Algebraic Effects and Handlers

Ningning Xie

University of Toronto
OPLSS 2025

Acknowledgments

- *Lectures on Algebraic Effects*, Gordon Plotkin, NII Shonan meeting No. 146, 2019
- *Effect-Handler Oriented Programming*, Sam Lindley, OPLSS'22
- My collaborators: Daan Leijen, Jonathan Brachthäuser, Daniel Hillerström, Philipp Schuster, Youyou Cong, Kazuki Ikemori, Daniel D. Johnson, Dougal Maclaurin, Adam Paszke, Gordon Plotkin

Effects are everywhere

- Input/output
- Read/Write
- Exceptions
- Mutable states
- Concurrency
- Backtracking
- ...

Effects are everywhere

- Input/output
- Read/Write
- Exceptions
- Mutable states
- Concurrency
- Backtracking
- ...

Effects are often ad-hoc and hard-coded.

Algebraic effects and effect handlers

Composable and **structured** control-flow abstraction.



FoSSaCS'01

Adequacy for Algebraic Effects

Gordon Plotkin and John Power *

Division of Informatics, University of Edinburgh, King's Buildings, Edinburgh EH9 3JZ, Scotland

Abstract. Moggi proposed a monadic account of computational effects. He also presented the computational λ-calculus, λ_c, a core call-by-value functional programming language for effects; the effects are obtained by adding appropriate operations. The question arises as to whether one can give a corresponding treatment of operational semantics. We do this in the case of algebraic effects where the operations are given by a single-sorted algebraic signature, and their semantics is supported by the monad, in a certain sense. We consider call-by-value PCF with—and without—recursion, an extension of λ_c with arithmetic. We prove general adequacy theorems, and illustrate these with two examples: nondeterminism and probabilistic nondeterminism.

ESOP'09

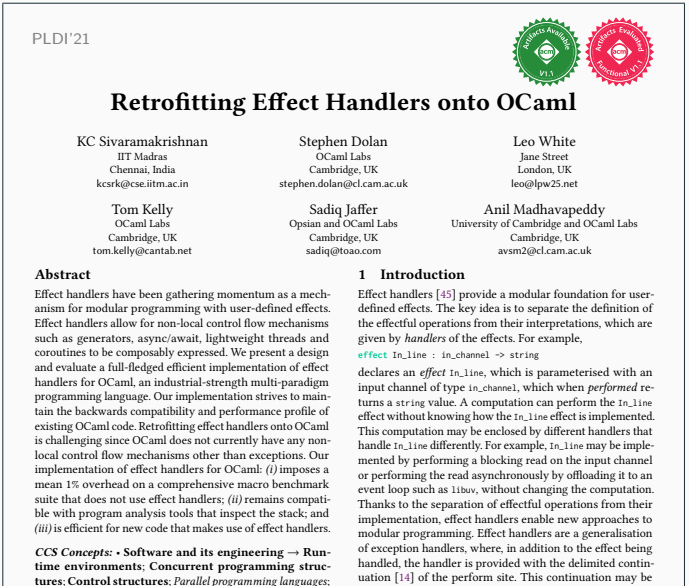
Handlers of Algebraic Effects

Gordon Plotkin * and Matija Pretnar **

Laboratory for Foundations of Computer Science,
School of Informatics, University of Edinburgh, Scotland

Abstract. We present an algebraic treatment of exception handlers and, more generally, introduce handlers for other computational effects representable by an algebraic theory. These include nondeterminism, interactive input/output, concurrency, state, time, and their combinations; in all cases the computation monad is the free-model monad of the theory. Each such handler corresponds to a model of the theory for the effects at hand. The handling construct, which applies a handler to a computation, is based on the one introduced by Benton and Kennedy, and is interpreted using the homomorphism induced by the universal property

Growing interest in academia



PLDI'21

Artifacts Available v1.1

Artifacts Evaluated Functional v1.1

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

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

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

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*;

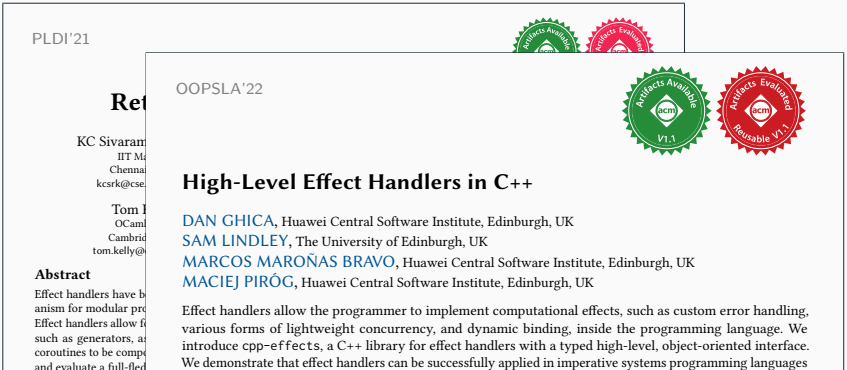
1 Introduction

Effect handlers [45] 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 [14] of the perform site. This continuation may be

Growing interest in academia



PLDI'21

Ret

KC Sivaram
IIT Ma
Chennai
kcsrk@cse.

Tom K
OCaml
Cambrid
tom.kelly@

Abstract

Effect handlers have b
anism for modular pro
Effect handlers allow f
such as generators, as
coroutines to be compo
and evaluate a full-fled
handlers for OCaml, a
programming language
tain the backwards cor
existing OCaml code. R
is challenging since OC
local control flow mec
implementation of effe
mean 1% overhead on
suite that does not use
ble with program anal
(iii) is efficient for new

CCS Concepts: • **Soft**
time environments;
tures; **Control struct**

OOPSLA'22

Artifacts Available acm V1.1

Artifacts Evaluated acm Reusable V1.1

High-Level Effect Handlers in C++

DAN GHICA, Huawei Central Software Institute, Edinburgh, UK
SAM LINDLEY, The University of Edinburgh, UK
MARCOS MAROÑAS BRAVO, Huawei Central Software Institute, Edinburgh, UK
MACIEJ PIRÓG, Huawei Central Software Institute, Edinburgh, UK

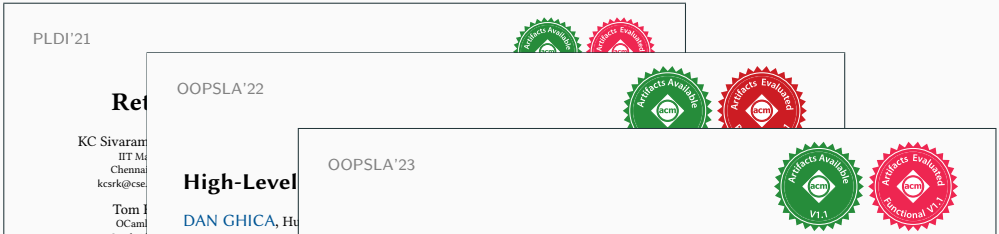
Effect handlers allow the programmer to implement computational effects, such as custom error handling, various forms of lightweight concurrency, and dynamic binding, inside the programming language. We introduce `cpp-effects`, a C++ library for effect handlers with a typed high-level, object-oriented interface. We demonstrate that effect handlers can be successfully applied in imperative systems programming languages with manual memory management. Through a collection of examples, we explore how to program effectively with effect handlers in C++, discuss the intricacies and challenges of the implementation, and show that despite its limitations, `cpp-effects` performance is competitive and in some cases even outperforms state-of-the-art approaches such as C++20 coroutines and the `libmprompt` library for multiprompt delimited control.

CCS Concepts: • **Software and its engineering → Control structures**; *Coroutines*; *Concurrent programming structures*.

Additional Key Words and Phrases: Effect handlers, algebraic effects, lightweight concurrency, context switching

ACM Reference Format:
Dan Ghica, Sam Lindley, Marcos Maroñas Bravo, and Maciej Piróg. 2022. High-Level Effect Handlers in C++. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 183 (October 2022), 29 pages. https://doi.org/10.1145/3563445

Growing interest in academia



PLDI'21

Ret

KC Sivaram
IIT Ma
Chenna
kcsrk@cse.

Tom K
OCaml
Cambrid
tom.kelly@

Abstract

Effect handlers have b
anism for modular pro
Effect handlers allow f
such as generators, as
coroutines to be compo
and evaluate a full-fled
handlers for OCaml, a
programming language
tain the backwards com
existing OCaml code. R
is challenging since OC
local control flow mec
implementation of effe
mean 1% overhead on
suite that does not use
ble with program anal
(iii) is efficient for new

CCS Concepts: • **Soft**
time environments;
tures; **Control struct**

OOPSLA'22

High-Level

DAN GHICA, Hu
SAM LINDLEY,
MARCOS MAR
MACIEJ PIRÓG,

Effect handlers allo
various forms of lig
introduce `cpp-effe`
We demonstrate that
with manual memor
with effect handlers
its limitations, `cpp-`
approaches such as

CCS Concepts: • **Sof**
structures.

Additional Key Word
ing

ACM Reference Fo
Dan Ghica, Sam Lin
Proc. ACM Program.

OOPSLA'23

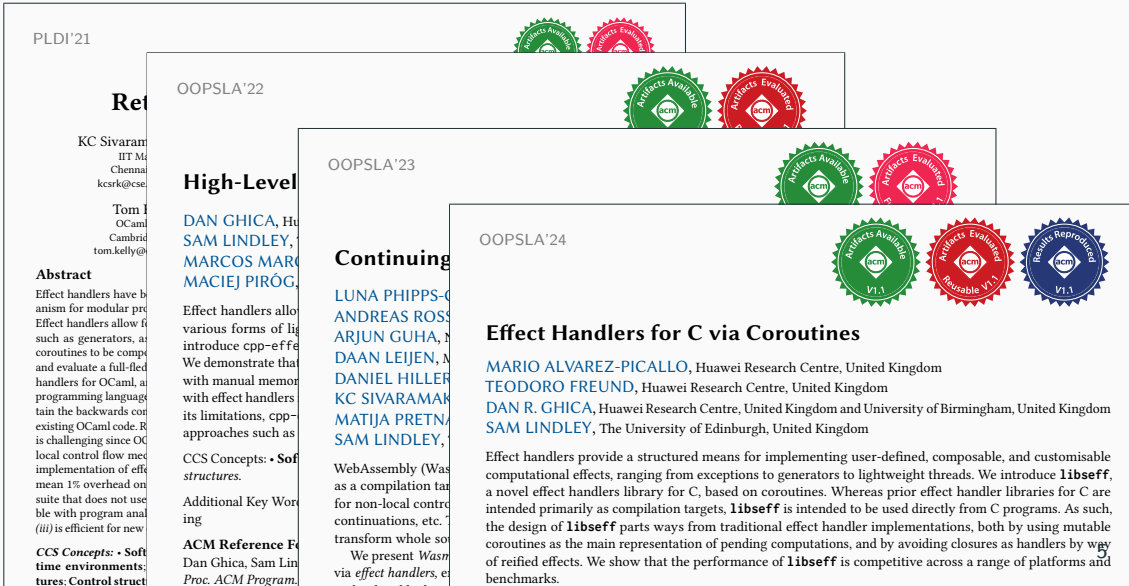
Continuing WebAssembly with Effect Handlers

LUNA PHIPPS-COSTIN, Northeastern University, United States
ANDREAS ROSSBERG, Independent, Germany
ARJUN GUHA, Northeastern University and Roblox, United States
DAAN LEIJEN, Microsoft Research, United States
DANIEL HILLERSTRÖM, Huawei Zurich Research Center, Switzerland
KC SIVARAMAKRISHNAN, Tarides and IIT Madras, India
MATIJA PRETNAR, University of Ljubljana and Institute of Mathematics, Physics & Mechanics, Slovenia
SAM LINDLEY, The University of Edinburgh, United Kingdom

WebAssembly (Wasm) is a low-level portable code format offering near native performance. It is intended as a compilation target for a wide variety of source languages. However, Wasm provides no direct support for non-local control flow features such as async/await, generators/iterators, lightweight threads, first-class continuations, etc. This means that compilers for source languages with such features must ceremoniously transform whole source programs in order to target Wasm.

We present *WasmFX*, an extension to Wasm which provides a universal target for non-local control features via *effect handlers*, enabling compilers to translate such features directly into Wasm. Our extension is minimal

Growing interest in academia



Growing interest in industry

SEMANTICS

code analysis service

JavaScript library for building UI

Python probabilistic programming

Fusing industry and academia at Github

ICFP'22

Fusing Industry and Academia at GitHub (Experience Report)

PATRICK THOMSON, GitHub, Inc., United States
ROB RIX, GitHub, Inc., Canada
NICOLAS WU, Imperial College London, United Kingdom
TOM SCHRIJVERS, KU Leuven, Belgium

GitHub hosts hundreds of millions of code repositories written in hundreds of different programming languages. In addition to its hosting services, GitHub provides data and insights into code, such as vulnerability analysis and code navigation, with which users can improve and understand their software development process. GitHub has built Semantic, a program analysis tool capable of parsing and extracting detailed information from source code. The development of Semantic has relied extensively on the functional programming literature; this paper describes how connections to academic research inspired and informed the development of an industrial-scale program analysis toolkit.

CCS Concepts: • **Software and its engineering → General programming languages**; • **Social and professional topics** → *History of programming languages.*

Additional Key Words and Phrases: effects, Haskell, data types, industry

ACM Reference Format:
Patrick Thomson, Rob Rix, Nicolas Wu, and Tom Schrijvers. 2022. Fusing Industry and Academia at GitHub (Experience Report). *Proc. ACM Program. Lang.* 6, ICFP, Article 108 (August 2022), 16 pages. https://doi.org/10.1145/3547639

1 INTRODUCTION

GitHub is a service that provides storage for repositories of source code tracked with the Git

Fusing industry and academia at Github

ICFP'22

Fusing Industry and Academia at GitHub (Experience Report)

PATRICK THOMSON, GitHub, Inc., United States

"An example of the utility and flexibility of algebraic effects is an effect we developed to extract telemetry data from our production Haskell systems."

" In a production context, we wanted these data to be uploaded to an aggregator; in a development context we wanted to see them reported on the command-line; and when running automated tests, we wanted to discard them entirely"

Additional Key Words and Phrases: effects, Haskell, data types, industry

ACM Reference Format:
Patrick Thomson, Rob Rix, Nicolas Wu, and Tom Schrijvers. 2022. Fusing Industry and Academia at GitHub (Experience Report). *Proc. ACM Program. Lang.* 6, ICFP, Article 108 (August 2022), 16 pages. https://doi.org/10.1145/3547639

1 INTRODUCTION

GitHub is a service that provides storage for repositories of source code tracked with the Git

Effect handler research languages

Eff `https://www.eff-lang.org/`

Koka `https://koka-lang.github.io`

Links `https://links-lang.org/`

Effekt `https://effekt-lang.org/`

1. Algebraic effects

1. Algebraic effects

1.1. Introduction

A question

How do effects arise?

i.e., how do we “construct” them in a programming language?

Algebraic effects

An algebraic effect is given by

- operations (i.e. *effect constructors*) with signatures
- a set of axioms

Algebraic effects

An algebraic effect is given by

- operations (i.e. *effect constructors*) with signatures
- a set of axioms

Example: one boolean location

signatures: $put_t : 1$, $put_f : 1$, $get : 2$

- Read $put_b(m)$ as "put b, and continue with m".
- Read $get(m, n)$ as "get the boolean value, continue with m if true, and n otherwise".

Algebraic effects

An algebraic effect is given by

- operations (i.e. *effect constructors*) with signatures
- a set of axioms

Example: one boolean location

signatures: $put_t : 1$, $put_f : 1$, $get : 2$

- Read $put_b(m)$ as "put b, and continue with m".
- Read $get(m, n)$ as "get the boolean value, continue with m if true, and n otherwise".

axioms: the set of axioms

- $get(m, m) = m$
- $get(get(m, m'), get(n, n')) = get(m, m')$
- $put_b(put_{b'}(m)) = put_{b'}(m)$
- $get(put_t(m), put_f(n)) = get(m, n)$
- $put_b(get(m_t, m_f)) = put_b(m_b)$

Interpretation of one boolean location

Interpretation $T_b(X) = B \to (X \times B)$

$$
\begin{aligned}
[\![c]\!] &= \lambda s.\ (c, s) \\
[\![get(m, n)]\!] &= \lambda s.\ \textbf{if } s \textbf{ then } [\![m]\!]s \textbf{ else } [\![n]\!]s \\
[\![put_b(m)]\!] &= \lambda s.\ [\![m]\!]b
\end{aligned}
$$

Interpretation of one boolean location

Interpretation $T_b(X) = B \to (X \times B)$

$$\begin{aligned} [\![c]\!] &= \lambda s.\ (c, s) \\ [\![get(m,n)]\!] &= \lambda s.\ \textbf{if } s \textbf{ then } [\![m]\!]s \textbf{ else } [\![n]\!]s \\ [\![put_b(m)]\!] &= \lambda s.\ [\![m]\!]b \end{aligned}$$

Sound and complete with respect to the equations:

$$m = n \iff [\![m]\!] = [\![n]\!]$$

Interpretation of one boolean location

Interpretation $T_b(X) = B \to (X \times B)$

$$\begin{aligned} [\![c]\!] &= \lambda s.\ (c, s) \\ [\![get(m,n)]\!] &= \lambda s.\ \textbf{if } s \textbf{ then } [\![m]\!]s \textbf{ else } [\![n]\!]s \\ [\![put_b(m)]\!] &= \lambda s.\ [\![m]\!]b \end{aligned}$$

Sound and complete with respect to the equations:

$$m = n \iff [\![m]\!] = [\![n]\!]$$

Example: $put_t(get(m_t, m_f)) = put_t(m_t)$

Interpretation of one boolean location

Interpretation $T_b(X) = B \to (X \times B)$

$$\begin{aligned} [\![c]\!] &= \lambda s.\ (c, s) \\ [\![get(m,n)]\!] &= \lambda s.\ \textbf{if } s \textbf{ then } [\![m]\!]s \textbf{ else } [\![n]\!]s \\ [\![put_b(m)]\!] &= \lambda s.\ [\![m]\!]b \end{aligned}$$

Sound and complete with respect to the equations:

$$m = n \iff [\![m]\!] = [\![n]\!]$$

Example: $put_t(get(m_t, m_f)) = put_t(m_t)$

- $[\![put_t(get(m_t, m_f))]\!] = \lambda s.\ [\![get(m_t, m_f)]\!]t = \lambda s.\ (\lambda s.\ \textbf{if } s \textbf{ then } [\![m_t]\!]s \textbf{ else } [\![m_f]\!]s)t$
  $= \lambda s.\ \textbf{if } t \textbf{ then } [\![m_t]\!]t \textbf{ else } [\![m_f]\!]t = \lambda s.\ [\![m_t]\!]t$

Interpretation of one boolean location

Interpretation $T_b(X) = B \to (X \times B)$

$$\begin{array}{rcl} [\![c]\!] & = & \lambda s.\ (c, s) \\ [\![get(m,n)]\!] & = & \lambda s.\ \textbf{if}\ s\ \textbf{then}\ [\![m]\!]s\ \textbf{else}\ [\![n]\!]s \\ [\![put_b(m)]\!] & = & \lambda s.\ [\![m]\!]b \end{array}$$

Sound and complete with respect to the equations:

$$m = n \iff [\![m]\!] = [\![n]\!]$$

Example: $put_t(get(m_t, m_f)) = put_t(m_t)$

- $[\![put_t(get(m_t, m_f))]\!] = \lambda s.\ [\![get(m_t, m_f)]\!]t = \lambda s.\ (\lambda s.\ \textbf{if}\ s\ \textbf{then}\ [\![m_t]\!]s\ \textbf{else}\ [\![m_f]\!]s)t$
  $= \lambda s.\ \textbf{if}\ t\ \textbf{then}\ [\![m_t]\!]t\ \textbf{else}\ [\![m_f]\!]t = \lambda s.\ [\![m_t]\!]t$
- $[\![put_t(m_t)]\!] = \lambda s.\ [\![m_t]\!]t$

Interpretation of one boolean location

Interpretation $T_b(X) = B \to (X \times B)$

$$\begin{aligned} [\![c]\!] &= \lambda s.\ (c, s) \\ [\![get(m,n)]\!] &= \lambda s.\ \textbf{if } s \textbf{ then } [\![m]\!]s \textbf{ else } [\![n]\!]s \\ [\![put_b(m)]\!] &= \lambda s.\ [\![m]\!]b \end{aligned}$$

Sound and complete with respect to the equations:

$$m = n \iff [\![m]\!] = [\![n]\!]$$

Example: $put_t(get(m_t, m_f)) = put_t(m_t)$

- $[\![put_t(get(m_t, m_f))]\!] = \lambda s.\ [\![get(m_t, m_f)]\!]t = \lambda s.\ (\lambda s.\ \textbf{if } s \textbf{ then } [\![m_t]\!]s \textbf{ else } [\![m_f]\!]s)t$
  $= \lambda s.\ \textbf{if } t \textbf{ then } [\![m_t]\!]t \textbf{ else } [\![m_f]\!]t = \lambda s.\ [\![m_t]\!]t$
- $[\![put_t(m_t)]\!] = \lambda s.\ [\![m_t]\!]t$

Exercise 1: Show other equations hold.

Interpretation of one boolean location

Interpretation $T_b(X) = B \to (X \times B)$

$$\begin{aligned} [\![c]\!] &= \lambda s.\ (c, s) \\ [\![get(m,n)]\!] &= \lambda s.\ \textbf{if } s \textbf{ then } [\![m]\!]s \textbf{ else } [\![n]\!]s \\ [\![put_b(m)]\!] &= \lambda s.\ [\![m]\!]b \end{aligned}$$

Sound and complete with respect to the equations:

$$m = n \iff [\![m]\!] = [\![n]\!]$$

Example: $put_t(get(m_t, m_f)) = put_t(m_t)$

- $[\![put_t(get(m_t, m_f))]\!] = \lambda s.\ [\![get(m_t, m_f)]\!]t = \lambda s.\ (\lambda s.\ \textbf{if } s \textbf{ then } [\![m_t]\!]s \textbf{ else } [\![m_f]\!]s)t$
  $= \lambda s.\ \textbf{if } t \textbf{ then } [\![m_t]\!]t \textbf{ else } [\![m_f]\!]t = \lambda s.\ [\![m_t]\!]t$
- $[\![put_t(m_t)]\!] = \lambda s.\ [\![m_t]\!]t$

Exercise 1: Show other equations hold.

Exercise 2: Show that the get-get equation is redundant.

Interpretation of one boolean location (2)

A different interpretation: $T_{log}(X) = B \to (X \times List\ B)$

$$
\begin{array}{rcl}
[\![c]\!] & = & \lambda s.\ (c, [s]) \\
[\![get(m,n)]\!] & = & \lambda s.\ \textbf{if}\ s\ \textbf{then}\ [\![m]\!]s\ \textbf{else}\ [\![n]\!]s \\
[\![put_b(m)]\!] & = & \lambda s.\ \textbf{let}\ (n,s) \Leftarrow [\![m]\!]b\ \textbf{in}\ (n, b :: s)
\end{array}
$$

Interpretation of one boolean location (2)

A different interpretation: $T_{log}(X) = B \to (X \times List\ B)$

$$\begin{array}{rcl} [\![c]\!] & = & \lambda s.\ (c, [s]) \\ [\![get(m,n)]\!] & = & \lambda s.\ \textbf{if}\ s\ \textbf{then}\ [\![m]\!]s\ \textbf{else}\ [\![n]\!]s \\ [\![put_b(m)]\!] & = & \lambda s.\ \textbf{let}\ (n,s) \Leftarrow [\![m]\!]b\ \textbf{in}\ (n, b :: s) \end{array}$$

Complete with respect to the equations:

$$m = n \Longleftarrow [\![m]\!] = [\![n]\!]$$

Interpretation of one boolean location (2)

A different interpretation: $T_{log}(X) = B \to (X \times List\ B)$

$$\begin{array}{rcl} [\![c]\!] & = & \lambda s.\ (c, [s]) \\ [\![get(m,n)]\!] & = & \lambda s.\ \textbf{if}\ s\ \textbf{then}\ [\![m]\!]s\ \textbf{else}\ [\![n]\!]s \\ [\![put_b(m)]\!] & = & \lambda s.\ \textbf{let}\ (n,s) \Leftarrow [\![m]\!]b\ \textbf{in}\ (n, b :: s) \end{array}$$

Complete with respect to the equations:

$$m = n \Longleftarrow [\![m]\!] = [\![n]\!]$$

But not sound:

$$[\![put_b(put_{b'}(m))]\!] \neq [\![put_{b'}(m)]\!]$$

- the left hand side logs twice, while the right hand side only logs once

Interpretation of one boolean location (3)

Another interpretation: $T_{discard}(X) = B \to X$

$$
\begin{aligned}
[\![c]\!] &= \lambda s.\ c \\
[\![get(m,n)]\!] &= \lambda s.\ \textbf{if } s \textbf{ then } [\![m]\!]s \textbf{ else } [\![n]\!]s \\
[\![put_b(m)]\!] &= \lambda s.\ [\![m]\!]b
\end{aligned}
$$

Interpretation of one boolean location (3)

Another interpretation: $T_{discard}(X) = B \to X$

$$\begin{aligned} [\![c]\!] &= \lambda s.\ c \\ [\![get(m,n)]\!] &= \lambda s.\ \textbf{if } s \textbf{ then } [\![m]\!]s \textbf{ else } [\![n]\!]s \\ [\![put_b(m)]\!] &= \lambda s.\ [\![m]\!]b \end{aligned}$$

Sound with respect to the equations:

$$m = n \Longrightarrow [\![m]\!] = [\![n]\!]$$

Interpretation of one boolean location (3)

Another interpretation: $T_{discard}(X) = B \to X$

$$\begin{array}{rcl} [\![c]\!] & = & \lambda s.\ c \\ [\![get(m,n)]\!] & = & \lambda s.\ \textbf{if}\ s\ \textbf{then}\ [\![m]\!]s\ \textbf{else}\ [\![n]\!]s \\ [\![put_b(m)]\!] & = & \lambda s.\ [\![m]\!]b \end{array}$$

Sound with respect to the equations:

$$m = n \Longrightarrow [\![m]\!] = [\![n]\!]$$

But incomplete:

$$[\![put_b(m)]\!] = [\![m]\!] \text{ but } put_b(m) \neq m$$

Another example: exception

Example: exception

signatures: $raise_e : 0 \qquad (e \in E)$

axioms: none

interpretation: $T_{exc}(X) = X + E$

$$\begin{aligned} [\![c]\!] &= inl(c) \\ [\![raise_e]\!] &= inr(e) \end{aligned}$$

Yet another example: non-determinisim

Example: non-determinisim

signatures: $or : 2$

- Read $or(m, n)$ as "non-deterministically runs m or n".

axioms:

- $or(m, or(n, p)) = or(or(m, n), p)$ *(associativity)*
- $or(m, n) = or(n, m)$ *(commutativity)*
- $or(m, m) = m$ *(absorption)*

interpretation: $T_{ndet}(X) = \mathcal{F}^+(X)$ collection of non-empty finite subsets of X

$$\begin{aligned} [\![c]\!] &= \{c\} \\ [\![or(m, n)]\!] &= [\![m]\!] \cup [\![n]\!] \end{aligned}$$

Remarks

An algebraic effect is given by

- operations (i.e. *effect constructors*) with signatures
- a set of axioms

Remarks

An algebraic effect is given by

- operations (i.e. *effect constructors*) with signatures
- a set of axioms

Is a set of axioms the right set of axioms?

- An equational theory is *equationally inconsistent* if it proves $x = y$.
- An equational theory is *Hilbert-Post complete* if adding an unprovable equation makes it equationally inconsistent.

Remarks

An algebraic effect is given by

- operations (i.e. *effect constructors*) with signatures
- a set of axioms

Is a set of axioms the right set of axioms?

- An equational theory is *equationally inconsistent* if it proves $x = y$.
- An equational theory is *Hilbert-Post complete* if adding an unprovable equation makes it equationally inconsistent.

Different interpretations are useful in practice, so sometimes people use effects *without* equations.

Algebraicity

From the literature:

Fix a finitary equational axiomatic theory Ax. Then for any set X and operation symbol $\mathrm{op} : n$ we have the function:

$$T_{\mathrm{Ax}}(X)^n \xrightarrow{\mathrm{op}_{F_{\mathrm{Ax}}(X)}} T_{\mathrm{Ax}}(X)$$

Further for any function $f : X \to T_{\mathrm{Ax}}(Y)$, $f^\dagger$ is a homomorphism:

$$\begin{array}{ccc} T_{\mathrm{Ax}}(X)^n & \xrightarrow{\mathrm{op}_{F_{\mathrm{Ax}}(X)}} & T_{\mathrm{Ax}}(X) \\ \downarrow{\scriptstyle (f^\dagger)^n} & = & \downarrow{\scriptstyle f^\dagger} \\ T_{\mathrm{Ax}}(Y)^n & \xrightarrow[\mathrm{op}_{F_{\mathrm{Ax}}(Y)}]{} & T_{\mathrm{Ax}}(Y) \end{array}$$

We call such a polymorphic family of functions $T_{\mathrm{Ax}}(X)^n \xrightarrow{\varphi_X} T_{\mathrm{Ax}}(X)$ algebraic.

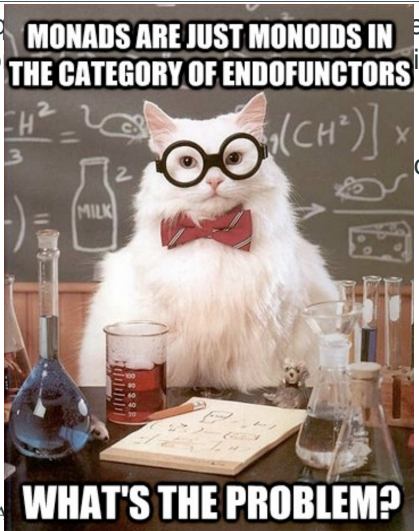
Algebraicity

From the literature:

Fix a finitary eq … en for any set X and operatio … ion:

Further for any … omomorphism:

We call such a …
$T_{\mathrm{Ax}}(X)^n \xrightarrow{\varphi_X} T_{\mathrm{A}}$



Programming counterpart of being algebraic

Evaluation context $E ::= \Box \mid E\ n \mid (\lambda x.\ m)\ E$

For any operation $op : n$, we have:

$$E[op(m_1, \ldots, m_n)] = op(E[m_1], \ldots, E[m_n])$$

Programming counterpart of being algebraic

Evaluation context $E ::= \square \mid E\ n \mid (\lambda x.\ m)\ E$

For any operation $op : n$, we have:

$$E[op(m_1, \ldots, m_n)] = op(E[m_1], \ldots, E[m_n])$$

Example

- $raise_e()\ n = raise_e()$
- $(\lambda x.\ m)\ raise_e() = raise_e()$
- $or(m, n)\ p = or(m\ p, n\ p)$
- $(\lambda x.\ p)\ or(m, n) = or((\lambda x.\ p)\ m, (\lambda x.\ p)\ n)$

Summary of algebraic effects

An algebraic effect is given by

- operations (i.e. *effect constructors*) with signatures
- a set of axioms

Algebracity

- operations commute with evaluation contexts

Summary of algebraic effects

An algebraic effect is given by

- operations (i.e. *effect constructors*) with signatures
- a set of axioms

Algebracity

- operations commute with evaluation contexts

Next:

- computational trees and free monads
- generic effects

1. Algebraic effects

1.2. Computational trees and free monad

An example program

Recall the boolean location with signature

$$put_t : 1, put_f : 1, get : 2$$

An example program

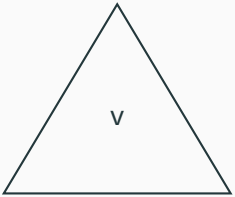
Recall the boolean location with signature

$$put_t : 1, put_f : 1, get : 2$$

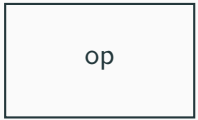
Consider a term $toggle = get(put_f(t), put_t(f))$

Question: What is this term doing?

Computational trees



Values

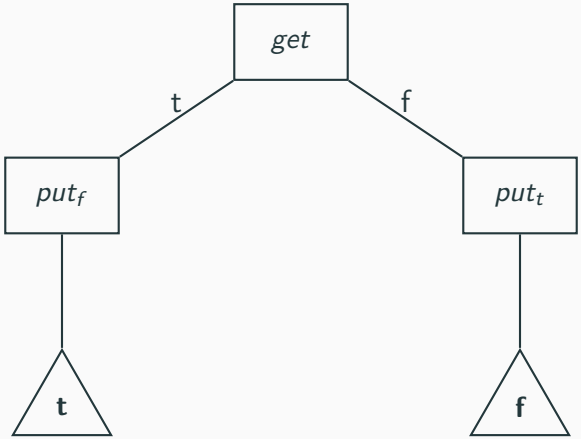


Operations

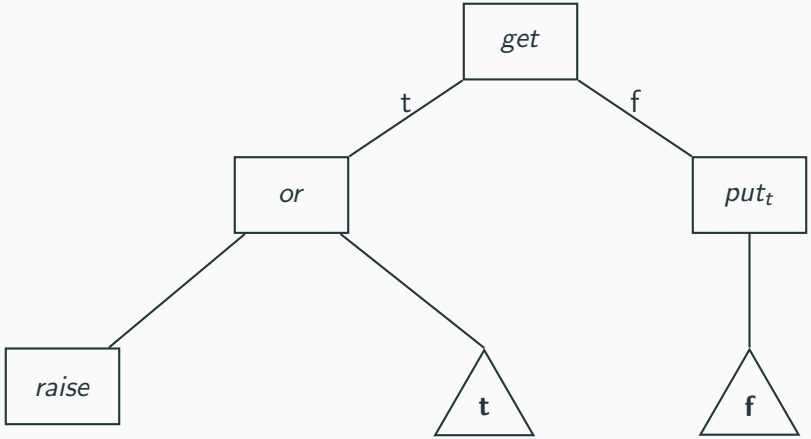
- A computational tree is a tree whose leaves are values, and internal nodes are operations.

Computational trees

$toggle = get(put_f(t), put_t(f))$

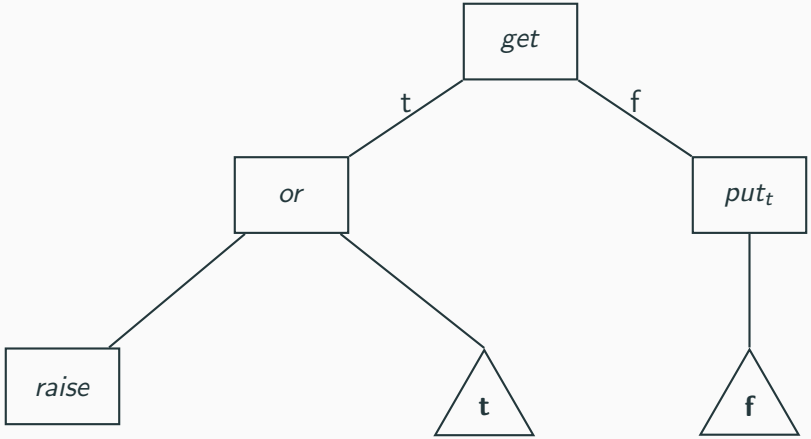


Computational trees



Question: what is this program doing?

Computational trees



Question: what is this program doing?

$$get(or(raise, t), put_t(f))$$

Computational trees

A computational tree is a tree whose leaves are values, and internal nodes are operations.

Computational trees as **free monads**

```
data Free f a = Pure a | Free (f (Free f a))
```

- `Pure a`: Represents a pure value, effectively the **return** of a monad
- `Free (f (Free f a))`: Represents an operation that produces another `Free f a` computation.

The **bind** performs substitution at the leaves

```
        return c >>= r  =  r c
op(m1, ..., mn) >>= r  =  op (m1 >>=r, ..., mn >== r)
```