

A **CASE STUDY** IN
MATHEMATICALLY INSPIRED
LANGUAGE **CONSTRUCTS**

Matija Pretnar



FMF

UNIVERSITY OF LJUBLJANA
Faculty of Mathematics and Physics

Xavier Leroy held a seminar on **control structures**



COLLÈGE
DE FRANCE
— 1530 —



08 → 14
FÉV → MAR
2024 → 2024

■ SÉMINAIRE

Structures de contrôle : de « *goto* » aux effets algébriques

Partager ▾

Du jeudi 8 février au
jeudi 14 mars 2024

Voir aussi :

- [Cours associé](#)
- [Xavier Leroy](#)



How would the **seminar series** in 75 years be titled?



COLLÈGE
DE FRANCE
— 1530 —



05 → 12
FÉV → MAR
2099 → 2099

■ SÉMINAIRE

Structures de contrôle : des effets algébriques au
« ??? »

Partager ▾

Du jeudi 5 février au
jeudi 12 mars 2099

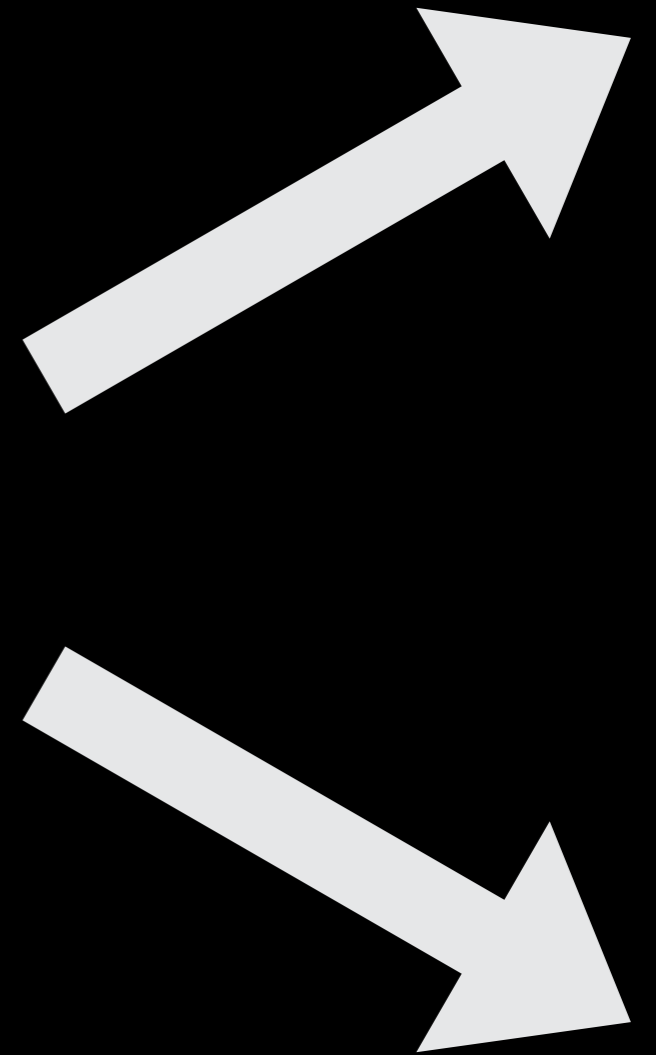
Voir aussi :

- [Cours associé](#)
- [Xavier Leroy](#)



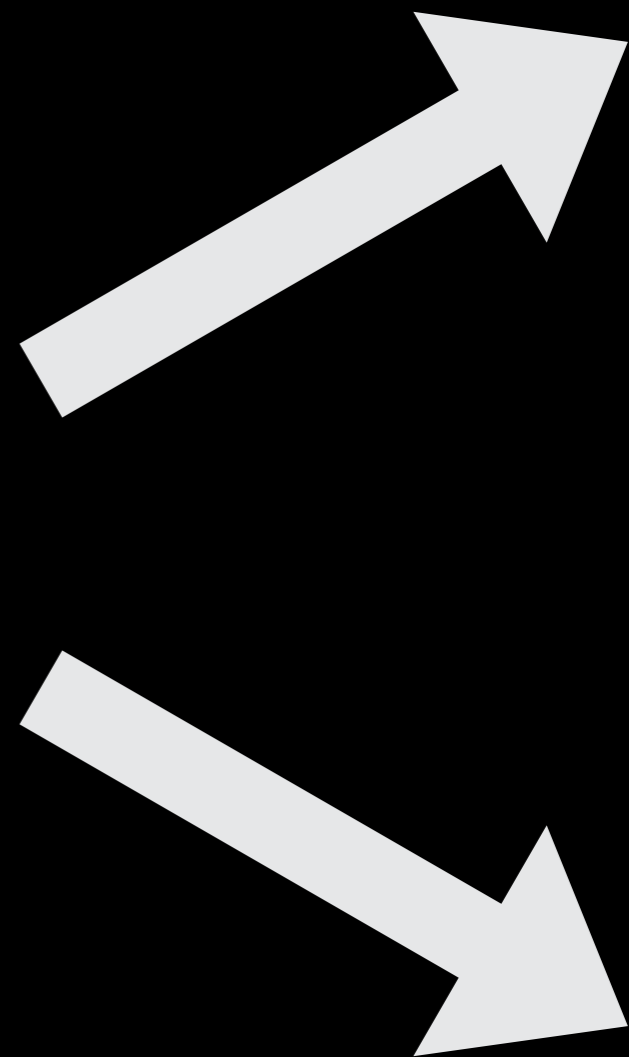


HANDLERS





HANDLERS



Moggi recognised **monads** in the **semantics** of effectful computations

Computational lambda-calculus and monads

Eugenio Moggi*
Lab. for Found. of Comp. Sci.
University of Edinburgh
EH9 3JZ Edinburgh, UK
On leave from Univ. di Pisa

Abstract

The λ -calculus is considered a useful mathematical tool in the study of programming languages. However, if one uses $\beta\eta$ -conversion to prove equivalence of programs, then a gross simplification¹ is introduced. We give a calculus based on a categorical semantics for *computations*, which provides a correct basis for proving equivalence of programs, independent from any specific computational model.

Introduction

This paper is about logics for reasoning about programs, in particular for proving equivalence of programs. Following a consolidated tradition in theoretical computer science we identify programs with the closed λ -terms, possibly containing extra constants, corresponding to some features of the programming language under consideration. There are three approaches to proving equivalence of programs:

- The **operational** approach starts from an **operational semantics**, e.g. a partial function mapping every program (i.e. closed term) to its resulting value (if any), which induces a congruence relation on open terms called **operational equivalence** (see e.g. [10]). Then the problem is to prove that two terms are operationally equivalent.
- The **denotational** approach gives an interpretation of the (programming) language in a mathematical structure, the **intended model**. Then the problem is to prove that two terms denote the same object in the intended model.

*Research partially supported by EEC Joint Collaboration Contract # ST2J-0374-C(EDB).

¹Programs are identified with total functions from *values* to *values*.

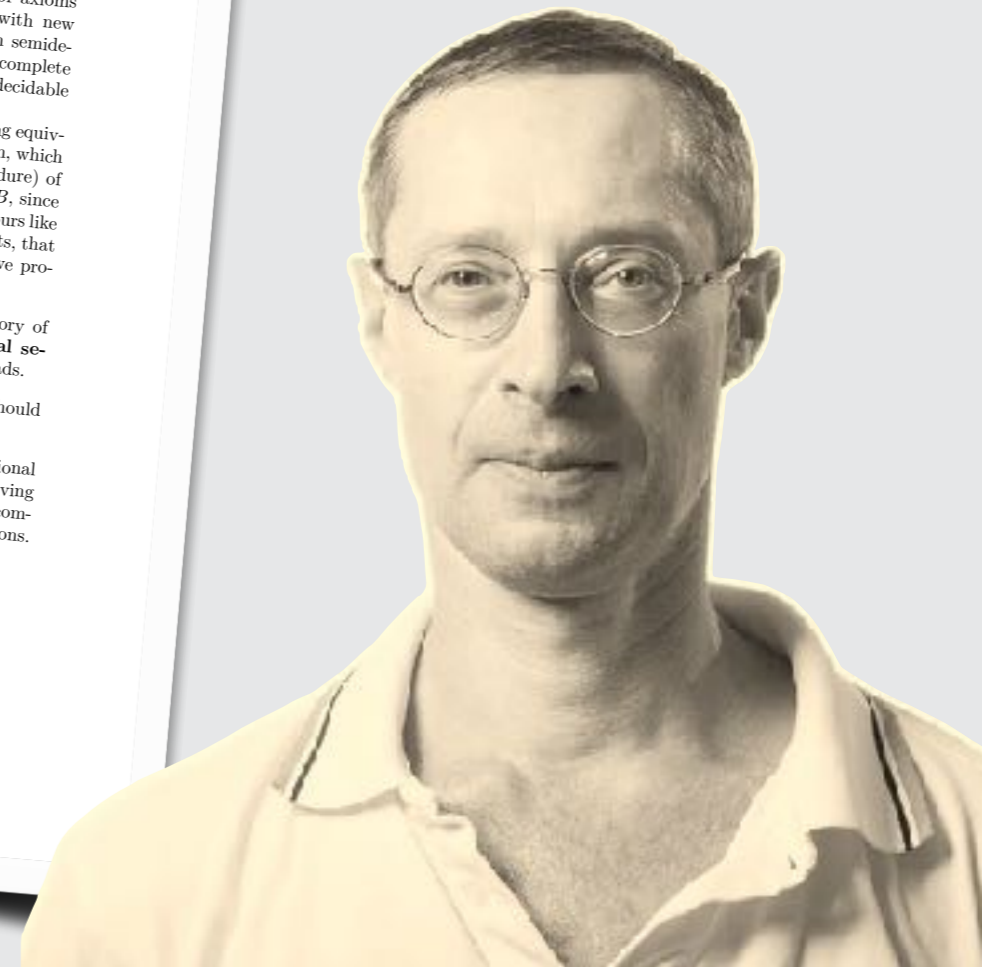
- The **logical** approach gives a class of **possible models** for the language. Then the problem is to prove that two terms denotes the same object in all possible models.

The operational and denotational approaches give only a theory (the operational equivalence \approx and the set Th of formulas valid in the intended model respectively), and they (especially the operational approach) deal with programming languages on a rather case-by-case basis. On the other hand, the logical approach gives a consequence relation $\vdash (Ax \vdash A$ iff the formula A is true in all models of the set of formulas Ax), which can deal with different programming languages (e.g. functional, imperative, non-deterministic) in a rather *uniform* way, by simply changing the set of axioms Ax , and possibly extending the language with new constants. Moreover, the relation \vdash is often semidecidable, so it is possible to give a sound and complete formal system for it, while Th and \approx are semidecidable only in oversimplified cases.

We do not take as a starting point for proving equivalence of programs the theory of $\beta\eta$ -conversion, which identifies the denotation of a program (procedure) of type $A \rightarrow B$ with a total function from A to B , since this identification wipes out completely behaviours like non-termination, non-determinism or side-effects, that can be exhibited by real programs. Instead, we proceed as follows:

1. We take category theory as a general theory of functions and develop on top a **categorical semantics of computations** based on monads.
2. We consider how the categorical semantics should be extended to interpret λ -calculus.

At the end we get a formal system, the computational lambda-calculus (λ_c -calculus for short), for proving **equivalence** of programs, which is sound and complete w.r.t. the categorical semantics of computations.



The initial specification was taken to be **mathematically** more natural

Example 1.3 Non-deterministic computations:

- $T(-)$ is the power set $\mathcal{P}(A)$ and
- $\eta_A(a)$ is $\{a\}$
- $\mu_A(X)$ is $\bigcup X$

There is an alternative description of a monad (see [7]), which is easier to justify computationally.

Definition 1.2 A Kleisli triple over \mathcal{C} is a triple $(T, \eta, -^*)$, where $T: \text{Obj}(\mathcal{C}) \rightarrow \text{Obj}(\mathcal{C})$, $\eta_A: A \rightarrow TA$, $f^*: TA \rightarrow TB$ for $f: A \rightarrow TB$ and the following equations hold:

- $\eta_A^* = \text{id}_{TA}$
- $\eta_A; f^* = f$
- $f^*; g^* = (f; g^*)^*$

Every Kleisli triple $(T, \eta, -^*)$ corresponds to a monad (T, η, μ) where $T(f: A \rightarrow B) = (f; \eta_B)^*$ and $\mu_A = \text{id}_{TA}^*$.

S is a
computation
together

and then returns
the pair
 $\lambda = f's'$.

In his subsequent work, a **computationally natural** approach was taken

DEFINITION 1.2 (Manes, 1976). A Kleisli triple over a category \mathcal{C} is a triple $(T, \eta, -^*)$, where $T: \text{Obj}(\mathcal{C}) \rightarrow \text{Obj}(\mathcal{C})$, $\eta_A: A \rightarrow TA$ for $A \in \text{Obj}(\mathcal{C})$, $f^*: TA \rightarrow TB$ for $f: A \rightarrow TB$ and the following equations hold:

- $\eta_A^* = \text{id}_{TA}$
- $\eta_A; f^* = f$ for $f: A \rightarrow TB$
- $f^*; g^* = (f; g^*)^*$ for $f: A \rightarrow TB$ and $g: B \rightarrow TC$.

EXAMPLE 1.4. We go through the notions of computation given in Example 1.1 and show that they are indeed part of suitable Kleisli triples.

- **partiality** $TA = A_{\perp} (= A + \{\perp\})$
 η_A is the inclusion of A into A_{\perp}
if $f: A \rightarrow TB$, then $f^*(\perp) = \perp$ and $f^*(a) = f(a)$ (when $a \in A$)
- **nondeterminism** $TA = \mathcal{P}_{\text{fin}}(A)$
 η_A is the singleton map $a \mapsto \{a\}$
if $f: A \rightarrow TB$ and $c \in TA$, then $f^*(c) = \bigcup_{x \in c} f(x)$
- **side-effects** $TA = (A \times S)^S$
 η_A is the map $a \mapsto (\lambda s: S. \langle a, s \rangle)$
if $f: A \rightarrow TB$ and $c \in TA$, then $f^*(c) = \lambda s: S. (\text{let } \langle a, s' \rangle = c(s) \text{ in } f(a)(s'))$

Wadler **transformed** the semantic notion into a **programming construct**

7.1 Parsers

The monad of parsers is given by

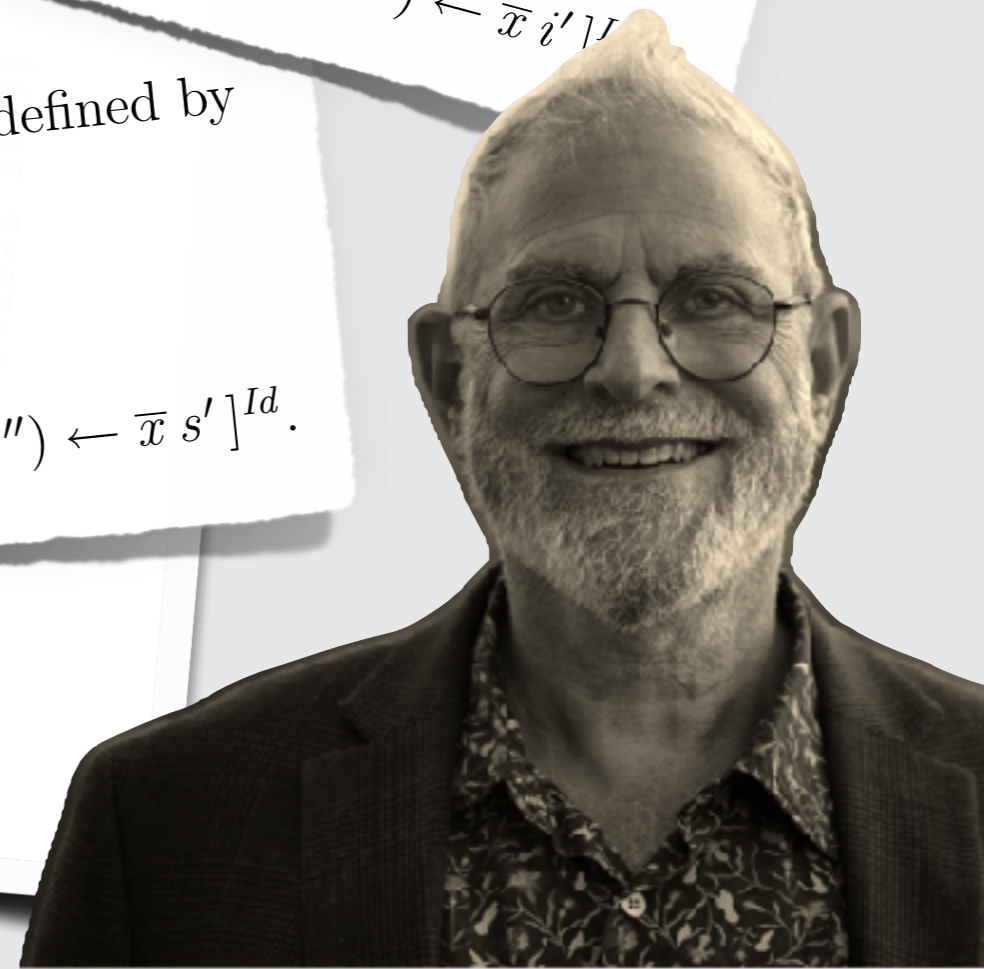
$$\begin{aligned} \text{type Parse } x &= \text{String} \rightarrow \text{List}(x, \text{String}) \\ \text{map}^{\text{Parse}} f \bar{x} &= \lambda i \rightarrow [(f\ x, i') \mid (x, i') \leftarrow \bar{x}\ i] \text{List} \\ \text{unit}^{\text{Parse}} x &= \lambda i \rightarrow [(x, i)] \text{List} \\ \text{join}^{\text{Parse}} \bar{\bar{x}} &= \lambda i \rightarrow [(x, i'') \mid (\bar{x}, i') \leftarrow \bar{\bar{x}}\ i, (x, i'') \leftarrow \bar{x}\ i'] \text{List} \end{aligned}$$

1 Introduction

Is there a way to combine the indulgent
Impure, strict function
[RC86]

4.1 State transformers

Fix a type S of states. The monad of state transformers ST is defined by

$$\begin{aligned} \text{type } ST\ x &= S \rightarrow (x, S) \\ \text{map}^{ST} f \bar{x} &= \lambda s \rightarrow [(f\ x, s') \mid (x, s') \leftarrow \bar{x}\ s] \text{Id} \\ \text{unit}^{ST} x &= \lambda s \rightarrow (x, s) \\ \text{join}^{ST} \bar{\bar{x}} &= \lambda s \rightarrow [(x, s'') \mid (\bar{x}, s') \leftarrow \bar{\bar{x}}\ s, (x, s'') \leftarrow \bar{x}\ s'] \text{Id}. \end{aligned}$$


An effect is specified with a **monad**

monad

$$TX = \mathcal{P}X$$

$$\eta(x) = \{x\}$$

$$c \gg k = \bigcup_{x \in c} k(x)$$

An effect is specified with a monad and **additional operations**

monad

$$TX = \mathcal{P}X$$

$$\eta(x) = \{x\}$$

$$c \gg= k = \bigcup_{x \in c} k(x)$$

effect-specific operations

$$\text{fail} : TX$$

$$\text{fail} = \{\}$$

$$\text{choose} : TX \times TX \rightarrow TX$$

$$\text{choose}(c_1, c_2) = c_1 \cup c_2$$

Plotkin & Power recognised **algebraic theories as sources** of effects

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: non-determinism and probabilistic nondeterminism.

1 Introduction

Moggi introduced the idea of a general account of computational effects, proposing encapsulating them via monads $T : \mathbf{C} \rightarrow \mathbf{C}$; the main idea is that $T(x)$ is the type of computations of elements of x . He also presented the computational λ -calculus λ_c as a core call-by-value functional programming language for effects [21]. The effects themselves are obtained by adding appropriate operations, specified by a signature Σ . Moggi introduced the consideration of these operations in the context of his metalanguage $ML(\Sigma)$ whose purpose is to give the semantics of programming languages [22, 23], but which is not itself thought of as a programming language.

In our view any complete account of computation should incorporate a treatment of operational semantics; this has been lacking for the monadic approach. In this paper we give such a treatment in the case of algebraic effects. Operations are given by a single-sorted algebraic signature Σ . An n -ary operation f is taken to denote a family of morphisms

$$f_x : T(x)^n \rightarrow T(x)$$

parametrically natural with respect to morphisms in \mathbf{C} . T is then said to *support* the family f_x . (In [22] only morphisms in \mathbf{C} is considered; we use the stronger

* This work has been done with the support of EPSRC grant GR/R014567/1: Algebraic Theories of Programming Languages: Syntax and Semantics.



Exception **handling failed** to be algebraic

exceptions

state

choice

I/O

probability

operations

fail **try**


get set

choose

read write

flip

*not
algebraic*



Exception handling indicated a **different nature**

On the other hand, for example, the exceptions monad does not support its exception handling operation: only the weaker naturality holds there. This monad is a free algebra functor for an equational theory, viz the one that has a constant for each exception and no equations; however the exception handling operation is not definable: only the exception raising operations are. Other standard monads present further difficulties. So while our account of operational semantics is quite general, it certainly does not cover all cases; it remains to be seen if it can be further extended.

Of the various operations, **handle** is of a different computational character and, although natural, it is not algebraic. Andrzej Filinski (personal communication) describes **handle** as a *deconstructor*, whereas the other operations are *constructors* (of effects). In this paper, we make the notion of constructor precise by identifying it with the notion of *algebraic* operation.

Mathematics was already suggesting **unrevealed constructs**

constructors **deconstructors**

exceptions

`fail`

`try`

state

`get set`

choice

`choose`

I/O

`read write`

probability

`flip`



In fact, a suitable interpretation was **there all along**

Definition 2.4 A model of a countable Lawvere theory L in any category C with countable products is a countable-product preserving functor $M : L \rightarrow C$.

so Mn must be the product of n copies of $M1$. So, to give a model M is equivalent to giving a set $X = M1$ together with, for each map of the form $f : m \rightarrow 1$ in L , a function from X^m to X , subject to the equations given by the composition and product structure of L . This analysis routinely extends to any category C with

The monad generated by L_E is $T_E = - + E$. More generally, if C is a category with countable powers and countable coproducts, $Mod(L_E, C)$ is equivalent to the category of algebras for the monad $- + \underline{E}$, where \underline{E} for the E -fold coproduct, i.e., $\coprod_E 1$.



Exception handlers are **homomorphisms** and they **generalise to other effects**

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 of the free model. This general construct can be used to describe previously unrelated concepts from both theory and practice.

1 Introduction

In seminal work, Moggi proposed a uniform representation of computational effects by monads [1–3]. The computations that return values from a set X are represented by elements of TX , for a suitable monad T . Examples include exceptions, nondeterminism, interactive input/output, concurrency, state, time, continuations, and combinations thereof. Plotkin and Power later proposed to focus on *algebraic* effects, that is effects that allow a representation by operations and equations [4–6]; the operations give rise to the effects at hand. All of the effects mentioned above are algebraic, with the notable exception of continuations [7], which have to be treated differently (see [8] for initial ideas).

In the algebraic approach the arguments of an operation represent possible computations after an occurrence of an effect. For example, using a binary choice operation $\text{or}: 2$, a nondeterministically chosen boolean is represented by the term $\text{or}(\text{return true}, \text{return false}): F\text{bool}$, where $F\sigma$ stands for the type of computations that return values of type σ . The equations of the theory, for example the ones stating that or is a semi-lattice operation, generate the free-model functor, which is exactly the monad proposed by Moggi to model the corresponding effect [9] (modulo the forgetful functor) and which is used to interpret the type $F\sigma$. The operations are then interpreted by the model structure. When viewed as a family of functions parametric in X , e.g., $\text{or}_X: TX^2 \rightarrow TX$, one obtains a so-called

* Supported by EPSRC grant GR/586371/01 and a Royal Society-Wolfson Award Fellowship.

** Supported by EPSRC grant GR/586371/01.



The Programming Languages Zoo

A potpourri of programming languages

> [home](#)

About the zoo

The Programming Languages Zoo is a collection of miniature programming languages which demonstrates various concepts and techniques used in programming language design and implementation. It is a good starting point for those who would like to implement their own programming language, or just learn how it is done.

The following features are demonstrated:

- >> functional, declarative, object-oriented, and procedural languages
- >> source code parsing with a parser generator
- >> keep track of source code positions
- >> pretty-printing of values
- >> interactive shell (REPL) and non-interactive file processing
- >> untyped, statically and dynamically typed languages
- >> type checking and type inference
- >> subtyping, parametric polymorphism, and other kinds of type systems
- >> eager and lazy evaluation strategies
- >> recursive definitions
- >> exceptions
- >> interpreters and compilers
- >> abstract machine

Installation

See the [installation & compilation instructions](#).

We wanted to **do the same for handlers** as Wadler did for monads

Moggi

*Computational
lambda-calculus
and monads*

1989

Plotkin & P.

*Handlers of
algebraic effects*

2009

Wadler

*Comprehending
monads*

1991



Initial version of Eff had a **Python-like syntax** and was **untyped**

Mathematics and Computation

A blog about mathematics for computers

```
type Store a:  
  operation lookup: () -> a  
  operation update: a -> ()
```

```
x = new Store  
x.update 10  
a = x.lookup ()  
x.update (a + 5)  
x.lookup ()
```

Installation

If you have [Mercurial](#) installed (type hg at command prompt to find out) you can get eff like this:

```
$ hg clone http://hg.andrej.com/eff/ eff
```

Otherwise, you may also download the latest source as a [.zip](#) or [.tar.gz](#), or [visit the repository with your browser](#) for other versions. Eff

```
ref loc =  
  handler:  
    return x:  
      lambda s: x  
    operation loc.update s' k:  
      lambda s: (k ()) s' k:  
    operation loc.lookup () k:  
      lambda s: (k s) s
```

```
ref : Store -> (A => (Int -> A))
```

Mathematics and Computation

A blog

Posts

```
type 'a ref = effect  
  operation get: unit -> 'a  
  operation set: 'a -> unit  
end
```

```
let state r x = handler  
  | r#get () k -> (fun s -> k s s)  
  | r#set s' k -> (fun s -> k () s')  
  val y -> (fun s -> (y, s))  
  finally f -> f x
```

- Eff now clearly separates three basic concepts: effect types, effect instances, and handlers.
- How eff works is explained in our paper on [Programming with Algebraic Effects and Handlers](#).
- We moved the [source code to GitHub](#), so go ahead and fork it!

Comments



Dan Doel

02 April 2012 at 22:05

The new version of Eff also had an accompanying **research paper**

Moggi

Computational
lambda-calculus
and monads

Plotkin & P.

Handlers of
algebraic effects

Philip Wadler once opined [21] that monads as a programming concept would not have been discovered without their category-theoretic counterparts, but once they were, programmers could live in blissful ignorance of their origin. Because the same holds for algebraic effects and handlers, we streamlined the paper for the benefit of programmers, trusting that connoisseurs will recognize the connections with the underlying mathematical theory.

The paper is organized as follows. Section 1 describes the syntax of Eff, Section 2 informally introduces constructs specific to Eff, Section 3 is devoted to type checking, in Section 4 we give a domain-theoretic semantics of Eff, and in Section 5 we briefly discuss our prototype implementation. The examples in Section 6 demonstrate how effects and handlers can be

wadler

Comprehending
monads

1991

Programming with algebraic effects and handlers

Andrey Raiser, Matija Pretnar*

Faculty of Mathematics and Physics, University of Ljubljana, Slovenia

ARTICLE INFO

Article history:
Received 20 February 2012
Revised in final form 22 November 2013
Accepted 22 February 2014
Available online 20 March 2014

ABSTRACT

Eff is a programming language based on the algebraic approach to computational effects, in which effects are viewed as algebraic operations and effect handlers as homomorphisms. In this paper, we define new computational effects and effect handlers as homomorphisms in a novel way. We give a domain-theoretic semantics of Eff and discuss a prototype implementation based on it. Through examples we demonstrate how the standard effects are handled in Eff, and how Eff supports programming constructs that use various forms of delimited computation, such as backtracking, break, first-class selectors, functionals, cooperative multi-threading, and others.

© 2014 Elsevier Inc. All rights reserved.

0. Introduction

Eff is a programming language based on the algebraic approach to effects, in which computational effects are modelled as operations of a suitably chosen algebraic theory [17]. Concrete computational effects such as input, output, state, effect handlers, and many others. These are modelled as homomorphisms between monads, from which the algebraic theory gives rise to a monad [1, 17]. Although the operations cannot be reconstructed from the effect handlers, the algebraic approach offers new ways of programming. An experiment in the design of a programming language category-theoretic constructs, but each they were, programmers could live in blissful ignorance of their origin. Because the same holds for algebraic effects and handlers, we streamlined the paper for the benefit of programmers, trusting that connoisseurs will recognize the connections with the underlying mathematical theory.

* Corresponding author.
E-mail addresses: andrey.raisar@upf.edu (A. Raiser), matija.pretnar@upf.edu (M. Pretnar).

http://dx.doi.org/10.1016/j.scs.2014.02.005
2352-2466/© 2014 Elsevier Inc. All rights reserved.



Moving from **mathematics to programming** gave **extra flexibility**

Plotkin & P.

$$\frac{\mathbf{x}_p:\sigma, \mathbf{x}:\beta; \mathbf{z}_p:\chi, (z_i:(\alpha_i) \rightarrow \chi)_{i=1}^n \vdash h_{op}:\chi \quad (op:\beta; \alpha_1, \dots, \alpha_n \in \Sigma_{\text{eff}})}{\vdash (\mathbf{x}_p:\sigma; \mathbf{z}_p:\chi). \{op_x(z) \mapsto h_{op}\}_{op \in \Sigma_{\text{eff}}} : (\sigma; \chi) \rightarrow \chi \text{ handler}}$$
$$e ::= x \mid n \mid b \mid \text{true} \mid \text{false} \mid () \mid (e_1, e_2) \mid \text{Left } e \mid \text{Right } e \mid \text{fun } x:A \mapsto c \mid e \# op \mid h,$$

Bauer & P.

Notion of models got absorbed in homomorphisms

Plotkin & P.

Benton and Kennedy noted a few issues about the syntax of their construct when used for programming [13]. It is not obvious that t is handled whereas t' is not, especially when t' is large and the handler is obscured. An alternative they propose is $\text{try } x \Leftarrow t \text{ unless } \{e_1 \Rightarrow t_1 \mid \dots \mid e_n \Rightarrow t_n\}_i \text{ in } t'$, but then it is not obvious that x is bound in t' , but not in the handler. The syntax of our construct $\text{try } t \text{ with } H(u; t) \text{ as } x \text{ in } t'$ addresses those issues and clarifies the order of evaluation: after t is handled with H , its results are bound to x and used in t' .

A handler

$$h = \text{handler } (e_i \# \text{op}_i \ x \ k \mapsto c_i)_i \mid \text{val } x \mapsto c_v \mid \text{finally } x \mapsto c_f$$

may be applied to a computation c with the handling construct

with h handle c ,

Bauer & P.

Equations disappeared

Plotkin & P.

framework [15, 11]. Section 3, describes (base) values and the algebraic theory of effects. A natural need for two languages arises: one to describe handlers, given in Section 4, and one where they are used to handle computations, given in Section 5. The second parts of these sections give the relevant denotational semantics; readers may wish to see Section 6, where

ensuring correctness

~~programmer
writes and uses
handlers~~

language designer
writes handlers

programmer
uses them

maximum result

operations

or : 2

handlers

$H_{\max} = \{ \text{or}(x_1, x_2) \rightarrow \max(x_1, x_2) \}$

try or(or(3, 2), 5) with $H_{\max} = 5$

~~$H_{\text{sum}} = \{ \text{or}(x_1, x_2) \rightarrow x_1 + x_2 \}$~~

~~try or(3, 3) with $H_{\text{sum}} = 6$~~

~~try 3 with $H_{\text{sum}} = 3$~~

Bauer & P.

Shallow handlers were visible only when looking operationally

Handlers in Action

Ohad Kammar

Another possible behaviour is for the continuation to return an unhandled computation, which must then be handled explicitly. We call such handlers *shallow handlers* because each handler only handles one step of a computation, in contrast to Plotkin and Pretnar's *deep handlers*. Shallow handlers are to deep handlers as case analysis is to a fold on an algebraic data type.

Keywords algebraic effects; effect handlers; effect typing; monads; continuations; Haskell; modularity

1. Introduction

Monads have proven remarkably successful over effectful computations [4, 30]. The programming language primitive via the monad principle: program to an interface.

Modular programs are constructed from building blocks. This is modular programming. Given an abstract interface, we instantiate it independently instantiated with concrete data. This is modular instantiation.

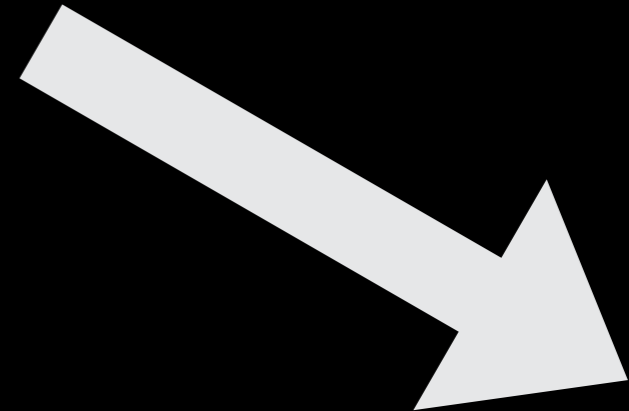
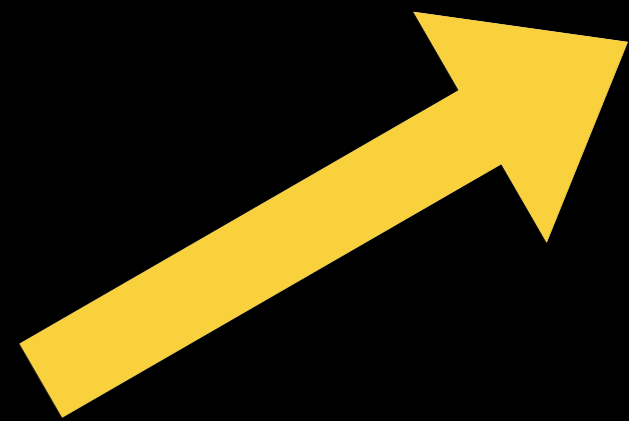
The monadic approach to effectful computation is a concrete implementation rather than an abstract one. For instance, in Haskell

Permission to make digital or hard copies of this work for classroom use is granted without fee provided that the copy is made on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting and publishing, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org. ICFP '13, September 25–27, 2013, Boston, MA. Copyright is held by the owner/author(s). ACM 978-1-4503-2326-0/13/09...\$15.00. <http://dx.doi.org/10.1145/2500365.2500590>





HANDLERS



Can **equations** also be tracked in an effect system?

Under consideration for publication in *J. Functional Programming*

1

Local Algebraic Effect Theories

Žiga Lukšič and Matija Pretnar*
University of Ljubljana, Faculty of Mathematics and Physics, Slovenia
(e-mail: ziga.luksic@fmf.uni-lj.si, matija.pretnar@fmf.uni-lj.si)

Abstract

Algebraic effects are computational effects that can be described with a set of basic operations and equations between them.

$\Gamma \vdash M : \sigma \mid \varphi / \mathcal{E}$



PUTTING **REASON**
BACK INTO **HANDLERS**

now with up to
37% shorter* proofs

*Findings based on a survey of 2 proofs conducted today

(Plotkin, 2018) — of the later work on handlers (Kammar *et al.*, 2013; Bauer and Pretnar, 2017; Biernacki *et al.*, 2018), resulting in a weaker reasoning logic.

This is achieved by reintroducing effect theories into the type system, tracking the effects of different parts of a program. On one hand, the induced logic allows us to reason about programs that are equivalent to ones with respect to the effect theory, while on the other hand, the logic enforces that handlers preserve equivalences, further specifying the operational semantics. In an informal overview in Section 1, we proceed as follows:

1. We start with a working language, its operational semantics, and the typing rules.

2. We show that whether a handler respects an effect theory is in general undecidable (Plotkin, 2018). There is no canonical way of defining such a judgement. Therefore, we give several parameterized judgements, and in Section 3, we discuss some interesting choices.

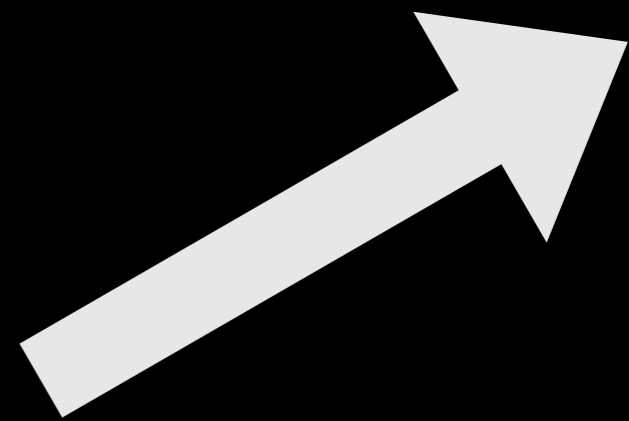
3. We show that typing judgements is intertwined with a reasoning logic, we define the denotation of types and terms. Thus, in Section 4, we define a soundness-based denotational semantics that disregards effect theories and prove its meta-theoretic properties.

This work is supported by the Air Force Office of Scientific Research under





HANDLERS



Efficient execution is just **fusion** with **purity-aware** compilation

Tom Schrijvers, *Efficient Compilation of Algebraic Effects and Handlers* (Tuesday, February 21, 2017, 14:55):

The popularity of algebraic effect handlers as a programming language feature for user-defined computational effects, is steadily growing. Yet, even though efficient runtime representations have already been studied, most handler-based programs are still much slower than hand-written code. In this paper we show that the performance gap can be drastically narrowed (in some cases even closed) by means of type- and-effect-directed optimising compilation. Our approach consists of two stages. Firstly, we combine elementary source-to-source transformations with judicious function specialisation in order to aggressively reduce handler applications. Secondly, we show how to elaborate the source language into a handler-less target language in a way that incurs no overhead for pure computations. This work comes with a practical implementation: an optimizing compiler from Effy, a small functional language with algebraic effects and handlers, to OCaml. Experimental evaluation with this implementation demonstrates that in a number of benchmarks our approach eliminates much of the overhead of handlers and yields competitive performance with hand-written OCaml code. This is joint work with Matija Pretnar, Amr Hany Saleh Shehata and Axel Faes.

Eff

Schrijvers
et al.

submitted

Benchmark 3: there is no benchmark 3

The experimental **evaluation of the optimization is very thin** and significantly below the kind of evaluation that one expects of an optimization paper at a venue like ICFP.

reviewer #113A

Only my concern is that **the benchmark set is rather small**. It remains to be seen if this improvement scales to larger programs.

reviewer #113B

Your compiler doesn't seem to support implementing high-level effects with OCaml's native effects, like references and console input/output. At least, **there are no examples in the paper**.

reviewer #113C

The evaluation of the work is only done using **two very small benchmarks**: a looping counter and nqueens.

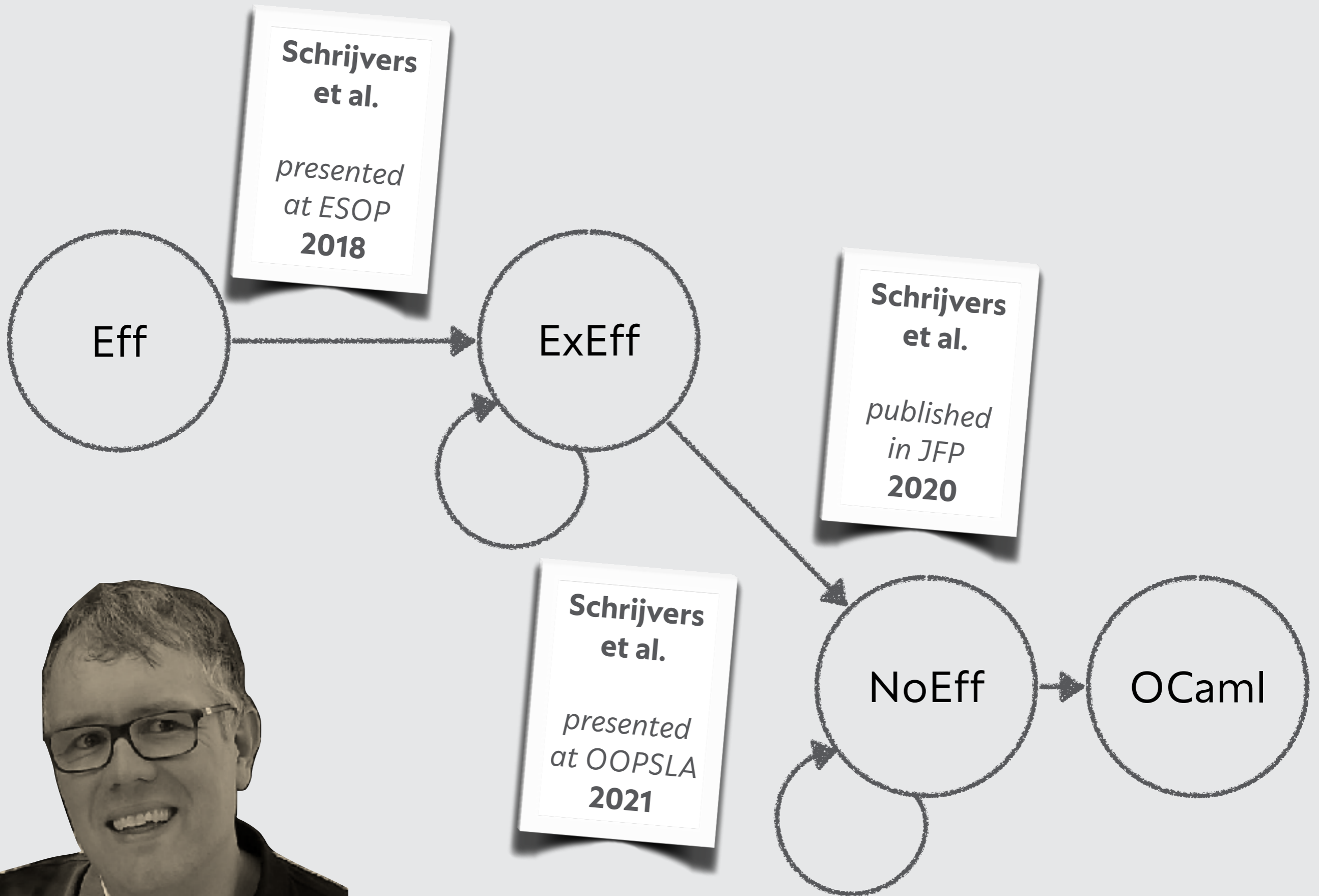
reviewer #113D

verdict: **REJECT**

OCaml



Purity-aware compilation required **coercions** as witnesses of **subtyping**



Adding **polymorphism** incurs significant **overhead**

ALGOL 2030

```
let applyZero f = f 0
applyZero cos
```

EXALGOL 2030

```
let applyZero $\alpha\beta$  (f:  $\alpha \rightarrow \beta$ ) = f (0Dw)
(w: int  $\leq$   $\alpha$ )
 $\forall \alpha, \beta. (\text{int} \leq \alpha) \Rightarrow (\alpha \rightarrow \beta) \rightarrow \beta$ 
applyZero float float int2float cos
```

OCAML

```
let applyZero w f = f (w 0)
(int  $\rightarrow$  'a)  $\rightarrow$  ('a  $\rightarrow$  'b)  $\rightarrow$  'b
applyZero int_to_float cos
```

```
let applyZero'  $\beta$  (f: int  $\rightarrow$   $\beta$ ) = f 0
applyZero' float (cosD (int2float  $\rightarrow$  (float))
cos o int2float
```

SIMPLIFYING EXPLICIT SUBTYPING COERCIONS IN A POLYMORPHIC CALCULUS WITH EFFECTS

FILIP KOPRIVEC^{a,b} AND MATIJA PRETNAR^{a,b}

^aUniversity of Ljubljana, Faculty of Mathematics and Physics, Jadranska 19, SI-1000 Ljubljana, Slovenia

^bInstitute of Mathematics, Physics and Mechanics, Jadranska 19, SI-1000 Ljubljana, Slovenia
Email address: filip.koprivec@fmf.uni-lj.si
Email address: matija.pretnar@fmf.uni-lj.si

ABSTRACT. Algebraic effect handlers are becoming increasingly popular way of structuring and reasoning about effectful computations, and their performance is often a concern. One of the proposed approaches towards efficient compilation is tracking effect information through explicit subtyping coercions. However, in the presence of polymorphism, these coercions are compiled to additional arguments of compiled functions, incurring significant overhead. In this paper, we present a polymorphic effectful calculus, identify simplification phases to reduce the number of unnecessary constraints, and prove they preserve the semantics. In addition, we implement the simplification algorithm in the Eff language, and evaluate its performance on a number of benchmarks. Though we do not prove optimality of the presented simplifications, the results show that the algorithm eliminates all the coercions, resulting in a code as efficient as manually monomorphised one.

INTRODUCTION

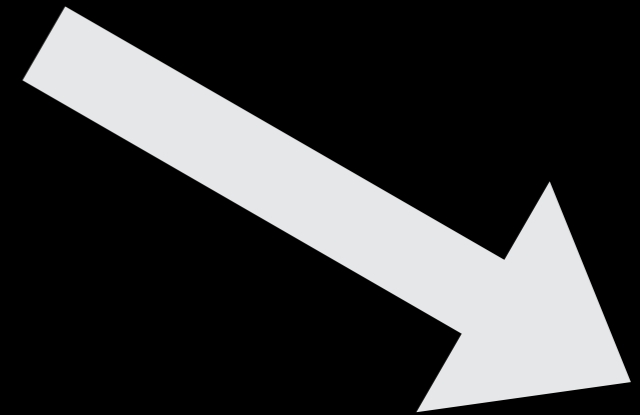
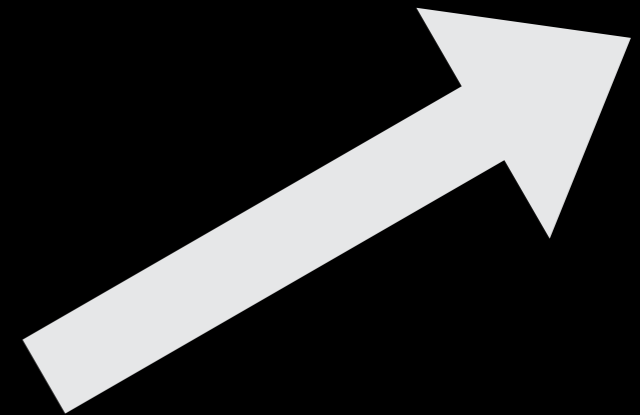
We have seen an increase in the number of programming languages that support effect handlers [PP03, PP13]. With a widespread usage, the need for performance becomes more important. And there are two main ways for achieving it: an efficient compiler [PP15, SDW⁺21], or an optimising compiler [SBO20, XL21, KKPS21], which is the focus of this paper. The work of Kopylov et al. [KKPS21] has shown how an optimising compiler can take code written in a language with effect handlers, infer precise information about which parts of it use effects, and produce code that matches conventional handcrafted one. However, to track effect information through explicit subtyping coercions [KPS⁺20], and to pass these coercions as additional parameters. Since

Keywords: Computational effects, Optimizing compilation, Algebraic effects, Polymorphic compilation, Denotational semantics.
This work is based upon work supported by the Air Force Office of Scientific Research under awards FA9550-21-1-0024 and FA9550-21-1-0024.





HANDLERS



Lindley & P.

*A survey of
algebraic
effect handlers*

soon



QUESTIONS?