Towards efficient implementations of effect handlers

– Extended Abstract –

Steven Keuchel
Universiteit Gent
steven.keuchel@ugent.be

Tom Schrijvers
Universiteit Gent
tom.schrijvers@ugent.be

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Functional Programming

General Terms Languages

Keywords effect handlers, modularity, delimited control, monads

1. Introduction

In recent years algebraic effects and effect handlers emerged as a compelling alternative to monads as a basis for effectful programming in functional programming languages. This approach provides language primitives for defining new abstract effectful operations, which the programmer uses to write his own programs, and effect handlers that instantiate the abstract operations with concrete implementations.

Abstract operations are composable and each effect handler instantiates a specific subset of the operations of a computation. Thus this approach yields modular abstraction and modular instantiation of effects similar to monad transformers and monad type classes. Transporting monads and monad transformers to different languages is possible but difficult or awkward. Some monads, like for example monadic parser combinators, crucially rely on infinite recursion and lazy evaluation. Modular abstraction with monads is achieved by means of type classes for each kind of effect and modular instantiation is achieved by type class instances that lift monad type classes over monad transformers. This crucially relies on automatic type class resolution.

Effect handlers are one possible alternative to monads and monad transformers for getting modularity in the handling of effects into a variety of functional programming languages, especially those that use strict evaluation and do not provide type classes.

2. Effect handlers

We present the general use of effect handlers using pseudo-syntax that is similar to the Frank language by Lindley and McBride [9]. The first step is to declare the signature of the abstract operations of an effect. The state effect for instance has two abstract operations: 1. get that retrieves the current state and 2. put that updates the state.

$$
\begin{array}{l}
sig\ State\ S \\
\quad = get : [\,]\ S \\
\quad \mid\ put : S \to [\,]\ Unit
\end{array}
$$

This declares get to be an effectful operation that results in a value of type S. put is an effectful operation that takes an a and returns a unit value. We use the notation $[\,]\ X$ to denote *computations* that upon execution return a value of type X.

We can write computations using these abstract operations. The $next$ computation performs state effects to increment a natural number. The first line is the type signature of $next$ which states that $next$ results in a value of type Nat and can perform $State\ Nat$ effects. The second line is the implementation of $next$ in terms of the abstract operations.

$$
\begin{array}{l}
next : [State\ Nat]\ Nat \\
next = get \to n; put\ (suc\ n); n
\end{array}
$$

The second step is to define a handler $state$ that implements the operations of the $State\ S$ effect. The handler $state$ takes as parameters an initial state s and a suspended computation on which it 'pattern matches'. The first two lines of the implementation handle the cases of the abstract operations get and put. The third line handles the case of a finished computation that resulted in a value v where we allow the handler to transform the result value.

$$
\begin{array}{l}
state : S \to [State\ S\ ?\ X] \to [\,]\ X \\
state\ s\ [get\ \ \ ?\ k] = state\ s\ ?\ k\ s \\
state\ \_\ [put\ s\ ?\ k] = state\ s\ ?\ k\ () \\
state\ \_\ [v] \qquad\quad = v
\end{array}
$$

3. Problem statement

If effect handlers are to be used as the principal paradigm of a language to model side-effects, their efficient implementation becomes an important point. The focus of existing work on languages [1, 9] and embedded domain-specific languages [3, 8] for programming with effect handlers is still to explore the expressivity of this alternative approach to effects. It should come as no surprise that benchmarking existing implementations shows that the performance of these systems is not yet competitive to monad transformers.

One of the main reasons for this is that these systems implement effect handlers indirectly, either by reducing them to a free-monad implementation in lazy languages or to (delimited) continuations in strict languages. This leaves a lot of room for improvement.

It is easier for a direct implementation to avoid technical pitfalls that impact performance and to leverage the structure of effect handlers to perform optimizations. More specifically the following concerns can be addressed in a better way.

1. The shift/reset operators are the most popular way to describe delimited continuations and are also the most commonly provided primitives by languages that implement delimited continuations. However, there is an impedance mismatch between effect handlers and delimited continuations using the shift/reset operators. In exception and effect handlers the delimiter describes the handling of the effectful operation in contrast to shift/reset where shift determines the *handler*. This also means for each reset delimiter there can be different handlers provided to different invocations of shift. As a consequence an indirect implementation might lose the opportunity to optimize using the fact that there is a constant handler for each effect handler. A conceptually better fit are delimited continuations using run/fcontrol [11].

2. State is an important concept that arises very often in the implementation of handlers of a variety of effects. Each handler invocation can potentially alter the state of the handler and the state needs to be threaded properly between invocations.

   Brady [3] treats the state of handlers explicitly by keeping track of a list of resources for a given handler stack. Kammar et al. [6] allow handlers to be parameterized and alter the parameters for handling the operations of the continuation.

   In the reduction of effect handlers to delimited control operators, the state of handlers is captured inside a closure that implements the handlers. A direct implementation of effect handlers should keep track of state explicitly to make state passing more efficient. Ideally, passing the state should be as efficient as passing an argument to a function.

3. Direct implementations of delimited continuations for call-by-value languages [7, 10] capture the continuation by copying the part of the stack up to the delimiter to the heap. When the continuation is invoked, this copy is pushed back onto the control stack. A direct implementation of effect handlers will necessarily perform comparable operations.

   However, if the implementation of a handler uses the continuation in a restricted way several optimizations are possible. If a continuation is only used in tail positions, the copying of the stack segment is unnecessary[7]; if the handler does not use the continuation at all, e.g. traditional exceptions, we can unwind the stack immediately before passing control to the handler function and thus free resources early.

   As it turns out, in practice a lot of effects fall into this these restricted categories [2]. It is therefore important to perform an analysis and optimize accordingly.

4. Towards efficient handlers

We want to address the performance concerns of effect handlers to help their adoption. Specifically we want to address the implementation and optimization of common cases that do not use the full power of effect handlers.

To this end, we are developing a definitional machine for effect handlers based on a definitional machine for delimited continuations [4, 7] that provides generic low-level primitives for the implementation of effect-handlers and a small call-by-value λ-calculus with effect handlers. In our development we address the following performance concerns:

4.1 Handler resolution

At the invocation point of an abstraction operation control is passed to handler for the effect. For this the concrete handler needs to be looked up. One of the problems appearing is how to do this efficiently.

In the implementation of exception handlers, the delimiter try is either pushing exception-handler marks explicitly on the control stack or is creating a table that maps code return addresses to exception handling code [5]. The throwing of an exception will unwind the control stack – executing cleanup code along the way – and use runtime-type information about the thrown value and the type of exception handlers to find the matching handler.

Obviously this dynamic lookup of effect handlers and the use of type-information is utterly slow. Using effect typing we can explicitly resolve the possible handlers statically and keep track of stack marks, handler functions and handler state explicitly and efficiently.

4.2 Fast linear code

We perform an analyses to detect handlers that invoke the continuations only tail position and handlers that discard the continuations. In these two cases we can avoid unnecessary work for capturing the continuation and also avoid duplicating state which would be necessary for potentially different continuations.

In this restricted setting the specialization of code for a specific set of handlers also becomes easier. We envision the inlining of known handlers to remove any overhead introduced by abstractions as much as possible.

References

[1] A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 2014.

[2] J. Berdine, P. O'hearn, U. Reddy, and H. Thielecke. Linear continuation-passing. *Higher-Order and Symbolic Computation*, 15 (2-3):181–208, 2002.

[3] E. Brady. Programming and reasoning with algebraic effects and dependent types. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pages 133–144. ACM, 2013.

[4] R. Dyvbig, S. P. Jones, and A. Sabry. A monadic framework for delimited continuations. *Journal of Functional Programming*, 17(06): 687–730, 2007.

[5] L. Goldthwaite. Technical report on c++ performance. *ISO/IEC PDTR*, 18015, 2006.

[6] O. Kammar, S. Lindley, and N. Oury. Handlers in action. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pages 145–158. ACM, 2013.

[7] O. Kiselyov. Delimited control in Ocaml, abstractly and concretely. *Theoretical Computer Science*, 2012.

[8] O. Kiselyov, A. Sabry, and C. Swords. Extensible effects: an alternative to monad transformers. In *Proceedings of the 2013 ACM SIGPLAN symposium on Haskell*, pages 59–70. ACM, 2013.

[9] S. Lindley and M. Conor. Do Be Do Be Do. draft, 2014.

[10] M. Masuko and K. Asai. Direct implementation of shift and reset in the mincaml compiler. In *Proceedings of the 2009 ACM SIGPLAN workshop on ML*, pages 49–60. ACM, 2009.

[11] D. Sitaram. Handling control. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 147–155, New York, NY, USA, 1993. ACM. ISBN 0-89791-598-4. . URL `http://doi.acm.org/10.1145/155090.155104`.