Bridging Theory and Practice in Interaction

Stefan K. Muller[1] and Umut A. Acar[1,2]

1 Carnegie Mellon University, Pittsburgh, PA, USA. {umut,smuller}@cs.cmu.edu
2 Inria, Paris, France.

Abstract

Many software programs perform some interaction with the user or outside world through a GUI, console, file, network, etc. Several techniques have evolved for developing interactive applications. Probably the most widely-used, *event-driven programming*, has the advantage that it inherently supports concurrency. However, such programs are notoriously difficult to check and maintain (e.g., [1, 3, 5]), in part because callbacks break the key abstraction of a function call (callbacks may invoke each other and may not return to their caller), and often share global state. Because callbacks may run at any time, invariants on this state must hold in the presence of this very general form of concurrency. For example, a Unix shell may handle SIGCHLD signals (indicating that a child process has terminated) using a callback that removes the process from a global job list. When a new process is forked, it could run to completion, triggering the signal handler to remove the new job from the job list, before the shell process adds it. Foreseeing and fixing this race condition requires reasoning globally about unrelated callbacks.

A contrasting approach to event-driven programming is functional reactive programming (FRP) [4], in which a program transforms time-varying values (called *behaviors* or *signals*) from inputs to outputs. FRP is generally synchronous: all signal transformations run at all time steps, though some may only produce values at certain times. As a result, most FRP implementations do not handle concurrency. Elm [2] introduces some asynchrony but is still limited and does not have a concurrent implementation. In addition, to avoid time and space leaks, many FRP implementations also limit the expressiveness of the language by restricting the ways in which signals can be used.

We thus ask the question: is there a better way? We believe that, to be widely useful, a technique for writing interactive programs should excel at four properties: **expressiveness**, the ability to perform a wide variety of functions; **control over sampling or polling frequency**; **concurrency** and **usability**, the ability to write, reason about, and enforce invariants on the code of an interactive application. Event-driven programming excels at expressiveness and concurrency (and does not perform polling), but not at usability because it requires a very low level of abstraction. FRP is much more usable, but does not have the other three properties. We seek a middle ground between these two extremes, drawing an analogy with work on parallelism that shows that implicit parallelism, which offers a layer of abstraction on top of concurrency, is powerful enough to express many interesting parallel applications without requiring the programmer to deal with the full complexity of concurrent programming.

We have developed an interactivity abstraction which we call a *factor*. Using a higher-order operation, such as a map or fold, over a factor results in clean, functional-style code. Factors can be used in more complex ways, for example asynchronously, allowing programmers to control concurrency and sampling. As in implicit parallelism, the full complexity of concurrency and interaction is tamed by the abstractions. We have implemented factors (as an OCaml library) and a number of applications, including various physics-based simulations, several GUI programs, an internet-based music streaming server, and Unix shell as described above. In our implementation of the Unix shell, we are able to handle asynchronous signals locally and can easily avoid the race condition that arises in the event-driven implementation.

As part of our ongoing research, we started evaluating our approach against existing approaches both empirically and theoretically. The empirical evaluation involves developing a performance-testing framework as well as benchmarks that isolate and test specific forms of interaction. The theoretical evaluation involves formally specifying the semantics of the abstractions and establishing correspondences between them and well-understood forms of computation such as functional programming, with benign effects such as non-determinism. We hope these evaluation techniques will be of independent interest as we and others continue to explore the rich design space of concurrent interactive programs.

Conference title on which this volume is based on.
Editors: Billy Editor and Bill Editors; pp. 1–2

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

References

1. Brian Chin and Todd Millstein. Responders: Language support for interactive applications. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP'06, pages 255–278, 2006.
2. Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, PLDI '13, pages 411–422, 2013.
3. Jonathan Edwards. Coherent reaction. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 925–932, 2009.
4. Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN International Conference on Functional Programming*, pages 263–273. ACM, 1997.
5. Jeffrey Fischer, Rupak Majumdar, and Todd Millstein. Tasks: Language support for event-driven programming. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '07, pages 134–143, New York, NY, USA, 2007. ACM.