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 FÉV 2024 → 14 MAR 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 FÉV 2099 → 12 MAR 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 an useful mathematical tool in the study of programming languages. However, if one uses $\beta\eta$-conversion to prove equivalence of programs, then a gross simplification[1] 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.
- 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.

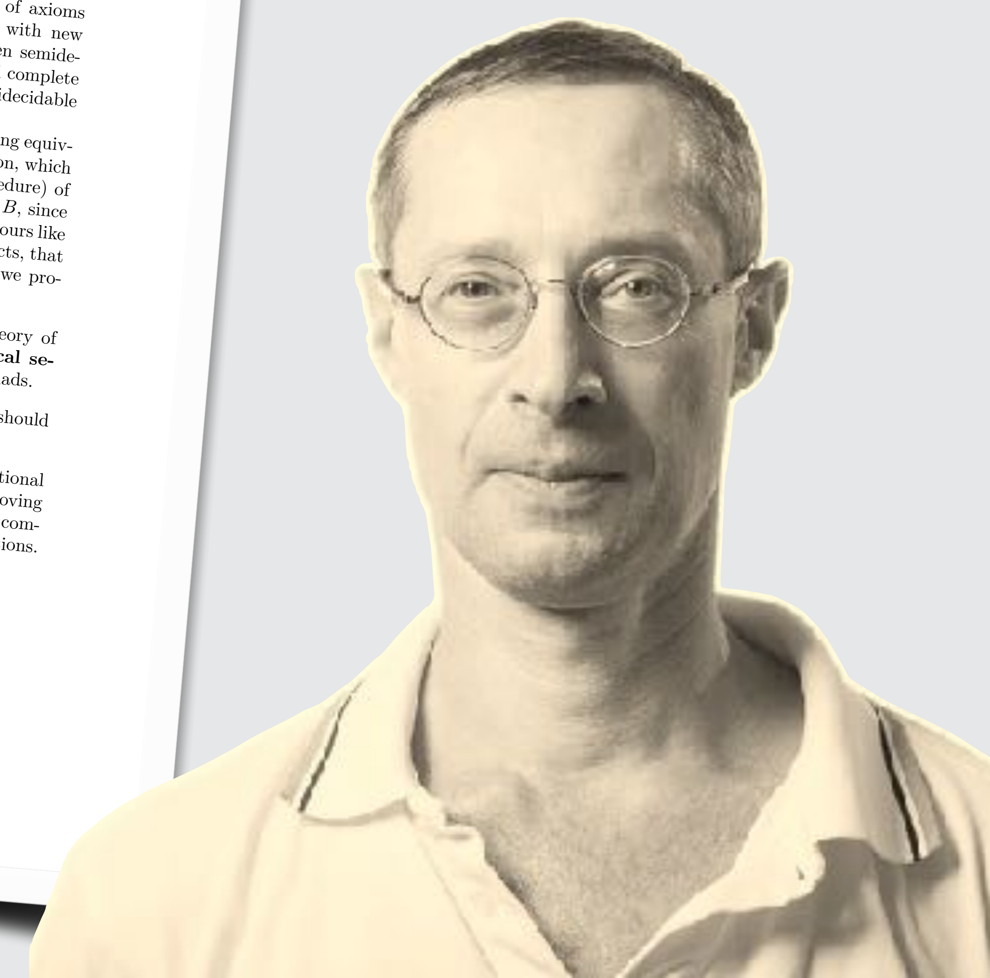
We do not take as a starting point for proving equivalence of programs the theory of $\beta\eta$-conversion, which identifies the denotation of a program (procedure) of type $A \to B$ with a total function from A to B, since this identification wipes out completely behaviours like non-termination, non-determinism 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.

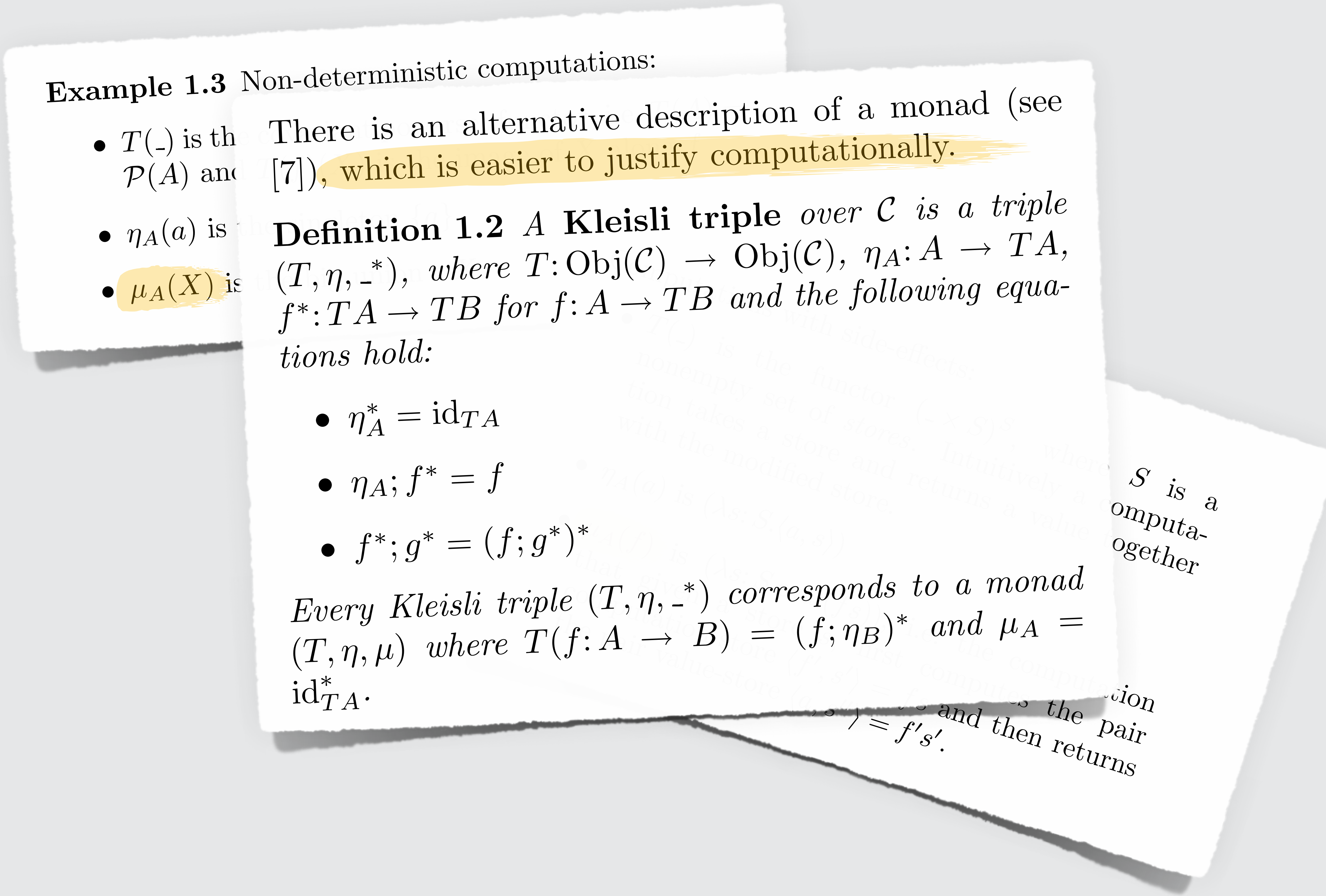
At the end we get a formal system, the computational lambda-calculus (λ_c-calculus for short), for proving **equivalence** of programs, which is sound and complete w.r.t. the categorical semantics of computations.

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

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



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



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* $T\colon \mathrm{Obj}(\mathcal{C}) \to \mathrm{Obj}(\mathcal{C})$, $\eta_A\colon A \to TA$, $f^*\colon TA \to TB$ *for* $f\colon A \to TB$ *and the following equations hold:*

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

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

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

Definition 1.2 (Manes, 1976). A *Kleisli triple over a category* $\mathscr{C}$ is a triple $(T, \eta, \_^*)$, where T: $\mathrm{Obj}(\mathscr{C}) \to \mathrm{Obj}(\mathscr{C})$, $\eta_A: A \to TA$ for $A \in \mathrm{Obj}(\mathscr{C})$, $f^*: TA \to TB$ for $f: A \to TB$ and the following equations hold:

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

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** $TA = A_\bot (= A + \{\bot\})$

η_A is the inclusion of A into $A_\bot$
if $f: A \to TB$, then $f^*(\bot) = \bot$ and $f^*(a) = f(a)$ (when $a \in A$)

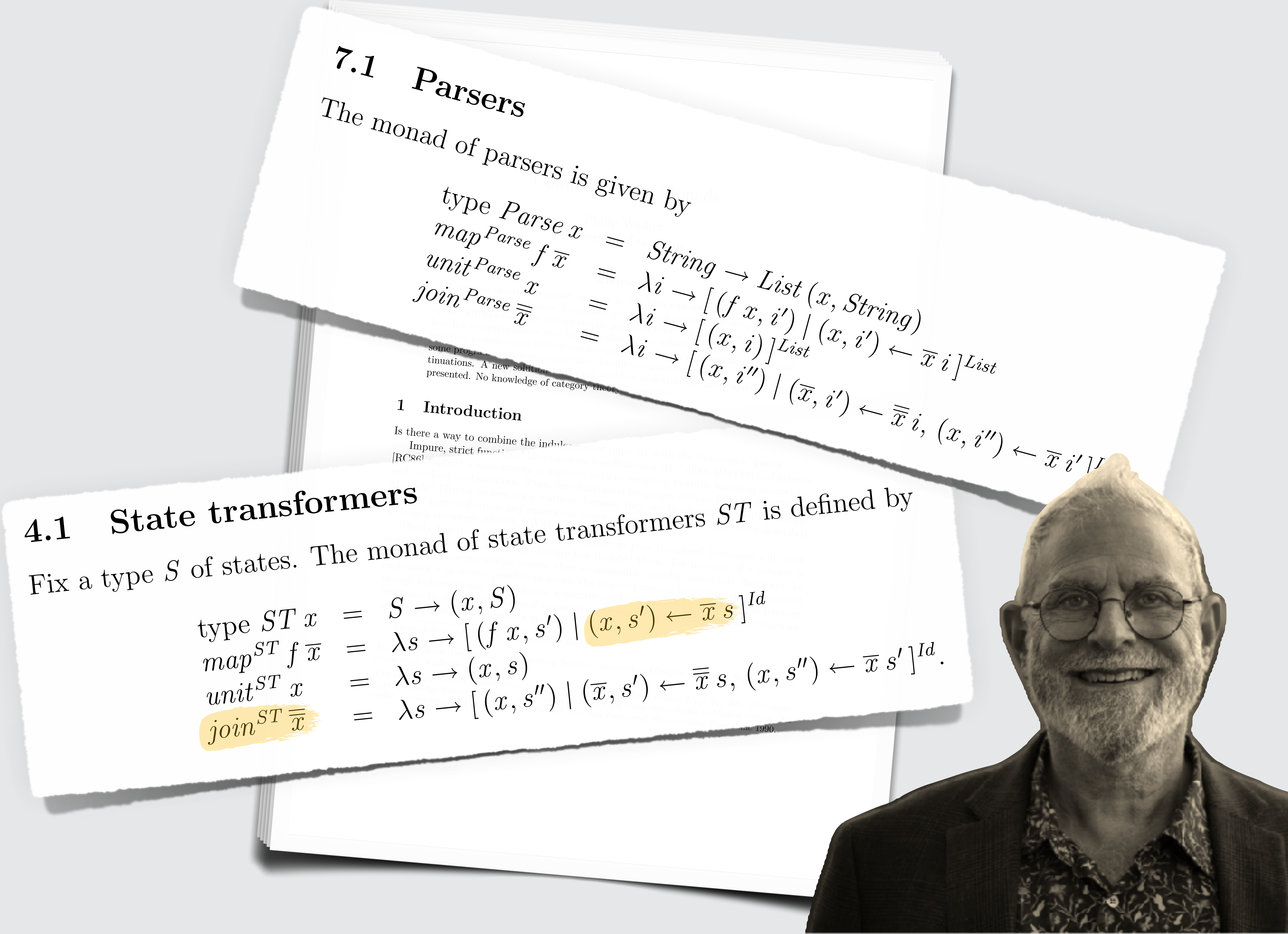
- **nondeterminism** $TA = \mathscr{P}_{\mathrm{fin}}(A)$

η_A is the singleton map $a \mapsto \{a\}$
if $f: A \to TB$ 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 $f: A \to TB$ and $c \in TA$, then $f^*(c) = \lambda s: S.(\mathrm{let}\ \langle a, s'\rangle = c(s)\ \mathrm{in}\ f(a)(s'))$

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



7.1 Parsers

The monad of parsers is given by

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

4.1 State transformers

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

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

An effect is specified with a **monad**

monad

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

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

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

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

monad

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

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

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

effect-specific operations

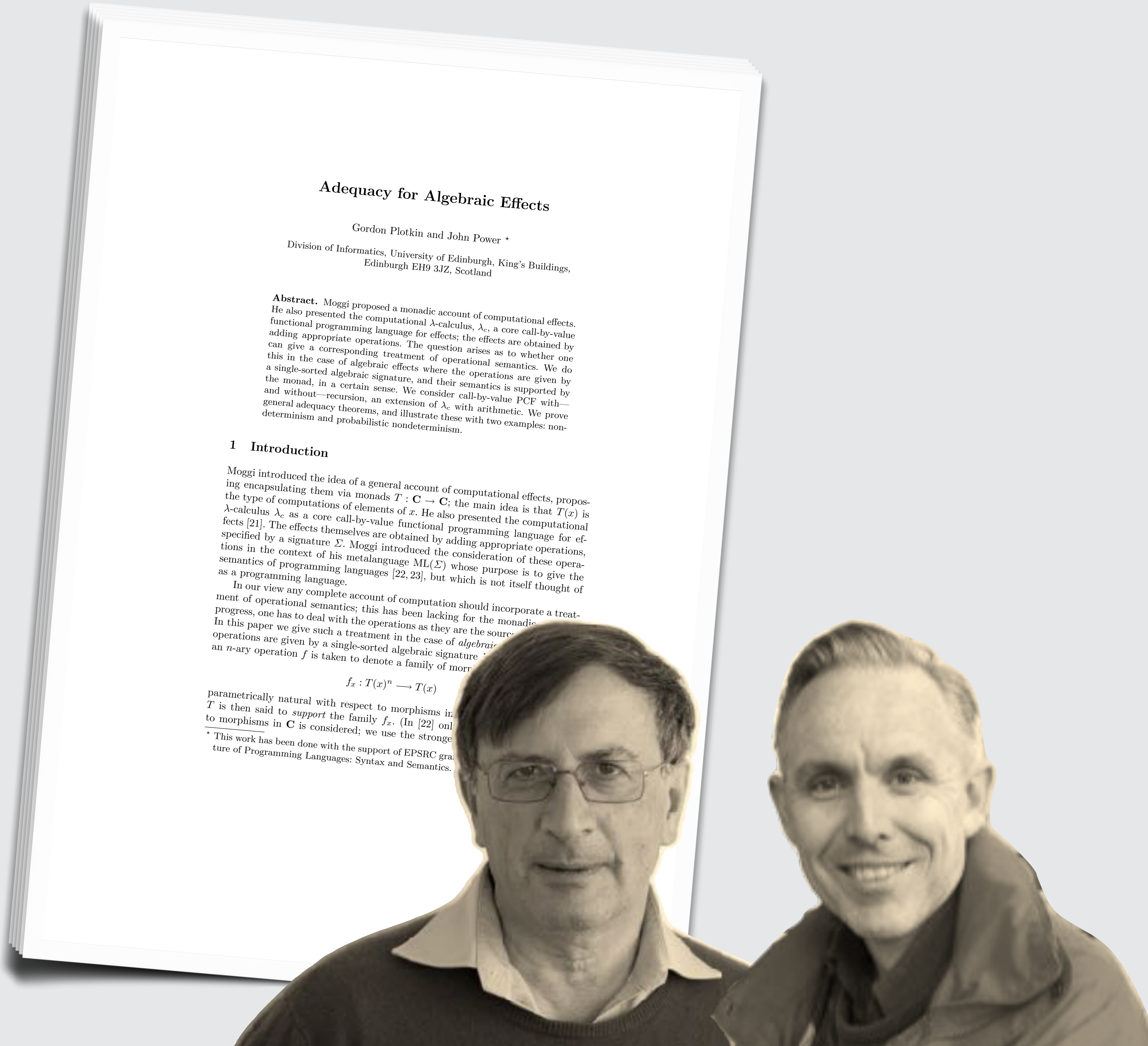
$$\texttt{fail} : TX$$

$$\texttt{fail} = \{\}$$

$$\texttt{choose} : TX \times TX \to TX$$

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

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



Exception **handling failed** to be algebraic

	operations
exceptions	`fail` `try`
state	`get` `set`
choice	`choose`
I/O	`read` `write`
probability	`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 $M : L \longrightarrow C$.

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 $f : m \longrightarrow 1$ 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 with countable powers and countable coproducts, $Mod(L_E, C)$ is equiv category of algebras for the monad $- + \underline{E}$, where $\underline{E}$ for the E-fold c 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 or : 2, a nondeterministically chosen boolean is represented by the term or(return true, return false) : F**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., $\mathsf{or}_X : TX^2 \to TX$, 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 next step was implementing handlers in **practice**

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

Moggi

Computational lambda-calculus and monads

1989

Wadler

Comprehending monads

1991

=

Plotkin & P.

Handlers of algebraic effects

2009

?

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 ()
```

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

ref : Store → (A ⇒ (Int → A))

ith effects I: Theory →

combined.

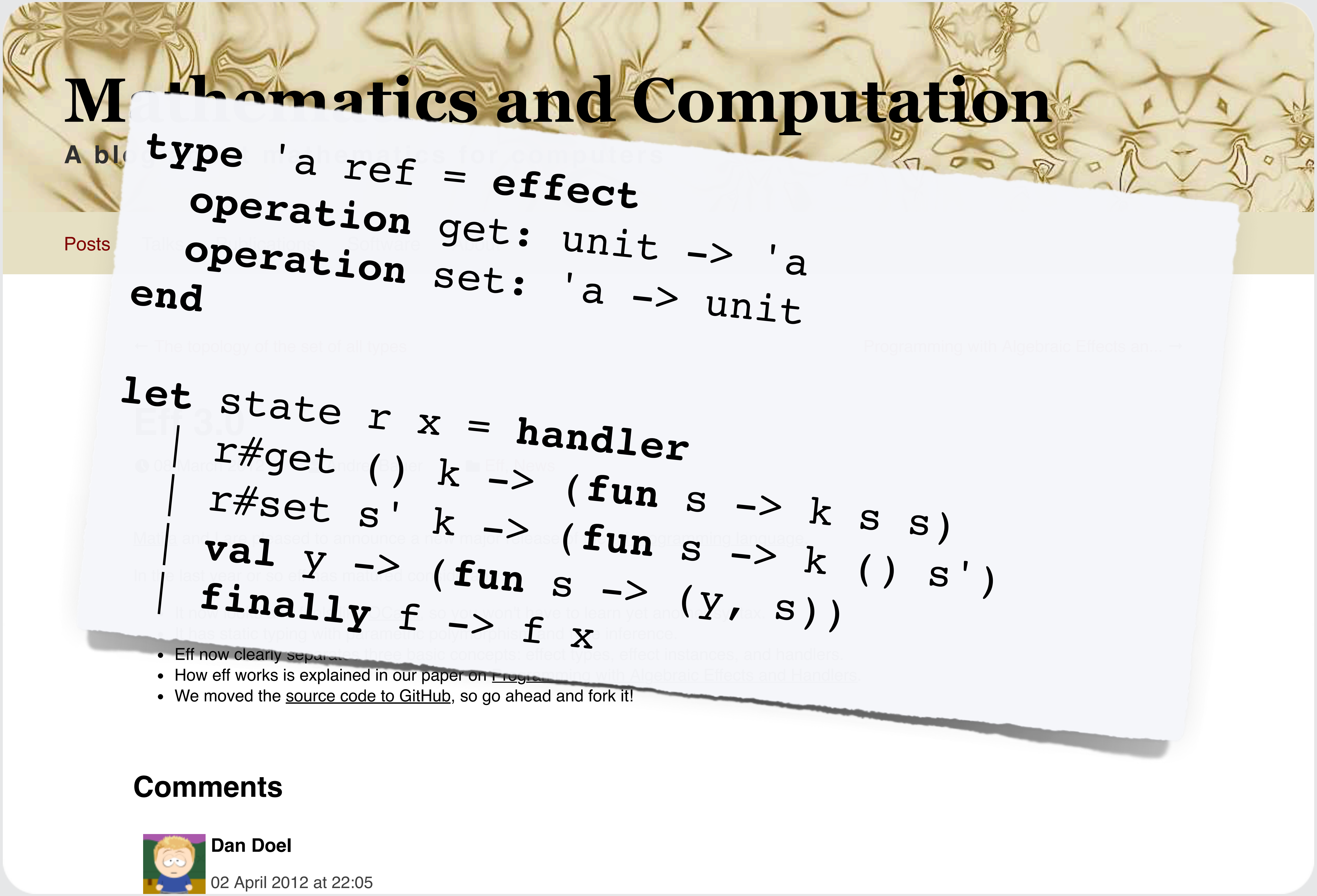
Installation

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 is

Next version **added types** and moved much closer to **OCaml**



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

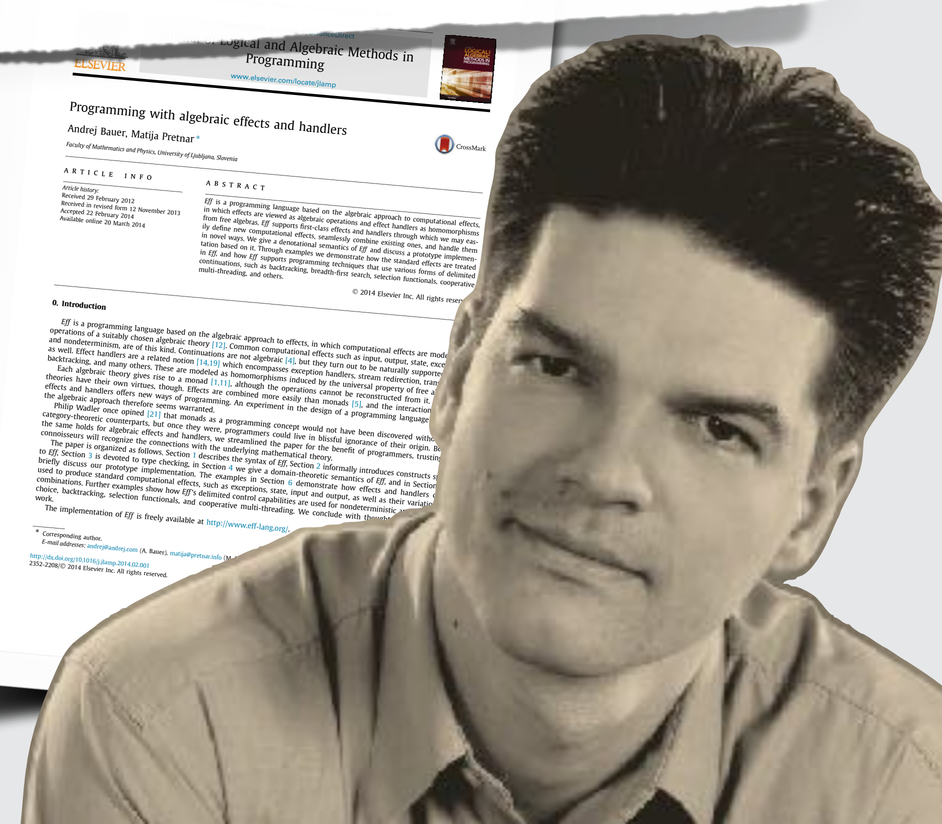
Comprehending monads

1991

Logical and Algebraic Methods in Programming

Programming with algebraic effects and handlers

Andrej Bauer, Matija Pretnar



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

Plotkin & P.

$$\frac{\boldsymbol{x}_p:\boldsymbol{\sigma}, \boldsymbol{x}:\boldsymbol{\beta}; \boldsymbol{z}_p:\boldsymbol{\chi}, (z_i:(\boldsymbol{\alpha}_i) \to \chi)_{i=1}^{n} \vdash h_{\mathrm{op}}:\chi \quad (\mathrm{op}:\boldsymbol{\beta};\boldsymbol{\alpha}_1,\ldots,\boldsymbol{\alpha}_n \in \Sigma_{\mathrm{eff}})}{\vdash (\boldsymbol{x}_p:\boldsymbol{\sigma}; \boldsymbol{z}_p:\boldsymbol{\chi}).\{\mathrm{op}_{\boldsymbol{x}}(\boldsymbol{z}) \mapsto h_{\mathrm{op}}\}_{\mathrm{op}\in\Sigma_{\mathrm{eff}}} : (\boldsymbol{\sigma};\boldsymbol{\chi}) \to \chi \ \textbf{handler}}$$

$$e ::= x \mid n \mid b \mid \mathtt{true} \mid \mathtt{false} \mid () \mid (e_1, e_2) \mid \mathtt{Left}\, e \mid \mathtt{Right}\, e \mid \mathtt{fun}\ x : A \mapsto c \mid e\,\#\,\mathtt{op} \mid h,$$

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 $\mathsf{try}\ x \Leftarrow t\ \mathsf{unless}\ \{e_1 \Rightarrow t_1 \mid \ldots \mid e_n \Rightarrow t_n\}_i\ \mathsf{in}\ t'$, but then it is not obvious that x is bound in t', but not in the handler. The syntax of our construct $\mathsf{try}\ t\ \mathsf{with}\ H(\boldsymbol{u}; \boldsymbol{t})\ \mathsf{as}\ x\ \mathsf{in}\ t'$ 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

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

may be applied to a computation c with the handling construct

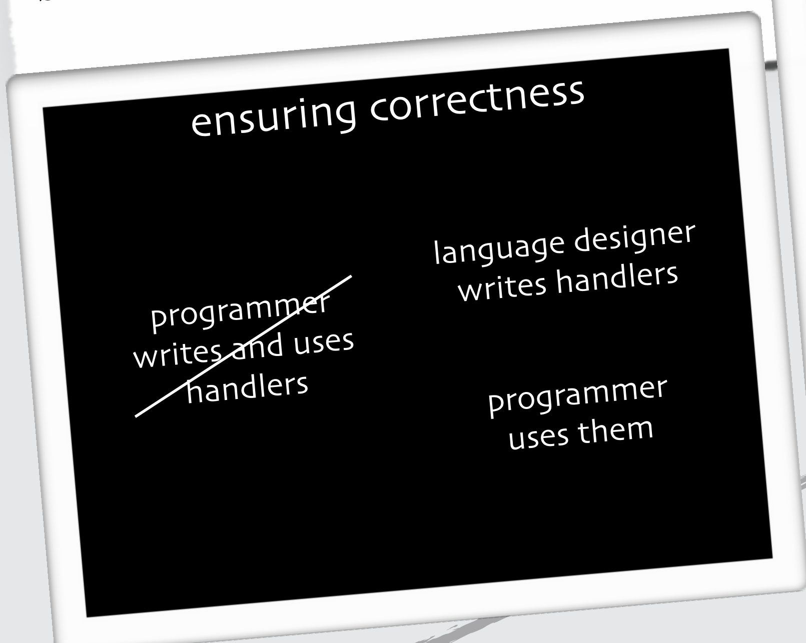
$$\mathtt{with}\ h\ \mathtt{handle}\ c,$$

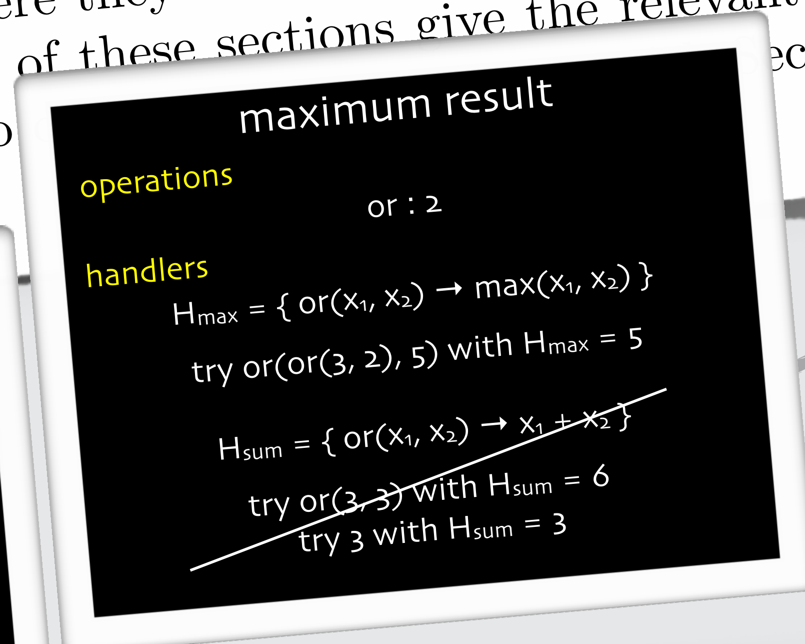
Bauer & P.

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 … ection 6, where





Bauer & P.

Shallow handlers were visible only when looking **operationally**

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.


HANDLERS

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


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

OCaml

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**.

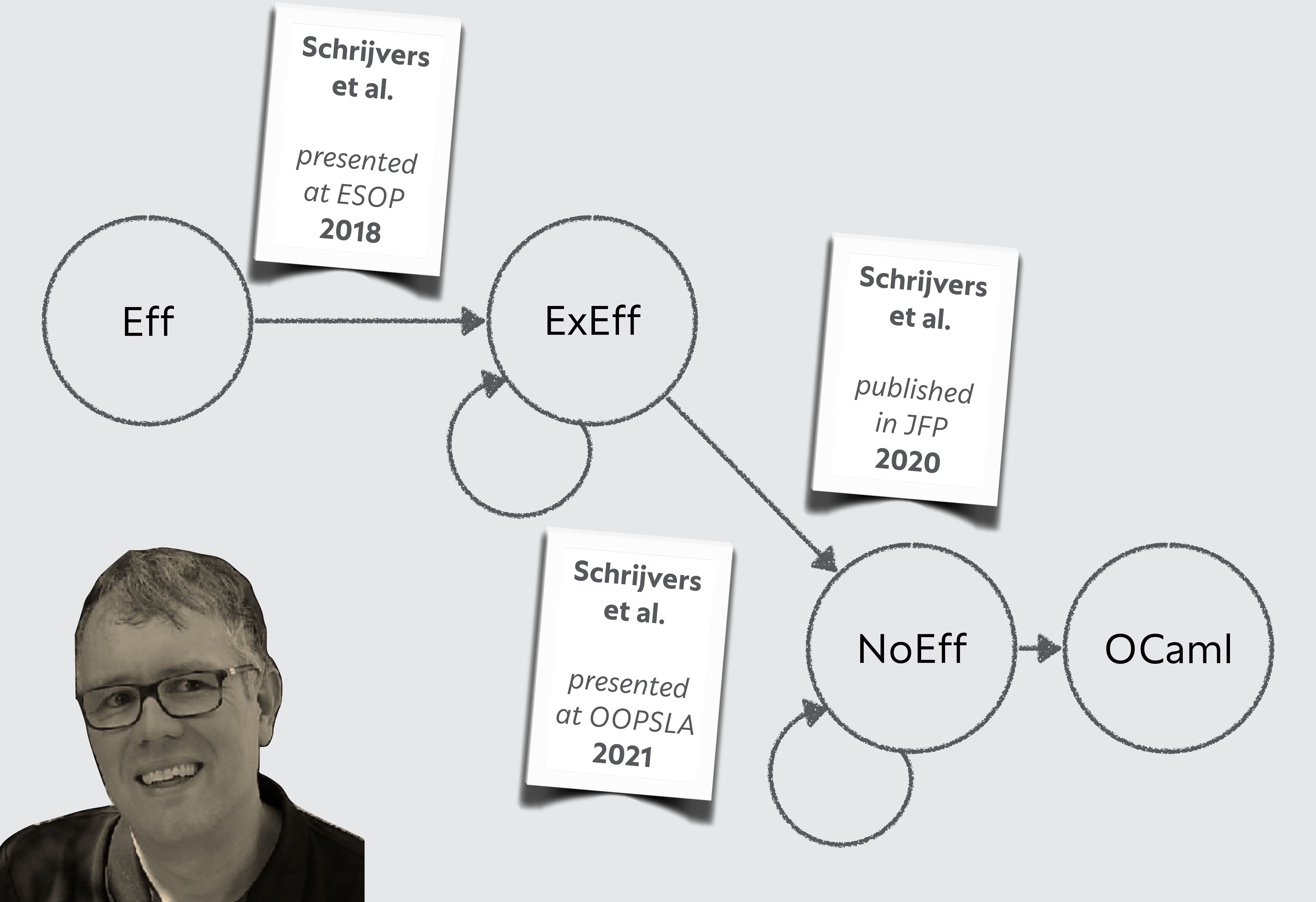
reviewer #113C

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

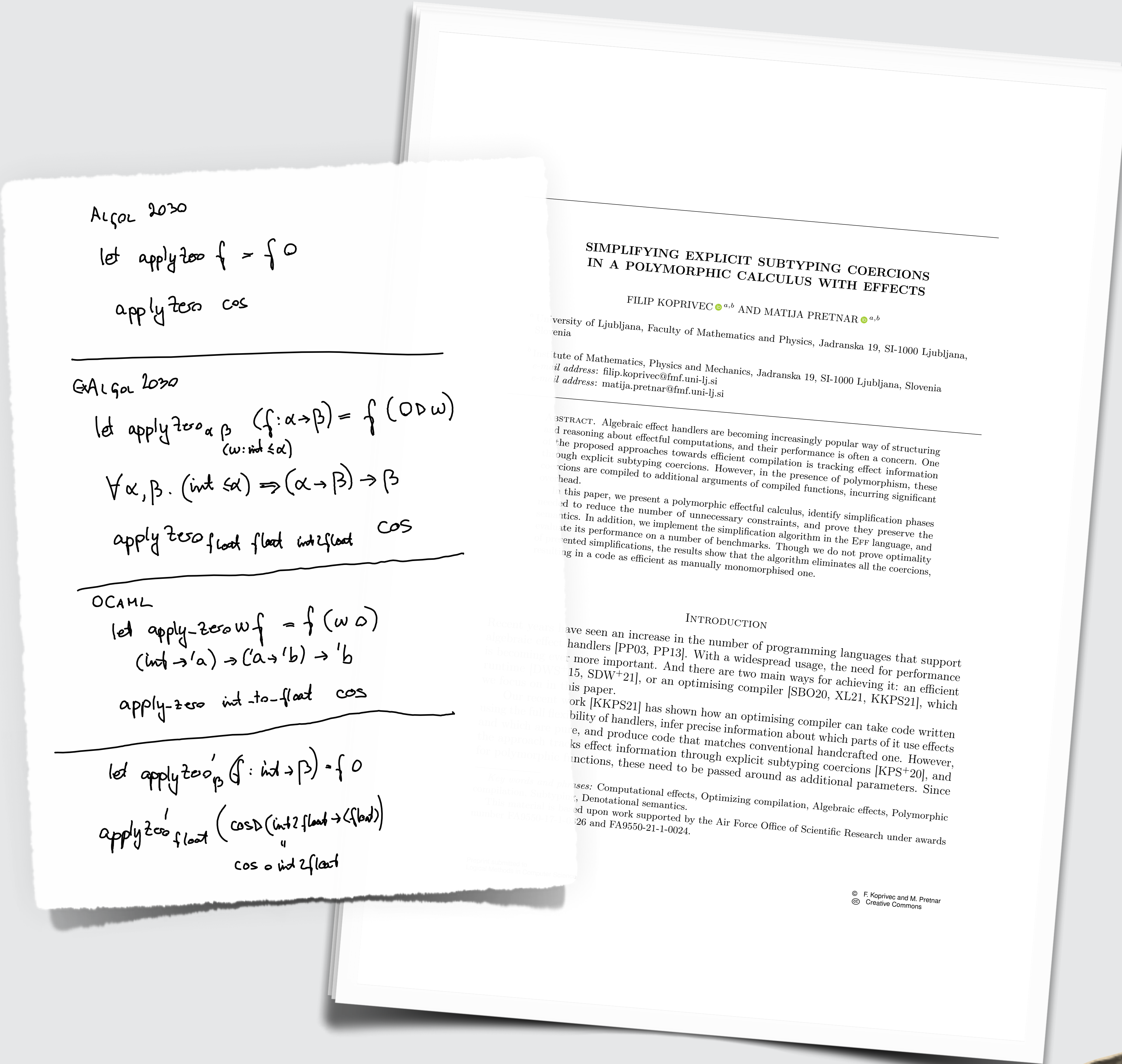
reviewer #113D

verdict: **REJECT**

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



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




HANDLERS

Lindley & P.

A survey of algebraic effect handlers

soon

QUESTIONS?