Bridging Theory and Practice in Interaction

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

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

Abstract

Many programs perform some interaction with the user or outside world by, for example, taking user input via a GUI or a file system, or interacting with remote processes via a network. Several techniques have evolved for developing interactive applications. Probably the most widely-used, *event-driven programming*, has the advantage that it inherently supports concurrency. Event-driven programs, however, are notoriously difficult to design, implement, and maintain (e.g., [1, 3, 5]), primarily because the invariants of the program are broken into different *callbacks* that interact using non-local control flow. For example, event-driven programs break the key abstraction of a function call (callbacks may invoke each other and may not return to their caller). Because callbacks may run at essentially any time, program invariants must hold in the presence of a rather general form of concurrency.

As an example of the complexities of event-driven programs, consider the following subtle race condition that might be found in an event-driven implementation of a Unix shell. Such a 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 starting a new job, the shell forks a new process and adds it to the job list. The race condition occurs when a process starts and runs to completion, and triggers the signal handler to remove the new job from the job list, before the shell process adds it to the job list. Foreseeing this race condition requires reasoning globally about when callbacks may run, and fixing it involves preventing the signal handler from running at an inopportune time (a simple fix like adding the process to the job list before forking doesn't work, since the process ID is obtained only after forking).

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.** Interactive applications should be able to interact with many sources of input and also perform complex computation.
- **Control over sampling/polling.** If an interactive system samples (polls) input sources at periodic intervals, frequency of sampling should be controllable by the programmer. Frequent polling may be important to ensure correctness and responsiveness, or may be wasteful, and this distinction can't, in general, be made by a runtime system.
- **Concurrency.** It should be possible to utilize concurrency to increase responsiveness.
- **Usability.** It should be possible to implement and reason about programs at a high level.

Event-driven programming excels in all aspects except for usability, because it provides a very low level of abstraction, requiring complex reasoning. FRP excels at usability, but does not have the other three properties. In other words, event-driven programming and FRP can

Conference title on which this volume is based on.
Editors: Billy Editor and Bill Editors; pp. 1–3

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

be viewed as the two extremes of the design space. We seek a middle ground. As an intuitive argument for why this should be possible, we draw an analogy with work on parallelism. Years of research in this area have shown 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. Similarly, we believe that it is possible to design abstractions that satisfy the four properties above.

We have developed a set of interactivity abstractions that revolve around the idea of a *factor*, which embodies the essence of interaction—two way communication—as a first-class value in a higher-order programming language. Factors can be defined and used just like ordinary values by, for example, employing higher-order operations, such as a map or fold. Since factors are first-class values, they require no restrictions on the programming language and thus lead to no loss of expressiveness. As a result, the programmer can write richly interactive code in a clean, functional style. Factors can be used concurrently by using several primitives that can create parallelism in a style similar to implicitly parallel languages. Thus, as in implicit parallelism, the abstractions tame the full complexity of concurrency.

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 a version of the Unix shell described above. In the factor implementation of the Unix shell, the race condition described above is not an issue because the asynchronous handling of signals is localized to the parts of the code where it should occur. For example, the code for the main input loop simultaneously waits for user input and signals. Because this asynchrony is local, rather than global, it is easy to reason about when signals may be handled. The diverse set of applications that we have implemented gives some practical evidence for the expressiveness of the factor-based abstractions.

As part of our ongoing research, we have 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.

Acknowledgments

This research is partially supported by the National Science Foundation under grant numbers CCF-1320563 and CCF-1408940, and by the European Research Council under grant number ERC-2012-StG-308246.

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.