The Essence of Event-Driven Programming

Jennifer Paykin[1], Neelakantan R. Krishnaswami[2], and Steve Zdancewic[3]

1 **University of Pennsylvania, Philadelphia, USA**
`jpaykin@seas.upenn.edu`
2 **University of Birmingham, Birmingham, United Kingdom**
`N.Krishnaswami@cs.bham.ac.uk`
3 **University of Pennsylvania, Philadelphia, USA**
`stevez@cis.upenn.edu`

Abstract

Event-driven programming is based on a natural abstraction: an event is a computation that can eventually return a value. This paper exploits the intuition relating events and time by drawing a Curry-Howard correspondence between a functional event-driven programming language and a *linear-time temporal logic*. In this logic, the eventually proposition $\Diamond A$ describes the type of events, and Girard's *linear logic* describes the effectful and concurrent nature of the programs. The correspondence reveals many interesting insights into the nature of event-driven programming, including a generalization of selective choice for synchronizing events, and an implementation in terms of callbacks where $\Diamond A$ is just $\neg\Box\neg A$.

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Event-driven programming is a popular approach to functional concurrency in which events, also known as "futures," "deferred values," or "lightweight threads," execute concurrently with each other and eventually produce a result. The abstraction of the event has been extremely successful in producing lightweight, extensible, and efficient concurrent programs. As a result, a wide range of programming languages and libraries use the event-driven paradigm to describe everything from message-passing concurrency [24] and lightweight threads [26] to graphical user interfaces [11] and I/O [13, 21, 25].

Although these systems vary considerably in the details of their implementations and APIs, they share a common, basic structure. They provide:

- the *abstraction* of the event, often given explicitly as a *monad*;
- the ability to *synchronize* across events;
- a continuation-passing style implementation in terms of *callbacks*; and
- a source or sources of *primitive events.*

In this paper we distill event-driven programming to its essence, demonstrating how to derive these components from first principles. Starting from a logical basis and building up the minimal machinery needed to explain the four points above, we proceed as follows:

The logic of events. In Section 2 we identify a Curry-Howard correspondence between events that eventually return a value and the $\Diamond$ ("eventually") modality from *temporal logic.* We define a core language of pure events where the monad from the event-driven abstraction is identified as $\Diamond$. We formulate the type system in the setting of Girard's *linear logic* to characterize the fact that events are effectful and execute concurrently. This linear and monadic logic serves as the basic scaffolding for event-driven computations.

Synchronization refers to the ability to execute two events concurrently and record which one happens first. In Section 3 we extend the type system of pure events to include a synchronization operator `choose` in the style of Concurrent ML [24]. We observe that, logically, `choose` corresponds to the linear-time axiom of temporal logic:

$$\Diamond A \wedge \Diamond B \to \Diamond((A \wedge \Diamond B) \vee (\Diamond A \wedge B)) \qquad \text{“eventually, } A \text{ happens before } B \text{ or } B \text{ happens before } A\text{”}$$

This logical interpretation of `choose` suggests a natural generalization in terms of McBride's derivatives of types [19], which we develop into a new technique for synchronizing events across arbitrary container data structures.

Callbacks, continuations, and the event loop. In the event-driven paradigm, events are implemented using callbacks that interact with an underlying event loop. For the language of pure events, we define in Section 4 a time-aware continuation-passing style (CPS) translation based on the property of temporal logic that $\Diamond A$ is equivalent to $\neg\Box\neg A$, where negation $\neg$ is the type of first-class continuations, and $\Box$ is the “always” operator from temporal logic.

In addition to the logic of pure events, the implementation should take into account the extralogical *sources of concurrency* that interact with the event loop. Throughout this paper we use a range of these concurrency primitives, including nondeterministic events, timeouts, user input, and channels, and argue that the choice of primitive is orthogonal to the logical structure of events.

In Section 5 we extend the CPS translation to account for these axiomatic sources of concurrency. We model the event loop in the *answer type* of the CPS translation [7] and show how to instantiate the answer type for a concrete choice of event primitives.

The essence of events. In this paper we argue that the logical interpretation of events is a unifying idea behind the vast array of real-world event-driven languages and libraries. We complete the story in Section 6 by comparing techniques used in practice with the approaches developed in this paper based on the essence of event-driven programming.

2 The Curry-Howard Connection

The important abstraction of event-driven programming is based on the relationship between events and time: an event is a computation that can eventually return a value. This abstraction takes the form of a *monad*, which means that there is a simple interface for interacting with events:

- `return` e is a trivial event that immediately returns the value e.
- `bind` $x = e_1$ `in` e_2 is an event that first waits for e_1 to return a value, binds that value to x, and then continues as the event e_2.

In this paper we go even further by identifying the particular monad for events with the “eventually” operator from temporal logic. Consider a simple intuitionistic logic for time in which a proposition is either *true now* or *true later*. A proposition that is true now is also true at every point in the future, but a proposition that is true later may not necessarily be true now.[1] A proposition that is true later is denoted $\Diamond A$, and pronounced “eventually A.”

[1] Other presentations of temporal logic include next ($\circ$), always ($\Box$), and until ($\mathcal{U}$) operators, and do not necessarily assume that just because a proposition is true now, it will always be true.

The "eventually" modality $\Diamond A$ is defined by two rules. The first says that if a proposition is true now, it is also true later. The second rule says that if A is true later, and if A now proves that some B is true later, then B itself is also true later. Through the Curry-Howard correspondence, these proofs correspond to typing rules for `return` and `bind`, respectively.

$$\frac{\Delta \vdash e : A}{\Delta \vdash \texttt{return}\, e : \Diamond A} \qquad \frac{\Delta_1 \vdash e_1 : \Diamond A \quad \Delta_2, x : A \vdash e_2 : \Diamond B}{\Delta_1, \Delta_2 \vdash \texttt{bind}\, x = e_1\, \texttt{in}\, e_2 : \Diamond B}$$

2.1 A logic for effects: linear logic

Consider the event $\texttt{nondet}\, e$ that returns e at some nondeterministic point in the future. The program $\texttt{foo} = (\texttt{let}\ x = \texttt{nondet}\, e\ \texttt{in}\ \texttt{in}_1\, (x, x))$ has state $\texttt{in}_1\, (\textit{undefined}, \textit{undefined})$ for some nondeterministic amount of time before simultaneously stepping to $\texttt{in}_1\, (\texttt{return}\, e, \texttt{return}\, e)$. In particular, `foo` is *not* equivalent to the substituted form $\texttt{in}_1\, (\texttt{nondet}\, e, \texttt{nondet}\, e)$, which at some point in the future may have its first component be $\texttt{return}\, e$ and its second component be *undefined*. On the other hand, the computation should not be blocking; the expression $\texttt{case foo of}\ (\texttt{in}_1\, y \to \texttt{True} \mid \texttt{in}_2\, z \to \texttt{False})$, which tests whether `foo` is a right or a left injection, should resolve immediately to `True`.

Events like `foo` are inherently effectful, because the state of an event changes as computation progresses. In order to describe events in a purely logical way, the Curry-Howard correspondence has to take these effectful relationships into account.

Linear logic [12] is one approach for typing effectful programs that has been successful for concurrency in the settings of session types [5, 27] and uniqueness types [14]. Linear-use (or "one shot") data structures show up in many event-driven languages, for example in the form of Ivars in Async [21], Futures in Scala [13], and widgets in GUI programming [17]. Linear types are so useful in concurrent programming because they disallow aliasing, meaning that there is no shared state between processes. By using linear variables we can define an *equational operational semantics* that restores the Curry-Howard correspondence with logic.[2]

2.2 A logic without effects: the unrestricted fragment

The linear and temporal type system will account for effects and events, but event-driven programs also include ordinary computations that don't require the event or linear-space abstractions. For these we should be able to write terms in a regular programming language—in this case, the (non-linear) simply-typed λ-calculus.

Benton et al. [4] introduced a way to combine linear and non-linear logics that has since been called *adjoint logic* [23] after the categorical structure. In an adjoint logic, we write non-linear types τ to distinguish them from linear types A, and non-linear typing contexts Γ to distinguish them from linear typing contexts Δ. The non-linear typing judgment has the form $\Gamma \vdash t : \tau$, while the linear one has the form $\Gamma; \Delta \vdash e : A$. That is, the linear typing judgment allows unrestricted access to the non-linear variables in Γ, but only linear-space access to the variables in Δ.

Adjoint logic also describes ways in which the linear and non-linear types relate to each other. A linear type A can be embedded into a persistent type $\lceil A \rceil$ when A does not rely

[2] Other event-driven languages solve this problem in different ways without using linear types. Implementations in strict functional languages like Async [21] ensure that every event normalizes immediately to an asynchronous primitive, which somewhat defeats the purpose of a strict evaluation order. In CML [24], all events evaluate strictly and synchronously unless wrapped in a thunk called `guard`.

$$\begin{aligned}
\tau &::= \mathsf{Unit} \mid \mathsf{Void} \mid \tau \times \tau \mid \tau + \tau \mid \tau \to \tau \mid \lceil A \rceil \\
A &::= 1 \mid 0 \mid A \otimes A \mid A \oplus A \mid A \multimap A \mid \Diamond A \mid \lfloor \tau \rfloor
\end{aligned}$$

$$\begin{aligned}
t &::= x \mid () \mid \texttt{case}\ \ t\ \ \texttt{of}\ () \mid (t_1, t_2) \mid \pi_i\, t \mid \texttt{in}_i\, t \mid \texttt{case}\ \ t\ \ \texttt{of}\ (\texttt{in}_1\, x_1 \to t_1 \mid \texttt{in}_2\, x_2 \to t_2) \\
&\quad \mid \lambda x.t \mid t_1\, t_2 \mid \texttt{suspend}\, e \\
e &::= x \mid () \mid \texttt{let}\ () = e_1\ \texttt{in}\ e_2 \mid \texttt{case}\ \ e\ \ \texttt{of}\ () \\
&\quad \mid (e_1, e_2) \mid \texttt{let}\ (x_1, x_2) = e_1\ \texttt{in}\ e_2 \mid \texttt{in}_i\, e \mid \texttt{case}\ \ e\ \ \texttt{of}\ (\texttt{in}_1\, x_2 \to e_1 \mid \texttt{in}_2\, x_2 \to t_2) \\
&\quad \mid \lambda x.e \mid e_1\, e_2 \mid \texttt{return}\, e \mid \texttt{bind}\, x = e_1\ \texttt{in}\ e_2 \mid \texttt{force}\, t \mid \lfloor t \rfloor \mid \texttt{let}\ \lfloor x \rfloor = e_1\ \texttt{in}\ e_2
\end{aligned}$$

Figure 1 Syntax of linear and non-linear types, terms, and expressions.

$$\frac{\Gamma; \cdot \vdash e : A}{\Gamma \vdash \texttt{suspend}\, e : \lceil A \rceil}\ \lceil - \rceil\text{-I} \qquad \frac{\Gamma \vdash t : \lceil A \rceil}{\Gamma; \cdot \vdash \texttt{force}\, t : A}\ \lceil - \rceil\text{-E}$$

$$\frac{\Gamma \vdash t : \tau}{\Gamma; \cdot \vdash \lfloor t \rfloor : \lfloor \tau \rfloor}\ \lfloor - \rfloor\text{-I} \qquad \frac{\Gamma; \Delta_1 \vdash e_1 : \lfloor \tau \rfloor \quad \Gamma, x : \tau; \Delta_2 \vdash e_2 : B}{\Gamma; \Delta_1, \Delta_2 \vdash \texttt{let}\ \lfloor x \rfloor = e_1\ \texttt{in}\ e_2 : B}\ \lfloor - \rfloor\text{-E}$$

Figure 2 Typing rules for moving between the linear and unrestricted fragments.

on any linear assumptions. On the other hand, a persistent type τ can always be treated as a linear type, written $\lfloor \tau \rfloor$. Figure 1 shows the syntax of types, (non-linear) terms, and (linear) expressions.

The typing rules for terms and expressions are mostly standard, but Figure 2 shows how to move in between the linear and non-linear fragments via the typing rules for $\lceil A \rceil$ and $\lfloor \tau \rfloor$. A linear expression e can be suspended to a persistent term $\texttt{suspend}\, e$, and can be unsuspended using `force`. On the other hand, a non-linear term can be used linearly in this type system by applying a floor operator, and unpacked in the same way.

The remaining typing rules are shown in Appendix A.

2.3 Completing the Curry-Howard connection

We have shown how the Curry-Howard correspondence relates the propositions of temporal linear logic to types and the proofs of such propositions to event-driven programs. In the remainder of this section we show how the operational semantics of event-based programs relates to the equational theory of proofs.

Events use a call-by-name evaluation strategy, because events themselves are not expected to normalize right away. Consider the nondeterministic event $\texttt{nondet}\, e$ introduced in the beginning of this section. In the expression $\texttt{let}\ x = \texttt{nondet}\, e\ \texttt{in}\ \texttt{in}_1\, x$, the variable x occurs linearly in the rest of the expression and so it is safe to substitute the unresolved event $\texttt{nondet}\, e$ for x.

On the other hand, terms have a call-by-value evaluation strategy, which is safe because a suspended expression $\texttt{suspend}\, e$ is a value in the term language.

A selection of the operational semantics for expressions is shown in Figure 3. Evaluation occurs under contexts, which have the form E^{P} for contexts with holes for non-linear terms, and E^{L} for contexts with holes for linear expressions. A full description of the operational

$$\begin{array}{rl} \texttt{bind}\, x = \texttt{return}\, e_1\, \texttt{in}\, e_2 \rightsquigarrow e_2\{e_1/x\} & \dfrac{t \rightsquigarrow t'}{E^{\mathsf{P}}[t] \rightsquigarrow E^{\mathsf{P}}[t']} \\ \texttt{force}(\texttt{suspend}\, e) \rightsquigarrow e & \\ \texttt{let}\, \lfloor x \rfloor = \lfloor v \rfloor\, \texttt{in}\, e \rightsquigarrow e\{v/x\} & \\ E^{\mathsf{P}} ::= \cdots \mid \texttt{force}[\,] \mid \lfloor [\,] \rfloor & \dfrac{e \rightsquigarrow e'}{E^{\mathsf{L}}[e] \rightsquigarrow E^{\mathsf{L}}[e']} \\ E^{\mathsf{L}} ::= \cdots \mid \texttt{bind}\, x = [\,]\, \texttt{in}\, e \mid \texttt{let}\, \lfloor x \rfloor = [\,]\, \texttt{in}\, e & \end{array}$$

Figure 3 Operational semantics of terms and expressions.

semantics can be found in Appendix B.

▶ **Theorem 1** (Preservation). *If $\vdash t : \tau$ and $t \rightsquigarrow t'$ then $\vdash t' : \tau$. If $\vdash e : A$ and $e \rightsquigarrow e'$ then $\vdash e' : A$.*

▶ **Theorem 2** (Progress). *If $\vdash t : \tau$ then either t is a value or t can take a step.*
If $\vdash e : A$ then either e is in weak head normal form or e can take a step.

3 Synchronization and Linear Time

In the previous section we described a simple logic for pure events that is not particularly expressive. Although it can express sequential events using `return` and `bind`, it cannot express concurrent events running in parallel. As an example, consider a timeout event `onTimeout` n that returns a unit value after the amount of time specified by n. With this operator we can limit the execution of another event e by running `onTimeout` n and e in parallel and keeping track of which one occurs first. We denote the operation that runs two events in parallel as `choose` and give it the type $\Diamond A \otimes \Diamond B \multimap \Diamond(A \otimes \Diamond B \oplus \Diamond A \otimes B)$. The `choose` operator is a kind of selective choice operator as seen in CML and Async.

```
timeout e |n| = bind z = choose (e, onTimeout |n|) in
                case z of | in1 (a,t)  -> let () = drop t in return (Some a)
                          | in2 (e,()) -> let () = drop e in return None
                          end
```

Here `drop`, of type $\Diamond A \multimap 1$, is an operation that explicitly aborts an event.

The type of `choose` can be interpreted as a property of temporal logic: if both A and B will be true at some point in the future, then either A will come before B, or B will come before A. This axiom distinguishes a *linear-time temporal logic* from a *branching-time temporal logic*, in which two different futures could occur in different timelines.

The logical interpretation of `choose` reveals that it is somehow fundamental to the event-driven interpretation. In fact, we can think of `choose` as part of the operational semantics of events.[3] We extend the evaluation contexts so that the two events being synchronized on can evaluate concurrently. Once one of the events resolves to a value, the synchronization operator can take a β-reduction step.

$$E^{\mathsf{L}} ::= \cdots \mid \texttt{choose}([\,], e) \mid \texttt{choose}(e, [\,]) \qquad \begin{array}{l} \texttt{choose}(\texttt{return}\, e_1, e_2) \rightsquigarrow \texttt{return}(\texttt{in}_1\,(e_1, e_2)) \\ \texttt{choose}(e_1, \texttt{return}\, e_2) \rightsquigarrow \texttt{return}(\texttt{in}_2\,(e_1, e_2)) \end{array}$$

[3] These rules make sense operationally, but not necessarily as part of the Curry-Howard correspondence, because as an equational theory they relate $\texttt{return}(\texttt{in}_1\,(e_1, \texttt{return}\, e_2))$ and $\texttt{return}(\texttt{in}_2\,(\texttt{return}\, e_1, e_2))$, which are certainly not equal. This stems from the fact that `choose` is not a pure logical axiom; it relates multiple connectives in a complicated way and is hence neither an introduction nor an elimination rule.

$$\tau ::= \cdots \mid \mathsf{Chan}\ A$$

$$\texttt{spawn} : \Diamond 1 \multimap 1$$

$$\texttt{new} : 1 \multimap \lfloor \mathsf{Chan}\ A \rfloor$$

$$\texttt{send} : A \multimap \lfloor \mathsf{Chan}\ A \rfloor \multimap \Diamond 1$$

$$\texttt{receive} : \lfloor \mathsf{Chan}\ A \rfloor \multimap \Diamond A$$

```
choose (eA,eB) =
  let |win| : Chan (1⊕1) = new () in
  let |cA|  : Chan A = new () in
  let |cB|  : Chan B = new () in
  spawn (bind a = eA in
         spawn (send |win| (in1 ()));
         send |cA| a);
  spawn (bind b = eB in
         spawn (send |win| (in2 ()));
         send |cB| b);
```

```
bind z = receive |win| in
case z of
| in1 () ->
  bind a = |receive |cA| in
  return (in1 (a,receive |cB|))
| in2 () ->
  bind b = receive |cB| in
  return (in2 (receive |cA|,b))
end
```

Figure 4 Signature of channels and the channel-based implementation of `choose`

3.1 Implementing `choose` with channels

As we hinted above, the `choose` operator cannot be implemented in the language of pure events. Logically this is because `choose` corresponds to the linear-time *axiom* from temporal logic, so it is not derivable from the other operators. However, as we describe here, we can implement `choose` in terms of primitive sources of concurrency: in this case, linear synchronous channels and the `spawn` operator.

The `spawn` operator takes an event with return type 1 and executes it asynchronously. A synchronous channel is a way to communicate linear information between processes. Although the data transmitted across channels is linear, the reference to the channel itself need not be, so $\mathsf{Chan}\ A$ is added as a non-linear type τ to our type system. Following Reppy [24], `send` and `receive` are synchronous in that they do not return a value until a send is matched with a receive. The signature of linear message-passing is summarized in Figure 4.

The implementation of `choose`, also shown in Figure 4, uses three channels. The first, called `win`, tracks which event of `eA` or `eB` occurs first. Based on that, `choose` will wait for either `eA` or `eB` explicitly, through the intermediate channels `cA` and `cB`. Then `choose` will return either `in1` or `in2` along with a reference to the intermediate event, the receive event on the unresolved channel.

3.2 Synchronizing more than pairs

For practical programming problems, we often need to synchronize more than two events at a time. For example, the `choose` operators in CML and Async operate over lists instead of pairs. In the linear type system of this paper, the type of such an operator would be

$$\texttt{chooseList} : \mathsf{List}(\Diamond A) \multimap \mathsf{List}(\Diamond A) \otimes A \otimes \mathsf{List}(\Diamond A)$$

where the input list is partitioned into the prefix and suffix of the first event to return a value. Unfortunately it is not possible to derive `chooseList`, or even a version over triples of events, from the binary version of choose. Like `choose` itself we need to implement `chooseList` using channels or some other concurrency primitive. By itself this solution is ad-hoc and unsatisfactory.

In the remainder of this section we describe a way to build up synchronization operators on *arbitrary finite containers of events* by induction on the type of the container. We do this by exploiting a uniform pattern on the structure of these primitives, inspired by Conor

$$\partial_\Diamond \Diamond A = A \qquad \partial_\Diamond (A \otimes B) = (\partial_\Diamond A \otimes B) \oplus (A \otimes \partial_\Diamond B) \qquad \partial_\Diamond (A \multimap B) = 0$$
$$\partial_\Diamond 1 = 0 \qquad \partial_\Diamond (A \oplus B) = \partial_\Diamond A \oplus \partial_\Diamond B \qquad \partial_\Diamond \lfloor \tau \rfloor = 0$$
$$\partial_\Diamond 0 = 0$$

$$E^{\mathsf{L}} ::= \cdots \mid \texttt{choose}\, E^\Diamond \qquad \mathit{fill}\,[\,]e = e$$
$$E^\Diamond ::= [\,] \mid \texttt{in}_i\, E^\Diamond \mid (E^\Diamond, e) \mid (e, E^\Diamond) \qquad \mathit{fill}\,(\texttt{in}_i\, E^\Diamond)e = \texttt{in}_i\,(\mathit{fill}\ E^\Diamond\, e)$$
$$\mathit{fill}\,(E^\Diamond, e')e = \texttt{in}_1\,(\mathit{fill}\ E^\Diamond\, e, e')$$
$$\texttt{choose}\, E^\Diamond[\texttt{return}\, e] \rightsquigarrow \texttt{return}(\mathit{fill}\ E^\Diamond\, e) \qquad \mathit{fill}\,(e', E^\Diamond)e = \texttt{in}_2\,(e', \mathit{fill}\ E^\Diamond\, e)$$

Figure 5 Derivatives with respect to time

McBride's 2001 observation that the derivative of a regular type is the type of its one-hole contexts [19]. We explore a variation on his idea and show that *the derivative of a type with respect to time is the type of its synchronization operator.*

Derivatives with respect to time. We define the *instant* of an event to be the time at which it returns a value. An event itself can be thought of as a context with a hole for time that is filled in by its instant. For example, the event $\texttt{return}\, n$ consists of the context $[\,]n$, where the hole $[\,]$ is filled in by its instant "now."

The semantics of synchronization say that the instant of $\texttt{choose}(e_1, e_2)$ is either the instant of e_1 or the instant of e_2, depending on which occurs first. The context containing that hole has one of two shapes. If e_1 returns a value n before e_2 does, then the context will have the the form $([\,]n, e_2)$, of type $A \otimes \Diamond B$. If e_2 returns a value first, the context will have the type $\Diamond A \otimes B$. Thus the return type of `choose`, $(A \otimes \Diamond B) \oplus (\Diamond A \otimes B)$, describes the possible shapes of its context with a hole for time.

McBride's partial derivative operation, written $\partial_X A$, records the possible shapes of a one-hole context of A with a hole for the type X.[4] This intuition extends from finite containers to recursive data types like lists. For example, the one-hole contexts of the type $\mathsf{List}\, A$ consist of a one-hole context of A, along with the prefix and suffix lists surrounding the element with the hole. That is, the derivative of $\mathsf{List}\, A$ is $\mathsf{List}\, A \otimes \partial_X A \otimes \mathsf{List}\, A$, which is reminiscent of the return type of `chooseList`.

In Figure 5 we define the syntactic operation $\partial_\Diamond A$ on types that describes the derivative with respect to time.[5] The derivative of an event type $\Diamond A$ is A itself, leaving the time at which the event occurred as the hole.

The general `choose` operator decomposes a type into two parts: its instant (designated by the $\Diamond$ prefix) at which synchronization will occur, and a context with a hole for time.

$$\texttt{choose}_A : A \multimap \Diamond(\partial_\Diamond A) \qquad \text{(when } \partial_\Diamond A \text{ is not degenerate, i.e. } \partial_\Diamond A \ncong 0)$$

This gives us a pattern for synchronizing events across arbitrary finite containers. McBride's treatment of recursive types provides a way to extend synchronization to arbitrary recursive containers such as lists, but we leave the details to future work.

[4] The syntax is inspired by the fact that this operation obeys the product and sum rules from calculus. For example, if we write X^2 for $X \times X$ then $\partial_X X^2 \cong 2 \times X = X + X$.

[5] The one-hole context interpretation of derivatives does not extend to higher-order types, so we make the simplification that the derivative of all higher-order types is 0.

Operational semantics. The operational behavior of $\texttt{choose}_A$ is also shown in Figure 5. The evaluation contexts E^{L} are extended with $\texttt{choose}\, E^{\Diamond}$ where $E^{\Diamond}$ is a special kind of context for expressions. We write $\Gamma;\Delta \vdash A :> E^{\Diamond} : B$ to mean that $E^{\Diamond}$ has a hole for expressions of type A.

The β-rule applies when a component of any context returns a value. In this case the synchronization operator must produce an expression of type $\partial_{\Diamond} A$. We define an operation *fill* $E^{\Diamond} e$ that constructs the element of the derivative from the context and the component filling in the hole.

▶ **Lemma 3.** *If* $\Gamma;\Delta_1 \vdash \Diamond A :> E^{\Diamond} : B$ *and* $\Gamma;\Delta_2 \vdash e : A$*, then* $\Gamma;\Delta_1,\Delta_2 \vdash \mathit{fill}\ E^{\Diamond} e : \partial_{\Diamond} B$.

The above lemma is enough to prove preservation in the extended system. The side condition on $\texttt{choose}_A$ that $\partial_{\Diamond} A \ncong 0$ ensures that progress also holds by ruling out ill-formed terms like $\texttt{choose}(\lambda x.e)$.

4 Logic and Callbacks

The interpretation of events as the eventually type from temporal logic has led to some interesting insights about their behavior, including the meaning of synchronization. In the following sections we discuss how the standard implementation of events as callbacks can also be given a logical interpretation.

Callbacks, which are also called continuations and event handlers, are functions that accept some input but do not return a value. Rather, the return type of a callback is invisible, so it can be represented as $A \to \mathsf{Answer}$ for any type Answer. We could instead write its type as $\neg A$, making the answer type opaque to the programmer.

Consider an implementation of a GUI library using callbacks and an event loop. The library provides a way to register handlers in the event loop that get triggered once the user performs some external action, such as a key press.

$$\texttt{onKeyPress} : (\mathsf{Char} \to \mathsf{Answer}) \to \mathsf{Answer} \qquad \text{or} \qquad \texttt{onKeyPress} : \neg\neg\mathsf{Char}$$

What does the type of a callback have to do with temporal logic? The callback being registered in the event loop will not be invoked immediately, but at some point in the future, once the user has pressed a key. The type of the callback itself should then reflect the fact that it will be available in the future. We write $\Box A$ to denote the fact that A will be true at every point in the future, and so the type of `onKeyPress` should be written

$$\texttt{onKeyPress} : \Box(\mathsf{Char} \to \mathsf{Answer}) \to \mathsf{Answer} \quad \text{or} \quad \texttt{onKeyPress} : \neg\Box\neg\mathsf{Char} \quad \text{or} \quad \texttt{onKeyPress} : \Diamond\mathsf{Char}$$

This last step follows from the isomorphism $\neg\Box\neg A \cong \Diamond A$, so we conclude: *the act of registering a callback that reacts to a key press is the same as an event that eventually returns the key that was pressed.*

4.1 An alternate temporal logic

While the event-based perspective on programming focused on a temporal logic of the eventually type $\Diamond A$, the callback-based perspective focuses on the "always" or "global" type, written $\Box A$. Under this logic, a proposition that is true now (denoted $A^{\mathbf{n}}$) is not necessarily true in the future. The modifier $\Box A^{\mathbf{n}}$ indicates that $A^{\mathbf{n}}$ is true both now and at every point in the future. However, since the temporal logic is still linear in space, the proposition $\Box A^{\mathbf{n}}$ is only available until it gets used.

$$\frac{\Gamma; \Delta, x : A^{\mathbf{n}} \vdash e : \mathsf{Answer}}{\Gamma; \Delta \vdash \lambda x.e : \neg A^{\mathbf{n}}} \qquad \frac{\Gamma; \Delta_1 \vdash e_1 : \neg A^{\mathbf{n}} \quad \Gamma; \Delta_2 \vdash e_2 : A^{\mathbf{n}}}{\Gamma; \Delta_1, \Delta_2 \vdash e_1\, e_2 : \mathsf{Answer}}$$

$$\frac{\Gamma; \Box\,\Delta \vdash e : A}{\Gamma; \Box\,\Delta \vdash \mathtt{box}\, e : \Box\, A} \qquad \frac{\Gamma; \Delta \vdash e : \Box\, A}{\Gamma; \Delta \vdash \mathtt{unbox}\, e : A}$$

Figure 6 Typing rules for tensor logic

The behavior of $\Box\, A^{\mathbf{n}}$ can be described using two simple rules. First, if $A^{\mathbf{n}}$ is provable using only hypotheses that are always true, then $A^{\mathbf{n}}$ is always true. On the other hand, a proposition $A^{\mathbf{n}}$ that is always true is also true now.

To describe continuations, we add the negation type $\neg A^{\mathbf{n}}$ and remove all other linear arrows. Because only linear expressions need to undergo a CPS translation, the *non-linear* arrow types are unchanged. We denote the linear propositions in the resulting *adjoint tensor logic* [20] as $A^{\mathbf{n}}$, and non-linear propositions as τ.

$$\begin{aligned} \tau &::= \mathsf{Unit} \mid \mathsf{Void} \mid \tau \times \tau \mid \tau + \tau \mid \tau \to \tau \mid \lceil A \rceil^{\mathbf{n}} \\ A^{\mathbf{n}} &::= 1 \mid 0 \mid A^{\mathbf{n}} \otimes A^{\mathbf{n}} \mid A^{\mathbf{n}} \oplus A^{\mathbf{n}} \mid \neg A \mid \Box\, A \mid \lfloor \tau \rfloor \end{aligned}$$

The syntax of terms and expressions is almost identical to that of the event-based language. The negation type $\neg A^{\mathbf{n}}$ is introduced with a λ-abstraction and is eliminated with an application. The monadic `bind` and `return` operators are replaced by the comonadic `box` and `unbox` operators, as shown in Figure 6. We assume a call-by-value operational semantics for both terms and expressions.

4.2 A temporal CPS translation

The callback-based implementation of events is based on a time-sensitive CPS transformation on types written $[\![A]\!]$. The CPS translation is time-sensitive in that it keeps track of when hypotheses are available. In particular, while expressions in the event-based language are available at any point in the future, expressions in the callback-based language expire. For example, the translation of the eventually type encodes the fact that the return value can be used at any point in the future: $[\![\Diamond A]\!] = \neg\Box\neg\Box[\![A]\!]$. Similarly, a function might use its argument not right away, but at some point in the future, so the type translation of linear functions is $[\![A_1 \multimap A_2]\!] = \neg(\Box[\![A_1]\!] \otimes \neg[\![A_2]\!])$. The full details of the CPS translation on types are shown in Appendix C.

As expected, the translation on (non-linear) terms is the identity, while the translation on expressions is guided by the type translation in a mostly standard way. We describe a few cases in Appendix C and have the following soundness theorem:

▶ **Theorem 4.** *If $\Gamma \vdash t : \tau$ then $[\![\Gamma]\!] \vdash [\![t]\!] : [\![\tau]\!]$. If $\Gamma; \Delta \vdash e : A$ then $[\![\Gamma]\!]; \Box[\![\Delta]\!] \vdash [\![e]\!] : [\![A]\!]$.*

In addition, the translation respects the operational semantics of the source language.

▶ **Theorem 5.** *If $e \rightsquigarrow e'$ then $[\![e]\!] \to^* [\![e']\!]$, modulo the administrative reduxes introduced by the CPS translation [10].*

5 Concurrency Sources and the Event Loop

The CPS translation in the previous section only describes the language of pure events, and not the additional *sources of concurrency* that are the backbone of event-driven programming. So far in this paper we have considered a number of different sources: nondeterminism in the event `nondet` e, timeouts of the form `onTimeout` e, synchronous channels that create events `send` and `receive`, and the GUI operation `onKeyPress`.

The following section sketches a way to implement these primitive sources of concurrency as part of the CPS translation. The trick is to integrate the concurrent actions of these primitives with the answer type of the continuation, as described by Claessen [7] in the *poor man's concurrency monad.*

Actions and the answer type. For the pure fragment of the eventually monad (consisting only of `return` and `bind`), the answer type of the continuation is invisible. What we write as $\neg A$ is in fact $A \multimap \mathsf{Answer}$ for some fixed type Answer. Claessen observed that this makes the answer type of the continuation the perfect place to hide the presence of effects.

We call these effects *actions* following Claessen. An action can be thought of as a distinct thread of computation [18], the primitive thread operations being `Halt` and `Fork`. These threads are also stateful, operating over an event queue monad, which we write $\mathsf{EventQueue}\,A$. The `Atom` action can execute an arbitrary monadic operation over the event queue. Using Haskell-like notation for the algebraic datatype of actions, we have:

```
data Action =
| Halt : Action
| Fork : Action -o Action -o Action
| Atom : EventQueue Action -o Action
```

Since actions represent threads of computation, they are executed by a scheduler of type $\mathsf{List\,Action} \multimap \mathsf{EventQueue\,Action}$ that schedules a list of actions inside the event queue monad. For example, the following is a simple round robin scheduler:

```
eventLoop []                 = return ()
eventLoop (Halt       :: ls) = eventLoop ls
eventLoop (Fork a1 a2 :: ls) = eventLoop (ls ++ [a1,a2])
eventLoop (Atom mA    :: ls) = bind a = mA in eventLoop (ls++[a])
```

Events and actions interact via a top-level `run` operator that converts a unit-valued event to an action. Its type is $\neg[\![\Diamond 1]\!]$ or $[\![\Diamond 1]\!] \multimap \mathsf{Action}$.

$$\texttt{run} = \lambda(k : \neg\,\Box\,\neg\,\Box[\![1]\!]).k(\texttt{box}(\lambda(x : \Box[\![1]\!]).(\texttt{unbox}\,x)(\lambda().\,\texttt{Halt})))$$

As a sanity check, observe that $\texttt{run}[\![\texttt{return}\,v]\!]$ evaluates to $[\![v]\!](\lambda().\,\texttt{Halt})$.

Spawn. Using actions we can encode the event-level concurrency primitives that have been used throughout this paper. For example, `spawn`, of type $\Diamond 1 \multimap 1$, is implemented as a member of its CPS-converted type $[\![\Diamond 1 \multimap 1]\!] = \neg(\Box[\![\Diamond 1]\!] \otimes \neg[\![1]\!])$ that takes in an event (of type $\Box[\![\Diamond 1]\!]$) and a continuation (of type $\neg[\![1]\!]$) and produces a `Fork` action that runs the event and calls the continuation in parallel. The definition is found in Appendix D, and we can check that $\texttt{run}[\![\texttt{return}(\texttt{spawn}\,e)]\!]$ evaluates to $\texttt{Fork}(\texttt{box}[\![e]\!])\,\texttt{Halt}$.

Linear Channels. We conclude this section with a sketch of how to implement synchronous message-passing in the style of CML, which we used to encode `choose` in Section 3. Other sources of concurrency can be implemented in a similar way.

In order to represent channels, the event queue underlying the action type should be stateful over a linear heterogeneous store. Linear references are indexed by some non-linear identifiers, which we write $\mathsf{Id}\,A$.

$$\begin{array}{rcl@{\qquad}r}
\tau & ::= & \cdots \mid \mathsf{Id}\,A & \texttt{newId} : A^{\mathbf{n}} \multimap \mathsf{EventQueue}\lfloor \mathsf{Id}\,A^{\mathbf{n}} \rfloor \\
A^{\mathbf{n}} & ::= & \cdots \mid \mathsf{EventQueue}\,A^{\mathbf{n}} & \texttt{updateId} : \lfloor \mathsf{Id}\,A^{\mathbf{n}} \rfloor \multimap (A^{\mathbf{n}} \multimap A^{\mathbf{n}} \otimes B^{\mathbf{n}}) \multimap \mathsf{EventQueue}\,B^{\mathbf{n}}
\end{array}$$

A channel is a reference to a linear cell that consists of either: (a) a list of messages to be sent, along with the event handlers to be triggered after rendezvous, or (b) a list of event handlers waiting for messages. The types of these two possible elements are written SendElt and RecvElt, respectively.

$$\mathsf{SendElt}\ A^{\mathtt{n}} = \Box\, A^{\mathtt{n}} \otimes \Box \neg \Box [\![1]\!] \qquad \mathsf{RecvElt}\ A^{\mathtt{n}} = \Box \neg \Box\, A^{\mathtt{n}}$$

$$\mathsf{Chan}\ A^{\mathtt{n}} = \mathsf{Id}(\mathsf{List}(\mathsf{SendElt}\ A^{\mathtt{n}}) \oplus \mathsf{List}(\mathsf{RecvElt}\ A^{\mathtt{n}}))$$

When a message of type $\Box\, A^{\mathtt{n}}$ is sent over the channel, we examine the current state of the cell. If the cell contains a list of messages to be sent, the new message will be added to the end of the list. If the cell contains any callbacks waiting for messages, the callback will be applied to the incoming message and stored as an action. This behavior is governed by the function `attachSend` of type $\lfloor \mathsf{Chan}\ A \rfloor \multimap \mathsf{SendElt}\ A \multimap \mathsf{EventQueue}\,\mathsf{Action}$.

```
attachSend c (a,k0) = updateId c (fun s =>
  case s of
  | in1 ls      -> (in1 (ls++[(a,k0)]), Halt)
  | in2 []      -> (in1 [a], Halt)
  | in2 (k::ks) -> (in2 ks, Fork ((unbox k) a) ((unbox k0) [()]))
  end)
```

The event-level `send` operator has the type $\lfloor \mathsf{Chan}\ A \rfloor \multimap A \multimap \Diamond 1$, so its implementation has type $[\![\lfloor \mathsf{Chan}\ A \rfloor \multimap A \multimap \Diamond 1]\!]$. Then $[\![$`send`$]\!]$ is a continuation that takes in a channel, a message of type $\Box [\![A]\!]$, and a continuation of type $[\![\Diamond 1]\!]$, and produces an `Atom` performing the monadic computation `attachSend`.

The interpretation of `receive` is governed by a similar protocol `attachReceive` of type $\lfloor \mathsf{Chan}\ A \rfloor \multimap \mathsf{RecvElt}\ A \multimap \mathsf{EventQueue}\,\mathsf{Action}$, the details of which are given in Appendix D.

6 Discussion

In this paper, a linear and temporal logic is the guiding principle in the design of a core language for events. But the connection between logic and programming is only significant if it has a basis in existing event-driven languages, which vary in the ways they embody the event-driven paradigm. To understand these variations and the design decisions of this paper, we revisit the four features of event-driven programming discussed in the introduction: the monadic abstraction of the event, the synchronization operator, the implementation in terms of callbacks, and the primitive sources of events.

Layers of abstraction for the monadic event. The language of pure events presented in Section 2 has two parts: a type of events ($\Diamond A$) and a monadic interface for interacting with them. The monadic structure is explicit in many existing languages, including CML's `'a event` type [24], Async's `'a Deferred.t` [21], Lwt's type for light-weight threads [26], and Scala's `Future[A]` [13].[6] Other languages don't have an explicit event type, but require programmers to work directly in CPS, including Python's Twisted library [16], JavaScript's Node.js [25], or Ruby's EventMachine [6]. Still others have the event abstraction but not in a monadic style, like Racket's synchronizable events [1] or Go's goroutines [2].

[6] Although all of these abstractions are monads, their interfaces are not standard. CML has a return operator `alwaysEvt` but in lieu of `bind` it has a functorial `wrap` along with a `sync` operator of type `'a event -> 'a` that synchronously executes an event. Async and Lwt both use the standard `return` and `bind` terminology but Async has an impure `peek` operation that polls whether or not an event has completed, and Lwt has the ability to explicitly put threads to sleep and wake them up again.

Synchronization. Selective choice as described in Section 3 is less universal than the monadic bind, but has proved useful in CML, Async, Lwt, and Racket, where the default choice operator acts on lists, not pairs, and has type $\mathsf{List}(\Diamond A) \to \Diamond A$.[7] In this paper, by considering a linear `choose` operator over pairs instead of lists, we are able to draw a connection with the linear-time axiom of temporal logic, and abstract away from the type of pairs to derive a synchronization operator not only for lists, but for any container data structure.

Implementation: synchronous or asynchronous. In Section 4 we give an implementation of events based on a CPS translation that is independent of the event loop. As a consequence, `return` and `bind` are necessarily synchronous, which means that an event, once running, does not yield control by default. Only once an explicitly asynchronous operation like `choose`, `spawn`, or `receive` is called does control shift back to the scheduler.[8] In the asynchronous style, used by Async and Scala, the bind operation $\texttt{bind}\, x = e_1 \,\texttt{in}\, e_2$ registers the event e_1 in the event queue so that it executes asynchronously by default. To encode synchrony in an asynchronous system requires a special case, such as Scala's blocking constructs [13], but the other direction is trivial using `spawn` [22].

Where do events come from? Despite the similarities between event-driven languages, programming in one versus another may feel very different depending on the intended application domain. CML feels most natural for describing message-passing concurrency due to its built-in channels and spawn operator. Async describes shared-state concurrency, where its Ivar data structure is a one-shot kind of shared state. Promises in Scala are more focused on long-running computations like I/O. However, these different concurrency abstractions can be implemented in one another, such as eXene's implementations of GUIs in CML [9], or Scala's async libraries [3]. We argue that the choice of concurrency primitive is orthogonal to the design of events themselves, and that the techniques presented in this paper are applicable to a wide range of primitives.

What about FRP? The event-driven paradigm described in this paper is closely connected to functional reactive programming (FRP), which targets many of the same domains. In the FRP model, the input to a program is modeled as a time-varying value, or a stream. FRP programs can be thought of as stream transformers, or as programs that *react* to the current state of the system. Recently, FRP's connection with linear-time temporal logic [15] was discovered, which in fact prompted us to search for similar connections to event-based programming.

In typical FRP systems, the type $\Box A$ denotes the type of time-varying values, as opposed to our interpretation as an expression that is available now or at any time in the future. FRP programs model events $\blacklozenge A$ coinductively as $\nu\alpha.\ A \vee \circ\alpha$, where $\circ$ is the "next" modality. Unfortunately, this forces an implementation based on polling, which means programs continuously check whether an event has resolved yet.[9] In the event-driven interpretation, the type of events $\Diamond A$ is interpreted as a continuation $\neg\Box\neg A$ and the structure of the event loop avoids polling.

[7] In CML, `choose` aborts the events that are not chosen by means of its negative acknowledgment mechanism, but Panangaden and Reppy [22] show that this feature is encodable.

[8] CML and Lwt both use synchronous implementation strategies. Notice that the question of synchronous versus asynchronous events is orthogonal to the choice of synchronous versus asynchronous channels.

[9] Modern FRP languages work hard to avoid these time and space leaks, either by restricting the expressivity of programs [8] or by mixing ideas from event-driven programming with FRP [9].

Conclusion The Curry-Howard correspondence reveals many interesting insights into the nature of event-driven programs. Synchronization via selective choice can be thought of as the linear-time axiom from temporal logic, and can be generalized to arbitrary container data structures. The standard implementation using callbacks can be explained in a temporal way by interpreting $\Diamond A$ as $\neg\Box\neg A$, and primitive sources of concurrency are implemented using a clever choice of answer type. The result is a top-to-bottom formulation of the essence of events: computations that eventually return a value.

References

1 Events (Racket documentation). Website. URL: `docs.racket-lang.org/reference/sync.html`.
2 The Go programming language. Website. URL: `www.golang.org/`.
3 scala-async. GitHub repository. URL: `github.com/scala/async`.
4 P.N. Benton. A mixed linear and non-linear logic: Proofs, terms and models. In *Computer Science Logic*. 1995.
5 Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*. 2010.
6 Francis Cianfrocca. About EventMachine. Website. URL: `www.rubydoc.info/gems/eventmachine`.
7 Koen Claessen. A poor man's concurrency monad. *Journal of Functional Programming*, 9:313–323, 1999.
8 Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Haskell Workshop*, 2001.
9 Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for GUIs. In *PLDI*, 2013.
10 Oliver Danvy and Andrzex Filinski. Representing control: a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2:361–391, 12 1992.
11 Emden R. Gansner and John H. Reppy. A multi-threaded higher-order user interface toolkit. In *User Interface Software*, volume 1 of *Software Trends*. 1993.
12 Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.
13 Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn, and Vojin Jovanovic. Futures and Promises (Scala documentation). Website, 2013. URL: `http://docs.scala-lang.org/overviews/core/futures`.
14 Dana Harrington. Uniqueness logic. *Theoretical Computer Science*, 354(1):24 – 41, 2006. Algebraic Methods in Language Processing.
15 Alan Jeffrey. LTL types FRP: Linear-time temporal logic propositions as types, proofs as functional reactive programs. In *PLPV*, 2012.
16 Ken Kinder. Event-driven programming with Twisted and Python. *Linux Journal*, March 2005.
17 Neel Krishnaswami and Nick Benton. A semantic model for graphical user interfaces. In *ICFP*, 2011.
18 Peng Li and Steve Zdancewic. Combining events and threads for scalable network services: Implementation and evaluation of monadic, application-level concurrency primitives. In *PLDI*, 2007.
19 Conor McBride. The derivative of a regular type is its type of one-hole contexts. 2001.
20 Paul-André Melliès and Nicolas Tabareau. Resource modalities in tensor logic. *Annals of Pure and Applied Logic*, 161(5):632–653, 2010.
21 Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml*. O'Reilly Media, 2013.
22 Prakash Panangaden and John Reppy. *ML with Concurrency: Design, Analysis, Implementation, and Application*, chapter The Essence of Concurrent ML, pages 5–29. 1997.
23 Frank Pfenning and Dennis Griffith. Polarized substructural session types. In *FSSCS*. 2015.
24 John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.
25 S. Tilkov and S. Vinoski. Node.js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83, Nov 2010.
26 Jérôme Vouillon. Lwt: A cooperative thread library. In *ML Workshop*, 2008.
27 Philip Wadler. Propositions as sessions. *ICFP*, 2012.

A Event-based language

$$\tau ::= \mathsf{Unit} \mid \mathsf{Void} \mid \tau \times \tau \mid \tau + \tau \mid \tau \to \tau \mid \lceil A \rceil$$
$$A ::= 1 \mid 0 \mid A \otimes A \mid A \oplus A \mid A \multimap A \mid \Diamond A \mid \lfloor \tau \rfloor$$

$$\begin{aligned}
t ::= {} & x \mid () \mid \texttt{case}\ \ t\ \ \texttt{of}\ \ () \mid (t_1, t_2) \mid \pi_i\, t \mid \texttt{in}_i\, t \mid \texttt{case}\ \ t\ \ \texttt{of}\ (\texttt{in}_1\, x_1 \to t_1 \mid \texttt{in}_2\, x_2 \to t_2) \\
& \mid \lambda x.t \mid t_1\, t_2 \mid \texttt{suspend}\, e \\
e ::= {} & x \mid () \mid \texttt{let}\ () = e_1\ \texttt{in}\ e_2 \mid \texttt{case}\ \ e\ \ \texttt{of}\ \ () \\
& \mid (e_1, e_2) \mid \texttt{let}\ (x_1, x_2) = e_1\ \texttt{in}\ e_2 \mid \texttt{in}_i\, e \mid \texttt{case}\ \ e\ \ \texttt{of}\ (\texttt{in}_1\, x_2 \to e_1 \mid \texttt{in}_2\, x_2 \to t_2) \\
& \mid \lambda x.e \mid e_1\, e_2 \mid \texttt{return}\, e \mid \texttt{bind}\, x = e_1\ \texttt{in}\ e_2 \mid \texttt{force}\, t \mid \lfloor t \rfloor \mid \texttt{let}\ \lfloor x \rfloor = e_1\ \texttt{in}\ e_2
\end{aligned}$$

$$\dfrac{}{\Gamma, x : \tau \vdash x : \tau}\ \textsc{Var} \qquad \dfrac{}{\Gamma \vdash () : \mathsf{Unit}}\ \text{Unit-I} \qquad \dfrac{\Gamma \vdash t : \mathsf{Void}}{\Gamma \vdash \texttt{case}\ \ t\ \ \texttt{of}\ () : \sigma}\ \text{Void-E}$$

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

$$\dfrac{\Gamma \vdash t : \tau_i}{\Gamma \vdash \texttt{in}_i\, t : \tau_1 + \tau_2}\ +\text{-I} \qquad \dfrac{\Gamma \vdash t : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash t_1 : \sigma \quad \Gamma, x_2 : \tau_2 \vdash t_2 : \sigma}{\Gamma \vdash \texttt{case}\ \ t\ \ \texttt{of}\ (\texttt{in}_1\, x_1 \to t_1 \mid \texttt{in}_2\, x_2 \to t_2) : \sigma}\ +\text{-E}$$

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

$$\dfrac{}{\Gamma; x : A \vdash x : A}\ \textsc{Var} \qquad \dfrac{\Gamma; \Delta \vdash e : 0}{\Gamma; \Delta \vdash \texttt{case}\ \ e\ \ \texttt{of}\ () : B}\ 0\text{-E}$$

$$\dfrac{}{\Gamma; \cdot \vdash () : 1}\ 1\text{-I} \qquad \dfrac{\Gamma; \Delta_1 \vdash e_1 : 1 \quad \Gamma; \Delta_2 \vdash e_2 : B}{\Gamma; \Delta_1, \Delta_2 \vdash \texttt{let}\ () = e_1\ \texttt{in}\ t_2 : B}\ 1\text{-E}$$

$$\dfrac{\Gamma; \Delta_1 \vdash e_1 : A_1 \quad \Gamma; \Delta_2 \vdash e_2 : A_2}{\Gamma; \Delta_1, \Delta_2 \vdash (e_1, e_2) : A_1 \otimes A_2}\ \otimes\text{-I} \qquad \dfrac{\Gamma; \Delta_1 \vdash e_1 : A_1 \otimes A_2 \quad \Gamma; \Delta_2, x_1 : A_1, x_2 : A_2 \vdash e_2 : B}{\Gamma; \Delta_1, \Delta_2 \vdash \texttt{let}\ (x_1, x_2) = e_1\ \texttt{in}\ e_2 : B}\ \otimes\text{-E}$$

$$\dfrac{\Gamma; \Delta \vdash e : A_i}{\Gamma; \Delta \vdash \texttt{in}_i\, e : A_1 \oplus A_2}\ \oplus\text{-I} \qquad \dfrac{\Gamma; \Delta_1 \vdash e : A_1 \oplus A_2 \quad \Gamma; \Delta_2, x_1 : A_1 \vdash e_1 : B \quad \Gamma; \Delta_2, x_2 : A_2 \vdash e_2 : B}{\Gamma; \Delta_1, \Delta_2 \vdash \texttt{case}\ \ e\ \ \texttt{of}\ (\texttt{in}_1\, x_1 \to e_1 \mid \texttt{in}_2\, x_2 \to e_2) : B}\ \oplus\text{-E}$$

$$\dfrac{\Gamma; \Delta, x : A \vdash e : B}{\Gamma; \Delta \vdash \lambda x.e : A \multimap B}\ \multimap\text{-I} \qquad \dfrac{\Gamma; \Delta_1 \vdash e_1 : A \multimap B \quad \Gamma; \Delta_2 \vdash e_2 : A}{\Gamma; \Delta_1, \Delta_2 \vdash e_1\, e_2 : B}\ \multimap\text{-E}$$

$$\dfrac{\Gamma; \Delta \vdash e : A}{\Gamma; \Delta \vdash \texttt{return}\, e : \Diamond A}\ \Diamond\text{-I} \qquad \dfrac{\Gamma; \Delta_1 \vdash e : \Diamond A \quad \Gamma; \Delta, x : A \vdash e' : \Diamond B}{\Gamma; \Delta_1, \Delta \vdash \texttt{bind}\, x = e\, \texttt{in}\, e' : \Diamond B}\ \Diamond\text{-E}$$

$$\dfrac{\Gamma; \cdot \vdash e : A}{\Gamma \vdash \texttt{suspend}\, e : \lceil A \rceil}\ \lceil - \rceil\text{-I} \qquad \dfrac{\Gamma \vdash t : \lceil A \rceil}{\Gamma; \cdot \vdash \texttt{force}\, t : A}\ \lceil - \rceil\text{-E}$$

$$\dfrac{\Gamma \vdash t : \tau}{\Gamma; \cdot \vdash \lfloor t \rfloor : \lfloor \tau \rfloor}\ \lfloor - \rfloor\text{-I} \qquad \dfrac{\Gamma; \Delta_1 \vdash e_1 : \lfloor \tau \rfloor \quad \Gamma, x : \tau; \Delta_2 \vdash e_2 : B}{\Gamma; \Delta_1, \Delta_2 \vdash \texttt{let}\ \lfloor x \rfloor = e_1\ \texttt{in}\ e_2 : B}\ \lfloor - \rfloor\text{-E}$$

B Operational Semantics

The normal forms of terms and expressions, respectively, are denoted v and n.

$$\begin{aligned}
v &::= () \mid (v_1, v_2) \mid \mathtt{in}_i\, v \mid \lambda x.t \mid \mathtt{suspend}\, e \\
n &::= () \mid (e_1, e_2) \mid \mathtt{in}_i\, e \mid \lambda x.e \mid \mathtt{return}\, e \mid \lfloor v \rfloor
\end{aligned}$$

$$\begin{aligned}
\pi_i\,(v_1, v_2) &\rightsquigarrow v_i & (\lambda x.t_1)v_2 &\rightsquigarrow t_1\{v_2/x\} \\
&& \mathtt{case}\ \mathtt{in}_i\, v\ \mathtt{of}\ (\mathtt{in}_1\, x_1 \to t_1 \mid \mathtt{in}_2\, x_2 \to t_2) &\rightsquigarrow t_i\{v/x_i\}
\end{aligned}$$

$$\begin{aligned}
\mathtt{let}\ () = ()\ \mathtt{in}\ e &\rightsquigarrow e & \mathtt{let}\ (x_1, x_2) = (e_1, e_2)\ \mathtt{in}\ e &\rightsquigarrow e\{e_1/x, e_2/x\} \\
(\lambda x.e_1)e_2 &\rightsquigarrow e_1\{e_2/x\} & \mathtt{bind}\, x = \mathtt{return}\, e_1\, \mathtt{in}\, e_2 &\rightsquigarrow e_2\{e_1/x\} \\
\mathtt{force}(\mathtt{suspend}\, e) &\rightsquigarrow e & \mathtt{let}\ \lfloor x \rfloor = \lfloor v \rfloor\ \mathtt{in}\ e &\rightsquigarrow e\{v/x\} \\
&& \mathtt{case}\ \mathtt{in}_i\, e\ \mathtt{of}\ (\mathtt{in}_1\, x_1 \to e_1 \mid \mathtt{in}_2\, x_2 \to e_2) &\rightsquigarrow e_i\{e/x_i\}
\end{aligned}$$

$$\begin{aligned}
E^{\mathsf{P}} ::= {} & ([\,], t) \mid (v, [\,]) \mid \pi_i\,[\,] \\
& \mid \mathtt{in}_i\,[\,] \mid \mathtt{case}\ [\,]\ \mathtt{of}\ (\mathtt{in}_1\, x_1 \to t_1 \mid \mathtt{in}_2\, x_2 \to t_2) \\
& \mid [\,]t \mid v[\,] \mid \mathtt{force}[\,] \mid \lfloor[\,]\rfloor \\
E^{\mathsf{L}} ::= {} & \mathtt{let}\ () = [\,]\ \mathtt{in}\ e \mid \mathtt{let}\ (x_1, x_2) = [\,]\ \mathtt{in}\ e \\
& \mid \mathtt{case}\ [\,]\ \mathtt{of}\ (\mathtt{in}_1\, x_1 \to e_1 \mid \mathtt{in}_2\, x_2 \to e_2) \\
& \mid [\,]e \mid \mathtt{bind}\, x = [\,]\, \mathtt{in}\, e \mid \mathtt{let}\ \lfloor x \rfloor = [\,]\ \mathtt{in}\ e
\end{aligned}
\qquad
\frac{t \rightsquigarrow t'}{E^{\mathsf{P}}[\![t]\!] \rightsquigarrow E^{\mathsf{P}}[\![t']\!]}
\qquad
\frac{e \rightsquigarrow e'}{E^{\mathsf{L}}[\![e]\!] \rightsquigarrow E^{\mathsf{L}}[\![e']\!]}$$

C CPS translation

$$\begin{aligned}
t ::= {} & x \mid () \mid \mathtt{case}\ \ t\ \ \mathtt{of}\ () \\
& \mid (t_1, t_2) \mid \pi_i\, t \\
& \mid \mathtt{in}_i\, t \mid \mathtt{case}\ \ t\ \ \mathtt{of}\ (\mathtt{in}_1\, x_1 \to t_1 \mid \mathtt{in}_2\, x_2 \to t_2) \\
& \mid \lambda x.t \mid t_1\, t_2 \mid \mathtt{suspend}\, e \\
e ::= {} & x \mid () \mid \mathtt{let}\ () = e_1\ \mathtt{in}\ e_2 \mid \mathtt{case}\ \ e\ \ \mathtt{of}\ () \\
& \mid (e_1, e_2) \mid \mathtt{let}\ (x_1, x_2) = e_1\ \mathtt{in}\ e_2 \\
& \mid \mathtt{in}_i\, e \mid \mathtt{case}\ \ e\ \ \mathtt{of}\ (\mathtt{in}_1\, x_2 \to e_1 \mid \mathtt{in}_2\, x_2 \to t_2) \\
& \mid \lambda x.e \mid e_1\, e_2 \\
& \mid \mathtt{box}\, e \mid \mathtt{unbox}\, e \\
& \mid \mathtt{force}\, t \mid \lfloor t \rfloor \mid \mathtt{let}\ \lfloor x \rfloor = t\ \mathtt{in}\ e
\end{aligned}$$

$$\begin{aligned}
[\![\mathsf{Unit}]\!] &= \mathsf{Unit} \\
[\![\mathsf{Void}]\!] &= \mathsf{Void} \\
[\![\tau_1 \times \tau_2]\!] &= [\![\tau_1]\!] \times [\![\tau_2]\!] \\
[\![\tau_1 + \tau_2]\!] &= [\![\tau_1]\!] + [\![\tau_2]\!] \\
[\![\tau_1 \to \tau_2]\!] &= [\![\tau_1]\!] \to [\![\tau_2]\!] \\
[\![\lceil A \rceil]\!] &= \lceil [\![A]\!] \rceil
\end{aligned}
\qquad
\begin{aligned}
[\![1]\!] &= \neg\neg 1 \\
[\![0]\!] &= \neg\neg 0 \\
[\![A_1 \otimes A_2]\!] &= \neg\neg([\![A_1]\!] \otimes [\![A_2]\!]) \\
[\![A_1 \oplus A_2]\!] &= \neg\neg([\![A_1]\!] \oplus [\![A_2]\!]) \\
[\![A_1 \multimap A_2]\!] &= \neg(\Box[\![A_1]\!] \otimes \neg[\![A_2]\!]) \\
[\![\Diamond A]\!] &= \neg\Box\neg\Box[\![A]\!] \\
[\![\lfloor \tau \rfloor]\!] &= \neg\neg\lfloor [\![\tau]\!] \rfloor
\end{aligned}$$

$$\begin{aligned}
[\![x]\!] &= \mathtt{unbox}\, x \\
[\![\lambda x.e]\!] &= \lambda(x, k).k[\![e]\!] \\
[\![e_1\, e_2]\!] &= \lambda k.[\![e_1]\!](\mathtt{box}[\![e_2]\!], \lambda z.zk)
\end{aligned}
\qquad
\begin{aligned}
[\![\mathtt{return}\ e]\!] &= \lambda k.(\mathtt{unbox}\, k)(\mathtt{box}[\![e]\!]) \\
[\![\mathtt{bind}\, x = e_1\ \mathtt{in}\ e_2]\!] &= \lambda k.[\![e_1]\!](\mathtt{box}(\lambda x.[\![e_2]\!]k))
\end{aligned}$$

Sketch of Theorem 5. Following Danvy and Filinski [10], we could easily consider a one-pass CPS translation where the administrative reduxes—those introduced only by the CPS translation—are treated as meta-operations on terms. These meta-operations are written with an overline, and follow the pattern

$$(\lambda x.e)\ \overline{@}\ v = e\{v/x\} \qquad\qquad \overline{\texttt{unbox}}(\texttt{box}\ e) = e.$$

The administrative reduces are introduced uniformly in the CPS translation of terms, with the exception of the λ abstraction rule, which requires an extra η-expansion.

$$\begin{aligned}
[x] &= \overline{\texttt{unbox}}\,x\\
[\lambda x.e] &= \lambda(y,z).(\lambda x.z\ \overline{@}\ [e])y\\
[e_1\,e_2] &= \lambda k.[e_1]\ \overline{@}\ (\texttt{box}[e_2], \lambda z.z\ \overline{@}\ k)\\
[\texttt{return}\ e] &= \lambda k.(\overline{\texttt{unbox}}\,k)(\texttt{box}[e])\\
[\texttt{bind}\,x = e_1\ \texttt{in}\ e_2] &= \lambda k.[e_1]\ \overline{@}\ (\texttt{box}(\lambda x.[e_2]\ \overline{@}\ k))
\end{aligned}$$

With this one-pass CPS translation we can prove the theorem directly. ◀

D CPS Implementation of Event Primitives

$$\begin{aligned}
\texttt{spawn} &: \Diamond 1 \multimap 1\\
[\![\texttt{spawn}]\!] &: [\![\Diamond 1 \multimap 1]\!] = \neg(\Box[\![\Diamond 1]\!] \otimes \neg[\![1]\!])\\
&= \lambda(x,k).\,\texttt{Fork}(\texttt{run}(\texttt{unbox}\,x))(k[\![(\,)]\!])
\end{aligned}$$

$$\begin{aligned}
\texttt{new} &: 1 \multimap \lfloor \mathsf{Chan}\ A \rfloor\\
[\![\texttt{new}]\!] &: \neg(\Box[\![1]\!] \otimes \neg[\![\lfloor \mathsf{Chan}\ A \rfloor]\!])\\
&= \lambda(u : \Box\neg\neg 1, k : \neg\lfloor \mathsf{Chan}[\![A]\!]\rfloor).(\texttt{unbox}\ u)(\lambda().\texttt{Atom (bind i = newId (inl []) in } k\texttt{i}))
\end{aligned}$$

$$\begin{aligned}
\texttt{send} &: \lfloor \mathsf{Chan}\ A \rfloor \multimap A \multimap \Diamond 1\\
[\![\texttt{send}]\!] &: \neg(\Box[\![\lfloor \mathsf{Chan}\ A \rfloor]\!] \otimes \Box[\![A]\!] \otimes \neg[\![\Diamond 1]\!])\\
&= \lambda(c : \Box\neg\neg\lfloor \mathsf{Chan}[\![A]\!]\rfloor, a : \texttt{box}[A], k : [\![\Diamond 1]\!]).\\
&\qquad (\texttt{unbox}\ c)(\lambda i.k(\lambda(k' : \Box\neg\Box[\![1]\!]).\texttt{Atom (attachSend } i\ k')))
\end{aligned}$$

$$\begin{aligned}
\texttt{receive} &: \lfloor \mathsf{Chan}\ A \rfloor \multimap \Diamond A\\
[\![\texttt{receive}]\!] &: \neg(\Box[\![\lfloor \mathsf{Chan}\ A \rfloor]\!] \otimes \neg[\![\Diamond A]\!])\\
&= \lambda(c : \Box\neg\neg\lfloor \mathsf{Chan}[\![A]\!]\rfloor, k : [\![\Diamond A]\!]).\\
&\qquad (\texttt{unbox}\ c)(\lambda i.k(\lambda(k' : \Box\neg\Box[\![A]\!]).\texttt{Atom (attachReceive } i\ k')))
\end{aligned}$$