

# Polymorphic algebraic effects: theoretical properties and implementation

Wiktor Kuchta

# Chapter 1

## Introduction

A programming language needs to be more than just a lambda calculus, capable only of functional abstraction and evaluation of expressions. Programs need to have an effect on the outside world and, thinking more locally, in our programs we would like to have fragments which do not merely reduce to a value in isolation, but affect the execution of surrounding code in interesting ways. This is what we call computational effects, the typical examples of which are: input/output, mutable state, exceptions, nondeterminism, and coroutines.

Today, we are at the mercy of programming language designers to include the effects we would like to use and make sure that they all interact with each other well. In recent years, however, a new approach, called *algebraic effects*, has emerged. Algebraic effects allow programmers to express all the usual computational effects as library code and use them in the usual direct style (as opposed to eg. monadic encodings, which do not always play well with the rest of the language and do not compose easily). These implementations of effects are instances of one concept and hence interact with each other predictably.

More concretely, algebraic effects may be thought of as resumable exceptions. We can *perform* an *operation*, just as we can raise an exception in a typical high-level programming language. The operation is then handled by the nearest enclosing *handler* for the specific operation. The handler receives the value given at perform-point and also, unlike normal exception handlers, a *continuation*, a first-class functional value representing the rest of the code to execute inside the handler from the perform-point. By calling the continuation with a value we can resume at the perform-point, as if performing the operation evaluated to the value. But more interestingly, we can resume multiple times, or never, or store the continuation for later use.

This access to the continuation, while powerful, might also be a recipe for disaster. That is why languages with algebraic effects typically have *type-and-effect systems*, not only tracking the type of value an expression might evaluate to, but also what kind of effects it may perform on the way.

This work will study a particular formal language equipped with algebraic effects.

Although most results in the domain of programming languages depend crucially on the exact calculus and type system used, we hope that the results, ideas, and techniques may apply elsewhere. This should not seem far-fetched, as the paper which introduced the language [2] is exactly about interexpressibility with other systems.

Possibly the most interesting result of this work is a potentially novel use of coinduction in the method of logical relations. It allowed us to get rid of step-indexing in the definition of the logical relation and ultimately prove termination of the language. This may seem surprising, as coinduction is connotated primarily with infinite processes.

[todo other contributions]

I would like to thank Dariusz Biernacki, Filip Sieczkowski, and Piotr Polesiuk for their help with this work.

# Chapter 2

## The language

We will study the deep handler calculus and type-and-effect system formulated in [2]. It is a refreshingly minimal language—the call-by-value lambda calculus with a few extensions to be able to express the essence of algebraic effects. There is only one unnamed universal operation, performed **do**  $v$ . To be able to simulate calculi with named effects (and more), the *lift* operator, written  $[e]$ , is introduced. When operations are performed inside the expression  $e$  enclosed by *lift*, the nearest handler will be skipped and the operation will be handled by the next one instead. Naturally, the operator composes, so multiple enclosing lifts means multiple handlers skipped. Interestingly, the effect-tracking system is structural, we do not have any concept of predefined or user-defined (named) signatures of effects. Finally, the language includes polymorphic expressions and polymorphic operations.

$$\begin{aligned} \text{TVar} &\ni \alpha, \beta, \dots \\ \text{Var} &\ni f, r, x, y, \dots \\ \text{Kind} &\ni \kappa ::= T \mid E \mid R \\ \text{Types} &\ni \sigma, \tau, \varepsilon, \rho ::= \alpha \mid \tau \rightarrow_\rho \tau \mid \forall \alpha :: \kappa. \tau \mid \iota \mid \varepsilon \cdot \rho \\ \text{Val} &\ni v ::= x \mid \lambda x. t \\ \text{Exp} &\ni e ::= v \mid e \ e \mid [e] \mid \text{do } v \mid \text{handle } e \{x, r. e; x. e\} \\ \text{ECont} &\ni K ::= \square \mid K \ e \mid v \ K \mid [K] \mid \text{handle } K \{x, r. e; x. e\} \end{aligned}$$

# Chapter 3

## The logical relation

The logical relation is adapted from [1] with a few changes: our relation is unary, we additionally have polymorphic effects, we do not have type lambdas.

$$\begin{aligned}\llbracket \mathbf{T} \rrbracket &= \mathcal{P}(\mathbf{Val}) = \mathbf{Type} \\ \llbracket \mathbf{E} \rrbracket &= \llbracket \mathbf{R} \rrbracket = \mathcal{P}(\mathbf{Exp} \times \mathbb{N} \times \mathbf{Type}) = \mathbf{Eff}\end{aligned}$$

$$\mathbf{Obs} = \{e \mid \exists v. e \rightarrow^* v\}$$

$$\begin{aligned}\mathcal{E}[\tau/\rho]_\eta &= \{t \mid \forall K \in \mathcal{K}[\tau/\rho]_\eta. K[t] \in \mathbf{Obs}\} \\ \mathcal{K}[\tau/\rho]_\eta &= \{K \mid \forall v \in \llbracket \tau \rrbracket_\eta. K[v] \in \mathbf{Obs} \wedge \forall s \in \mathcal{S}[\tau/\rho]_\eta. K[s] \in \mathbf{Obs}\} \\ \mathcal{S}[\tau/\rho]_\eta &= \{K[\mathbf{do} \ v] \mid \exists n, \mu. (v, n, \mu) \in \llbracket \rho \rrbracket_\eta \wedge n\text{-free}(K) \wedge \forall e \in \mu. K[e] \in \mathcal{E}[\tau/\rho]_\eta\}\end{aligned}$$

$$\begin{aligned}\llbracket \tau_1 \rightarrow_\rho \tau_2 \rrbracket_\eta &= \{\lambda x. t \mid \forall v \in \llbracket \tau_1 \rrbracket_\eta. [x \mapsto v]t \in \mathcal{E}[\tau_2/\rho]_\eta\} \\ \llbracket \forall \alpha :: \kappa. \tau \rrbracket_\eta &= \{v \mid \forall \mu \in \llbracket \kappa \rrbracket_\eta. v \in \llbracket \tau \rrbracket_{[\alpha \mapsto \mu]_\eta}\} \\ \llbracket \alpha \rrbracket_\eta &= \{v \mid \forall K. (\forall u \in \eta(\alpha). K[u] \in \mathbf{Obs}) \implies K[v] \in \mathbf{Obs}\}\end{aligned}$$

$$\begin{aligned}\llbracket \tau_1 \Rightarrow \tau_2 \rrbracket_\eta &= \{(v, 0, \llbracket \tau_2 \rrbracket_\eta) \mid v \in \llbracket \tau_1 \rrbracket_\eta\} \\ \llbracket \forall \alpha :: \kappa. \varepsilon \rrbracket_\eta &= \{t \mid \exists \mu \in \llbracket \kappa \rrbracket_\eta. t \in \llbracket \varepsilon \rrbracket_{[\alpha \mapsto \mu]_\eta}\} \\ \llbracket \varepsilon \cdot \rho \rrbracket_\eta &= \llbracket \varepsilon \rrbracket_\eta \cup \{(v, n+1, \mu) \mid (v, n, \mu) \in \llbracket \rho \rrbracket_\eta\}\end{aligned}$$

$$\begin{aligned}\mathcal{E}[\tau/\rho]_\eta(\mathbf{X}) &= \{t \mid \forall K \in \mathcal{K}[\tau/\rho]_\eta(\mathbf{X}). K[t] \in \mathbf{Obs}\} \\ \mathcal{K}[\tau/\rho]_\eta(\mathbf{X}) &= \{K \mid \forall v \in \llbracket \tau \rrbracket_\eta. K[v] \in \mathbf{Obs} \wedge \forall s \in \mathcal{S}[\tau/\rho]_\eta(\mathbf{X}). K[s] \in \mathbf{Obs}\} \\ \mathcal{S}[\tau/\rho]_\eta(\mathbf{X}) &= \{K[\mathbf{do} \ v] \mid \exists n, \mu. (v, n, \mu) \in \llbracket \rho \rrbracket_\eta \wedge n\text{-free}(K) \wedge \forall e \in \mu. K[e] \in \mathbf{X}\}\end{aligned}$$

# Bibliography

- [1] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Handle with care: relational interpretation of algebraic effects and handlers. *Proc. ACM Program. Lang.*, 2(POPL):8:1–8:30, 2018.
- [2] Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Typed equivalence of effect handlers and delimited control. In Herman Geuvers, editor, *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany*, volume 131 of *LIPIcs*, pages 30:1–30:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.