, which characterize stores at a particular moment in time. We embed every surface language type as a temporal proposition, which ensures

that the heaps they reference satisfy temporality. At the same time, we perform the double negation/CPS translation to unify the $\Box$ modality in the encoding of $\diamond$ with this notion of temporality.

6. Adjoint Tensor Logic

This section presents an intermediate logic known as adjoint tensor logic as the first step in the interpretation of the surface language. The logic serves three purposes.

First, tensor logic allows for an interpretation of the monad $\diamond$ in terms of negation and $\Box$. The negation operator provides a constructive logical formulation for reasoning about continuations. In the setting of temporal logic, it strengthens the connection between classical temporal logic and continuation passing style. We define a double negation translation from the surface language to derivations the CPS-style logic.

Second, the logic provides structure for the semantics presented in Section 7. We argue that the $\Box$ operator is more primitive than $\diamond$ for the purposes of classifying machine configurations, because it describes states that are valid at any point in the future. Linearity provides the structure to prove that disjoint parts of the heap do not interfere.

Third, the cut elimination theorem we prove about the logic provides assurance that the return and select operators from the surface language make up a sound API for the monadic event style. The cut elimination theorem shows that `select` is not just an arbitrary synchronization primitive—it forms a logical complement to `return` on top of being a generalization of `bind`.

6.1 Propositions and Judgments

The intermediate logic consists of three different structural fragments, as opposed to the two of the surface language. The *persistent fragment*, denoted $\mathbf{P}$, corresponds to persistent types τ in the surface language. These hypotheses can be used arbitrarily many times by applying weakening and contraction. The *temporal fragment*, denoted $\mathbf{T}$, represents propositions that are true at any point in time, which are nevertheless stateful, so they must be treated linearly. The *linear fragment*, denoted $\mathbf{L}$, consists of propositions that are linear and only true at one particular moment in time.

$$\begin{aligned} A^{\mathbf{L}} ::=& 1 \mid {A_1}^{\mathbf{L}} \otimes {A_2}^{\mathbf{L}} \mid 0 \mid {A_1}^{\mathbf{L}} \oplus {A_2}^{\mathbf{L}} \mid \neg\, A^{\mathbf{L}} \mid \mathbf{F}_{\mathbf{L}}\, A^{\mathbf{T}} \\ A^{\mathbf{T}} ::=& \mathbf{U}^{\mathbf{T}}\, A^{\mathbf{L}} \mid \mathbf{F}_{\mathbf{T}}\, A^{\mathbf{P}} \\ A^{\mathbf{P}} ::=& \mathsf{Unit} \mid {A_1}^{\mathbf{P}} \times {A_2}^{\mathbf{P}} \mid \mathsf{Void} \mid {A_1}^{\mathbf{P}} + {A_2}^{\mathbf{P}} \\ & \mid {A_1}^{\mathbf{P}} \to {A_2}^{\mathbf{P}} \mid \mathbf{U}^{\mathbf{P}}\, A^{\mathbf{T}} \end{aligned}$$

Note that the linear and temporal fragments do not contain linear implication, which is replaced by the negation operator $\neg\, A^{\mathbf{L}}$. Linear negation was introduced by Melliès and Tabareau [10] as tensor logic. Negation can be thought of as the type of continuations, which means that the computational interpretation of the logic satisfies CPS.

The adjoint operators, $!\,A$ and $|\tau|$, have been replaced by $\mathbf{U}^{\mathbf{P}}\, A^{\mathbf{T}}$ and $\mathbf{F}_{\mathbf{T}}\, A^{\mathbf{P}}$. The same operators relate the linear and temporal fragments. These are two instances of the same adjoint relationship, formalized by Reed [17] as adjoint logic. While $\mathbf{U}^{\mathbf{P}}$ and $\mathbf{F}_{\mathbf{T}}$ correspond to the ! connective from linear logic, the operators $\mathbf{U}^{\mathbf{T}}$ and $\mathbf{F}_{\mathbf{L}}$ correspond to the $\Box$ connective in temporal logic.

The union of adjoint and tensor logics is one of the contributions of this paper. In fact, the linear/temporal/persistent structure is an instance of a more general signature for adjoint tensor logic.

The adjoint operators are based on the idea that one level is "stronger" than another in a particular way. For example, any temporal proposition can be treated as if it were linear, so we write $\mathbf{L} \lhd \mathbf{T}$. Similarly, any persistent proposition (which is a pure value that does not depend on the store) can be treated as if it were only temporal: $\mathbf{T} \lhd \mathbf{P}$. These relationships can be generalized to an arbitrary signature of fragments r, with a DAG on fragments denoted $r_1 \lhd r_2$. We write $\leq$ for the reflexive transitive closure of the graph. Every fragment corresponds to a collection of propositions A^r, and for every edge $r \lhd s$ in the graph, the propositions include

$$A^r ::= \cdots \mid \mathbf{F}_r\, A^s \qquad\qquad A^s ::= \cdots \mid \mathbf{U}^s\, A^r$$

Judgments in the logic have one of two forms:

$$\Gamma \vdash A^r$$

where $\Gamma \geq r$ for arbitrary fragments r, and

$$\Gamma \vdash s$$

where $\Gamma \geq s$ for fragments s with negation. Judgments of this form are said to have no conclusion. We will sometimes write $\Gamma \vdash (A)^s$ to mean that the judgment has either 0 or 1 conclusion.

The logic is structurally linear, as can be seen in the hypothesis and cut rules:

$$\frac{}{A^r \vdash A^r}\ \textsc{hyp} \qquad \frac{\Gamma \vdash A^s \quad \Delta, A^s \vdash (B)^r}{\Gamma, \Delta \vdash (B)^r}\ \textsc{cut}$$

In the hypothesis rule the context must be empty—all resources must already be consumed. In the cut rule, the contexts Γ and Δ are concatenated together and their resources are not duplicated. There are explicit weakening and contraction rules for persistent propositions:

$$\frac{\Gamma \vdash (B)^r}{\Gamma, A^{\mathbf{P}} \vdash (B)^r}\ \textsc{weak} \qquad \frac{\Gamma, A^{\mathbf{P}}, A^{\mathbf{P}} \vdash (B)^r}{\Gamma, A^{\mathbf{P}} \vdash (B)^r}\ \textsc{contract}$$

Negation has the following inference rules whenever it is defined:

$$\frac{\Gamma \vdash A^s}{\Gamma, \neg A^s \vdash s}\ \neg\text{-L} \qquad \frac{\Gamma, A^s \vdash s}{\Gamma \vdash \neg\, A^s}\ \neg\text{-R}$$

The rules for linear logic operators are given in Figure 8, and the rules for persistent operators in Figure 9. These are straightforward from sequent calculus formulations of linear/non-linear logic [1].

Next are the adjoint operators: $\mathbf{U}^s$ and $\mathbf{F}_r$.[4] The inference rules for these operators are given in Figure 10. The **U**-R rule states that if A^r is provable using hypotheses that are at least as strong as s, then the s-proposition $\mathbf{U}^s\, A^r$ is provable. This is clearly a generalization of the !-I typing rule from the surface language, and similarly **F**-R is a generalization of $|-|$-I. The intuition should be that stronger fragments lay above the weaker fragments in the ordering, so the **U** operator moves **U**p the chain.

$$\frac{\Gamma \vdash (A)^{\mathbf{L}}}{\Gamma, 1 \vdash (A)^{\mathbf{L}}}\ 1\text{-L} \qquad \frac{}{\vdash 1}\ 1\text{-R}$$

$$\frac{\Gamma, {A_1}^{\mathbf{L}}, {A_2}^{\mathbf{L}} \vdash (B)^{\mathbf{L}}}{\Gamma, {A_1}^{\mathbf{L}} \otimes {A_2}^{\mathbf{L}} \vdash (B)^{\mathbf{L}}}\ \otimes\text{-L} \qquad \frac{\Gamma_1 \vdash {A_1}^{\mathbf{L}} \quad \Gamma_2 \vdash {A_2}^{\mathbf{L}}}{\Gamma_1, \Gamma_2 \vdash {A_1}^{\mathbf{L}} \otimes {A_2}^{\mathbf{L}}}\ \otimes\text{-R}$$

$$\frac{}{\Gamma, 0 \vdash (B)^{\mathbf{L}}}\ 0\text{-L} \qquad \text{(no 0-R rule)}$$

$$\frac{\Gamma, {A_1}^{\mathbf{L}} \vdash (B)^{\mathbf{L}} \quad \Gamma, {A_2}^{\mathbf{L}} \vdash (B)^{\mathbf{L}}}{\Gamma, {A_1}^{\mathbf{L}} \oplus {A_2}^{\mathbf{L}} \vdash (B)^{\mathbf{L}}}\ \oplus\text{-L}$$

$$\frac{\Gamma \vdash {A_i}^{\mathbf{L}}}{\Gamma \vdash {A_1}^{\mathbf{L}} \oplus {A_2}^{\mathbf{L}}}\ \oplus\text{-R}_i$$

Figure 8. Inference rules for linear operators.

[4] The name "adjoint" hints at a categorical relationship between the logical fragments. Indeed, the labels **F** and **U** derive from the **F**ree and **U**nderlying functors that make up free constructions in category theory [9]. A formal treatment of the categorical semantics of adjoint tensor logic is reserved

$$\frac{\Gamma \vdash (B)^r}{\Gamma, \mathsf{Unit} \vdash (B)^r}\ \mathsf{Unit\text{-}L} \qquad \frac{}{\vdash \mathsf{Unit}}\ \mathsf{Unit\text{-}R}$$

$$\frac{\Gamma, {A_1}^{\mathbf{P}}, {A_2}^{\mathbf{P}} \vdash (B)^r}{\Gamma, {A_1}^{\mathbf{P}} \times {A_2}^{\mathbf{P}} \vdash (B)^r}\ \times\text{-L} \qquad \frac{\Gamma_1 \vdash {A_1}^{\mathbf{P}} \quad \Gamma_2 \vdash {A_2}^{\mathbf{P}}}{\Gamma_1, \Gamma_2 \vdash {A_1}^{\mathbf{P}} \times {A_2}^{\mathbf{P}}}\ \times\text{-R}$$

$$\frac{}{\Gamma, \mathsf{Void} \vdash (B)^r}\ \mathsf{Void\text{-}L} \qquad \text{(no Void-R rule)}$$

$$\frac{\Gamma, {A_1}^{\mathbf{P}} \vdash (B)^r \quad \Gamma, {A_2}^{\mathbf{P}} \vdash (B)^r}{\Gamma, {A_1}^{\mathbf{P}} + {A_2}^{\mathbf{P}} \vdash (B)^r}\ +\text{-L}$$

$$\frac{\Gamma_1 \vdash {A_1}^{\mathbf{P}} \quad \Gamma_2, {A_2}^{\mathbf{P}} \vdash (B)^r}{\Gamma_1, \Gamma_2, {A_1}^{\mathbf{P}} \to {A_2}^{\mathbf{P}} \vdash (B)^r}\ \to\text{-L}$$

$$\frac{\Gamma \vdash {A_i}^{\mathbf{P}}}{\Gamma \vdash {A_1}^{\mathbf{P}} + {A_2}^{\mathbf{P}}}\ +\text{-R} \qquad \frac{\Gamma, {A_1}^{\mathbf{P}} \vdash {A_2}^{\mathbf{P}}}{\Gamma \vdash {A_1}^{\mathbf{P}} \to {A_2}^{\mathbf{P}}}\ \to\text{-R}$$

Figure 9. Inference rules for persistent operators.

$$\frac{\Gamma, A^r \vdash (B)^{r_0}}{\Gamma, \mathbf{U}^s A^r \vdash (B)^{r_0}}\ \mathbf{U}\text{-L} \qquad \frac{\Gamma \vdash A^r \quad \Gamma \geq s}{\Gamma \vdash \mathbf{U}^s\, A^r}\ \mathbf{U}\text{-R}$$

$$\frac{\Gamma, A^s \vdash (B)^{r_0} \quad r_0 \leq r}{\Gamma, \mathbf{F}_r A^s \vdash (B)^{r_0}}\ \mathbf{F}\text{-L} \qquad \frac{\Gamma \vdash A^s}{\Gamma \vdash \mathbf{F}_r\, A^s}\ \mathbf{F}\text{-R}$$

Figure 10. Inference rules for adjoint logic, assuming $r \lhd s$.

Linear-time temporal logic. Just as $\mathbf{U}^{\mathbf{P}}$ and $\mathbf{F}_{\mathbf{T}}$ correspond to the ! operator from linear logic, the $\mathbf{U}^{\mathbf{T}}$ and $\mathbf{F}_{\mathbf{L}}$ operators correspond to the $\Box$ operator from LTL. We write $\Box\, A^{\mathbf{L}}$ for the linear composition $\mathbf{F}_{\mathbf{L}}\, \mathbf{U}^{\mathbf{T}}\, A^{\mathbf{L}}$. Reed [17] and Pfenning and Davies [15] show that adjoint logic (without negation) is equivalent to an intuitionistic version of the modal logic S4. By integrating tensor logic, we align more closely with the classical version of S4, and the linear-time axiom, in the form of the SELECT rule, extends the model to cover LTL.

Using negation, we express the eventually modality $\diamond\, A^{\mathbf{L}}$ as

$$\diamond\, A^{\mathbf{L}} := \neg \Box \neg\, A^{\mathbf{L}} = \neg\, \mathbf{F}_{\mathbf{L}}\, \mathbf{U}^{\mathbf{T}}\, \neg\, A^{\mathbf{L}}.$$

In this setting, the linear-time axiom from LTL can be written as:

$$\diamond\, A \multimap \diamond\, B \multimap \diamond((A \otimes \diamond\, B) \oplus (\diamond\, A \otimes B)),$$

which characterizes the total order on time. Unfortunately, the decomposition of $\diamond$ into the subcomponents $\neg \mathbf{F}_{\mathbf{L}} \mathbf{U}^{\mathbf{T}} \neg$ makes the axiom significantly more difficult to integrate into a propositional logic. Our solution is the following inference rule for SELECT:

$$\frac{\Gamma, \neg A^{\mathbf{L}}, \neg \mathbf{F}_{\mathbf{L}} \mathbf{U}^{\mathbf{T}} B^{\mathbf{L}} \vdash \mathbf{L} \quad \Gamma, \neg \mathbf{F}_{\mathbf{L}} \mathbf{U}^{\mathbf{T}} A^{\mathbf{L}}, \neg B^{\mathbf{L}} \vdash \mathbf{L}}{\Gamma, \neg \mathbf{F}_{\mathbf{L}} \mathbf{U}^{\mathbf{T}} A^{\mathbf{L}}, \neg \mathbf{F}_{\mathbf{L}} \mathbf{U}^{\mathbf{T}} B^{\mathbf{L}} \vdash \mathbf{L}}\ \text{SELECT}$$

The structure varies slightly from the surface language variant, as it introduces an extra negation in the hypotheses. But the more familiar variant is derivable from the primitive SELECT rule, where $\diamond\, A$ is $\neg\, \mathbf{F}_{\mathbf{L}}\, \mathbf{U}^{\mathbf{T}}\, \neg\, A$:

$$\frac{\Gamma, A^{\mathbf{L}}, \diamond B^{\mathbf{L}} \vdash \mathbf{L} \quad \Gamma, \diamond A^{\mathbf{L}}, B^{\mathbf{L}} \vdash \mathbf{L} \quad \Gamma \geq \mathbf{T}}{\Gamma, \diamond A^{\mathbf{L}}, \diamond B^{\mathbf{L}} \vdash \mathbf{L}}\ \diamond\text{-L}$$

for future work, but we expect to build on the categorical semantics for its subsystems [1, 10, 14].

As in the surface language, SELECT is subsumed by the more general n-ary rule:

$$\frac{\forall i = 1..n:\ \Gamma, \neg A_i, \overrightarrow{\neg \mathbf{F}_{\mathbf{L}} \mathbf{U}^{\mathbf{T}} A_j}^{j \neq i} \vdash \mathbf{L} \qquad \Gamma \geq \mathbf{T}}{\Gamma, \neg \mathbf{F}_{\mathbf{L}} \mathbf{U}^{\mathbf{T}} A_1, \ldots, \neg \mathbf{F}_{\mathbf{L}} \mathbf{U}^{\mathbf{T}} A_n \vdash \mathbf{L}}\ \text{SELECT-}n$$

6.2 Cut Elimination

Viewed through the lens of the Curry-Howard correspondence, cut elimination is an extremely strong computational claim. The property says that every term, open or closed, can be reduced to normal form. What this means for the surface language is that the concurrency primitives, `return` and `select`, are logically well-founded. Other operators might be built on top of these two primitives, but the core language they form is sufficient.

In addition, cut elimination ensures that the event type $\diamond\, A$ is fully higher-order, so it is safe to compute with events of events or events of functions.

The cut elimination theorem is as follows:

Theorem 1 (Cut Elimination)**.** *If $\Gamma \vdash A^s$ and $\Delta, A^s \vdash (B)^r$ are cut-free derivations in adjoint tensor logic, then there is a cut-free derivation of $\Gamma, \Delta \vdash (B)^r$.*

We prove the theorem by induction on the cut term A^s and on the size of the subderivations. When the second subderivation is a SELECT rule, the proof is not straightforward from the induction hypotheses. We need the help of the following auxiliary lemma:

Lemma 2. *Let $\mathcal{D}_1$ be a cut-free derivation of $\Gamma \vdash \neg\, \mathbf{F}_{\mathbf{L}}\, \mathbf{U}^{\mathbf{T}}\, A$ and $\mathcal{D}_2$ be a cut-free derivation of*

$$\mathcal{D}_2 = \frac{\dfrac{\mathcal{D}_{21}}{\Delta, \neg A, \neg \mathbf{F}_{\mathbf{L}} \mathbf{U}^{\mathbf{T}} B \vdash \mathbf{L}} \qquad \dfrac{\mathcal{D}_{22}}{\Delta, \neg \mathbf{F}_{\mathbf{L}} \mathbf{U}^{\mathbf{T}} A, \neg B \vdash \mathbf{L}}}{\Delta, \neg \mathbf{F}_{\mathbf{L}} \mathbf{U}^{\mathbf{T}} A, \neg \mathbf{F}_{\mathbf{L}} \mathbf{U}^{\mathbf{T}} B \vdash \mathbf{L}}$$

where $\Delta \geq \mathbf{T}$. Then there exists a cut-free derivation of $\Gamma, \Delta, \neg \mathbf{F}_{\mathbf{L}} \mathbf{U}^{\mathbf{T}} B \vdash \mathbf{L}$.

The problem with this case is the fact that SELECT deals with multiple connectives at once: $\neg$, $\mathbf{F}$, and $\mathbf{U}$. Even if the last rule in $\mathcal{D}_1$ introduces the outermost constructor (the negation) the inductive hypothesis from Theorem 1 does not immediately apply to the subderivation. We strengthen the inductive hypothesis for Lemma 2 in the following way: Given $\mathcal{D}_2$,

1. If $\Gamma \vdash \neg\, \mathbf{F}_{\mathbf{L}}\, \mathbf{U}^{\mathbf{T}}\, A$ then $\Gamma, \Delta, \neg \mathbf{F}_{\mathbf{L}} \mathbf{U}^{\mathbf{T}} B \vdash \mathbf{L}$.
2. If $\Gamma, \mathbf{F}_{\mathbf{L}} \mathbf{U}^{\mathbf{T}} A \vdash (C)^{\mathbf{L}}$ then $\Gamma, \Delta, \neg \mathbf{F}_{\mathbf{L}} \mathbf{U}^{\mathbf{T}} B, (\neg C^{\mathbf{L}}) \vdash \mathbf{L}$.
3. If $\Gamma, \mathbf{U}^{\mathbf{T}} A \vdash (C)^{\mathbf{L}}$ then $\Gamma, \Delta, \neg \mathbf{F}_{\mathbf{L}} \mathbf{U}^{\mathbf{T}} B, (\neg C^{\mathbf{L}}) \vdash \mathbf{L}$.
4. If $\Gamma, \mathbf{U}^{\mathbf{T}} A \vdash (C)^{\mathbf{T}}$ then $\Gamma, \Delta, \neg \mathbf{F}_{\mathbf{L}} \mathbf{U}^{\mathbf{T}} B, (\neg \mathbf{F}_{\mathbf{L}}\, C^{\mathbf{T}}) \vdash \mathbf{L}$.
5. If $\Gamma, A \vdash (C)^{\mathbf{L}}$ then $\Gamma, \Delta, \neg \mathbf{F}_{\mathbf{L}} \mathbf{U}^{\mathbf{T}} B, (\neg C^{\mathbf{L}}) \vdash \mathbf{L}$.

The full proof of cut elimination can be found in Appendix A.

6.3 Double Negation Translation

In this section we describe how to transform surface language terms into derivations in the logic. Linear types A from the surface language will be interpreted as temporal propositions $A^{\mathbf{T}}$, since the surface language assumes that all terms are temporally persistent. Unrestricted types τ will be interpreted as persistent propositions $A^{\mathbf{P}}$, taking advantage of the shared type structure. The translation of linear types will integrate the double negation translation, while the persistent types are translated directly. This ensures that the unrestricted fragment of the surface language is semantically just the simply typed lambda calculus.

The linear translation is broken up into two parts. We define $[A] ::= \mathbf{U^T} \neg [A]^-$ where

$$\begin{aligned}
[1]^- &::= \neg 1 \\
[A_1 \otimes A_2]^- &::= \neg (\mathbf{F_L}\,[A_1] \otimes \mathbf{F_L}\,[A_2]) \\
[0]^- &::= \neg 0 \\
[A_1 \oplus A_2]^- &::= \neg (\mathbf{F_L}\,[A_1] \oplus \mathbf{F_L}\,[A_2]) \\
[A_1 \multimap A_2]^- &::= \mathbf{F_L}\,[A_1] \otimes \neg\,\mathbf{F_L}\,[A_2] \\
[\diamond\, A]^- &::= \mathbf{F_L}\,\mathbf{U^T}\,\neg\,\mathbf{F_L}\,[A] \\
[|\tau|]^- &::= \neg\,\mathbf{F_L}\,\mathbf{F_T}\,[\tau]
\end{aligned}$$

The following lemma ensures that the double negation translation is flexible enough to encode linear implication:

Lemma 3. $\Gamma \vdash [A]$ *if and only if* $\Gamma \geq \mathbf{T}$ *and* $\Gamma, [A]^- \vdash \mathbf{L}$.

The unrestricted types translation simply commutes around the operators.

$$\begin{aligned}
[\mathsf{Unit}] &::= \mathsf{Unit} \\
[\tau_1 \times \tau_2] &::= [\tau_1] \times [\tau_2] \\
[\mathsf{Void}] &::= \mathsf{Void} \\
[\tau_1 + \tau_2] &::= [\tau_1] + [\tau_2] \\
[\tau_1 \to \tau_2] &::= [\tau_1] \to [\tau_2] \\
[!\,A] &::= \mathbf{U^P}\,[A]
\end{aligned}$$

Next, well-typed terms in the surface language $\Gamma \vdash t : A$ are translated into doubly negated derivations $[\Gamma] \vdash [A]$ in the sequent calculus. Most of the definitions are straightforward from their types, but the following lemmas provide translations for the $\diamond$ typing rules.

Lemma 4. *The $\diamond$ introduction rule is admissible:*

$$\frac{\Gamma \vdash [A]}{\Gamma \vdash [\diamond\, A]}\ \diamond\textbf{-I}$$

Proof. Recall that $[\diamond\, A]^- = \mathbf{F_L}\,\mathbf{U^T}\,\neg\,\mathbf{F_L}\,[A]$. By Lemma 3, the following derivation is sufficient:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash [A]}{\Gamma \vdash \mathbf{F_L}\,[A]}}{\Gamma, \neg\mathbf{F_L}[A] \vdash \mathbf{L}}}{\Gamma, \mathbf{U^T}\neg\mathbf{F_L}[A] \vdash \mathbf{L}}}{\Gamma, [\diamond\, A]^- \vdash \mathbf{L}}$$

□

To interpret the select rule, we make the following observations:

Proposition 5. $\Gamma \vdash [\diamond\, A]$ *if and only if* $\Gamma, \mathbf{U^T}\neg\mathbf{F_L}[A] \vdash \mathbf{L}$ *where* $\Gamma \geq \mathbf{T}$.

Proposition 6. $\Gamma, [\diamond A] \vdash \mathbf{L}$ *if and only if* $\Gamma, \neg\mathbf{F_L}\mathbf{U^T}\neg\mathbf{F_L}[A] \vdash \mathbf{L}$.

Now we can interpret the surface language select rule.

Lemma 7. *The following derivation is admissible:*

$$\frac{\forall i : \Gamma_i \vdash [\diamond\, A_i] \qquad \Gamma, [A_i], \overrightarrow{[\diamond A_j]}^{j \neq i} \vdash [\diamond\, B]}{\Gamma_1, \ldots, \Gamma_n, \Gamma \vdash [\diamond\, B]}$$

Proof. In order to prove the lemma, it suffices to give a derivation of

$$\Gamma, \mathbf{U} \neg\, \mathbf{F}[B], [\diamond A_1], \ldots, [\diamond A_n] \vdash \mathbf{L},$$

which can be turned into the stated conclusion by cutting against the derivations of $[\diamond\, A_i]$ and applying Proposition 5.

From the statement of the lemma, for each $1 \leq i \leq n$ we have the following derivation:

$$\Gamma, \mathbf{U^T}\neg\mathbf{F_L}[B], [A_i], \overrightarrow{[\diamond A_j]}^{j \neq i} \vdash \mathbf{L}.$$

By Proposition 6, this is equivalent to

$$\Gamma, \mathbf{U^T}\neg\mathbf{F_L}[B], \mathbf{F_L}[A_i], \overrightarrow{\neg\mathbf{F_L}\mathbf{U^T}\neg\mathbf{F_L}[A_j]}^{j \neq i} \vdash \mathbf{L}$$

for each i. Because $\Gamma, \mathbf{U^T}\neg\mathbf{F_L}[B] \geq \mathbf{T}$, we can then apply the $\diamond$-left rule to obtain a derivation of

$$\Gamma, \mathbf{U^T}\neg\mathbf{F_L}[B], \neg\mathbf{F_L}\mathbf{U^T}\neg\mathbf{F_L}[A_1], \ldots, \neg\mathbf{F_L}\mathbf{U^T}\neg\mathbf{F_L}[A_n] \vdash \mathbf{L}.$$

Applying Proposition 6 again yields the desired conclusion:

$$\Gamma, \mathbf{U}\,\neg\mathbf{F_L}[B], [\diamond A_1], \ldots, [\diamond A_n] \vdash \mathbf{L}.$$ □

6.4 GUI primitives

The GUI specific primitives in the surface language do not have meaning in the intermediate logic—we will give their semantics directly in terms of machine configurations. However, to prove that the implementation is safe with respect to the rest of the semantics, we must provide a typing specification. We extend the signature of propositions as follows:

$$\begin{aligned}
A^{\mathbf{T}} &::= \cdots \mid \mathbf{W} \\
A^{\mathbf{P}} &::= \cdots \mid \mathbf{WData} \mid \mathbf{EData}
\end{aligned}$$

We then add interpretation for the GUI-specific types. Since **W** is a temporal proposition, it must satisfy Lemma 3.

$$\begin{aligned}
[\mathbf{W}] &::= \mathbf{U^T}\,\neg\,[\mathbf{W}]^- & [\mathbf{WData}] &::= \mathbf{WData} \\
[\mathbf{W}]^- &::= \neg\,\mathbf{F_L}\,\mathbf{W} & [\mathbf{EData}] &::= \mathbf{EData}
\end{aligned}$$

7. Semantics

With the logical structure of adjoint tensor logic in place, we can move on to defining a Kripke logical relation for the target language semantics.

First, however, we must account for synchronization in the implementation of the select operator. Recall Figure 6, which describes a protocol governing the use of a shared channel c in the synchronization of select. To prove the implementation abides by the protocol, we augment the target language with a notion of *resources*, denoted A and B in the diagram, that govern which expressions have the *permission* to modify a shared channel, and how.

Semantic types are collections of expressions along with the permissions that they own in a particular heap. We say an expression *owns* the resources that allow it to modify the heap. When an expression embedded at a channel in the heap owns some resources, we say that the channel itself is the owner.

Permissions and resources actually form the basis of all heap ownership in the system. We prove that non-shared references treat ownership like a linear resource, taking advantage of the linear logical structure to prove it.

Thus permissions actually track two distinct protocols. The first, required by the select rule, is based on the state transition diagram of Figure 6, and regulates the use of shared state. The second protocol governs the rest of the heap, which is not shared but treated linearly. To simplify the presentation we distinguish shared state as *channels*, with locations given by a set of channel locations Chan. This set is disjoint from the locations for linear-use references, which is denoted Loc. This induces two different kinds of heaps:

- Linear heaps, as in Section 5, are partial functions h from locations in Loc to values.
- Shared heaps are partial functions c from locations in Chan to values.

All linear references will have type **WData**, while channels may have arbitrary types.

We extend the operational semantics to act on updated stores, which consist of the event queue, the heap, and the shared heap. The syntax of expressions for modifying channels remains the same, except that there are two distinct constructors for references versus channels:

$$\frac{\langle \sigma; e\rangle \Downarrow \langle (q, h, c); v\rangle \quad l \notin \mathrm{dom}(h)}{\langle \sigma; \mathsf{ref}(e)\rangle \Downarrow \langle (q, [h \mid l : v], c); l\rangle}\ \textsc{ref}$$

$$\frac{\langle \sigma; e\rangle \Downarrow \langle (q, h, c); v\rangle \quad l \notin \mathrm{dom}(c)}{\langle \sigma; \mathsf{chan}(e)\rangle \Downarrow \langle (q, h, [c \mid l : v]); l\rangle}\ \textsc{chan}$$

The sketch of the semantics is as follows: propositions in adjoint tensor logic will be interpreted as as semantic types. Derivations will then be interpreted as expressions with the permissions they own. The fundamental theorem will show soundness in that every derivation $\Gamma \vdash (A)^r$ in the logic induces an element of the semantic type given by Γ and $(A)^r$. Combined with the double-negation translation of surface language terms, the fundamental theorem defines a compilation strategy from the surface to the target language.

7.1 Resources and Permissions

There are two kinds of resources: one to govern the ownership of references and one to govern the ownership of channels.

$$\mathrm{RefResources} := \{\mathsf{Full}\}$$
$$\mathrm{ChanResources} := \mathbb{N}$$

References are governed by a single resource: an expression either has permission to modify a location, or it does not. For channels, an expression may have permission to take one action but not another. The channel permissions are countably infinite to account for the arbitrary n-ary versions of select. In the binary case, as in Section 5, we only use two of these resources, so we write A for 1 and B for 2.

A *permission* is a partial map from locations to resources. Given some sets of domains, $L \subseteq \mathrm{Loc}$ and $C \subseteq \mathrm{Chan}$, we define $\mathrm{Perm}(L, C)$ to be the type of permissions over L and C:

$$\mathrm{Perm}(L, C) := (L \to \mathcal{P}(\mathrm{RefResources})) \times (C \to \mathcal{P}(\mathrm{ChanResources}))$$

Next we define the protocols. Protocols govern how the permissions owned by an expression affect how that expression can modify the heap. For a specific location l, the relation $v' \sqsubset_X v$ means that the the owner of the resources in X is allowed to mutate the value stored at l from v to v'.

If an expression owns the resource Full for a reference, it may freely modify the reference's value. That is, for any values v and v' we have

$$v' \sqsubset_X v \qquad \text{when } \mathsf{Full} \in X$$

For channels, the only possible values are those given by the states in Figure 6. The necessary permissions to move from one state to the other are specified by the labels on the transitions.

WaitingForA	$\sqsubset_X$	Empty	when $B \in X$
WaitingForB	$\sqsubset_X$	Empty	when $A \in X$
ValueA (v)	$\sqsubset_X$	WaitingForA	when $A \in X$
ContextA (v)	$\sqsubset_X$	WaitingForA	when $B \in X$
ValueB (v)	$\sqsubset_X$	WaitingForB	when $B \in X$
ContextB (v)	$\sqsubset_X$	WaitingForB	when $A \in X$
Done	$\sqsubset_X$	ValueA (v)	when $B \in X$
Done	$\sqsubset_X$	ContextA (v)	when $A \in X$
Done	$\sqsubset_X$	ValueB (v)	when $A \in X$
Done	$\sqsubset_X$	ContextB (v)	when $B \in X$

Protocols can be lifted to express the allowable actions an expression can take for an entire heap. For a permission $R = (H, P)$ and stores $\sigma = (q, h, c)$ and $\sigma' = (q', h', c')$, we define the extension order $\sigma' \leq_R \sigma$ to hold exactly when

$$\forall l \in L, q(l) \subseteq q'(l)$$
$$\wedge\ \forall l \in L, h'(l) \sqsubset_{H(l)} h(l)$$
$$\wedge\ \forall l \in C, c'(l) \sqsubset_{P(l)} c(l)$$

This means that the expression owning R has permission to mutate the store σ into σ'.

7.2 Types and Worlds

In this section we start to define the step-indexed Kripke logical relation that will be used to prove soundness of the logic. The full development can be found in Appendix C, but we present a simplified overview of the main ideas. In particular we leave out the details of step-indexing for the sake of clarity, even though they are necessary for well-founded definitions.

Stores now consist of three parts: a queue, a linear heap, and a shared heap. We define the type of a store to consist of the domains of its two heaps as well as a typing assignment for its channels.

$$\mathrm{StoreType} := \left\{ (L, C, \tau) \;\middle|\; \begin{array}{c} L \subseteq \mathrm{Loc} \ \wedge C \subseteq \mathrm{Chan} \ \wedge \\ \tau \in C \to \mathrm{Type} \end{array} \right\}$$

$$\mathrm{Store}(L, C, \tau) := \left\{ (q, h, c) \;\middle|\; \begin{array}{l} \mathrm{dom}(h) = L \wedge \mathrm{dom}(c) = C \ \wedge \\ \forall l \notin L,\ q(l) = \emptyset \end{array} \right\}$$

We say a store σ is well-typed if there exists some store type Σ such that $\sigma \in \mathrm{Store}(\Sigma)$. Well-typed stores play the role of worlds in the Kripke semantics, meaning that the logical relation is indexed by them.

$$\mathrm{World} := \{(\sigma, \Sigma) \mid \Sigma \in \mathrm{StoreType} \wedge \sigma \in \mathrm{Store}(\Sigma)\}.$$

Semantic types, referenced in the definition of StoreType, map worlds to values with permissions saying how they can modify those worlds.

$$\mathrm{Type} := \mathrm{World} \to \mathcal{P}(\mathrm{Perm} \times \mathrm{Val})$$

In Section 5 we considered what it meant for a store to be safe. We now define Safe to be the semantic type that maps worlds to the unit value with the empty set of permissions.

$$\mathrm{Safe}(w) = \{(\epsilon, ())\}$$

Expressions in Safe don't have permission to modify the heap at all, so intuitively they cannot do any harm.

A permission map ascribes a permission to every event handler in the event queue and to every channel in the shared heap. Let $[n]$ denote the set $\{0, \ldots, n-1\}$ of natural numbers. Given the sets L and C of locations and a size function $s : L \to \mathbb{N}$, a permission map is a function

$$M : (C + \Sigma\, l \in L : [s(l)]) \to \mathrm{Perm}(L, C).$$

The intuition is that $s(l)$ numbers the callbacks registered at each location l, and M associates a permission with each callback and channel. We write $\hat{M}$ to mean the pointwise union of all the permissions in the image of M.

Let w be the world (σ, Σ) where $\sigma = (q, h, c)$ and $\Sigma = (L, C, \tau)$. A permission map M is well-typed with respect to the world w if the following properties hold:

1. $\forall l \in L, h(l) \in$ **WData**;
2. $\forall l \in L, \forall k_i \in q(l), (M(l, i), k_i) \in \Box \neg$ **EData**(w); and
3. $\forall l \in C, (M(l), c(l)) \in \tau(l)(w)$.

In this case we write $M : w$.

This definition relies on two operators on semantic types: $\Box$ and $\neg$. The $\Box$ operator implements temporality by ignoring the event queue:

$$\Box\, A((q,h,c),\Sigma) := A((\emptyset,h,c),\Sigma).$$

What this means is that an expression in $\Box\, A(w)$ has the same permissions in A given any updated world w' derived from executing callbacks in w.

The negation operator describes continuations.

$$\neg\, A(w) := \left\{ (R,u) \;\middle|\; \begin{array}{l} \forall T \# R, \forall w' \leq_T w, \\ \text{whenever } (S,v) \in A(w'), \\ (R \cdot S, uv) \in \mathcal{E}(Safe)(w') \end{array} \right\}$$

Finally, the type $\mathcal{E}(A)$ relates expressions to the values in the semantic type of A.

$$\mathcal{E}(A)(w) = \left\{ (R,u) \;\middle|\; \begin{array}{l} \forall w' \leq_R w, \forall v, \forall M, \\ \text{if } M : w \text{ and } \langle w; e\rangle \Downarrow \langle w'; v\rangle\,, \\ \text{then } (R,v) \in A(w') \end{array} \right\}$$

As a reminder, these definitions are not completely accurate, but are intended to convey the flavor of the precise definitions given in Appendix C. The two biggest omissions are leaving out step indexing (to ensure well-foundedness) and leaving out the possibility of interference on the shared state from other parties with the rights to modify it.

7.3 Logical Relation

Finally, we can interpret propositions from the logic as semantic types. Recall the structure of propositions from Section 6:

$$\begin{aligned} A^{\mathbf{L}} ::=& 1 \mid {A_1}^{\mathbf{L}} \otimes {A_2}^{\mathbf{L}} \mid 0 \mid {A_1}^{\mathbf{L}} \oplus {A_2}^{\mathbf{L}} \mid \neg\, A^{\mathbf{L}} \mid \mathbf{F_L}\, A^{\mathbf{T}} \\ A^{\mathbf{T}} ::=& \mathbf{U^T}\, A^{\mathbf{L}} \mid \mathbf{F_T}\, A^{\mathbf{P}} \\ A^{\mathbf{P}} ::=& \mathsf{Unit} \mid {A_1}^{\mathbf{P}} \times {A_2}^{\mathbf{P}} \mid \mathsf{Void} \mid {A_1}^{\mathbf{P}} + {A_2}^{\mathbf{P}} \\ & \mid {A_1}^{\mathbf{P}} \to {A_2}^{\mathbf{P}} \mid \mathbf{U^P}\, A^{\mathbf{T}} \end{aligned}$$

We will define three mutually-recursive step-indexed Kripke logical relations, shown in Figure 11, to interpret the three classes of propositions.

Contexts Γ are interpreted as pairs $(R;\gamma)$ where R is a permission and γ is a substitution of values for variables. We define an indexed family of substitutions for contexts Γ by induction:

$$\begin{aligned} [\![\cdot]\!](w) &= \{(\epsilon;\cdot)\} \\ [\![\Gamma, x : A^{\mathbf{P}}]\!](w) &= \left\{ (R;\gamma, v/x) \;\middle|\; (R;\gamma) \in [\![\Gamma]\!](w) \wedge v \in [\![A^{\mathbf{P}}]\!] \right\} \\ [\![\Gamma, x : A^{\mathbf{T}}]\!](w) &= \left\{ (R_\Gamma \cdot R_A; \gamma, v/x) \;\middle|\; \begin{array}{l} (R_\Gamma;\gamma) \in [\![\Gamma]\!](w) \wedge \\ (R_A; v) \in [\![A^{\mathbf{T}}]\!](w) \end{array} \right\} \\ [\![\Gamma, x : A^{\mathbf{L}}]\!](w) &= \left\{ (R_\Gamma \cdot R_A; \gamma, v/x) \;\middle|\; \begin{array}{l} (R_\Gamma;\gamma) \in [\![\Gamma]\!](w) \wedge \\ (R_A; v) \in [\![A^{\mathbf{L}}]\!](w) \end{array} \right\} \end{aligned}$$

Finally we can prove soundness of the semantics. The fundamental property says that every derivation in the logic corresponds to an expression in the correct semantic type. The proof of the theorem, which is provided in Appendix C, constructs the expressions, which leads to a compilation strategy similar to the one set out in Section 5.

Theorem 8 (Fundamental Property)**.** *If $\Gamma \vdash (A)^r$ then there is an expression e with free variables in Γ such that for all worlds w and $\langle\sigma;\gamma\rangle \in [\![\Gamma]\!](w)$, we have $\langle\sigma;\gamma(e)\rangle \in \mathcal{E}[\![(A)^r]\!]$.*

8. Discussion

Liveness. An important correctness condition for event-based programs is liveness, which says that the event queue is always responsive enough to handle incoming events. Blocking computations are those computations that never return control to the scheduler. In the target language, these just correspond to non-terminating expressions.

$$\begin{aligned} [\![1]\!](w) &= \{(\epsilon, (\,))\} \\ [\![A^{\mathbf{L}} \otimes B^{\mathbf{L}}]\!](w) &= \left\{ (R \cdot S, (v_1, v_2)) \;\middle|\; \begin{array}{l} (R, v_1) \in [\![A^{\mathbf{L}}]\!](w) \wedge \\ (S, v_2) \in [\![B^{\mathbf{L}}]\!](w) \end{array} \right\} \\ [\![0]\!](w) &= \emptyset \\ [\![A^{\mathbf{L}} \oplus B^{\mathbf{L}}]\!](w) &= \begin{array}{l} \left\{ (R, \mathbf{in}_1\, v) \;\middle|\; (R,v) \in [\![A^{\mathbf{L}}]\!](w) \right\} \cup \\ \left\{ (R, \mathbf{in}_2\, v) \;\middle|\; (R,v) \in [\![B^{\mathbf{L}}]\!](w) \right\} \end{array} \\ [\![\neg\, A^{\mathbf{L}}]\!](w) &= (\neg\, [\![A^{\mathbf{L}}]\!])(w) \\ [\![\mathbf{F_L}\, A^{\mathbf{T}}]\!](w) &= [\![A^{\mathbf{T}}]\!](w) \\ [\![\mathbf{U^T}\, A^{\mathbf{L}}]\!](w) &= (\Box \neg \neg\, [\![A^{\mathbf{L}}]\!])(w) \\ [\![\mathbf{F_T}\, A^{\mathbf{P}}]\!](w) &= \left\{ (\epsilon, v) \;\middle|\; v \in [\![A^{\mathbf{P}}]\!] \right\} \\ [\![\mathsf{Unit}]\!] &= \{(\,)\} \\ [\![A^{\mathbf{P}} \times B^{\mathbf{P}}]\!] &= \left\{ (v_1, v_2) \;\middle|\; v_1 \in [\![A^{\mathbf{P}}]\!] \wedge v_2 \in [\![B^{\mathbf{P}}]\!] \right\} \\ [\![0]\!] &= \emptyset \\ [\![A^{\mathbf{P}} + B^{\mathbf{P}}]\!] &= \left\{ \mathbf{in}_1\, v \;\middle|\; v \in [\![A^{\mathbf{P}}]\!] \right\} \cup \left\{ \mathbf{in}_2\, v \;\middle|\; v \in [\![B^{\mathbf{P}}]\!] \right\} \\ [\![A^{\mathbf{P}} \to B^{\mathbf{P}}]\!] &= \left\{ v \;\middle|\; \forall v' \in [\![A^{\mathbf{P}}]\!], v\, v' \in [\![B^{\mathbf{P}}]\!] \right\} \\ [\![\mathbf{U^P}\, A^{\mathbf{T}}]\!] &= \left\{ v \;\middle|\; (\epsilon, v\,(\,)) \in [\![A^{\mathbf{T}}]\!](\cdot) \right\} \end{aligned}$$

Figure 11. Kripke interpretation of propositions.

The `sfix` operator defines non-terminating computations that are also non-blocking by attaching each recursive call of the function to the event queue. This technique is commonly used in even non-monadic event programming languages to define non-blocking recursion—see for example the `setTimeout` paradigm in JavaScript [18].

Unfortunately, the step-indexed semantics described in Section 7 is not equipped to prove liveness guarantees, even for the language without `sfix`. We leave a thorough analysis of liveness to future work.

Message passing. We have shown that the surface language in Section 4 makes a robust core language for expressing GUI programs. But the monadic style is equally well-suited for other domains of event-driven programming. One promising domain is message-passing, in which primitive operations include sending and receiving messages over communication channels (not to be confused with the synchronization channels from Section 7). Channels replace widgets as the type of mutable references in the event queue. The act of receiving along a channel is equivalent to waiting for GUI events by invoking the `onEvent` primitive. The final operator, the `send` function, would correspond to the invocation of callbacks in the event queue.

Affine events. The linear restriction on types in the surface language provides a surprisingly clean way to incorporate stateful computations in a pure functional setting. It may be the case that this restriction can be weakened to only *affine* types, allowing variables to be dropped but not duplicated. Computationally this would correspond to flushing event handlers from the event queue.

This paper draws on ideas from type theory, programming language design, propositional logic, and logical relations to study event-based reactive programming from multiple perspectives. We discover that the monadic style used by many real-world languages

has logical foundations in temporal and linear logic. We exploit this foundation to prove the existence of a sound interpretation in an effectful language. The result of this study is a valuable application of the Curry-Howard correspondence in a surprising domain.

References

[1] P. Benton. A mixed linear and non-linear logic: Proofs, terms and models. In L. Pacholski and J. Tiuryn, editors, *Computer Science Logic*, volume 933 of *Lecture Notes in Computer Science*, pages 121–135. Springer Berlin Heidelberg, 1995. doi: 10.1007/BFb0022251.

[2] K. Crary, A. Kliger, and F. Pfenning. A monadic analysis of information flow security with mutable state. *Journal of Functional Programming*, 15:249–291, 2005. ISSN 1469-7653. doi: 10.1017/S0956796804005441.

[3] R. Davies and F. Pfenning. A modal analysis of staged computation. 48(3):555–604, May 2001. doi: 10.1145/382780.382785.

[4] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1 – 101, 1987. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/0304-3975(87)90045-4.

[5] P. Haller, A. Prokopec, H. Miller, V. Klang, R. Kuhn, and V. Jovanovic. Futures and promises. Scala Documentation, 2013. URL `http://docs.scala-lang.org/overviews/core/futures`.

[6] A. Jeffrey. LTL types FRP: Linear-time temporal logic propositions as types, proofs as functional reactive programs. In *Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification*, PLPV '12, pages 49–60, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1125-0. doi: 10.1145/2103776.2103783.

[7] W. Jeltsch. Towards a common categorical semantics for linear-time temporal logic and functional reactive programming. *Electronic Notes in Theoretical Computer Science*, 286(0):229 – 242, 2012. doi: http://dx.doi.org/10.1016/j.entcs.2012.08.015. Proceedings of the 28th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVIII).

[8] N. R. Krishnaswami and N. Benton. Ultrametric semantics of reactive programs. In *Logic in Computer Science (LICS), 2011 26th Annual IEEE Symposium on*, pages 257–266. IEEE, June 2011. doi: 10.1109/LICS.2011.38.

[9] S. Mac Lane. *Categories for the working mathematician*, volume 5. Springer Science & Business Media, 1978.

[10] P.-A. Melliès and N. Tabareau. Resource modalities in tensor logic. *Annals of Pure and Applied Logic*, 161(5):632–653, 2010. ISSN 0168-0072. doi: http://dx.doi.org/10.1016/j.apal.2009.07.018. The Third workshop on Games for Logic and Programming Languages (GaLoP) Galop 2008.

[11] Y. Minsky, A. Madhavapeddy, and J. Hickey. *Real World OCaml*. O'Reilly Media, 2013.

[12] N. Mix. Narrative JavaScript, 2007. URL `http://www.neilmix.com/narrativejs`.

[13] V. Murphy, Tom, K. Crary, R. Harper, and F. Pfenning. A symmetric modal lambda calculus for distributed computing. In *Logic in Computer Science, 2004. Proceedings of the 19th Annual IEEE Symposium on*, pages 286–295, July 2004. doi: 10.1109/LICS.2004.1319623.

[14] J. Paykin and S. Zdancewic. A linear/producer/consumer model of classical linear logic. In S. Alves and I. Cervesato, editors, *Proceedings Third International Workshop on Linearity, LINEARITY 2014, Vienna, Austria, 13th July, 2014.*, volume 176 of *EPTCS*, pages 9–23, 2015. doi: 10.4204/EPTCS.176.2.

[15] F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. *Mathematical structures in computer science*, 11(04):511–540, 8 2001. ISSN 1469-8072. doi: 10.1017/S0960129501003322.

[16] A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

[17] J. Reed. A judgmental deconstruction of modal logic. 2009.

[18] S. Stefanov. *JavaScript Patterns*. O'Reilly Media, 2010. ISBN 9781449396947.

[19] H. Thielecke. From control effects to typed continuation passing. *SIGPLAN Not.*, 38(1):139–149, Jan. 2003. ISSN 0362-1340. doi: 10.1145/640128.604144.