Handlers.Js (Presentation)

A Comparative Study of Implementation Strategies for Effect Handlers on the Web

Daniel Hillerström
Sam Lindley
The University of Edinburgh, UK

Robert Atkey
University of Strathclyde, UK

KC Sivaramakrishnan
Jeremy Yallop
University of Cambridge, UK

Abstract

Handlers for algebraic effects provide a compelling and modular basis for effectful programming by separating the use of an effectful operation from its meaning. Previous work studies implementation strategies, although, most often focusing on a particular compilation strategy or a particular embedding technique. We compare a range of implementation strategies specifically for handlers on the web.

1 Motivation

Despite being a quite recent programming abstraction *handlers for algebraic effects* [17, 18] have already been widely adopted by the research community as witnessed by the many implementations and studies of handlers [1, 4, 7, 9–11, 13, 14]. Handlers are also gaining traction in industry, for instance, finding their way into Uber's probabilistic programming language Pyro [5], and inspiring the design of Facebook's rendering engine React Fiber [15] which underpins the popular JavaScript user interface framework React.

An attractive aspect of effect handlers is that they provide a modular abstraction for effectful programming which generalises a range of contemporary programming abstractions such as generators/iterators and async/await [3, 12]. This modularity is achieved by separating invocation of an effectful operation from implementation of its handled behaviour; akin to the separation of raising an exception from its handled behaviour. In this sense effect handlers generalise exception handlers, as they permit handling of arbitrary user-definable effects, including exceptions, non-determinism, concurrency, state, and so forth. Operationally, a handler captures the continuation of an operation invocation in its scope. The handler exposes this continuation as a first-class value to the programmer inside the handler, allowing the programmer to drop or stash away the continuation, or to invoke it, possibly multiple times, in order to resume execution at the invocation point of the operation.

Existing systems use different strategies to implement effect handlers. Eff [1] translates handlers to a free monad representation [19]. Multicore OCaml [4] adapts the stack-switching design by Bruggeman et al. [2] to provide an efficient native implementation. The web programming languages Koka and Links use different strategies. Koka performs a selective continuation passing style (CPS) translation to lift effectful code into a free monad [13]. Links uses a generalised CEK machine to implement handlers on the server-side [7] and makes use of a higher-order CPS translation on the client-side [8].

In this work we consider compilation of effect handlers to the web. We choose JavaScript as the compilation target as it presently remains the only viable option for the web; at least until WebAssembly has been brought up to speed [6]. We detail the advantages and disadvantages of each implementation strategy through a comparative study.

2 Five implementation strategies

We identify five implementation strategies for compiling handlers to JavaScript including two novel translations based on generators and iterators and another via generalised stack inception [16]. The other three strategies are the well-known free monad, CPS, and abstract machine approaches.

3 Methodology

As our starting point we implement each strategy as a separate JavaScript backend for Links. For each of the backends we evaluate the quality of the generated code on the five most popular JavaScript execution engines: i) Chakra used by Microsoft Edge, ii) JavaScriptCore used by Safari, iii) SpiderMonkey used by Firefox, iv) V8 used by Chrome and Opera, v) and Node.Js. To assess the quality of the generated code we measure the execution time and the memory consumption as well as the warm-up time on each engine.

4 Preliminary results

Preliminary results suggest that the CPS strategy outperforms the other strategies when running on a platform that supports tail call optimisation. Making it robust requires a trampoline which has a negative impact on the performance. The CEK machine is robust, but it has a notable interpretative overhead. This overhead could likely be reduced by using partial evaluation techniques. We are currently in the process of tweaking each of the backends.

5 Talk objectives

The talk will cover: i) motivating examples for programming with handlers on the web, ii) a high-level discussion of how each implementation strategy works, iii) and a comparative evaluation of the aforementioned strategies.

References

[1] Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.* 84, 1 (2015), 108–123.

[2] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. 1996. Representing Control in the Presence of One-Shot Continuations. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI), Philadephia, Pennsylvania, May 21-24, 1996*, Charles N. Fischer (Ed.). ACM, 99–107.

[3] Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, KC Sivaramakrishnan, and Leo White. 2017. Concurrent System Programming with Effect Handlers. TFP. (2017).

[4] Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy. 2015. Effective Concurrency through Algebraic Effects. OCaml Workshop. (2015).

[5] Noah Goodman. 2017. Uber AI Labs Open Sources Pyro, a Deep Probabilistic Programming Language. (Nov. 2017). https://eng.uber.com/pyro/

[6] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. 2017. Bringing the web up to speed with WebAssembly. In *PLDI*. ACM, 185–200.

[7] Daniel Hillerström and Sam Lindley. 2016. Liberating effects with rows and handlers. In *TyDe@ICFP*. ACM, 15–27.

[8] Daniel Hillerström, Sam Lindley, Robert Atkey, and K. C. Sivaramakrishnan. 2017. Continuation Passing Style for Effect Handlers. In *FSCD (LIPIcs)*, Vol. 84. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 18:1–18:19.

[9] Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. In *ICFP*. ACM, 145–158.

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

[11] Daan Leijen. 2017. Implementing Algebraic Effects in C - "Monads for Free in C". In *APLAS (Lecture Notes in Computer Science)*, Vol. 10695. Springer, 339–363.

[12] Daan Leijen. 2017. Structured Asynchrony with Algebraic Effects. In *TyDe@ICFP*. ACM, 16–29.

[13] Daan Leijen. 2017. Type directed compilation of row-typed algebraic effects. In *POPL*. ACM, 486–499.

[14] Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do be do be do. In *POPL*. ACM, 500–514.

[15] Sebasitian Markbaage. 2016. How React Fiber Works. (2016). https://www.facebook.com/groups/2003630259862046/permalink/2054053404819731/ Facebook discussion.

[16] Greg Pettyjohn, John Clements, Joe Marshall, Shriram Krishnamurthi, and Matthias Felleisen. 2005. Continuations from generalized stack inspection. In *ICFP*. ACM, 216–227.

[17] Gordon D. Plotkin and John Power. 2001. Adequacy for Algebraic Effects. In *FoSSaCS (LNCS)*, Vol. 2030. Springer, 1–24.

[18] Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4 (2013).

[19] Matija Pretnar, Amr Hany Saleh, Axel Faes, and Tom Schrijvers. 2017. *Efficient compilation of algebraic effects and handlers.* Technical Report CW 708. KU Leuven, Belgium.