Effect handlers

$$\textbf{handle}\ (\textbf{do}\ x \leftarrow op(v;\ y.c_1)\ \textbf{in}\ c_2)\ \textbf{with}\ h$$

Jeremy Yallop

jeremy.yallop@cl.cam.ac.uk

Evaluation & the stack

Expression evaluation (fine-grain CBV)

Values

$$\begin{array}{rcll} U, V & ::= & x & \textit{variable} \\ & | & \lambda x.C & \textit{abstraction} \end{array}$$

Computations

$$\begin{array}{rcll} C, D & ::= & \mathbf{return}\ V & \textit{value} \\ & | & U\ V & \textit{application} \\ & | & \mathbf{do}\ x \leftarrow C\ \mathbf{in}\ D & \textit{sequencing} \end{array}$$

Computation rules

$$\frac{}{(\lambda x.C)\ V \rightsquigarrow C\{V/x\}}$$

$$\frac{}{\mathbf{do}\ x \leftarrow \mathbf{return}\ V\ \mathbf{in}\ C \rightsquigarrow C\{V/x\}}$$

Congruence rules

$$\frac{C \rightsquigarrow C'}{\mathbf{do}\ x \leftarrow C\ \mathbf{in}\ D \rightsquigarrow \mathbf{do}\ x \leftarrow C'\ \mathbf{in}\ D}$$

Values

$$U, V ::= x \mid \lambda x.C$$

Computations

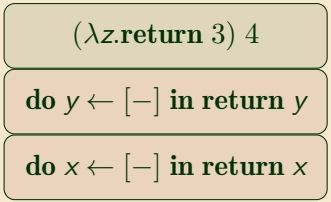
$$C, D ::= \mathbf{return}\ V \mid U\ V \mid \mathbf{do}\ x \leftarrow C\ \mathbf{in}\ D$$

Continuations

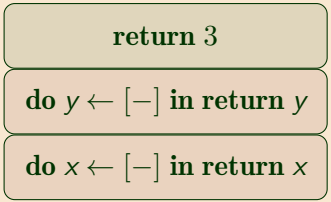
$$E[\,\cdot\,] ::= [\,\cdot\,] \mid E[\mathbf{do}\ x \leftarrow [\,\cdot\,]\ \mathbf{in}\ C]$$

Computation rules

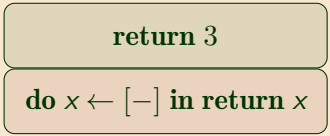
$$E[(\lambda x.C)\ V] \rightsquigarrow E[C\{V/x\}]$$
$$E[\mathbf{do}\ x \leftarrow \mathbf{return}\ V\ \mathbf{in}\ C] \rightsquigarrow E[C\{V/x\}]$$



$$\mathbf{do}\ x \leftarrow (\mathbf{do}\ y \leftarrow (\lambda z.\mathbf{return}\ 3)\ 4\ \mathbf{in\ return}\ y)\ \mathbf{in\ return}\ x$$



$$\textbf{do}\ x \leftarrow (\textbf{do}\ y \leftarrow \textbf{return}\ 3\ \textbf{in return}\ y)\ \textbf{in return}\ x$$



$$\textbf{do}\ x \leftarrow \textbf{return}\ 3\ \textbf{in return}\ x$$

return 3

Effect handlers

Values

$$\begin{array}{rcl} U, V & ::= & x \\ & \mid & \lambda x.C \end{array}$$

Computations

$$\begin{array}{rcl} C, D & ::= & \mathbf{return}\ V \\ & \mid & U\ V \\ & \mid & \mathbf{do}\ x \leftarrow C\ \mathbf{in}\ D \\ & \mid & \mathbf{op}(V; y.C) \\ & \mid & \mathbf{handle}\ C\ \mathbf{with}\ \{\mathbf{return}\ x \mapsto C_x, \overline{\mathbf{op}_i(x_i; k_i) \mapsto C_i}\} \end{array}$$

Continuations

$$\begin{array}{rcl} E[\,\cdot\,] & ::= & [\,\cdot\,] \\ & \mid & E[\mathbf{do}\ x \leftarrow [\,\cdot\,]\ \mathbf{in}\ C] \\ & \mid & E[\mathbf{handle}\ [\,\cdot\,]\ \mathbf{with}\ h] \end{array}$$

Computation rules

$E[(\lambda x.C)\ V]$
$\rightsquigarrow E[C\{V/x\}]$

$E[\mathbf{do}\ x \leftarrow \mathbf{return}\ V\ \mathbf{in}\ C]$
$\rightsquigarrow E[C\{V/x\}]$

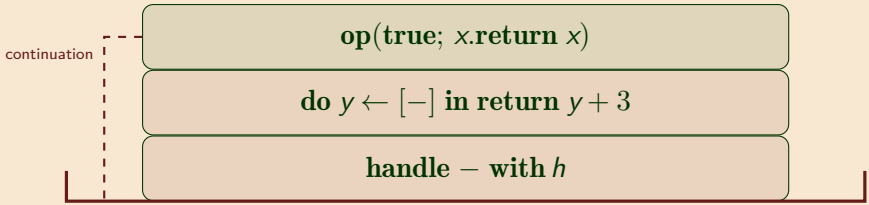
$E[\mathbf{do}\ x \leftarrow \mathbf{op}(V;\ y.C)\ \mathbf{in}\ D]$
$\rightsquigarrow E[\mathbf{op}(V;\ y.\mathbf{do}\ x \leftarrow C\ \mathbf{in}\ D)]$ (continuation capture)

$E[\mathbf{handle}\ \mathbf{return}\ V\ \mathbf{with}\ \{\mathbf{return}\ x \mapsto C, \overline{\mathbf{op}_i(x_i; k_i) \mapsto C_i}\}]$
$\rightsquigarrow E[C\{V/x\}]$ (return a value from a handler)

$E[\mathbf{handle}\ \mathbf{op_i}(V;\ y.C)\ \mathbf{with}\ \{\mathbf{return}\ x \mapsto C, \overline{\mathbf{op}_i(x_i; k_i) \mapsto C_i}\}]$
$\rightsquigarrow E[C_i\{V/x, (\lambda y.\mathbf{handle}\ C\ \mathbf{with}\ \ldots)/k\}]$ (handle an effect)

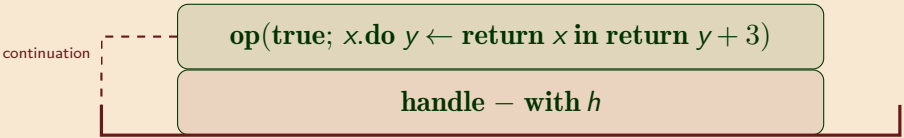
$E[\mathbf{handle}\ \mathbf{op_j}(V;\ y.C)\ \mathbf{with}\ \{\mathbf{return}\ x \mapsto C, \overline{\mathbf{op}_i(x_i; k_i) \mapsto C_i}\}]$
$\rightsquigarrow E[\mathbf{op_j}(V;\ y.\mathbf{handle}\ C\ \mathbf{with}\ \ldots)]$ (forward an effect)

Example



$$\textbf{handle do } y \leftarrow \textbf{op}(\textbf{true};\ x.\textbf{return } x) \textbf{ in return } y + 3 \textbf{ with } h$$

$$h = \{\textbf{return } v \mapsto v + 1, \textbf{op}(v; k) \mapsto \textbf{if } v \textbf{ then } k\ 10 \textbf{ else return } 0\}$$



$$\textbf{handle}\ \textbf{op}(\textbf{true};\ x.\textbf{do}\ y \leftarrow \textbf{return}\ x\ \textbf{in return}\ y + 3)\ \textbf{with}\ h$$

$$h = \{\textbf{return}\ v \mapsto v + 1, \textbf{op}(v; k) \mapsto \textbf{if}\ v\ \textbf{then}\ k\ 10\ \textbf{else return}\ 0\}$$

$$\textbf{if true then } (\lambda x. \ldots)\ 10 \textbf{ else return } 0$$

$$\textbf{if true then } (\lambda x.\textbf{handle do } y \leftarrow \textbf{return } x \textbf{ in return } y + 3 \textbf{ with } h)\ 10 \textbf{ else return } 0$$

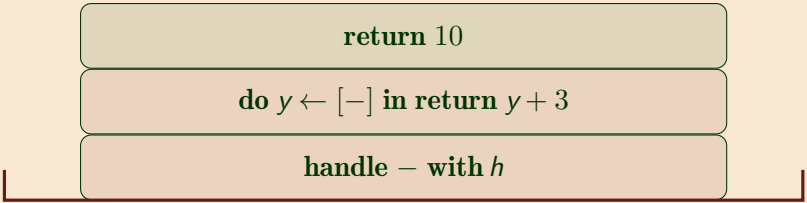
$$h = \{\textbf{return } v \mapsto v + 1, \textbf{op}(v; k) \mapsto \textbf{if } v \textbf{ then } k\ 10 \textbf{ else return } 0\}$$

$$(\lambda x.\textbf{handle do } y \leftarrow \textbf{return } x \textbf{ in return } y + 3 \textbf{ with } h)\ 10$$

$$(\lambda x.\textbf{handle do } y \leftarrow \textbf{return } x \textbf{ in return } y + 3 \textbf{ with } h)\ 10$$

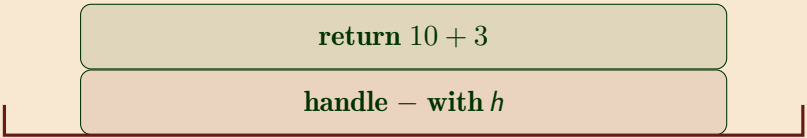
$$h = \{\textbf{return } v \mapsto v + 1, \textbf{op}(v; k) \mapsto \textbf{if } v \textbf{ then } k\ 10 \textbf{ else return } 0\}$$

Example



$$\textbf{handle do } y \leftarrow \textbf{return } 10 \textbf{ in return } y + 3 \textbf{ with } h$$

$$h = \{\textbf{return } v \mapsto v + 1, \textbf{op}(v; k) \mapsto \textbf{if } v \textbf{ then } k\ 10 \textbf{ else return } 0\}$$



handle return $10 + 3$ **with** h

$h = \{\textbf{return}\ v \mapsto v + 1, \textbf{op}(v; k) \mapsto \textbf{if}\ v\ \textbf{then}\ k\ 10\ \textbf{else return}\ 0\}$

$$\textbf{return}\ (10 + 3) + 1$$

$$\textbf{return}\ (10 + 3) + 1$$

Applications

(demo)

Reading

Paper 1: *Retrofitting Effect Handlers onto OCaml*

The stack

Handlers

Applications

Reading

Retrofitting Effect Handlers onto OCaml

KC Sivaramakrishnan
IIT Madras
Chennai, India
kcsrk@cse.iitm.ac.in

Stephen Dolan
OCaml Labs
Cambridge, UK
stephen.dolan@cl.cam.ac.uk

Leo White
Jane Street
London, UK
leo@lpw25.net

Sadiq Jaffer
Opsian and OCaml Labs
Cambridge, UK
sadiq@tuxo.com

Tom Kelly
OCaml Labs
Cambridge, UK
tom.kelly@cantab.net

Anil Madhavapeddy
University of Cambridge and OCaml Labs
Cambridge, UK
avsm2@cl.cam.ac.uk

Abstract

Effect handlers have been gathering momentum as a mechanism for modular programming with user-defined effects. Effect handlers allow for non-local control flow mechanisms such as generators, async/await, lightweight threads and coroutines to be composably expressed. We present a design and evaluate a full-fledged efficient implementation of effect handlers for OCaml, an industrial-strength multi-paradigm programming language. Our implementation strives to maintain the backwards compatibility and performance profile of existing OCaml code. Retrofitting effect handlers onto OCaml is challenging since OCaml does not currently have any non-local control flow mechanisms other than exceptions. Our implementation of effect handlers for OCaml: *(i)* imposes a mean 1% overhead on a comprehensive macro benchmark suite that does not use effect handlers; *(ii)* remains compatible with program analysis tools that inspect the stack; and *(iii)* is efficient for new code that makes use of effect handlers.

CCS Concepts: • **Software and its engineering** → **Runtime environments**; **Concurrent programming structures**; **Control structures**; *Parallel programming languages*; **Concurrent programming languages**.

Keywords: Effect handlers, Backwards compatibility, Fibers, Continuations, Backtraces

ACM Reference Format:
KC Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, and Anil Madhavapeddy. 2021. Retrofitting Effect Handlers onto OCaml. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, UK.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3453483.3454039

PLDI '21, June 20–25, 2021, Virtual, UK

ACM ISBN 978-1-4503-8391-2/21/06.
https://doi.org/10.1145/3453483.3454039

1 Introduction

Effect handlers [46] provide a modular foundation for user-defined effects. The key idea is to separate the definition of the effectful operations from their interpretations, which are given by handlers of the effects. For example,

```
effect In_line : in_channel -> string
```

declares an *effect* `In_line`, which is parameterised with an input channel of type `in_channel`, which when *performed* returns a `string` value. A computation can perform the `In_line` effect without knowing how the `In_line` effect is implemented. This computation may be enclosed by different handlers that handle `In_line` differently. For example, `In_line` may be implemented by performing a blocking read on the input channel or performing the read asynchronously by offloading it to an event loop such as `libuv`, without changing the computation. Thanks to the separation of effectful operations from their implementation, effect handlers enable new approaches to modular programming. Effect handlers are a generalisation of exception handlers, where, in addition to the effect being handled, the handler is provided with the delimited continuation [19] of the perform site. This continuation may be used to resume the suspended computation later. This enables non-local control-flow mechanisms such as resumable exceptions, lightweight threads, coroutines, generators and asynchronous I/O to be composably expressed.

One of the primary motivations to extend OCaml with effect handlers is to natively support asynchronous I/O in order to express highly scalable concurrent applications such as web servers in *direct style* (as opposed to using callbacks). Many programming languages, including OCaml, require non-local changes to source code in order to support asynchronous I/O, often leading to a dichotomy between synchronous and asynchronous code [11]. For asynchronous I/O, OCaml developers typically use libraries such as Lwt [54] and Async [41, §18], where asynchronous functions are represented as monadic computations. In these libraries, while asynchronous functions can call synchronous functions directly, the converse is not true. In particular, any function that calls an asynchronous function will also have to be marked as asynchronous. As a result, large parts of the applications using these libraries end up being in monadic form.

> *"Effect handlers allow for non-local control flow mechanisms such as generators, async/await, lightweight threads and coroutines to be composably expressed. We present a design and evaluate a full-fledged efficient implementation of effect handlers for OCaml, an industrial-strength multi-paradigm programming language.*
>
> *[...]*
>
> *Our implementation of effect handlers for OCaml: (i) imposes a mean 1% overhead on a comprehensive macro benchmark suite that does not use effect handlers; (ii) remains compatible with program analysis tools that inspect the stack; and (iii) is efficient for new code that makes use of effect handlers."*

Paper 2: *Generalized Evidence Passing for Effect Handlers*

The stack

Handlers

Applications

Reading

Generalized Evidence Passing for Effect Handlers

Efficient Compilation of Effect Handlers to C

NINGNING XIE, University of Hong Kong, China
DAAN LEIJEN, Microsoft Research, USA

This paper studies compilation techniques for algebraic effect handlers. In particular, we present a sequence of refinements of algebraic effects, going via multi-prompt delimited control, *generalized evidence passing*, yield bubbling, and finally a monadic translation into plain lambda calculus which can be compiled efficiently to many target platforms. Along the way we explore various interesting points in the design space. We provide two implementations of our techniques, one as a library in Haskell, and one as a C backend for the Koka programming language. We show that our techniques are effective, by comparing against three other best-in-class implementations of effect handlers: multi-core OCaml, the *Ev.Eff* Haskell library, and the libhandler C library. We hope this work can serve as a basis for future designs and implementations of algebraic effects.

CCS Concepts: • **Software and its engineering → Control structures**; **Polymorphism**; • **Theory of computation → Type theory**.

Additional Key Words and Phrases: Algebraic Effects, Handlers, Evidence Passing

ACM Reference Format:
Ningning Xie and Daan Leijen. 2021. Generalized Evidence Passing for Effect Handlers: Efficient Compilation of Effect Handlers to C. *Proc. ACM Program. Lang.* 5, ICFP, Article 71 (August 2021), 30 pages. https://doi.org/10.1145/3473576

1 INTRODUCTION

Algebraic effects and handlers [Plotkin and Power 2003; Plotkin and Pretnar 2013] provide a powerful and flexible way to add structured control-flow abstraction to programming languages. Unfortunately, it is not straightforward to compile effect handlers into efficient code: effect operations are generally able to capture- and resume a delimited continuation, which usually requires special runtime support to do efficiently. For example, the effect handler implementation in multi-core OCaml [Dolan et al. 2017; Sivaramakrishnan et al. 2021] relies on a runtime system that uses segmented stacks which can be captured efficiently [Farvardin and Reppy 2020]. Then, a natural question that arises is whether it is possible to compile effect handlers efficiently where the target platform does not directly support delimited continuations, for example, when compiling to C/LLVM, WASM [Haas et al. 2017], JavaScript, Java VM,. NET, etc.

In this paper we give a formalized translation and evaluation semantics from a typed effect handler calculus into a plain typed lambda calculus as a sequence of refinements:

(1) First we show how effect handler semantics can be expressed using standard *multi-prompt*

> *"This paper studies compilation techniques for algebraic effect handlers. In particular, we present a sequence of refinements of algebraic effects, going via multi-prompt delimited control, generalized evidence passing, yield bubbling, and finally a monadic translation into plain lambda calculus which can be compiled efficiently to many target platforms.*
>
> *[…]*
> *We show that our techniques are effective, by comparing against three other best-in-class implementations of effect handlers: multi-core OCaml, the Ev.Eff Haskell library, and the libhandler C library."*

Paper 3: *Do Be Do Be Do*

Do Be Do Be Do

Sam Lindley
The University of Edinburgh, UK
sam.lindley@ed.ac.uk

Conor McBride
University of Strathclyde, UK
conor.mcbride@strath.ac.uk

Craig McLaughlin
The University of Edinburgh, UK
craig.mclaughlin@ed.ac.uk

Abstract

We explore the design and implementation of Frank, a strict functional programming language with a bidirectional effect type system designed from the ground up around a novel variant of Plotkin and Pretnar's effect handler abstraction.

Effect handlers provide an abstraction for modular effectful programming: a handler acts as an interpreter for a collection of commands whose interfaces are statically tracked by the type system. However, Frank eliminates the need for an additional effect handling construct by generalising the basic mechanism of functional abstraction itself. A function is simply the special case of a Frank *operator* that interprets no commands. Moreover, Frank's operators can be *multihandlers* which simultaneously interpret commands from several sources at once, without disturbing the direct style of functional programming with values.

Effect typing in Frank employs a novel form of effect polymorphism which avoids mentioning effect variables in source code. This is achieved by propagating an *ambient ability* inwards, rather than accumulating unions of potential effects outwards.

We introduce Frank by example, and then give a formal account of the Frank type system and its semantics. We introduce Core Frank by elaborating Frank operators into functions, case expressions, and unary handlers, and then give a sound small-step operational semantics for Core Frank.

Programming with effects and handlers is in its infancy. We contribute an exploration of future possibilities, particularly in combination with other forms of rich type system.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Applicative (functional) languages

Keywords algebraic effects, effect handlers, effect polymorphism, call-by-push-value, pattern matching, continuations, bidirectional typing

1. Introduction

Shall I be pure or impure? —Philip Wadler [60]

We say 'Yes.': purity is a choice to make *locally*. We introduce **Frank**, an applicative language where the meaning of 'impure' computations is open to negotiation, based on Plotkin and Power's algebraic effects [45–48] in conjunction with Plotkin and Pretnar's handlers for algebraic effects [49]—a rich foundation for effectful programming. By separating effect interfaces from their implementation, algebraic effects offer a high degree of modularity. Programmers can express effectful programs independently of the concrete interpretation of their effects. A handler gives one interpretation of the effects of a computation. In Frank, effect types (sometimes called simply *effects* in the literature) are known as *abilities*. An ability denotes the permission to invoke a particular set of commands.

Frank programs are written in direct style in the spirit of effect type systems [34, 57]. Frank *operators* generalise call-by-value functions in two dimensions. First, operators handle effects. A unary operator is an effect handler, acting as an interpreter for a specified set of commands whose types are statically tracked by the type system. A unary function is simply the special case of a unary operator whose handled command set is empty. Second, operators are *n*-ary, handling multiple computations over distinct command sets simultaneously. An *n*-ary function is simply the special case of an *n*-ary operator whose handled command sets are all empty.

The contributions of this paper are:

- the definition of Frank, a strict functional programming language featuring a bidirectional effect type system, effect polymorphism, and effect handlers;
- operators as both *multihandlers* for handling multiple computations over distinct effect sets simultaneously and as *functions* acting on values;
- a novel approach to effect polymorphism which avoids mentioning effect variables in source code, crucially relying on the observation that one must always instantiate the effects of an operator being applied with the *ambient ability*, that is, precisely those algebraic effects permitted by the current typing context;
- a description of pattern matching compilation from Frank into a fairly standard call-by-value language with unary effect handlers, *Core Frank*;
- a straightforward small-step operational semantics for Core Frank and a proof of type soundness;
- an exploration of directions for future research, combining effect-and-handlers programming with features including substructural typing, dependent types, and totality.

A number of other languages and libraries are built around effect handlers and algebraic effects. Bauer and Pretnar's Eff [7] language is an ML-like language extended with effect handlers. A significant difference between Frank and the original version of Eff is that the latter provides no support for effect typing. Recently Bauer and Pretnar have designed an effect type system for Eff [6]. Their implementation [50] supports Hindley-Milner type inference and the type system incorporates effect subtyping.

> "*A novel approach to effect polymorphism which avoids all mention of effect variables*
>
> *[…]*
> *Multi-handlers as both an abstraction for handling multiple computations over different effect sets simultaneously and a characterisation of effect-handlers as generalised functions.*
>
> "

Expressiveness

Are some programs easier to express with effect handlers?

Do effect handlers add extra power that makes reasoning more difficult?

Efficiency

Can effect handlers be implemented efficiently?

Can existing languages be extended with efficient handler implementations?

Types

How are effect handlers typed?

Are types used in compilation?

Variants

Shallow vs deep, single- vs multi-shot, multi-handlers, ...