ICFP: G: Curry-Howard for Callbacks

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

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 `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-*

*With Neelakantan R. Krishnaswami and Steve Zdancewic. Supported in part by the by the NSF Graduate Research Fellowship under Grant No. DGE-1321851.

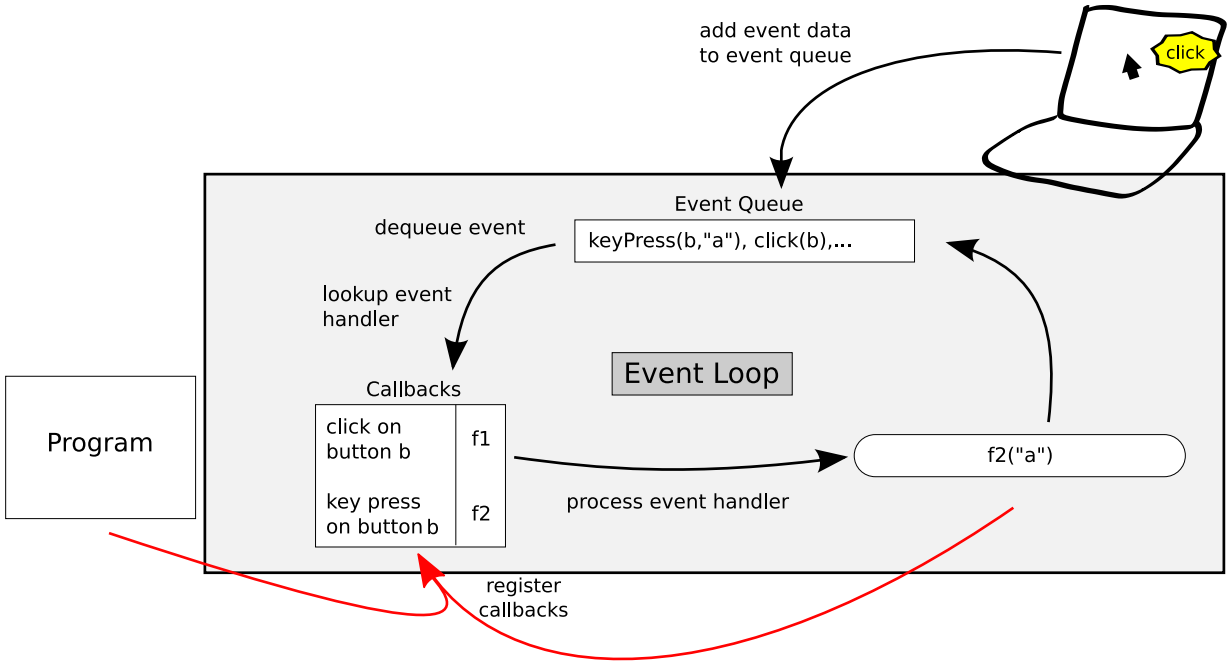

Figure 1: The event loop paradigm

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 $\Box A$, pronounced "always A," is true both now and at every point in the future. The proposition $\diamond A$, pronounced "eventually A," is

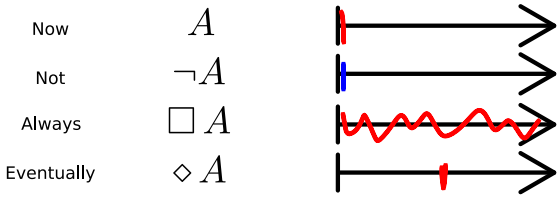

Figure 2: Propositions of temporal logic

not necessarily true now, but will be true at some point in the future. Finally, the negation operator $\neg A$ says only that A is not true now; it makes no claims on the future of A.

Judgments in this logic have the form $\Delta \vdash A$ and mean that under the hypotheses given by Δ, the proposition A is true now. The "always" operator can be characterized by how it constructs and uses propositions of the form $\Box A$. 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.

$$\frac{\Box\Delta \vdash A}{\Box\Delta \vdash \Box A} \qquad \frac{\Delta \vdash \Box A}{\Delta \vdash A}$$

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:

$$\frac{\Delta \vdash A}{\Delta \vdash \diamond A} \qquad \frac{\Delta \vdash \diamond A \qquad \Box\Delta', A \vdash \diamond B}{\Delta, \Box\Delta' \vdash \diamond B}$$

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 assuming A leads to a contradiction. Such a contradiction can be reached only if both A and $\neg A$ can be proven.

$$\frac{\Delta, A \vdash \bot}{\Delta \vdash \neg A} \qquad \frac{\Delta \vdash A \qquad \Delta \vdash \neg A}{\Delta \vdash \bot}$$

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.

Finally, linear-time temporal logic has an additional axiom that makes it "linear-time" as opposed to "branching-time," which says that all futures lie on the same timeline. This axiom has the following form: if A and B will both eventually be true, then either A will come before B or B will come before A.

$$\diamond A \wedge \diamond B \Rightarrow \diamond\left((A \wedge \diamond B) \vee (\diamond A \wedge B)\right)$$

3 Uniqueness of the approach

The strength of the Curry-Howard correspondence is that it combines things we know about programming languages and things we know about logic and uses each to gain a deeper understanding of the other. In this case, we make the observation that linear-time temporal logic can be seen as a type system for event-driven programming. We connect three aspects of event-driven programming to properties of the logic.

1. An event is a computation that eventually returns a value. As a result, we develop a type system where the type of an event is the "eventually" monad $\diamond A$ from temporal logic.

2. Under this interpretation, the linear-time temporal axiom corresponds to the kind of synchronization called selective choice. The choice of two events executes them both in parallel and returns a value recording which occurred first.

3. The implementation of the event-driven abstraction consists of a continuation-passing style (CPS) transformation that can too be explained by temporal logic. In particular, an event $\diamond A$ is interpreted as a continuation of continuations: $\neg\Box\neg A$.

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 `return e` for the trivial event that immediately returns the value `e`. We write `bind x = e1 in e2` to refer to the event that waits for `e1` to return a value, and then substitutes that value for `x` in `e2`. Operationally we can think of `bind` as waiting for an event, this being the main way of interacting with events in our language

$$\frac{\Gamma \vdash \texttt{e} : A}{\Gamma \vdash \texttt{return e} : \diamond A} \qquad \frac{\Gamma \vdash \texttt{e1} : \diamond A \quad \Gamma, x : A \vdash \texttt{e2} : \diamond B}{\Gamma \vdash \texttt{bind x = e1 in e2} : \diamond B}$$

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 `Button` of buttons, a way to update the text displayed on a button, and an event that records the user's click.

$$\begin{aligned} \texttt{newButton} &: \texttt{String} \to \texttt{Button} \\ \texttt{setText} &: \texttt{Button} \to \texttt{String} \to \texttt{Unit} \\ \texttt{onClick} &: \texttt{Button} \to \diamond\texttt{Unit} \end{aligned}$$

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

```
counter (b : Button) (n : Nat) =
  setText b (toString n);
  bind () = onClick b in
  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.

$$\texttt{choose} : \diamond A \times \diamond B \to \diamond(A \times \diamond B + \diamond A \times B)$$

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.

```
counter2 =
  let b1 = newButton (toString 0) in
  let b2 = newButton (toString 0) in
  choose (counter b1 0, counter b2 0)
```

For a more complex example, consider an inverse counter—where the label on `b1` represents the number of times `b2` has been clicked, and vice versa. This requires a more subtle interaction between the two clicks on the buttons.

```
counter2' =
  let b1 = newButton (toString 0) in
  let b2 = newButton (toString 0) in
  let loop e1 e2 n1 n2 =
    bind z = choose (e1,e2) in
    case z of
    | in1((),e2) ->
        let e1 = onClick b1 in
        setText b2 (n1+1);
        loop e1 e2 (n1+1) n2
    | in2(e1,()) ->
        let e2 = onClick b2 in
        setText b1 (n2+1)
        loop e1 e2 n1 (n2+1)
    end in
  loop (onClick b1) (onClick b2) 0 0
```

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.onClick(fun k -> e)` is an effectful computation that registers the continuation `(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 → Answer. However, since the result type is uniform over all continuations, we could instead write the type of continuations as ¬Unit, the function that accepts a Unit and doesn't return a value.

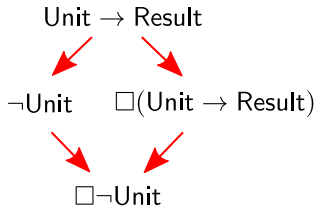

Figure 3: Type of a temporal callback

$$\begin{aligned}
[\![\diamond A]\!] &= \neg\Box\neg\Box[\![A]\!] \\
[\![\mathsf{Unit}]\!] &= \neg\neg\mathsf{Unit} \\
[\![A \times B]\!] &= \neg\neg([\![A]\!] \times [\![B]\!]) \\
[\![A + B]\!] &= \neg\neg([\![A]\!] + [\![B]\!]) \\
[\![A \to B]\!] &= \neg(\Box[\![A]\!] \times \neg[\![B]\!])
\end{aligned}$$

Figure 4: CPS translation on types

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 $\Box A$, to indicate that will be available at any point in the future. A continuation thus has the type $\Box(\mathsf{Unit} \to \mathsf{Answer})$ or, equivalently, $\Box\neg\mathsf{Unit}$. This relationship is described in Figure 3.

Since `b.onClick` accepts one of these continuations, its type is $\neg\Box\neg\mathsf{Unit}$, which is an intuitive analogue of $\diamond\mathsf{Unit}$, 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 $\diamond A = \neg\Box\neg A$. 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

Γ 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 $\neg A$ in a way inspired by Claessen's *poor man's concurrency monad* [2].

References

[1] The Go programming language. Website. URL `www.golang.org/`.

[2] Koen Claessen. A poor man's concurrency monad. *Journal of Functional Programming*, 9:313–323, 1999.

[3] Emden R. Gansner and John H. Reppy. A multi-threaded higher-order user interface toolkit. In *User Interface Software, Bass and Dewan (Eds.)*, volume 1 of *Software Trends*, pages 61–80. John Wiley & Sons, 1993.

[4] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987. doi: 10.1016/0304-3975(87)90045-4.

[5] Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn, and Vojin Jovanovic. Futures and promises. Scala Documentation, 2013. URL `http://docs.scala-lang.org/overviews/core/futures`.

[6] Conor McBride. The derivative of a regular type is its type of one-hole contexts. 2001.

[7] Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml*. O'Reilly Media, 2013.

[8] Jennifer Paykin, Neelakantan R Krishnaswami, and Steve Zdancewic. The essence of event-driven programming. 2016.

[9] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical structures in computer science*, 11(04):511–540, 8 2001. doi: 10.1017/S0960129501003322.

[10] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999. doi: 10.1017/CBO9780511574962. Cambridge Books Online.

[11] S. Tilkov and S. Vinoski. Node.js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83, Nov 2010. doi: 10.1109/MIC.2010.145.

[12] Jérôme Vouillon. Lwt: A cooperative thread library. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ML '08, pages 3–12, New York, NY, USA, 2008. ACM. doi: 10.1145/1411304.1411307.