Higher-order Programming is an Effect

Oleg Kiselyov

Tohoku University, Japan
oleg@okmij.org

Abstract. We argue that the central problem of the interaction of higher-order programming with various kinds of effects can be tackled by eliminating the distinction: higher-order facility is itself an effect, not too different from state effect.
We demonstrate that first-class abstractions may be treated uniformly as any other effects, thus completing the Cartwright and Felleisen's program of "Extensible Denotational Language Specifications": Variable substitution is indistinguishable from dereference of a C-like variable; "lambda", or creating a closure, is an effect as well. The (lexical) closure acts as a *handler* of all variable dereference effects arising during the execution of its body. Our approach uniformly handles dynamic and lexical binding and various calling conventions.
All in all, higher-order programming is essentially first-order single-assignment programming with first-class storage. A framework like extensible-effects that supports multiple effects should not hence have any problem with abstraction and substitution effects – which gives HOPE.

1 Motivation

This research has been a journey following the tantalizing lead: the founding papers on monads [5], monad transformers [7, 4], free monads [8] and extensible-effects [1, 3] were all about extensible interpreters. Along the way we have realized that the *expression problem*, *stable denotations*, and *extensible interpreters* are all different names for the same thing. We have eventually found [2, 6] the organizing principle to structure computation: *interaction* – between a client and a server, an interpreter and the interpreted code, an expression and its context.

It may well be that interaction is best expressed in a process calculus. In sequential calculi, Cartwright and Felleisen's "Extensible Denotational Language Specifications" [1] is the earliest fully worked-out rigorous presentations of effects as interactions.

2 Contributions

- The talk gives the modern reconstruction of Cartwright and Felleisen's "Extensible Denotational Language Specifications". Our reconstruction is simpler, aided by the tagless-final presentation, and immediately executable.
- We extend the original approach, making the effect handlers themselves be modular and extensible – and be part of a program rather than permanently stationed outside. This makes possible the following step:
- Cartwright and Felleisen had to bake the variable environment into the basic semantic framework, even though integers, or first-order sub-languages in general, make no use of it. We avoid postulating the variable environment and any special treatment of higher-order. After all, we treat Lambda on par with the State, in the same uniform formalism. Dynamic or lexical binding or various calling conventions boil down to different ways of handing the effect of dereferencing a bound variable.

Like call-by-push-value and the languages ATS or FX, we regard a lambda-abstraction an effectful expression. *Unlike* these calculi and languages, we treat a variable as an effectful expression as well. Bound variables are essentially single-assignment reference cells allocated in first-class storage, which is how they typically turned up in compiled code. Our approach hence shows promise of a faithful description of distributed and heterogeneous systems, where variable dereference is a non-trivial effect.

3 Supplementary material

The content of the talk is in the well-commented OCaml code
http://okmij.org/ftp/Computation/edlsng.ml.

The longer version of the talk uses Haskell98 and gives more examples and motivation:
http://okmij.org/ftp/Computation/having-effect.html.

4 Pure and Impure: Contingent rather than Absolute Distinction

What is an effect then? It is an interaction with the context. An expression is effectful if it needs context to interpret it. A lexically-scoped lambda-abstraction has to ask the context for the binding environment, and is hence effectful. A variable reference is effectful too, since it cannot be interpreted without a context. A closure, however, handles all variable dereference requests that occur during the execution of its body – and (absent state, IO, etc) requires no further context interactions. Thus, whereas the body of a lambda-abstraction is effectful, the entire lexical closure is pure.

Whether something is side-effect or not, is thus a matter of perspective. When focusing on a small part of computation, all interactions with the context – be they accessing an external memory or accessing a bound variable – are a side-effect.

References

[1] Robert Cartwright and Matthias Felleisen. Extensible denotational language specifications. In Masami Hagiya and John C. Mitchell, editors, *Theor. Aspects of Comp. Soft.*, number 789 in LNCS, pages 244–272. Springer, 1994.

[2] Carl Hewitt. Viewing control structures as patterns of passing messages. Memo 410, MIT Artificial Intelligence Laboratory, December 1976.

[3] Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects: an alternative to monad transformers. In *Haskell*, pages 59–70. ACM, 2013.

[4] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *POPL '95*, pages 333–343. ACM Press, 1995.

[5] Eugenio Moggi and Sonia Fagorzi. A monadic multi-stage metalanguage. In Andrew D. Gordon, editor, *Proceedings of FoSSaCS 2003: Foundations of Software Science and Computational Structures, 6th International Conference*, number 2620 in Lecture Notes in Computer Science, pages 358–374, Berlin, 7–11 April 2003. Springer.

[6] Manuel Silva. Half a century after Carl Adam Petri's Ph.D. thesis: A perspective on the field. *Annual Reviews in Control*, 37(2):191–219, 2013.

[7] Guy L. Steele, Jr. Building interpreters by composing monads. In *POPL '94*, pages 472–492. ACM Press, 1994.

[8] Wouter Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, July 2008.