1 Problem and motivation

Event-driven programming has become an increasingly popular programming model for lightweight concurrency. It has been used successfully in applications ranging from message-passing in Concurrent ML [10] to asynchronous I/O operations in Node.js [11] to the dynamic behavior of GUIs in eXene [3], among many others. The unifying idea is that an event is a computation that executes independently from other events and can eventually return a value.

Unfortunately, event-driven programming can be difficult because it is based on a combination of features that do not play well together: callbacks and state. Callbacks are higher-order first-class functions. When combined with stateful abstractions like Ivars in OCaml’s Async library [7] or Promises in Scala [5], the resulting programs can be convoluted and error-prone.

Many event-driven languages have tried to alleviate this difficulty by introducing various abstractions for serializing and synchronizing events. Unfortunately this has been done in a somewhat ad hoc way, resulting in many competing abstractions formulated in slightly different ways. For example, while events in Go [1] and Lwt [12] are thought of as lightweight threads, in Scala they are thought of as write-once shared state in the form of Futures [5], and in Concurrent ML they are thought of as concurrent processes [10].

This work aims to understand the space of event-driven programming by identifying the structures common to it with an intuitive perspective from temporal logic. Under this interpretation, an event is a computation that can eventually return a value, and so we identify its type with the “eventually” proposition $\Diamond A$ of temporal logic.

From this observation we are able to better understand several parts of the event-driven paradigm, including the selective choice synchronization operator as an axiom of linear-time temporal logic, and the callback-based implementation of events as the logical isomorphism $\Diamond A = \neg \Box \neg A$.

This kind of correspondence between a type system and a logic is known as the Curry-Howard isomorphism. Our work describes a new instance of the Curry-Howard isomorphism that relates continuation-based event-driven programming and linear-time temporal logic.

2 Background and related work

2.1 Event-driven GUIs

We will use graphical user interfaces (GUIs) as a running example of event-driven programming in this paper. In a GUI, events capture user behaviors such as key presses and button clicks. In practice, programs react to events by registering callbacks, or event handlers, that are triggered by an underlying event loop once the event occurs.

The structure of the graphical display, on the other hand, is based on a tree of widgets, which include buttons, text boxes, and frames. When users interact with these widgets on the screen, their actions are represented as events associated with those widgets. The programmer specifies how to react when one of these events occurs by registering callbacks with particular widgets. For example, the command

```javascript
b.onClick(fun () -> e)
```

will register the callback `fun () -> e` to the button `b`. When the button gets clicked, the event loop will trigger that callback with input `()`. The callback code can then register handlers to other events, creating a complex web of dependencies illustrated in Figure 1.

One of the fundamental abstractions used by existing event-driven languages is the monadic struc-
ture of callbacks. Instead of registering a callback in the event loop, instead think of `b.onClick` as an event that will return a unit value once the button has been clicked. The monadic bind operator `bind x = b.onClick in e` can be thought of as waiting for the click to occur, and then continuing as `e`. This sequential interpretation of events leads to a much more natural control flow for event-driven programs.

Another abstraction used in existing implementations is the selective choice approach to synchronization. In this approach, `choose (e1,e2)` is itself an event that waits for the first of `e1` or `e2` to return a value. For example, `choose(b1.onClick, b2.onClick)` is an event that will trigger when a user clicks on either the button `b1` or the button `b2`.

2.2 Linear-time Temporal Logic

Temporal logic is a kind of logic for reasoning about time. Although there are many variations on temporal logic in the literature, for our purposes we only need a simple propositional fragment, illustrated in Figure 2. In this fragment, a proposition that is true is only considered true at the current point in time. The proposition `□A`, pronounced “always A,” is true both now and at every point in the future. The proposition `⋄A`, pronounced “eventually A,” is not necessarily true now, but will be true at some point in the future. Finally, the negation operator `¬A` says only that `A` is not true now; it makes no claims on the future of `A`.

Judgments in this logic have the form `∆ ⊢ A` and mean that under the hypotheses given by `∆`, the proposition `A` is true now. The “always” operator can be characterized by how it constructs and uses propositions of the form `□A`. If a proposition `A` is true now, it is not necessarily always true. However, if it can be proved only using hypotheses that are always true, then `A` itself is always true. On the other hand, if a proposition `A` is true always, then it is also true now.

\[
\begin{align*}
□\Delta ⊢ A & \quad \Delta ⊢ □A \\
□\Delta ⊢ □A & \quad \Delta ⊢ A
\end{align*}
\]

For the “eventually” operator, consider that if a
proposition is true now, it is also true later. On the other hand, if $A$ proves that $B$ is true later, and $A$ is true later, then $B$ is true later without $A$. Represented as inference rules, we have the following:

$$
\Delta \vdash A \quad \Delta, \Diamond \Delta' \vdash \Diamond B \\
\Delta, \Box \Delta' \vdash \Diamond B \\
\Delta \vdash \Diamond A
$$

Note that if any of the hypotheses used to prove $\Diamond B$ were only true “now,” they wouldn’t necessarily be valid at the same time $A$ is ready.

The negation operator $\neg A$ is true if and only if $A$ leads to a contradiction. Such a contradiction can be reached only if both $A$ and $\neg A$ can be proven.

$$
\Delta \vdash \bot \\
\Delta, A \vdash \bot
$$

The propositional logic we present here, despite having a negation operator, is an intuitionistic logic where negation is a limited form of implication. In fact, negation can be thought of as the type of a callback—a function that does not return a value. Fact, negation can be thought of as the type of a function that does not return a value, where negation is a limited form of implication. In particular, an event $\Diamond A$ is interpreted as a continuation of continuations: $\Diamond \bot \rightarrow \bot$.

We describe these three contributions more thoroughly in the next section.

4 Results and Contributions

4.1 Events as “eventually”

The type system for events is a simple monadic type system based on the modal logics of Pfenning and Davies [9] and the temporal logic in Section 2.2. Using the monadic syntax, we write $\mathsf{return} e$ for the trivial event that immediately returns the value $e$. We write $\mathsf{bind} x = e_1 \mathsf{in} e_2$ to refer to the event that waits for $e_1$ to return a value, and then substitutes that value for $x$ in $e_2$. Operationally we can think of $\mathsf{bind}$ as waiting for an event, this being the main way of interacting with events in our language

$$
\Gamma \vdash e : A \\
\Gamma \vdash e_1 : \Diamond A \\
\Gamma, x : A \vdash e_2 : \Diamond B
$$

$$
\Gamma \vdash \mathsf{return} e : \Diamond A \\
\Gamma \vdash \mathsf{bind} x = e_1 \mathsf{in} e_2 : \Diamond B
$$

Logically, each typed expression $\Gamma \vdash e : A$ corresponds to a proof of $\Box \Gamma \vdash A$ in the temporal logic.

As an example of how to use these operators, consider a simple user interface. Assume that the library provides a type $\mathsf{Button}$ of buttons, a way to update the text displayed on a button, and an event that records the user’s click.

$$
\mathsf{newButton} : \mathsf{String} \rightarrow \mathsf{Button} \\
\mathsf{setText} : \mathsf{Button} \rightarrow \mathsf{String} \rightarrow \mathsf{Unit} \\
\mathsf{onClick} : \mathsf{Button} \rightarrow \Diamond \mathsf{Unit}
$$

Using this interface, we can define a counter where the text displayed on a button reflects the number of times it has been clicked.

$$
\mathsf{counter} (b : \mathsf{Button}) (n : \mathsf{Nat}) = \\
\mathsf{setText} b (\mathsf{toString} n); \\
\mathsf{bind} () = \mathsf{onClick} b \mathsf{in} \\
\mathsf{counter} b (n+1)
$$

4.2 Synchronization as linear-time

An important consequence of the event abstraction is the ability to execute multiple events in parallel
and record which one occurred first. This kind of synchronization, known as selective choice [10], can be thought of as an instance of the linear-time axiom from temporal logic.

\[
\text{choose} : \diamond A \times \diamond B \rightarrow \diamond(A \times \diamond B + \diamond A \times B)
\]

Using this synchronization operator, we can develop more complex GUIs. For example, consider a GUI of multiple counters, each recording the number of times it has been clicked. Since choose executes two events in parallel, we can reuse the former counter example.

\[
\text{counter2} = \\
\text{let } b_1 = \text{newButton (toString 0)} \text{ in} \\
\text{let } b_2 = \text{newButton (toString 0)} \text{ in} \\
\text{choose (counter b1 0, counter b2 0)}
\]

For a more complex example, consider an inverse counter—where the label on \(b_1\) represents the number of times \(b_2\) has been clicked, and vice versa. This requires a more subtle interaction between the two clicks on the buttons.

\[
\text{counter2'} = \\
\text{let } b_1 = \text{newButton (toString 0)} \text{ in} \\
\text{let } b_2 = \text{newButton (toString 0)} \text{ in} \\
\text{let loop } e_1 e_2 n_1 n_2 = \\
\text{bind } z = \text{choose (e1,e2) in} \\
\text{case } z \text{ of} \\
\begin{array}{l}
| \text{in1}(()),e2) -> \\
\quad \text{let } e_1 = \text{onClick b1 in} \\
\quad \text{setText b2 (n1+1);} \\
\quad \text{loop } e_1 e_2 (n1+1) n2 \\
\mid \text{in2} (e1,()) -> \\
\quad \text{let } e_2 = \text{onClick b2 in} \\
\quad \text{setText b1 (n2+1)} \\
\quad \text{loop } e_1 e_2 n_1 (n2+1)
\end{array} \\
\text{end in} \\
\text{loop (onClick b1) (onClick b2) 0 0}
\]

\subsection{4.3 A temporal CPS translation}

Our final contribution is to explain how our temporal interpretation of events relates to the implementation in terms of callbacks. In our ongoing GUI example, the standard treatment of the onClick event is to register a continuation in the heap. That is, \(b.\text{onClick (fun k -> e)}\) is an effectful computation that registers the continuation \((\text{fun k -> e})\) in the underlying event loop.

What is the type of a continuation? On the one hand, it takes a value as input (in this case with type Unit) but doesn’t return a meaningful value. We write the return type Answer, and so the continuation is a function Unit \(\rightarrow\) Answer. However, since the result type is uniform over all continuations, we could instead write the type of continuations as \(\neg\)Unit, the function that accepts a Unit and doesn’t return a value.

On the other hand, the type of a continuation should also have a temporal component. A callback is not intended to be used right away, but it will be used at some point in the future. Therefore the type of the continuation being registered in the event loop should have the “always” prefix \(\Box\), to indicate that it will be available at any point in the future. A continuation thus has the type \(\Box\)(Unit \(\rightarrow\) Answer) or, equivalently, \(\Box\neg\)Unit. This relationship is described in Figure 3.

Since \(b.\text{onClick}\) accepts one of these continuations, its type is \(\neg\Box\neg\)Unit, which is an intuitive analogue of \(\Box\text{Unit}\), the type of onClick b interpreted as an event. This tells us that the logical interpretation of events has a sound foundation in the continuation-based implementation.

The translation is based on the temporal CPS conversion inspired by the fact that \(\diamond A = \Box\neg\neg A\). Although the details, shown in Figure 4, understandably introduce some complexity, they are not surprising given other such CPS translations.

We extend the translation to expressions, written \([e]\), such that if \(\Gamma \vdash e : A\) then \(\Box ([\Gamma] \vdash [e] : [A])\). We add the extra \(\Box\) constructor around the hypotheses in

\[
\begin{align*}
\Box\neg\text{Unit} & \quad \Box\Box\text{Unit} \\
\Box\neg\text{Unit} & \quad \Box\Box\text{Unit} \\
\Box\neg\text{Unit} & \quad \Box\Box\text{Unit} \\
\Box\neg\text{Unit} & \quad \Box\Box\text{Unit}
\end{align*}
\]

Figure 3: Type of a temporal callback

\[
\begin{align*}
[\diamond A] & = \Box\neg\neg [A] \\
[\text{Unit}] & = \neg\Box\text{Unit} \\
[\Box A \times \Box B] & = \Box\Box([A] \times [B]) \\
[\Box A + \Box B] & = \Box\Box([A] + [B]) \\
[\Box A \rightarrow \Box B] & = \Box(\Box A \times \Box\neg B)
\end{align*}
\]

Figure 4: CPS translation on types
Because of the assumption, in the event language, that variables don’t expire—their types are implicitly precluded by an “always” operator.

4.4 But wait, there’s more!

This document describes only a rough overview of the Curry-Howard correspondence that relates events and temporal logic. Some of the details of the formalization have been left out for brevity and clarity. In order to turn our intuition about events and temporal logic into a usable programming language however, there are many additional factors to consider. We briefly review some of these, and for further details see [8].

Linear logic for effects. In order to define an operational semantics for the event-driven language, we need to account for a myriad of effects. Events themselves are effectful computations, buttons in GUIs are stateful, and so on. Programming languages have a number of techniques for describing type systems for effectful programs, but from the perspective of the Curry-Howard correspondence we rely on Girard’s linear logic [4].

Extend choose using derivatives. The choose operator described in this work is limited to synchronizing pairs of events, and is not strong enough to derive synchronization operators for lists or even tuples of events. By drawing a new connection to McBride’s derivatives of regular types [2001] we are able to generalize the selective choice operator not only to finite data structures containing events, but also to arbitrary containers of events including lists and trees.

The event loop implementation. In this document we use GUI programming as an example of the kind of primitive events encountered in existing languages. To implement these primitives requires interaction with the event loop. To encode the event loop in the context of our temporal CPS translation, we hide the effectful operations within the answer type of continuations ¬A in a way inspired by Claessen’s poor man’s concurrency monad [2].

References


