Curry-Howard for GUIs

Or, User Interfaces via Linear Temporal, Classical Linear Logic

Neelakantan R. Krishnaswami
University of Birmingham
<N.Krishnaswami@cs.bham.ac.uk>

Jennifer Paykin Steve Zdanciewic
University of Pennsylvania
<jpaykin@seas.upenn.edu>
<stevez@cis.upenn.edu>

1. Problem

Graphical user interface toolkits are one of the most commonly encountered higher-order interfaces. Even in languages (such as Java and C++) where programmers tend to avoid higher-order abstractions, the GUI libraries invariably expose a higher-order API.

The ruling abstraction for these libraries is the *event*. The user interface is thought of as something that changes over time, and programmers may register their interest in particular events by passing the UI a callback function to invoke whenever the event happens. Programming in an event-based style (even using standard patterns such as model-view-controller [11]) is notoriously difficult: programmers must write imperative, higher-order programs in continuation-passing style!

Within the functional programming community, a popular proposal for taming this complexity is *functional reactive programming* [6]. This model eliminates imperative state from the semantic model of interactive programs by treating time-varying values as first-class datatype, and eliminates concurrency by taking a synchronous view of time. This is known to be an expressive and principled approach — a recent line of work [9, 10, 12] shows that functional reactive programming can be thought of as a Curry-Howard style lambda calculus for *linear temporal logic*.

However, the conceptual basis of modern GUI toolkits and FRP are seemingly incompatible. Traditional GUI toolkits present an asynchronous model of time to the programmer. Conceptually, each widget in a user interface has its own logical sequence of events, and there is no intrinsic relative ordering between the events occuring in two different widgets: if a programmer wants to order the events occuring on two different widgets, they must write synchronization code themselves. In contrast, the semantics of FRP is based on a synchronous model of time. As a result, within the semantics it is always reasonable to compare any two events for when they occur, no matter where they come from — time is global. As a result of this mismatch, there have been very few serious attempts to integrate the FRP and the event-based model of GUIs ([8] being a notable exception), with most attempts to build FRP UI libraries taking a clean-slate approach.

However, the event-based model was not adopted idly! Used carefully, it offers some very significant advantages. First, the event-based style means that when a widget is not reacting to an event, it does no computation at all, which may offer significant performance benefits over the basic FRP style. Second, programmers do informal reasoning about GUIs, which treats each widget as a little process communicating with other widgets via message passing[1]. In particular, programmers rely heavily on the fact that widgets which do not communicate with each other at all do not affect each other's behavior. This "frame rule" style principle greatly simplifies building dynamic GUIs, where the widget hierarchy changes as the program executes.

2. Clues

To recap, FRP has a very solid logical foundation, and event-based programming has a very solid implementation strategy — but each style is weak where the other is strong. Can we combine these two? In approaching this questions, we have a few clues to work with.

From Widgets to Processes First, we will take seriously the fact that programmers talk about GUI widgets as if they were processes. This naturally suggests that a surface language for GUIs could be based on the process calculus. Furthermore, the fact that programmers use an informal notion of separation between different GUI components suggests that some kind of substructural discipline, like separation logic [15], might be appropriate.

Happily, there is a long line of research [1, 4, 16] showing how to use classical linear logic [7] to give types to process calculi. Thanks to the use of linearity as a type discipline, these process calculi can naturally encode ideas such as session types, and have very strong global reasoning properties, such as confluence and deadlock-freedom, which puts concurrency under solid control. This leads to our first idea:

Idea #1: Use a linearly-typed process calculus to capture the concurrent, interactive intuitions of programmers.

Moreover, we would like to smoothly embed a process calculus into a more conventional language. For intuitionistic linear logic, Benton [3] showed how to integrate linear and nonlinear calculi on an equal basis, and Paykin and Zdancewic [14] have recently extended this result to classical linear logic with their LPC (Linear/Producer/Consumer) calculus. So this is possible without giving up the logical character of the system.

Callbacks, Continuations, and Temporal Logic Our first clue comes from looking at the type of event handlers in GUI toolkits. In Haskell-style notation, an event handler method has the type:

```
onEvent    :    (Event → IO()) → IO()
```

[1] Messages can be *implemented* either through explicit event signalling or modifying shared state — but programmers *talking* with each other tend using the vocabulary of message passing.

Operationally, what will happen is that the `onEvent` function takes an event handler `k`, and then will store `k` in a queue until the event happens, at which point it will invoke `k` with the appropriate event data. So a more temporally-accurate type for `onEvent` is

$$\texttt{onEvent} \quad : \quad \Box(\texttt{Event} \to \texttt{IO}()) \to \texttt{IO}()$$

where $\Box A$ is the "always" modality of temporal logic, which says that the value `k` should be usable at all times in the future. So the argument type $\Box(\texttt{Event} \to \texttt{IO}())$ means that the callback should be invokable any time in the future (i.e., whenever the event actually occurs).

Now, if we read `IO()` as the answer type of a continuation, with $A \to \texttt{IO}()$ being read as the negation type $\neg A$, then we can read the type of `onEvent` as:

$$\texttt{onEvent} \quad : \quad \neg\Box(\neg\texttt{Event})$$

But this is precisely the encoding of the "eventually" operator in classical modal logic! This leads us to our next idea:

Idea #2: Look at continuation-passing style interpretations of classical temporal logic for hints about implementation.

3. Approach

Our clues have suggested we should *express* programs process-theoretically, and we want to *implement* them using continuations. To connect these two ideas, we will make use of a recently introduced substructural logic, called *tensorial logic* [13]. Tensorial logic can be thought of as a subset of intuitionistic linear logic, where the linear implication $A \multimap B$ is eliminated, and replaced with a negation $\neg A$. For us, the key feature of tensorial logic is that any proof of full classical linear logic can be translated into tensorial logic, by translating the connectives and derivation rules appropriately.

As a result, if we can give a sensible operational interpretation for a (temporal) tensorial logic, then we automatically have an operational interpretation for full classical linear logic. We give the grammar of types of tensorial logic below:

$$A \quad ::= \quad I \mid A \otimes B \mid \neg A \mid \Box A \mid \texttt{Widget} \mid \texttt{Event}$$

Our operational intuition is a stylized version of how event loops in the HTML DOM or other GUI toolkits work. Essentially, we we have a memory full of (mutable) widgets, each of which has an event queue in which it stores callbacks. On each tick of the event loop, some widget is chosen and its events are fired.

The I type is the unit type, and $A \otimes B$ is the type of pairs, where A and B access disjoint regions of memory. Operationally, the negation type $\neg A$ is the type of callbacks functions which take an A as an argument, and when executed perform some imperative action which respect the event loop's internal invariants. The type $\Box A$ is the type of A-values which are good both on the current tick, and on all future ticks. The type `Event` is the type of event data, and the `Widget` type denotes GUI widgets.

Formalizing these natural operational ideas is non-trivial, primarily because of the mathematical difficulties involved in handling callbacks. They are higher-order functions which are imperatively added and removed from the heap, which themselves perform actions on the heap. However, we are currently in the process of using some ideas from separation logic [5] and step-indexed models of state [2] to develop a logical relations model of callbacks as an implementation of tensorial logic.

4. Conclusion

Since this work is in-progress, and not complete, it should be regarded as conjectural. However, we think it is an extremely *interesting* conjecture, and we hope to:

1. Explain how there are secretly beautiful logical abstractions inside the apparent horror of windowing toolkits;
2. Illustrate how to write higher-order programs which automatically maintain complex imperative invariants, and
3. Show the audience some Javascript programs which we can claim are actually π-calculus terms in disguise.

References

[1] Samson Abramsky. Process realizability. *Electronic Notes in Theoretical Computer Science*, 23(1):1–2, 1999.

[2] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence. In *ACM SIGPLAN Notices (POPL 2009)*, volume 44, pages 340–353. ACM, 2009.

[3] Nick Benton and Philip Wadler. Linear logic, monads and the lambda calculus. In *Logic in Computer Science, 1996. LICS'96.*, pages 420–431. IEEE, 1996.

[4] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR 2010-Concurrency Theory*, pages 222–236. Springer, 2010.

[5] Cristiano Calcagno, Peter W O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *Logic in Computer Science, 2007. LICS 2007. 22nd Annual IEEE Symposium on*, pages 366–378. IEEE, 2007.

[6] Conal Elliott and Paul Hudak. Functional reactive animation. In *ACM SIGPLAN Notices (ICFP 1997)*, volume 32, pages 263–273. ACM, 1997.

[7] Jean-Yves Girard. Linear logic. *Theoretical computer science*, 50(1):1–101, 1987.

[8] Daniel Ignatoff, Gregory H Cooper, and Shriram Krishnamurthi. Crossing state lines: Adapting object-oriented frameworks to functional reactive languages. In *Functional and Logic Programming*, pages 259–276. Springer, 2006.

[9] Alan Jeffrey. LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. In *Proceedings of the sixth workshop on Programming languages meets program verification*, pages 49–60. ACM, 2012.

[10] Wolfgang Jeltsch. Towards a common categorical semantics for linear-time temporal logic and functional reactive programming. *Electronic Notes in Theoretical Computer Science*, 286:229–242, 2012.

[11] Glenn E Krasner, Stephen T Pope, et al. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, 1(3):26–49, 1988.

[12] Neelakantan R Krishnaswami. Higher-order functional reactive programming without spacetime leaks. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pages 221–232. ACM, 2013.

[13] Paul-André Mellies and Nicolas Tabareau. Resource modalities in tensor logic. *Annals of Pure and Applied Logic*, 161(5):632–653, 2009.

[14] Jennifer Paykin and Steve Zdancewic. A linear/producer/consumer model of classical linear logic. Technical report, University of Pennsylvania, April 2014.

[15] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002.

[16] Philip Wadler. Propositions as sessions. In *ACM SIGPLAN Notices (ICFP 2012)*, volume 47, pages 273–286. ACM, 2012.