EFFECT **HANDLERS**

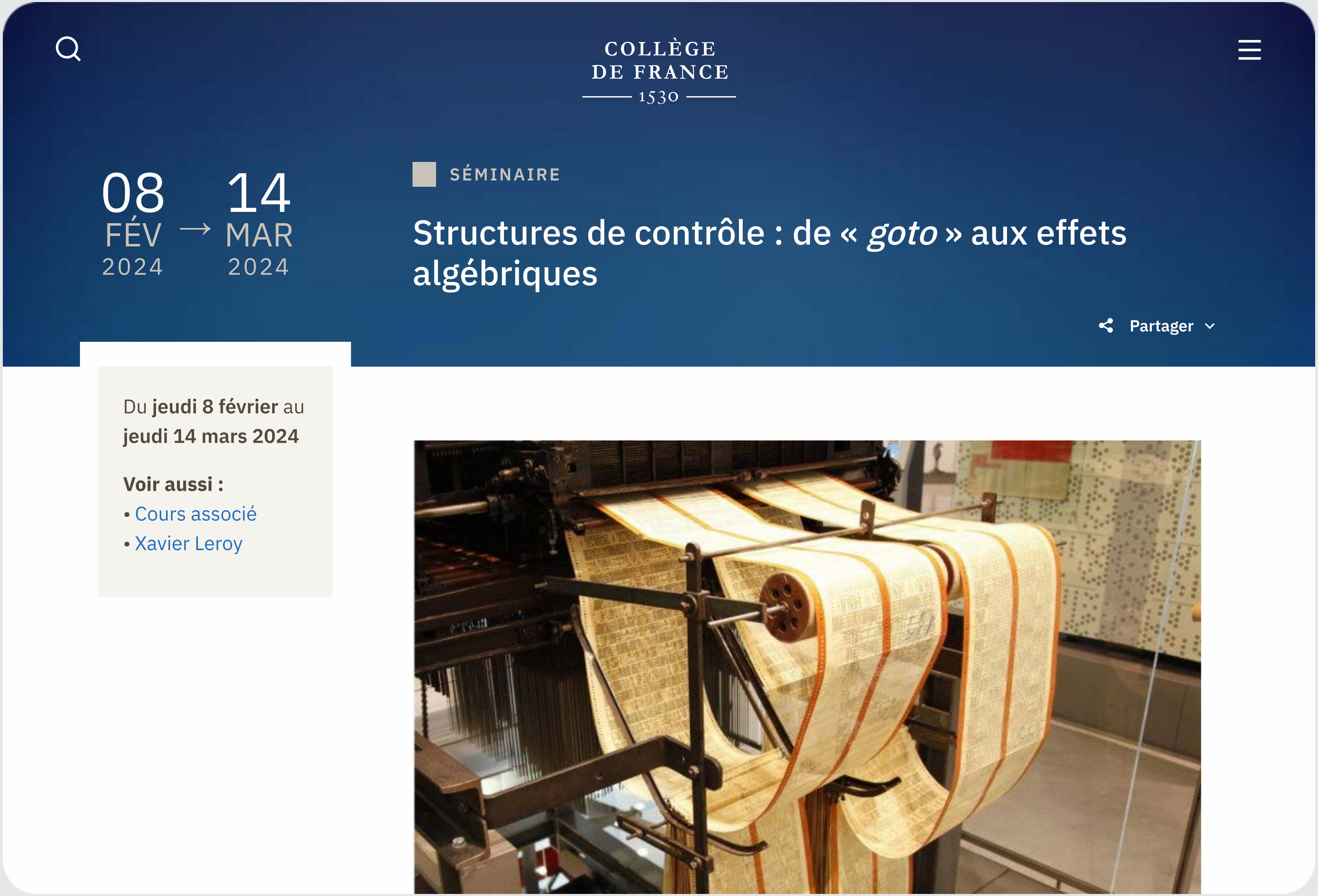
&

MATHEMATICALLY INSPIRED LANGUAGE **CONSTRUCTS**

Matija Pretnar

FMF UNIVERSITY OF LJUBLJANA
Faculty of Mathematics and Physics

This seminar series has covered the **past 75 years** of control structures



COLLÈGE
DE FRANCE
1530

08 FÉV 2024 → 14 MAR 2024

SÉMINAIRE

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

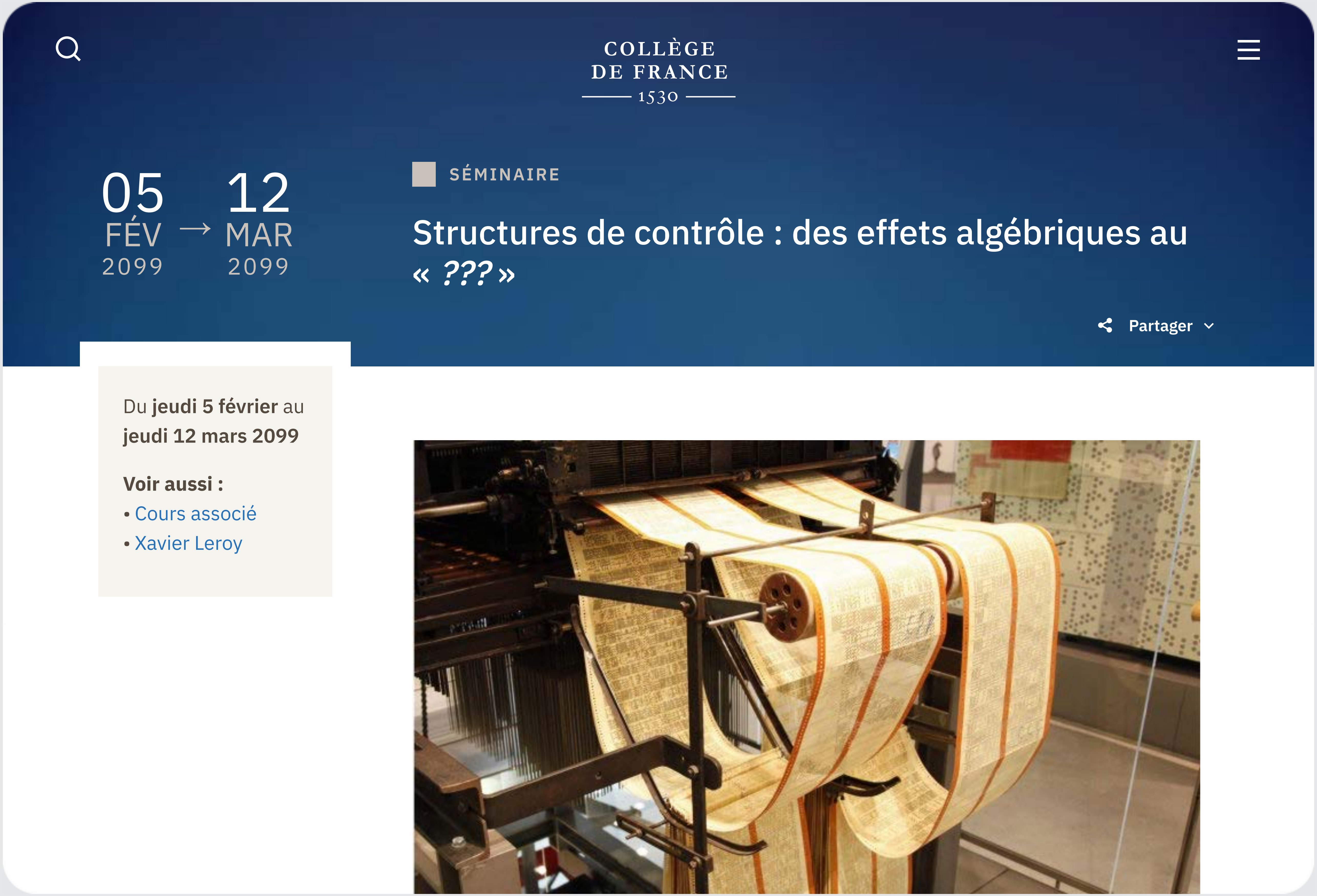
Partager

Du **jeudi 8 février** au **jeudi 14 mars 2024**

Voir aussi :

- Cours associé
- Xavier Leroy

How will the **next 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

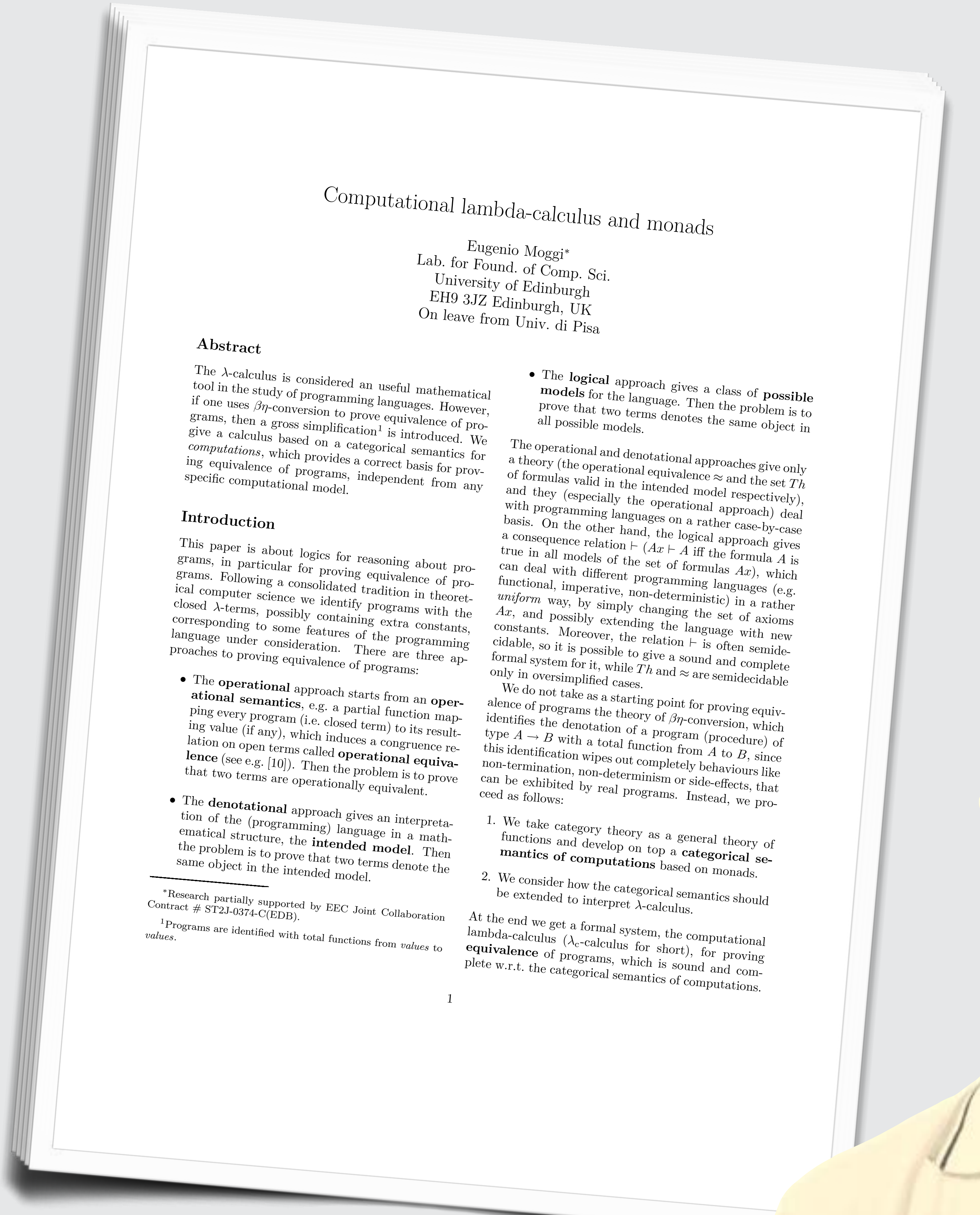
→ HANDLERS

HANDLERS

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:

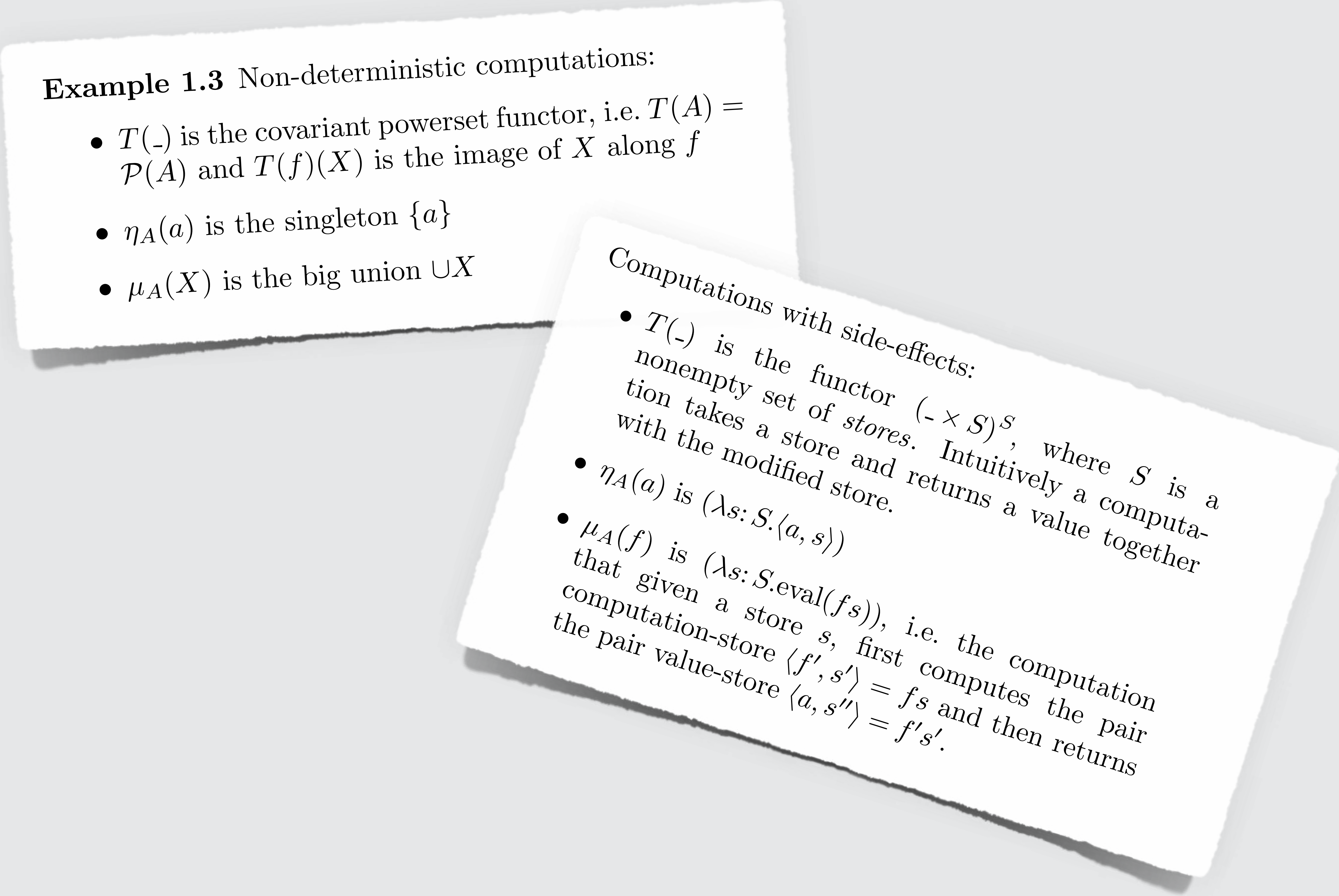
1. We take category theory as a general theory of functions and develop on top a **categorical semantics of computations** based on monads.
2. We consider how the categorical semantics should be extended to interpret λ-calculus.

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



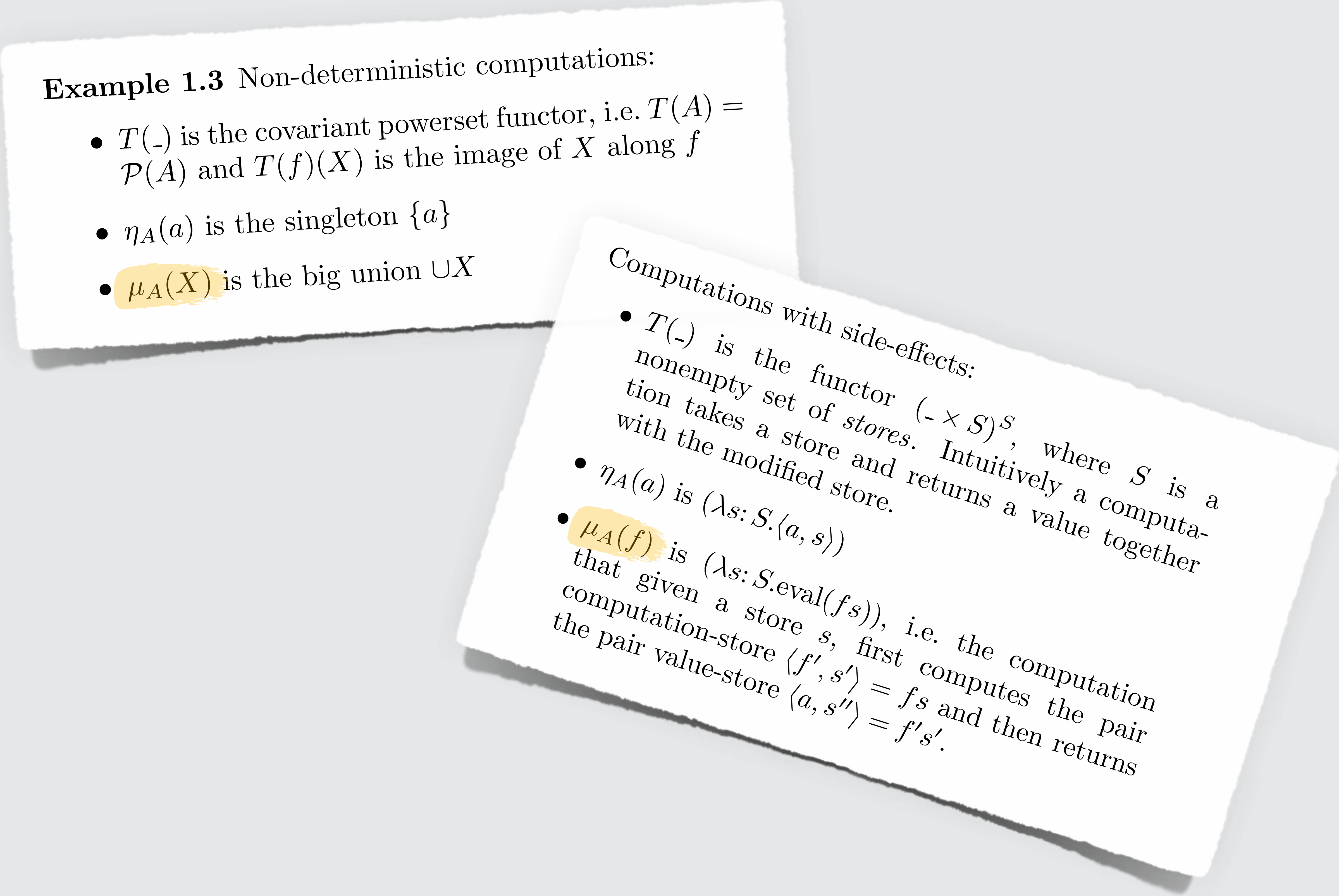
Example 1.3 Non-deterministic computations:

- $T(\_)$ is the covariant powerset functor, i.e. $T(A) = \mathcal{P}(A)$ and $T(f)(X)$ is the image of X along f
- $\eta_A(a)$ is the singleton $\{a\}$
- $\mu_A(X)$ is the big union $\cup X$

Computations with side-effects:

- $T(\_)$ is the functor $(\_ \times S)^S$, where S is a nonempty set of *stores*. Intuitively a computation takes a store and returns a value together with the modified store.
- $\eta_A(a)$ is $(\lambda s{:}\, S.\langle a, s\rangle)$
- $\mu_A(f)$ is $(\lambda s{:}\, S.\mathrm{eval}(fs))$, i.e. the computation that given a store s, first computes the pair computation-store $\langle f', s'\rangle = fs$ and then returns the pair value-store $\langle a, s''\rangle = f's'$.

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



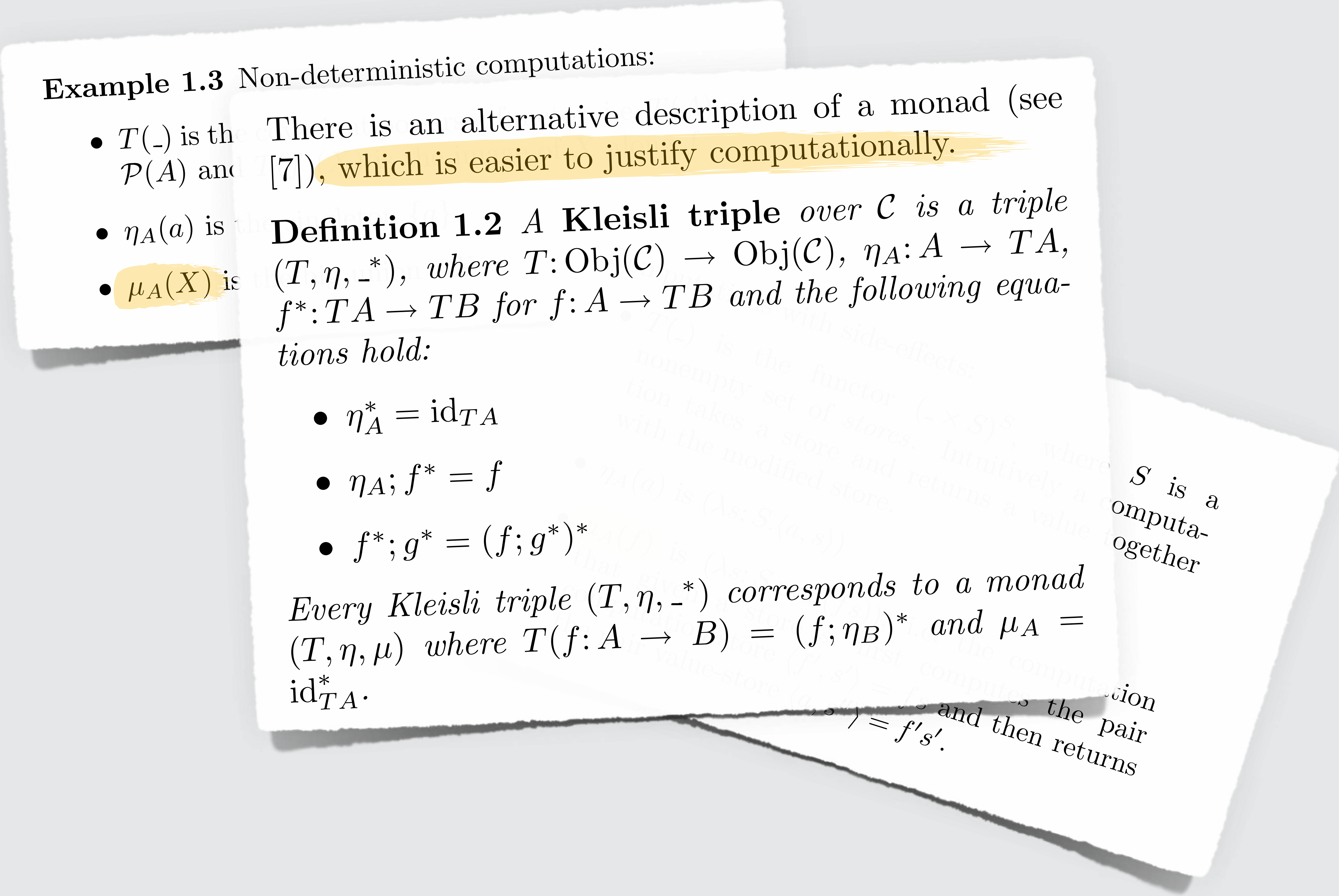
Example 1.3 Non-deterministic computations:

- $T(\_)$ is the covariant powerset functor, i.e. $T(A) = \mathcal{P}(A)$ and $T(f)(X)$ is the image of X along f
- $\eta_A(a)$ is the singleton $\{a\}$
- $\mu_A(X)$ is the big union $\cup X$

Computations with side-effects:

- $T(\_)$ is the functor $(\_ \times S)^S$, where S is a nonempty set of *stores*. Intuitively a computation takes a store and returns a value together with the modified store.
- $\eta_A(a)$ is $(\lambda s\colon S.\langle a, s\rangle)$
- $\mu_A(f)$ is $(\lambda s\colon S.\mathrm{eval}(fs))$, i.e. the computation that given a store s, first computes the pair computation-store $\langle f', s'\rangle = fs$ and then returns the pair value-store $\langle a, s''\rangle = f's'$.

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



Example 1.3 Non-deterministic computations:

- $T(\_)$ is th
  $\mathcal{P}(A)$ and
- $\eta_A(a)$ is
- $\mu_A(X)$ is

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:*

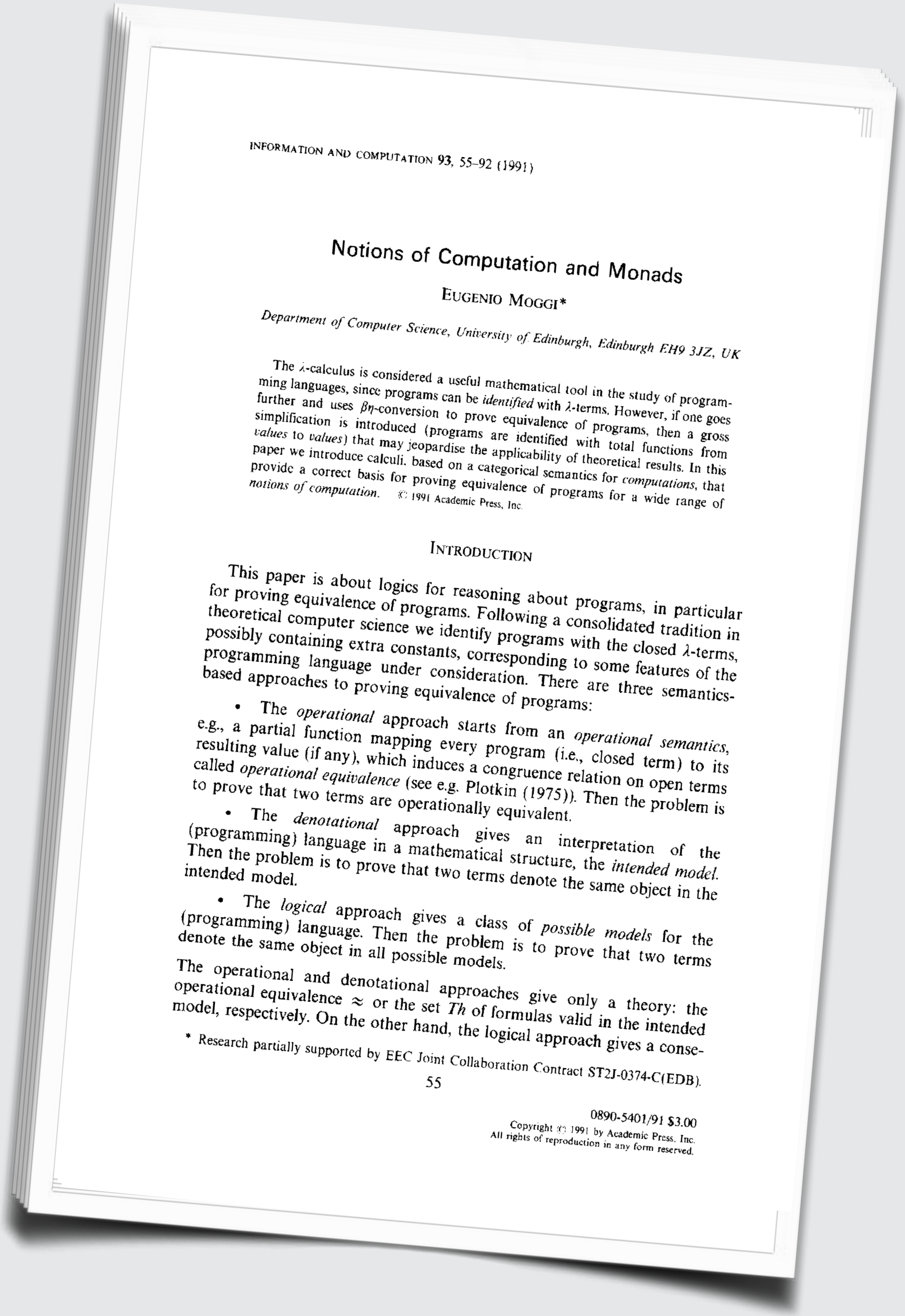
- $\eta_A^* = \mathrm{id}_{TA}$
- $\eta_A; f^* = f$
- $f^*; g^* = (f; g^*)^*$

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

S is a
omputa-
ogether

tion
the pair
and then returns
$= f's'$.

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



INFORMATION AND COMPUTATION **93**, 55–92 (1991)

Notions of Computation and Monads

EUGENIO MOGGI*

Department of Computer Science, University of Edinburgh, Edinburgh EH9 3JZ, UK

The λ-calculus is considered a useful mathematical tool in the study of programming languages, since programs can be *identified* with λ-terms. However, if one goes further and uses $\beta\eta$-conversion to prove equivalence of programs, then a gross simplification is introduced (programs are identified with total functions from *values* to *values*) that may jeopardise the applicability of theoretical results. In this paper we introduce calculi. based on a categorical semantics for *computations*, that provide a correct basis for proving equivalence of programs for a wide range of *notions of computation*. © 1991 Academic Press, Inc.

INTRODUCTION

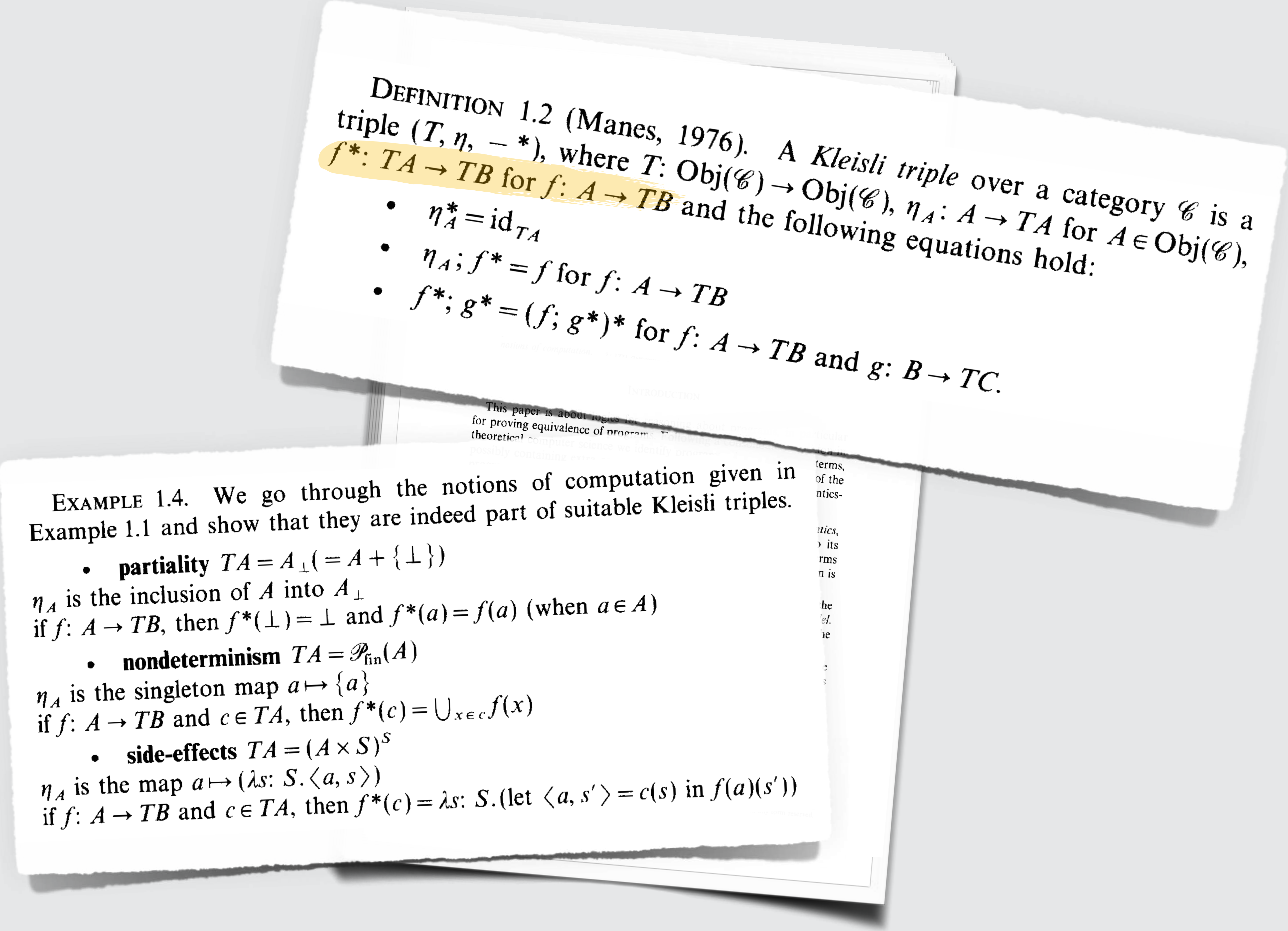
This paper is about logics for reasoning about programs, in particular for proving equivalence of programs. Following a consolidated tradition in theoretical computer science we identify programs with the closed λ-terms, possibly containing extra constants, corresponding to some features of the programming language under consideration. There are three semantics-based 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. Plotkin (1975)). 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 (programming) language. Then the problem is to prove that two terms denote the same object in all possible models.

The operational and denotational approaches give only a theory: the operational equivalence $\approx$ or the set *Th* of formulas valid in the intended model, respectively. On the other hand, the logical approach gives a conse-

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

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

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

This paper is about logics for

for proving equivalence of program

theoretical

terms,

of the

ntics-

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 $f\colon A \to TB$, then $f^*(\perp) = \perp$ and $f^*(a) = f(a)$ (when $a \in A$)

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

η_A is the singleton map $a \mapsto \{a\}$

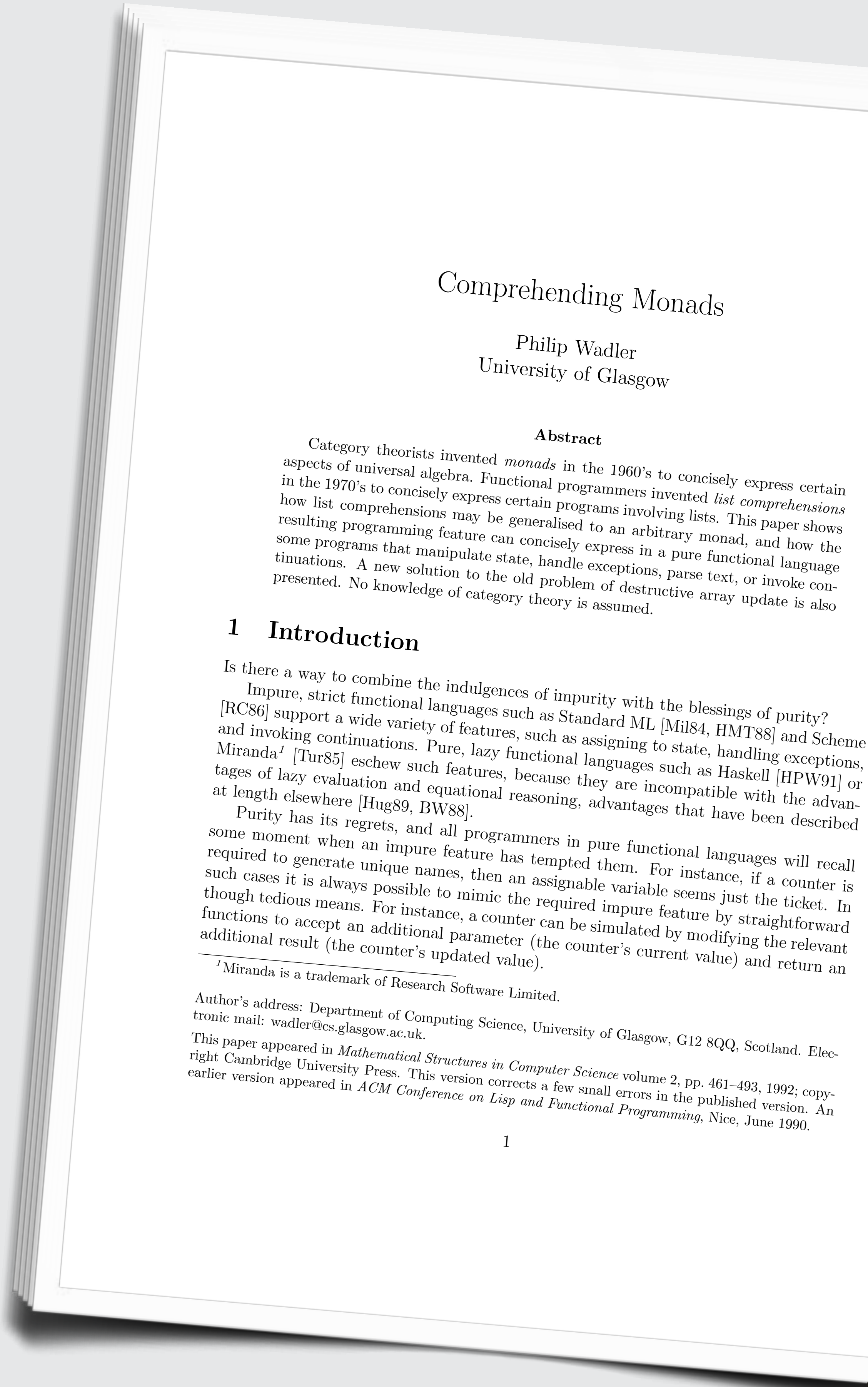
if $f\colon A \to TB$ and $c \in TA$, then $f^*(c) = \bigcup_{x \in c} f(x)$

- **side-effects** $TA = (A \times S)^S$

η_A is the map $a \mapsto (\lambda s\colon S.\langle a, s\rangle)$

if $f\colon A \to TB$ and $c \in TA$, then $f^*(c) = \lambda s\colon S.(\text{let } \langle a, s'\rangle = c(s) \text{ in } f(a)(s'))$

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



Comprehending Monads

Philip Wadler
University of Glasgow

Abstract

Category theorists invented *monads* in the 1960's to concisely express certain aspects of universal algebra. Functional programmers invented *list comprehensions* in the 1970's to concisely express certain programs involving lists. This paper shows how list comprehensions may be generalised to an arbitrary monad, and how the resulting programming feature can concisely express in a pure functional language some programs that manipulate state, handle exceptions, parse text, or invoke continuations. A new solution to the old problem of destructive array update is also presented. No knowledge of category theory is assumed.

1 Introduction

Is there a way to combine the indulgences of impurity with the blessings of purity?

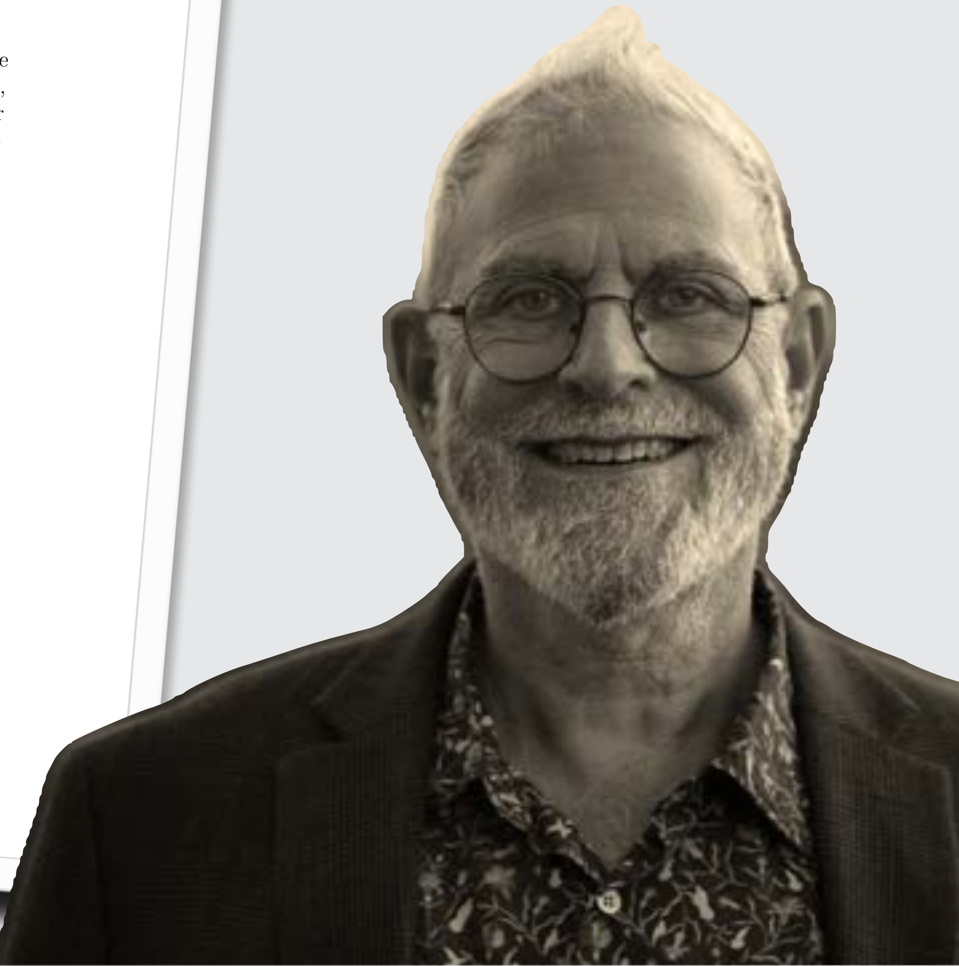
Impure, strict functional languages such as Standard ML [Mil84, HMT88] and Scheme [RC86] support a wide variety of features, such as assigning to state, handling exceptions, and invoking continuations. Pure, lazy functional languages such as Haskell [HPW91] or Miranda[1] [Tur85] eschew such features, because they are incompatible with the advantages of lazy evaluation and equational reasoning, advantages that have been described at length elsewhere [Hug89, BW88].

Purity has its regrets, and all programmers in pure functional languages will recall some moment when an impure feature has tempted them. For instance, if a counter is required to generate unique names, then an assignable variable seems just the ticket. In such cases it is always possible to mimic the required impure feature by straightforward though tedious means. For instance, a counter can be simulated by modifying the relevant functions to accept an additional parameter (the counter's current value) and return an additional result (the counter's updated value).

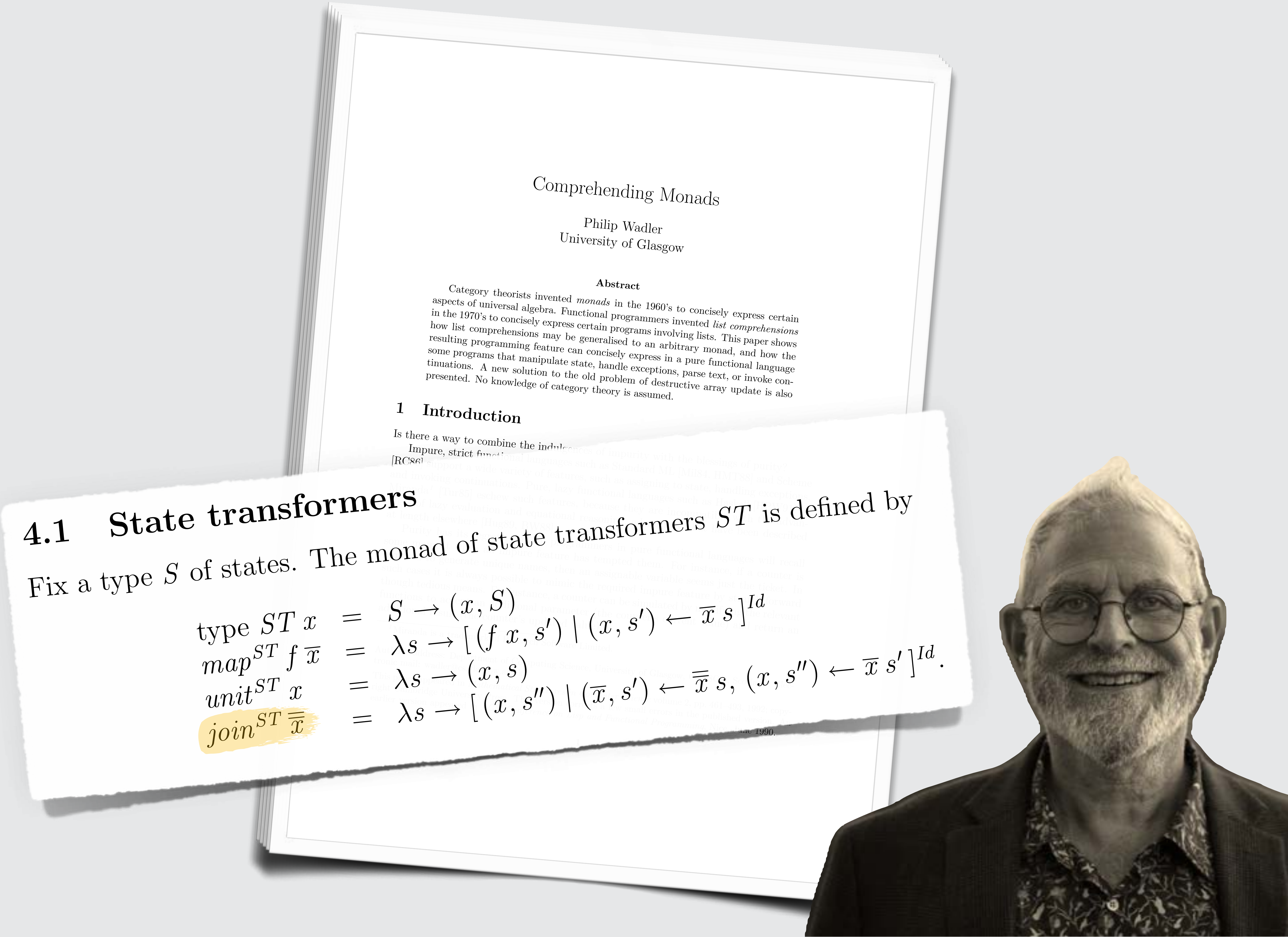
[1] Miranda is a trademark of Research Software Limited.

Author's address: Department of Computing Science, University of Glasgow, G12 8QQ, Scotland. Electronic mail: wadler@cs.glasgow.ac.uk.

This paper appeared in *Mathematical Structures in Computer Science* volume 2, pp. 461–493, 1992; copyright Cambridge University Press. This version corrects a few small errors in the published version. An earlier version appeared in *ACM Conference on Lisp and Functional Programming*, Nice, June 1990.



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



Comprehending Monads

Philip Wadler
University of Glasgow

Abstract

Category theorists invented *monads* in the 1960's to concisely express certain aspects of universal algebra. Functional programmers invented *list comprehensions* in the 1970's to concisely express certain programs involving lists. This paper shows how list comprehensions may be generalised to an arbitrary monad, and how the resulting programming feature can concisely express in a pure functional language some programs that manipulate state, handle exceptions, parse text, or invoke continuations. A new solution to the old problem of destructive array update is also presented. No knowledge of category theory is assumed.

1 Introduction

Is there a way to combine the indul

Impure, strict func

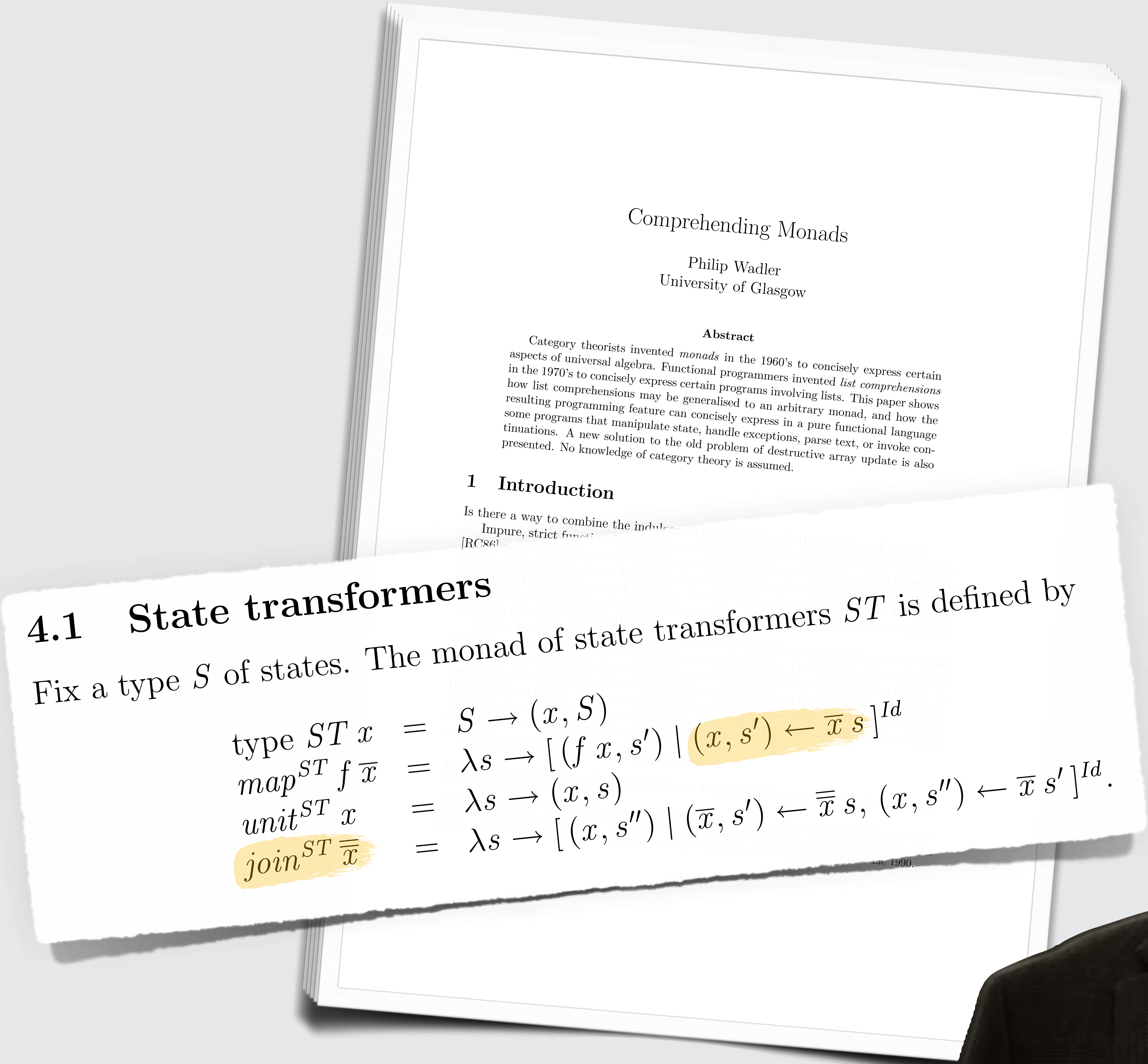
[RC86]

4.1 State transformers

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

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

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



Comprehending Monads

Philip Wadler
University of Glasgow

Abstract

Category theorists invented *monads* in the 1960's to concisely express certain aspects of universal algebra. Functional programmers invented *list comprehensions* in the 1970's to concisely express certain programs involving lists. This paper shows how list comprehensions may be generalised to an arbitrary monad, and how the resulting programming feature can concisely express in a pure functional language some programs that manipulate state, handle exceptions, parse text, or invoke continuations. A new solution to the old problem of destructive array update is also presented. No knowledge of category theory is assumed.

1 Introduction

Is there a way to combine the indul

Impure, strict fun

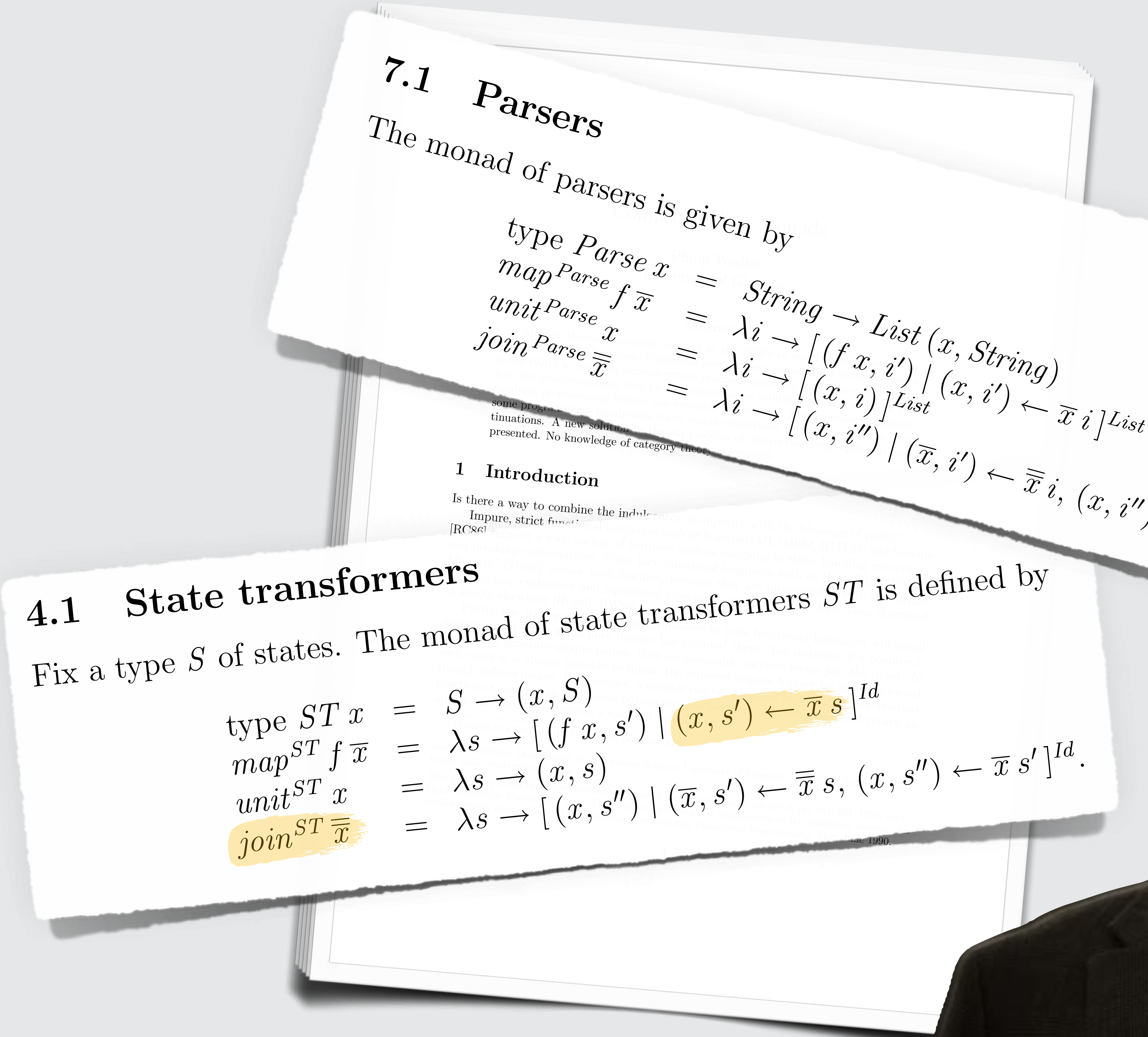
[RC86]

4.1 State transformers

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

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

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



7.1 Parsers

The monad of parsers is given by

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

4.1 State transformers

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

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

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

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

monad

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

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

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

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

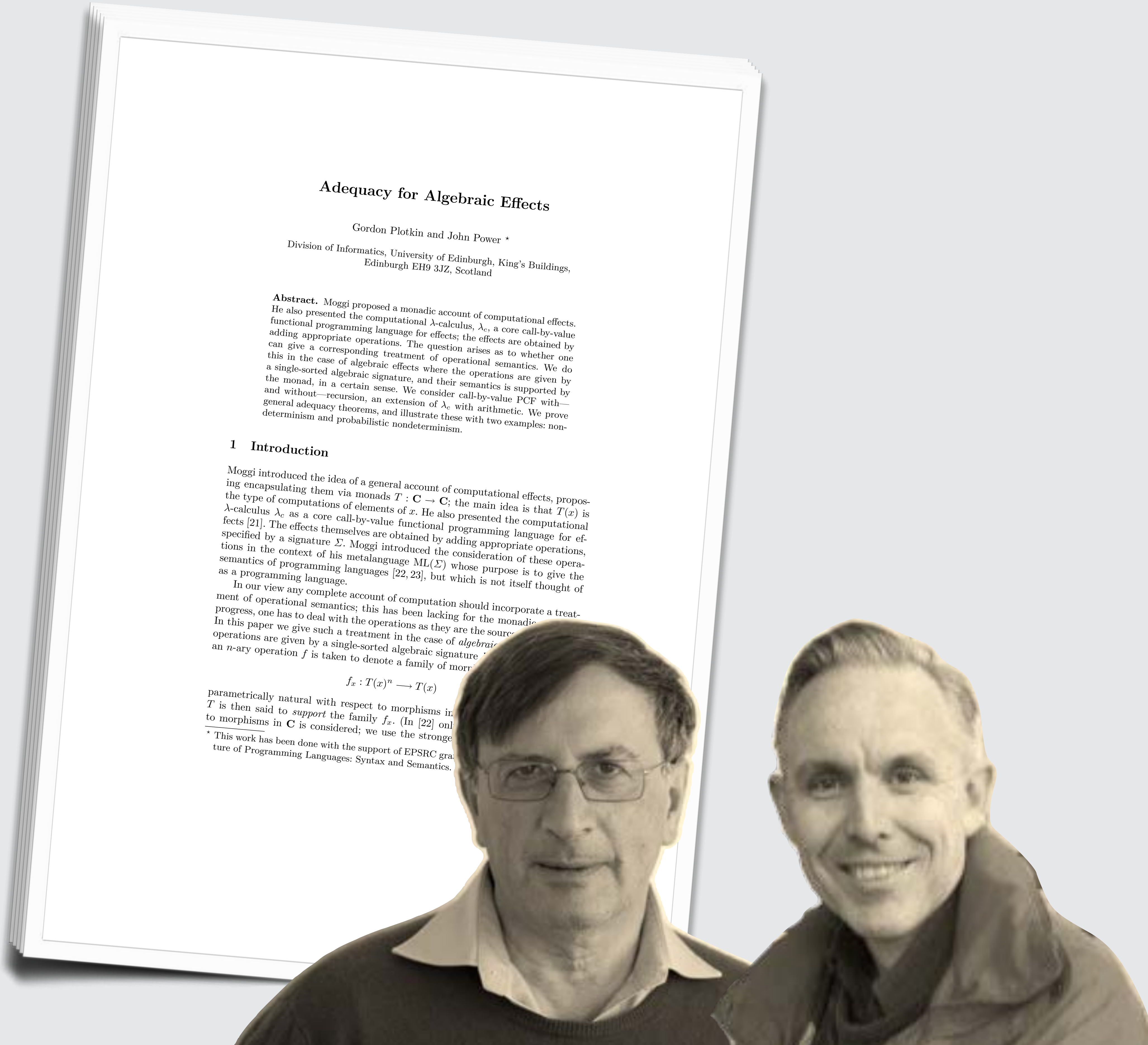
$$\texttt{fail} : TX$$

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

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

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

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



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 $T : \mathbf{C} \to \mathbf{C}$; 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(Σ) 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 treat-
ment of operational semantics; this has been lacking for the monadic
progress, one has to deal with the operations as they are the sourc
In this paper we give such a treatment in the case of *algebraic*
operations are given by a single-sorted algebraic signature
an n-ary operation f is taken to denote a family of morr

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

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

[*] This work has been done with the support of EPSRC gra
ture of Programming Languages: Syntax and Semantics.

An effect is specified with **operations** and **equations**

An effect is specified with **operations** and **equations**

operations

$$\begin{aligned} \mathtt{fail} &: 0 \\ \mathtt{choose} &: 2 \end{aligned}$$

An effect is specified with **operations** and **equations**

operations

$$\texttt{fail} : 0$$
$$\texttt{choose} : 2$$

equations

$$\texttt{choose}\,(\texttt{choose}\,M\,N)\,P = \texttt{choose}\,M\,(\texttt{choose}\,N\,P)$$
$$\texttt{choose}\,M\,N = \texttt{choose}\,N\,M$$
$$\texttt{choose}\,M\,M = M$$
$$\texttt{choose}\,\texttt{fail}\,M = M = \texttt{choose}\,M\,\texttt{fail}$$

An effect is specified with **operations** and **equations**

operations

$$\begin{aligned} \mathtt{fail} &: 0 \\ \mathtt{choose} &: 2 \end{aligned}$$

equations

$$\begin{aligned} \mathtt{choose}\,(\mathtt{choose}\,M\,N)\,P &= \mathtt{choose}\,M\,(\mathtt{choose}\,N\,P) \\ \mathtt{choose}\,M\,N &= \mathtt{choose}\,N\,M \\ \mathtt{choose}\,M\,M &= M \\ \mathtt{choose}\,\mathtt{fail}\,M &= M = \mathtt{choose}\,M\,\mathtt{fail} \end{aligned}$$

$$\mathtt{chs}\,B\,\big(\mathtt{chs}\,(\mathtt{chs}\,A\,C)\,(\mathtt{chs}\,B\,\mathtt{fail})\big)$$

An effect is specified with **operations** and **equations**

operations

$$\begin{aligned} \mathtt{fail} &: 0 \\ \mathtt{choose} &: 2 \end{aligned}$$

equations

$$\begin{aligned} \mathtt{choose}\,(\mathtt{choose}\,M\,N)\,P &= \mathtt{choose}\,M\,(\mathtt{choose}\,N\,P) \\ \mathtt{choose}\,M\,N &= \mathtt{choose}\,N\,M \\ \mathtt{choose}\,M\,M &= M \\ \mathtt{choose}\,\mathtt{fail}\,M = M &= \mathtt{choose}\,M\,\mathtt{fail} \end{aligned}$$

$$\begin{aligned} &\mathtt{chs}\,B\,\big(\mathtt{chs}\,(\mathtt{chs}\,A\,C)\,(\mathtt{chs}\,B\,\mathtt{fail})\big) \\ &= \mathtt{chs}\,B\,\Big(\mathtt{chs}\,A\,\big(\mathtt{chs}\,C\,(\mathtt{chs}\,B\,\mathtt{fail})\big)\Big) \end{aligned}$$

An effect is specified with **operations** and **equations**

operations

$$\begin{aligned} \mathtt{fail} &: 0 \\ \mathtt{choose} &: 2 \end{aligned}$$

equations

$$\begin{aligned} \mathtt{choose}\,(\mathtt{choose}\,M\,N)\,P &= \mathtt{choose}\,M\,(\mathtt{choose}\,N\,P) \\ \mathtt{choose}\,M\,N &= \mathtt{choose}\,N\,M \\ \mathtt{choose}\,M\,M &= M \\ \mathtt{choose}\,\mathtt{fail}\,M &= M = \mathtt{choose}\,M\,\mathtt{fail} \end{aligned}$$

$$\begin{aligned} &\mathsf{chs}\,B\left(\mathsf{chs}\,(\mathsf{chs}\,A\,C)\,(\mathsf{chs}\,B\,\mathtt{fail})\right) \\ &= \mathsf{chs}\,B\left(\mathsf{chs}\,A\left(\mathsf{chs}\,C\,(\mathsf{chs}\,B\,\mathtt{fail})\right)\right) \\ &= \mathsf{chs}\,\mathtt{fail}\left(\mathsf{chs}\,A\left(\mathsf{chs}\,B\,(\mathsf{chs}\,B\,C)\right)\right) \end{aligned}$$

An effect is specified with **operations** and **equations**

operations

$$\mathtt{fail} : 0$$
$$\mathtt{choose} : 2$$

equations

$$\begin{aligned}
\mathtt{choose}\,(\mathtt{choose}\,M\,N)\,P &= \mathtt{choose}\,M\,(\mathtt{choose}\,N\,P)\\
\mathtt{choose}\,M\,N &= \mathtt{choose}\,N\,M\\
\mathtt{choose}\,M\,M &= M\\
\mathtt{choose}\,\mathtt{fail}\,M &= M = \mathtt{choose}\,M\,\mathtt{fail}
\end{aligned}$$

$$\begin{aligned}
&\mathtt{chs}\,B\,\big(\mathtt{chs}\,(\mathtt{chs}\,A\,C)\,(\mathtt{chs}\,B\,\mathtt{fail})\big)\\
&= \mathtt{chs}\,B\,\Big(\mathtt{chs}\,A\,\big(\mathtt{chs}\,C\,(\mathtt{chs}\,B\,\mathtt{fail})\big)\Big)\\
&= \mathtt{chs}\,\mathtt{fail}\,\Big(\mathtt{chs}\,A\,\big(\mathtt{chs}\,B\,(\mathtt{chs}\,B\,C)\big)\Big)\\
&= \mathtt{chs}\,A\,(\mathtt{chs}\,B\ C) \approx \{A, B, C\}
\end{aligned}$$

Exception **handling failed** to be algebraic

	operations
exceptions	`fail` `try`
state	`get` `set`
choice	`choose`
I/O	`read` `write`
probability	`flip`

not algebraic

Why handling is **not** an **algebraic** operation?

Why handling is **not** an **algebraic** operation?

handling

$$\mathtt{try}(\mathtt{fail}, M) = M$$

$$\mathtt{try}(\mathtt{val}\ V, M) = \mathtt{val}\ V$$

Why handling is **not** an **algebraic** operation?

handling

$$\mathtt{try}(\mathtt{fail}, M) = M$$

$$\mathtt{try}(\mathtt{val}\ V, M) = \mathtt{val}\ V$$

algebraicity

$$\begin{aligned}&\mathtt{do}\ x \Leftarrow \mathtt{try}(M_1, M_2)\ \mathtt{in}\ N\\ &= \mathtt{try}(\mathtt{do}\ x \Leftarrow M_1\ \mathtt{in}\ N, \mathtt{do}\ x \Leftarrow M_2\ \mathtt{in}\ N)\end{aligned}$$

Why handling is not an algebraic operation?

handling

$$\mathtt{try}(\mathtt{fail}, M) = M$$

$$\mathtt{try}(\mathtt{val}\, V, M) = \mathtt{val}\, V$$

algebraicity

$$\mathtt{do}\, x \Leftarrow \mathtt{try}(M_1, M_2)\, \mathtt{in}\, N$$

$$= \mathtt{try}(\mathtt{do}\, x \Leftarrow M_1\, \mathtt{in}\, N, \mathtt{do}\, x \Leftarrow M_2\, \mathtt{in}\, N)$$

$$\mathtt{do}\, x \Leftarrow \mathtt{val}\, 0\, \mathtt{in}\, N$$

Why handling is not an algebraic operation?

handling

$$\mathtt{try}(\mathtt{fail}, M) = M$$

$$\mathtt{try}(\mathtt{val}\ V, M) = \mathtt{val}\ V$$

algebraicity

$$\mathtt{do}\ x \Leftarrow \mathtt{try}(M_1, M_2)\ \mathtt{in}\ N$$
$$= \mathtt{try}(\mathtt{do}\ x \Leftarrow M_1\ \mathtt{in}\ N, \mathtt{do}\ x \Leftarrow M_2\ \mathtt{in}\ N)$$

$$\mathtt{do}\ x \Leftarrow \mathtt{val}\ 0\ \mathtt{in}\ N$$
$$= \mathtt{do}\ x \Leftarrow \mathtt{try}(\mathtt{val}\ 0, \mathtt{val}\ 1)\ \mathtt{in}\ N$$

Why handling is **not** an **algebraic** operation?

handling

$$\mathtt{try}(\mathtt{fail}, M) = M$$

$$\mathtt{try}(\mathtt{val}\ V, M) = \mathtt{val}\ V$$

algebraicity

$$\begin{aligned} &\mathtt{do}\ x \Leftarrow \mathtt{try}(M_1, M_2)\ \mathtt{in}\ N \\ &= \mathtt{try}(\mathtt{do}\ x \Leftarrow M_1\ \mathtt{in}\ N, \mathtt{do}\ x \Leftarrow M_2\ \mathtt{in}\ N) \end{aligned}$$

$$\begin{aligned} &\mathtt{do}\ x \Leftarrow \mathtt{val}\ 0\ \mathtt{in}\ N \\ &= \mathtt{do}\ x \Leftarrow \mathtt{try}(\mathtt{val}\ 0, \mathtt{val}\ 1)\ \mathtt{in}\ N \\ &= \mathtt{try}(\mathtt{do}\ x \Leftarrow \mathtt{val}\ 0\ \mathtt{in}\ N, \mathtt{do}\ x \Leftarrow \mathtt{val}\ 1\ \mathtt{in}\ N) \end{aligned}$$

Why handling is **not** an **algebraic** operation?

handling

$$\mathtt{try}(\mathtt{fail}, M) = M$$

$$\mathtt{try}(\mathtt{val}\, V, M) = \mathtt{val}\, V$$

algebraicity

$$\begin{aligned}&\mathtt{do}\, x \Leftarrow \mathtt{try}(M_1, M_2)\,\mathtt{in}\, N\\ &= \mathtt{try}(\mathtt{do}\, x \Leftarrow M_1\,\mathtt{in}\, N, \mathtt{do}\, x \Leftarrow M_2\,\mathtt{in}\, N)\end{aligned}$$

$$\begin{aligned}&\boxed{\mathtt{do}\, x \Leftarrow \mathtt{val}\, 0\,\mathtt{in}\, N} \longleftarrow \text{failing}\\ &= \mathtt{do}\, x \Leftarrow \mathtt{try}(\mathtt{val}\, 0, \mathtt{val}\, 1)\,\mathtt{in}\, N\\ &= \mathtt{try}(\mathtt{do}\, x \Leftarrow \mathtt{val}\, 0\,\mathtt{in}\, N, \mathtt{do}\, x \Leftarrow \mathtt{val}\, 1\,\mathtt{in}\, N)\end{aligned}$$

Why handling is not an algebraic operation?

handling

$$\mathtt{try}(\mathtt{fail}, M) = M$$

$$\mathtt{try}(\mathtt{val}\ V, M) = \mathtt{val}\ V$$

algebraicity

$$\mathtt{do}\ x \Leftarrow \mathtt{try}(M_1, M_2)\ \mathtt{in}\ N$$

$$= \mathtt{try}(\mathtt{do}\ x \Leftarrow M_1\ \mathtt{in}\ N, \mathtt{do}\ x \Leftarrow M_2\ \mathtt{in}\ N)$$

$$\boxed{\mathtt{do}\ x \Leftarrow \mathtt{val}\ 0\ \mathtt{in}\ N} \longleftarrow \text{failing}$$

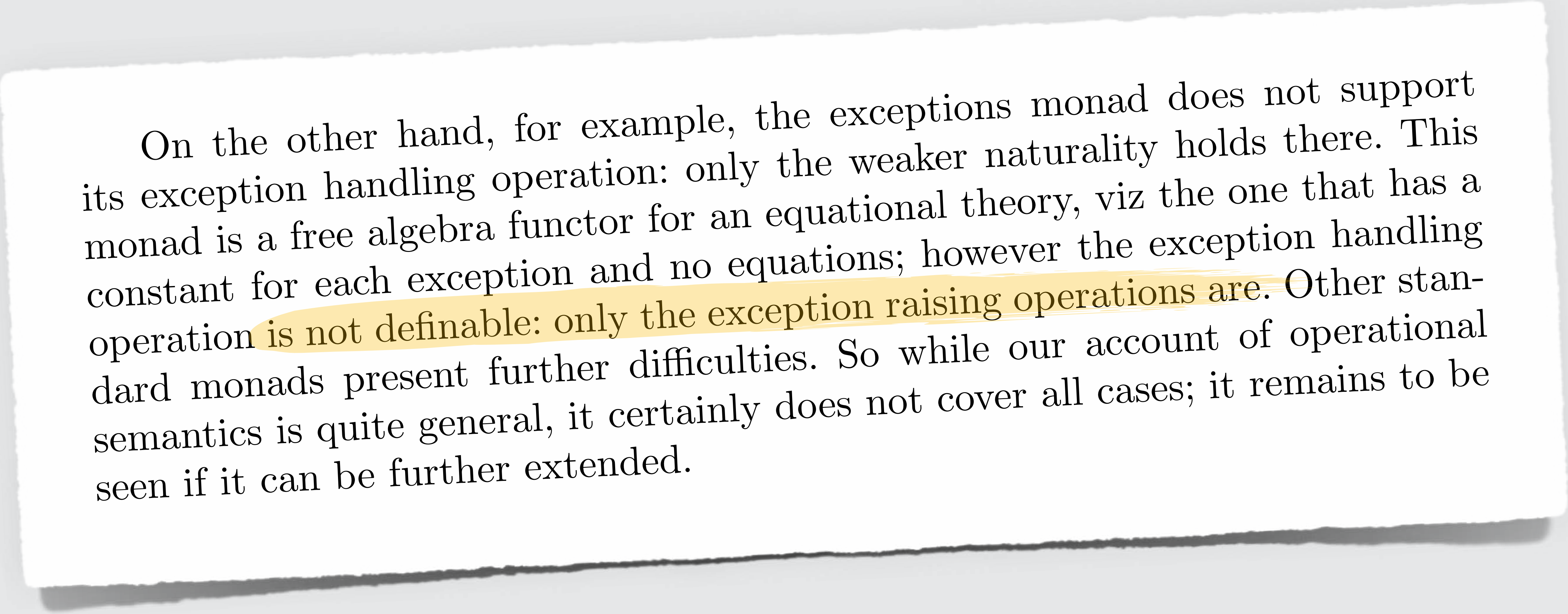
$$= \mathtt{do}\ x \Leftarrow \mathtt{try}(\mathtt{val}\ 0, \mathtt{val}\ 1)\ \mathtt{in}\ N$$

$$= \mathtt{try}(\mathtt{do}\ x \Leftarrow \mathtt{val}\ 0\ \mathtt{in}\ N, \mathtt{do}\ x \Leftarrow \mathtt{val}\ 1\ \mathtt{in}\ N)$$

$$= \mathtt{do}\ x \Leftarrow \mathtt{val}\ 1\ \mathtt{in}\ N$$

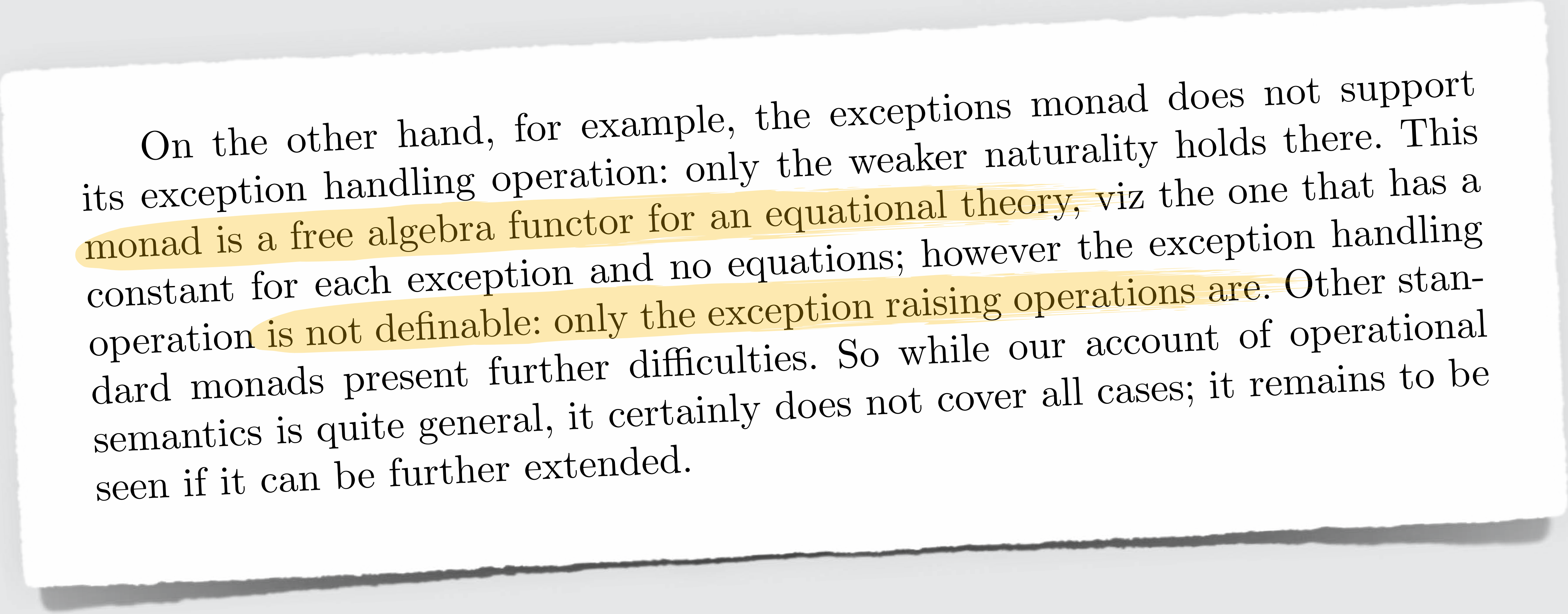
Exception handling indicated a **different nature**

Exception handling indicated a **different nature**



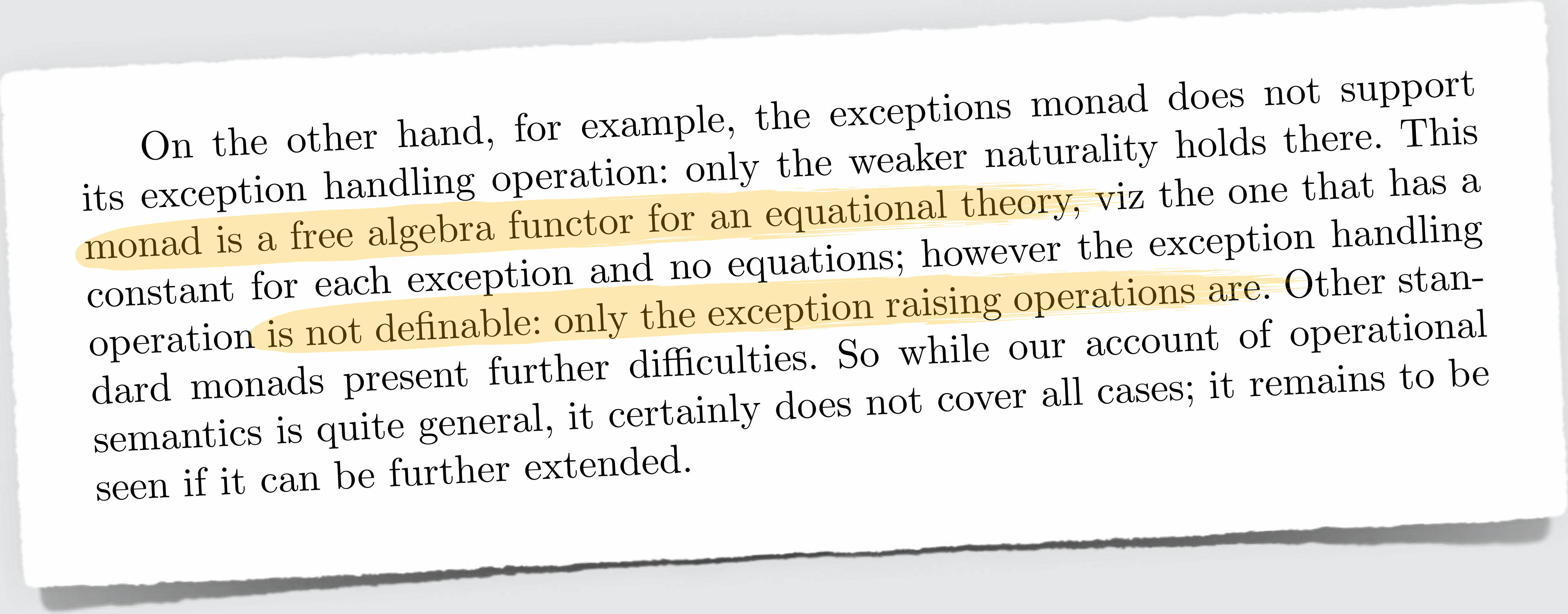
On the other hand, for example, the exceptions monad does not support its exception handling operation: only the weaker naturality holds there. This monad is a free algebra functor for an equational theory, viz the one that has a constant for each exception and no equations; however the exception handling 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.

Exception handling indicated a **different nature**

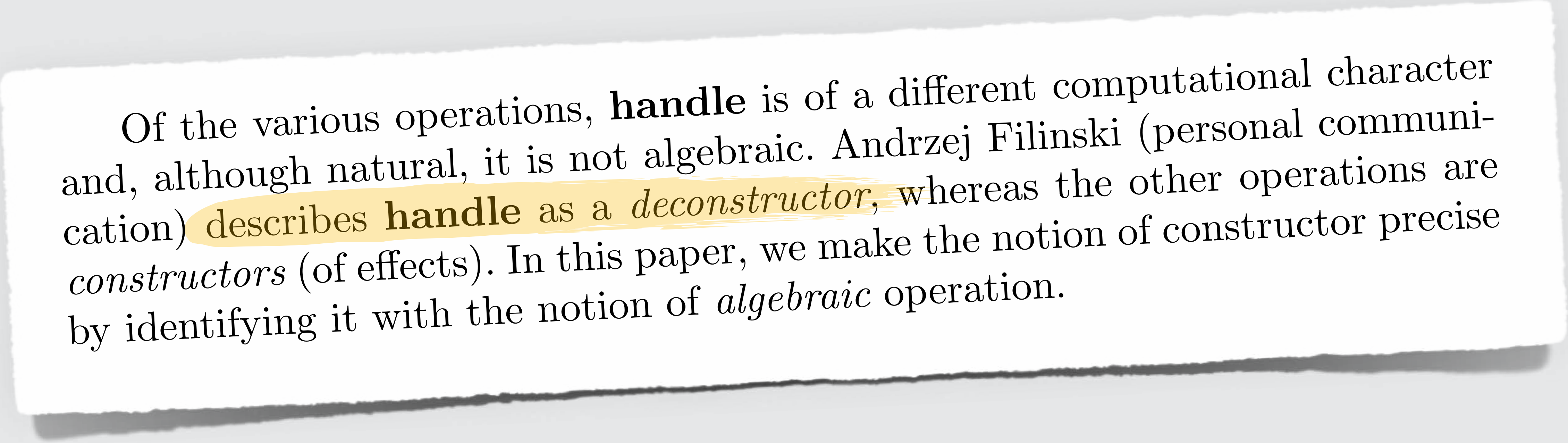


On the other hand, for example, the exceptions monad does not support its exception handling operation: only the weaker naturality holds there. This monad is a free algebra functor for an equational theory, viz the one that has a constant for each exception and no equations; however the exception handling 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.

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.

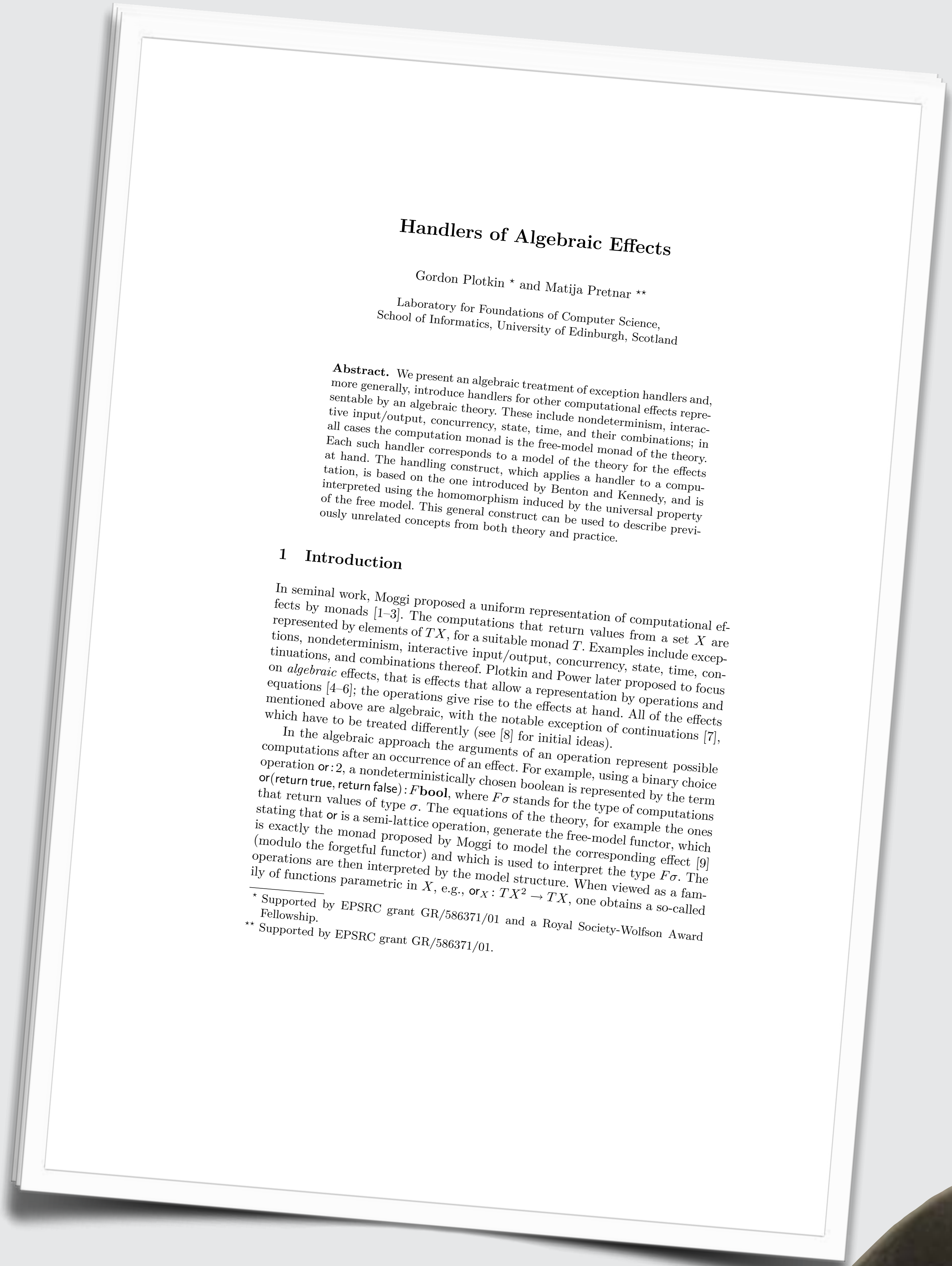
Mathematics was already suggesting **unrevealed constructs**

	constructors	**deconstructors**
exceptions	`fail`	`try`
state	`get set`	
choice	`choose`	
I/O	`read write`	
probability	`flip`	

Mathematics was already suggesting **unrevealed constructs**

	constructors	deconstructors
exceptions	`fail`	`try`
state	`get set`	?
choice	`choose`	
I/O	`read write`	
probability	`flip`	

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

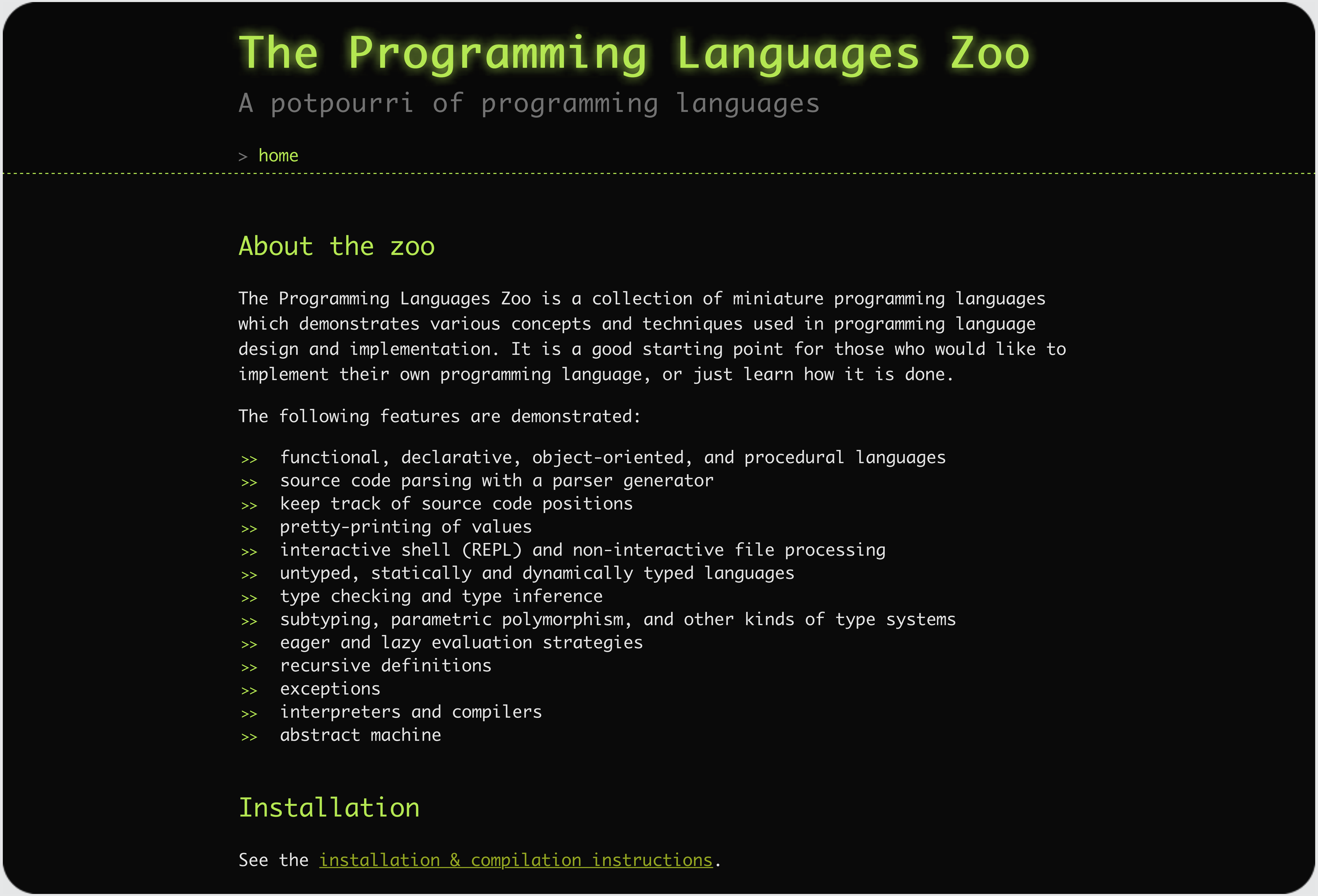
In seminal work, Moggi proposed a uniform representation of computational effects by monads [1–3]. The computations that return values from a set X are represented by elements of TX, for a suitable monad T. Examples include exceptions, nondeterminism, interactive input/output, concurrency, state, time, 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 $\mathsf{or}: 2$, a nondeterministically chosen boolean is represented by the term $\mathsf{or}(\mathsf{return\ true}, \mathsf{return\ false}) : F\mathbf{bool}$, 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**



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.

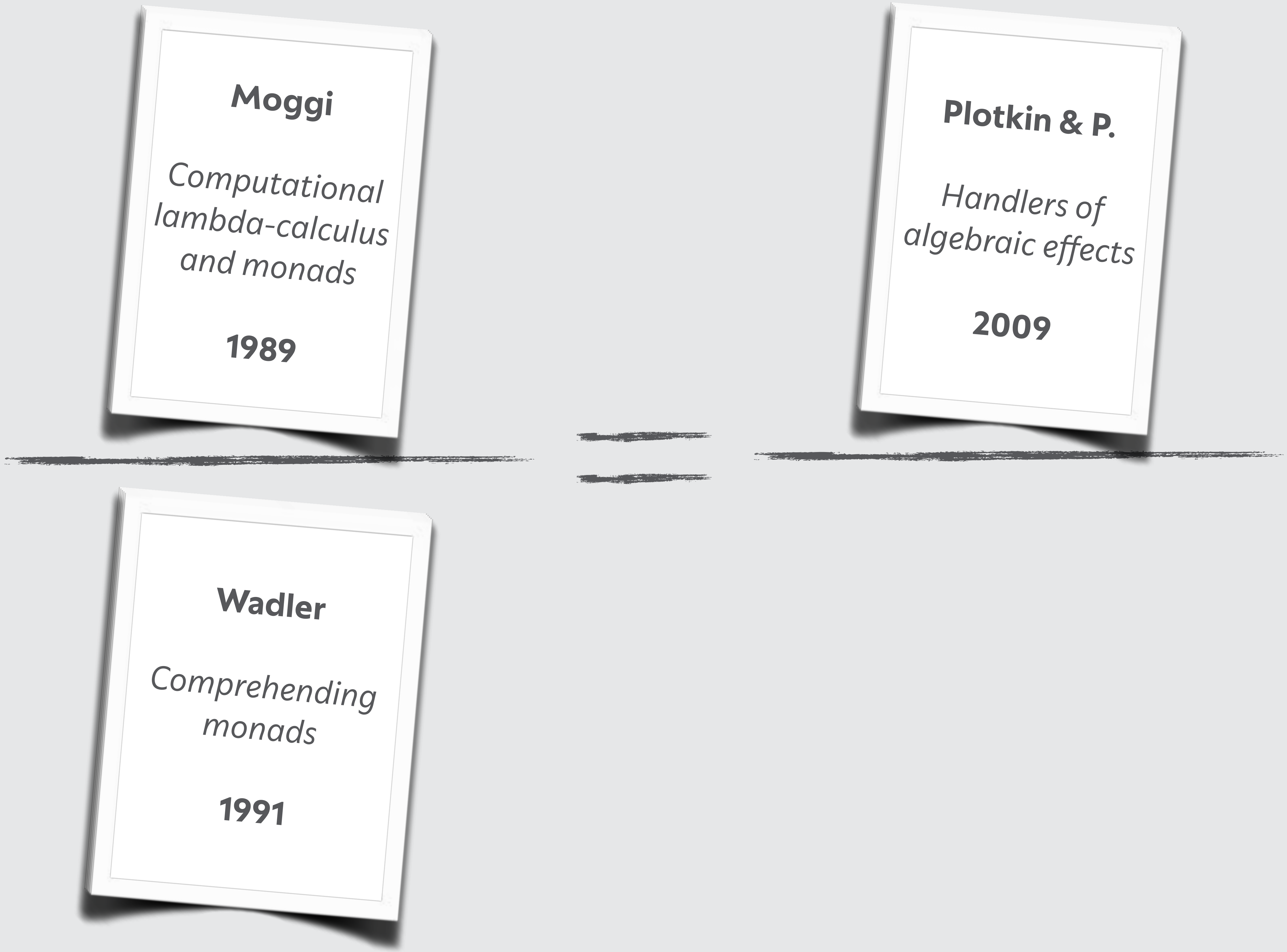
The following features are demonstrated:

- 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

Wadler

Comprehending monads

1991

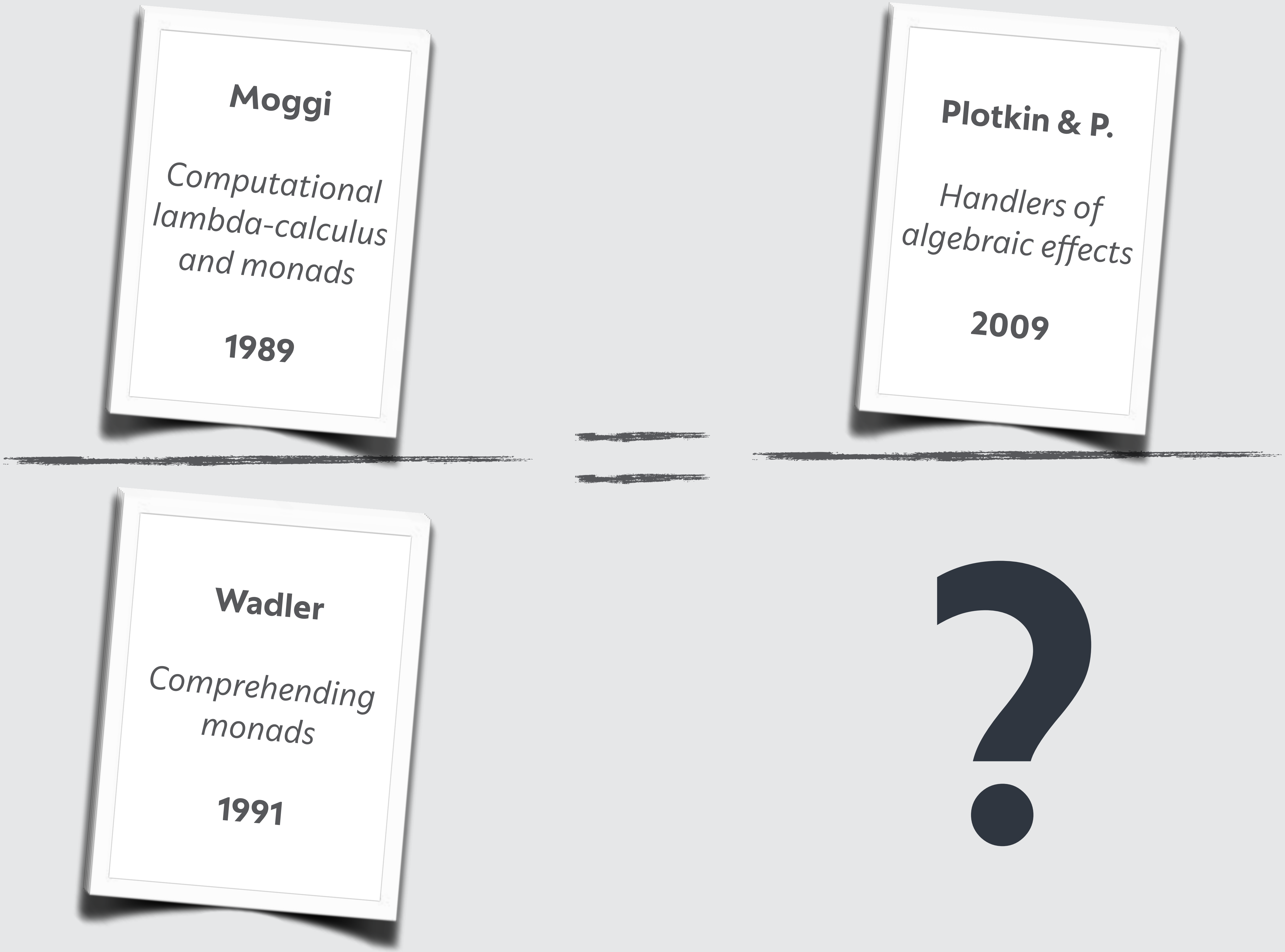
=

Plotkin & P.

Handlers of algebraic effects

2009

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



Moggi

Computational lambda-calculus and monads

1989

Wadler

Comprehending monads

1991

=

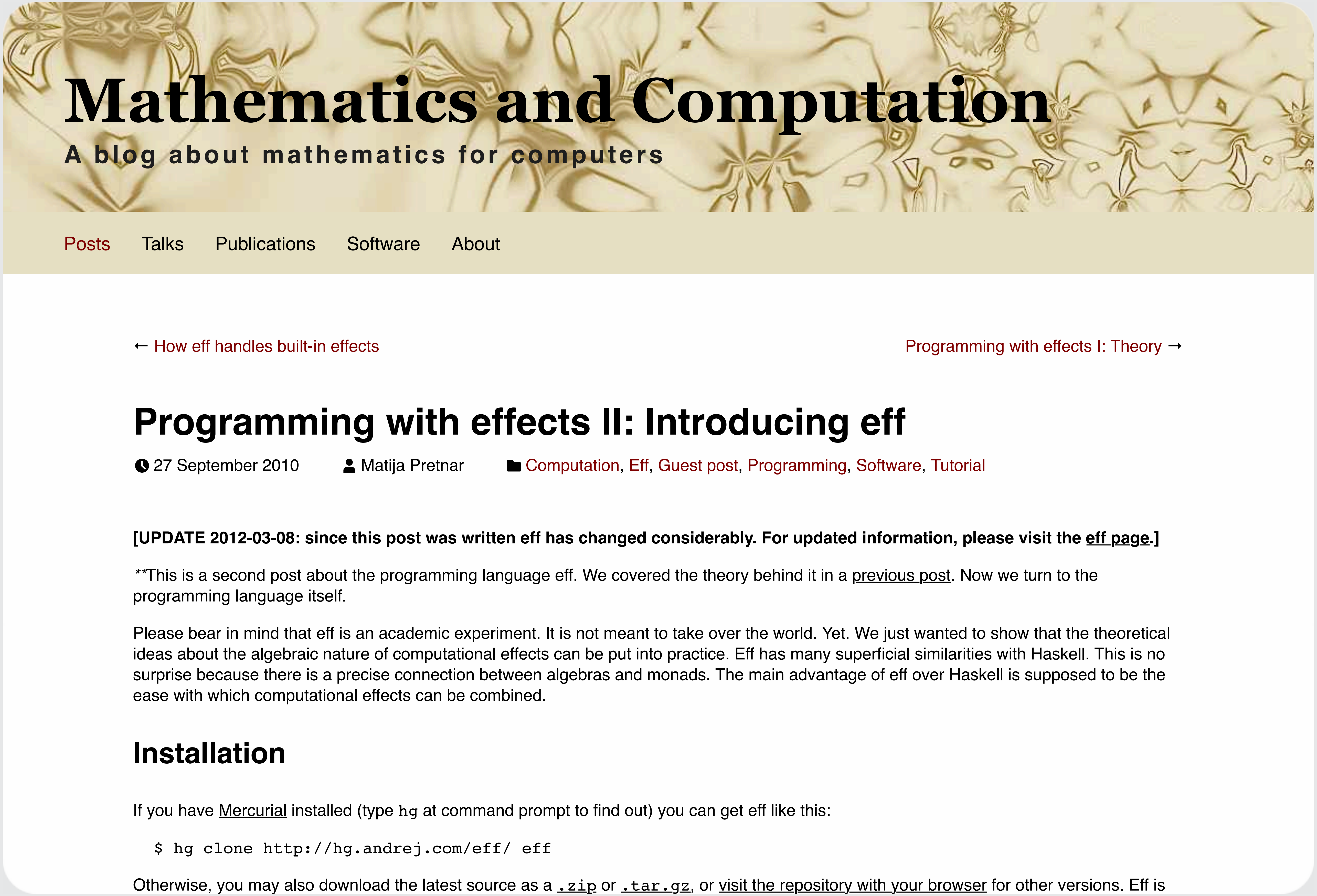
Plotkin & P.

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

← How eff handles built-in effects

Programming with effects I: Theory →

Programming with effects II: Introducing eff

27 September 2010 · Matija Pretnar · Computation, Eff, Guest post, Programming, Software, Tutorial

[UPDATE 2012-03-08: since this post was written eff has changed considerably. For updated information, please visit the eff page.]

**This is a second post about the programming language eff. We covered the theory behind it in a previous post. Now we turn to the programming language itself.

Please bear in mind that eff is an academic experiment. It is not meant to take over the world. Yet. We just wanted to show that the theoretical ideas about the algebraic nature of computational effects can be put into practice. Eff has many superficial similarities with Haskell. This is no surprise because there is a precise connection between algebras and monads. The main advantage of eff over Haskell is supposed to be the ease with which computational effects can be combined.

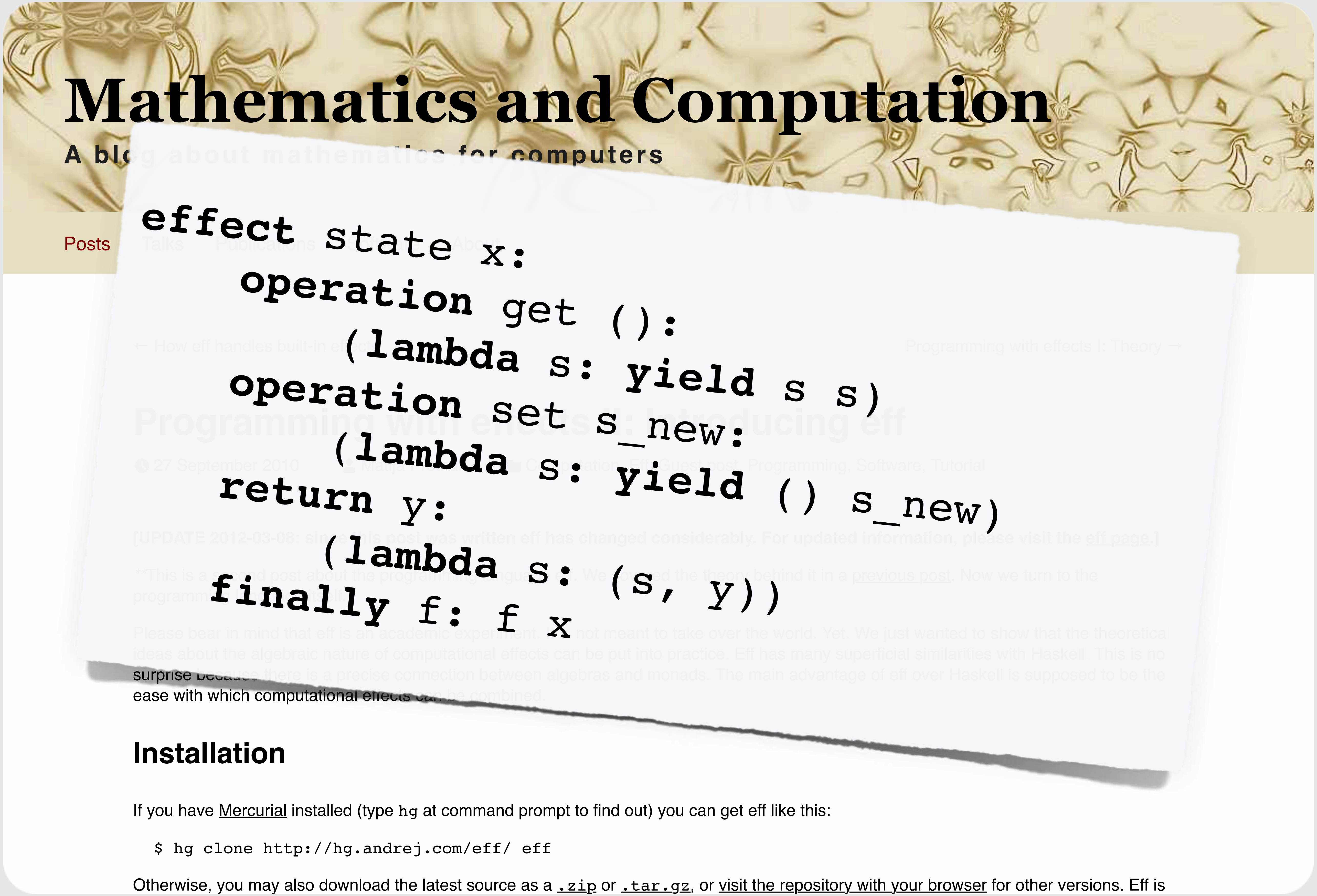
Installation

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

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



Mathematics and Computation

A blog about mathematics for computers

Posts Talks Publications About

```
effect state x:
    operation get ():
        (lambda s: yield s s)
    operation set s_new:
        (lambda s: yield () s_new)
    return y:
        (lambda s: (s, y))
    finally f: f x
```

← How eff handles built-in effects   Programming with effects I: Theory →

Programming with effects II: Introducing eff

27 September 2010   Matija Pretnar   Computation, Eff, Guest post, Programming, Software, Tutorial

[UPDATE 2012-03-08: since this post was written eff has changed considerably. For updated information, please visit the eff page.]

**This is a second post about the programming language eff. We covered the theory behind it in a previous post. Now we turn to the programming side of things.

Please bear in mind that eff is an academic experiment. It is not meant to take over the world. Yet. We just wanted to show that the theoretical ideas about the algebraic nature of computational effects can be put into practice. Eff has many superficial similarities with Haskell. This is no surprise because there is a precise connection between algebras and monads. The main advantage of eff over Haskell is supposed to be the ease with which computational effects can be combined.

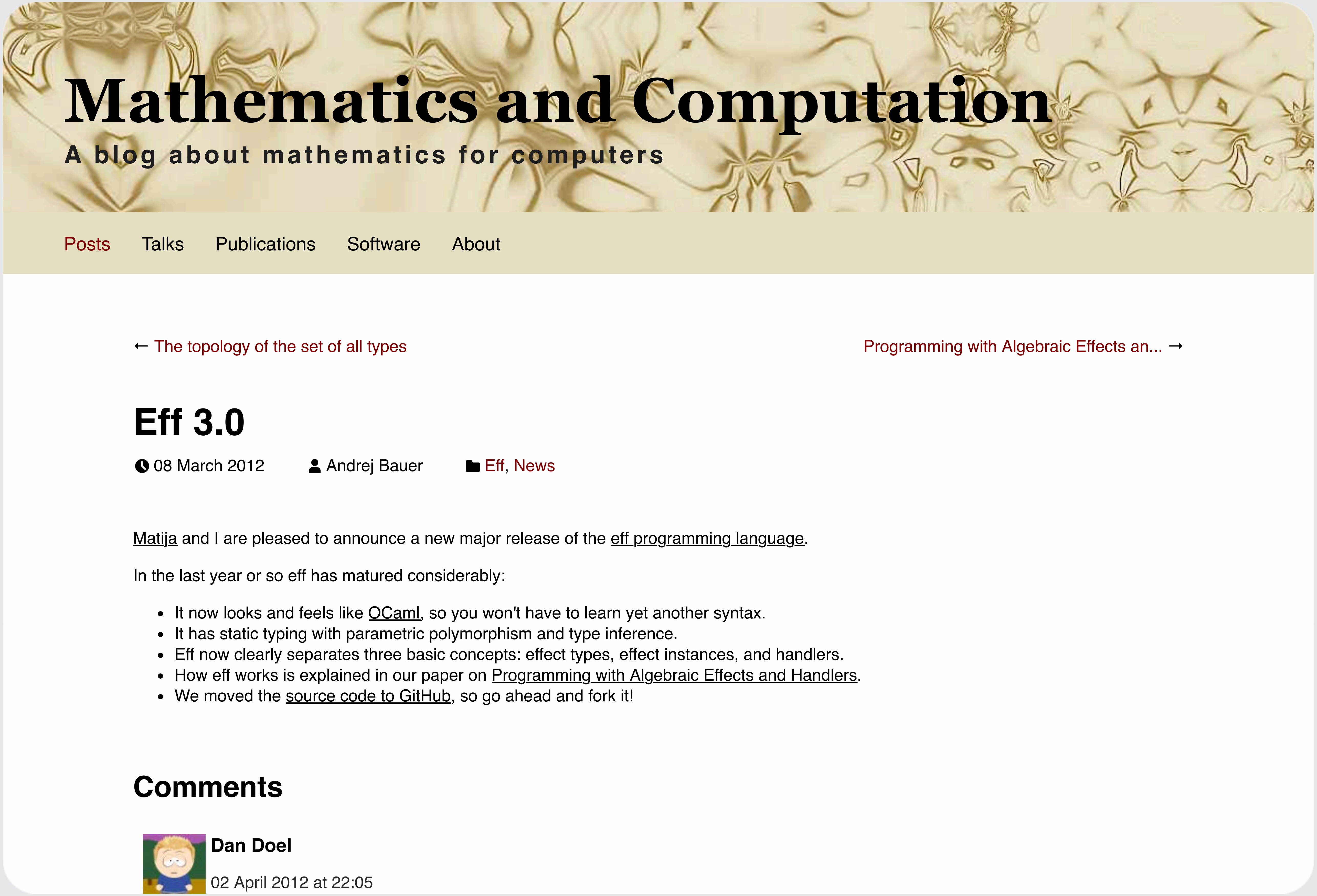
Installation

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**



Mathematics and Computation

A blog about mathematics for computers

Posts Talks Publications Software About

← The topology of the set of all types

Programming with Algebraic Effects an... →

Eff 3.0

08 March 2012 Andrej Bauer Eff, News

Matija and I are pleased to announce a new major release of the eff programming language.

In the last year or so eff has matured considerably:

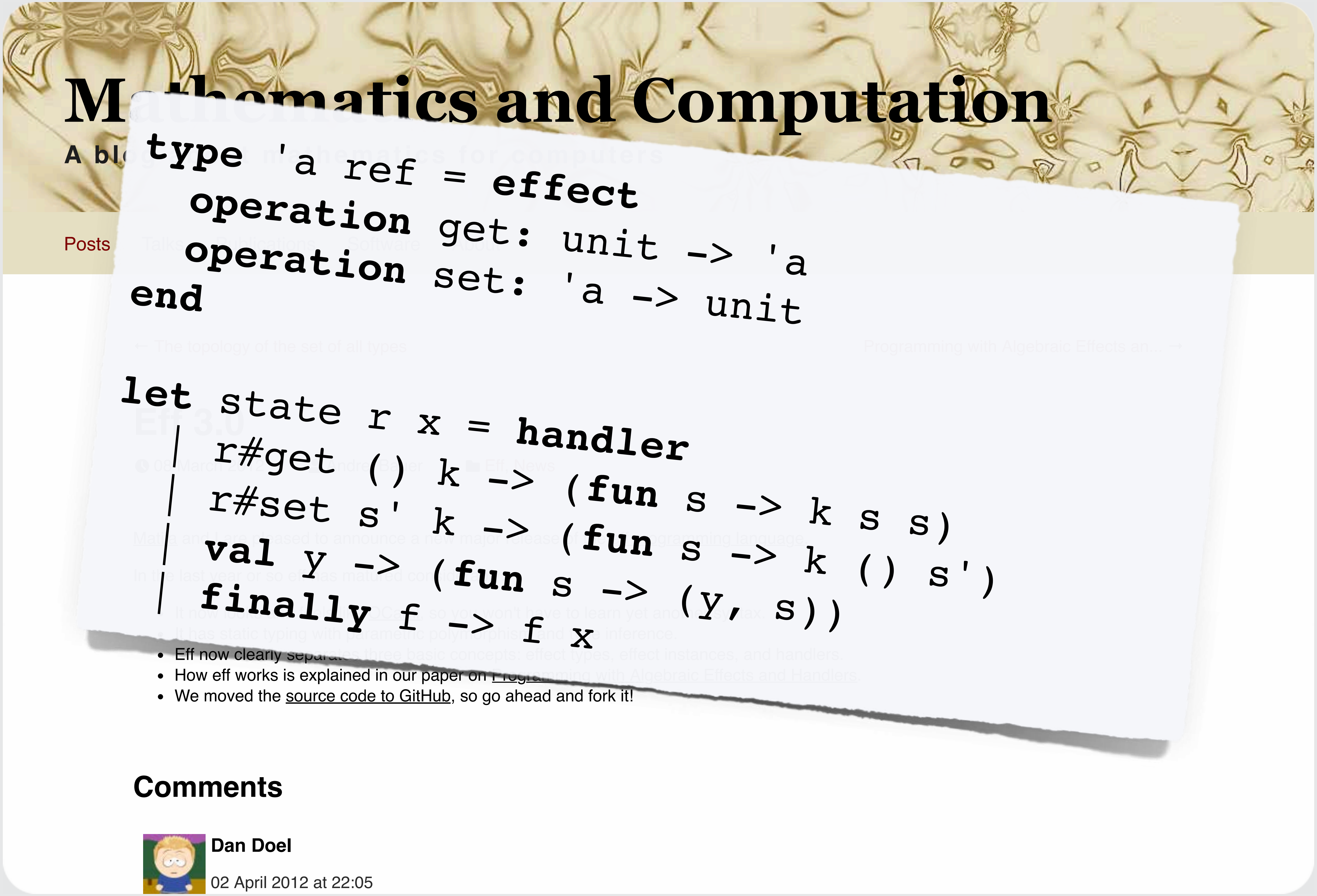
- It now looks and feels like OCaml, so you won't have to learn yet another syntax.
- It has static typing with parametric polymorphism and type inference.
- Eff now clearly separates three basic concepts: effect types, effect instances, and handlers.
- How eff works is explained in our paper on Programming 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

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



Mathematics and Computation

A blog about mathematics for computers

Posts Talks Publications Software About

```
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
```

← The topology of the set of all types

Programming with Algebraic Effects an... →

Eff 3.0

08 March 2012 Andrej Bauer Eff, News

Matija and I are pleased to announce a new major release of eff programming language.

In the last year or so eff has matured con...

- It now ... OCaml, so you won't have to learn yet another syntax.
- It has static typing with parametric polymorphism and type inference.
- 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

1989

Wadler

Comprehending monads

1991

Plotkin & P.

Handlers of algebraic effects

2009

Journal of Logical and Algebraic Methods in Programming 84 (2015) 108–123

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Programming with algebraic effects and handlers

Andrej Bauer, Matija Pretnar *

Faculty of Mathematics and Physics, University of Ljubljana, Slovenia

ARTICLE INFO

Article history:
Received 29 February 2012
Received in revised form 12 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 from free algebras. *Eff* supports first-class effects and handlers through which we may easily define new computational effects, seamlessly combine existing ones, and handle them in novel ways. We give a denotational semantics of *Eff* and discuss a prototype implementation based on it. Through examples we demonstrate how the standard effects are treated in *Eff*, and how *Eff* supports programming techniques that use various forms of delimited continuations, such as backtracking, breadth-first search, selection functionals, cooperative multi-threading, and others.

0. Introduction

Eff is a programming language based on the algebraic approach to effects, in which computational effects are mode
operations of a suitably chosen algebraic theory [12]. Common computational effects such as input, output, state, exce
and nondeterminism, are of this kind. Continuations are not algebraic [4], but they turn out to be naturally supporte
as well. Effect handlers are a related notion [14,19] which encompasses exception handlers, stream redirection, tran
backtracking, and many others. These are modeled as homomorphisms induced by the universal property of free a

Each algebraic theory gives rise to a monad [1,11], although the operations cannot be reconstructed from it.
theories have their own virtues, though. Effects are combined more easily than monads [5], and the interaction
effects and handlers offers new ways of programming. An experiment in the design of a programming language
the algebraic approach therefore seems warranted.

Philip Wadler once opined [21] that monads as a programming concept would not have been discovered witho
category-theoretic counterparts, but once they were, programmers could live in blissful ignorance of their origin. Be
the same holds for algebraic effects and handlers, we streamlined the paper for the benefit of programmers, trustin
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 s
to *Eff*, Section 3 is devoted to type checking, in Section 4 we give a domain-theoretic semantics of *Eff*, and in Section
briefly discuss our prototype implementation. The examples in Section 6 demonstrate how effects and handlers c
used to produce standard computational effects, such as exceptions, state, input and output, as well as their variatio
combinations. Further examples show how *Eff*'s delimited control capabilities are used for nondeterministic a
choice, backtracking, selection functionals, and cooperative multi-threading. We conclude with though
work.

The implementation of *Eff* is freely available at http://www.eff-lang.org/.

* Corresponding author.
E-mail addresses: andrej@andrej.com (A. Bauer), matija@pretnar.info (M.

http://dx.doi.org/10.1016/j.jlamp.2014.02.001
2352-2208/© 2014 Elsevier Inc. All rights reserved.

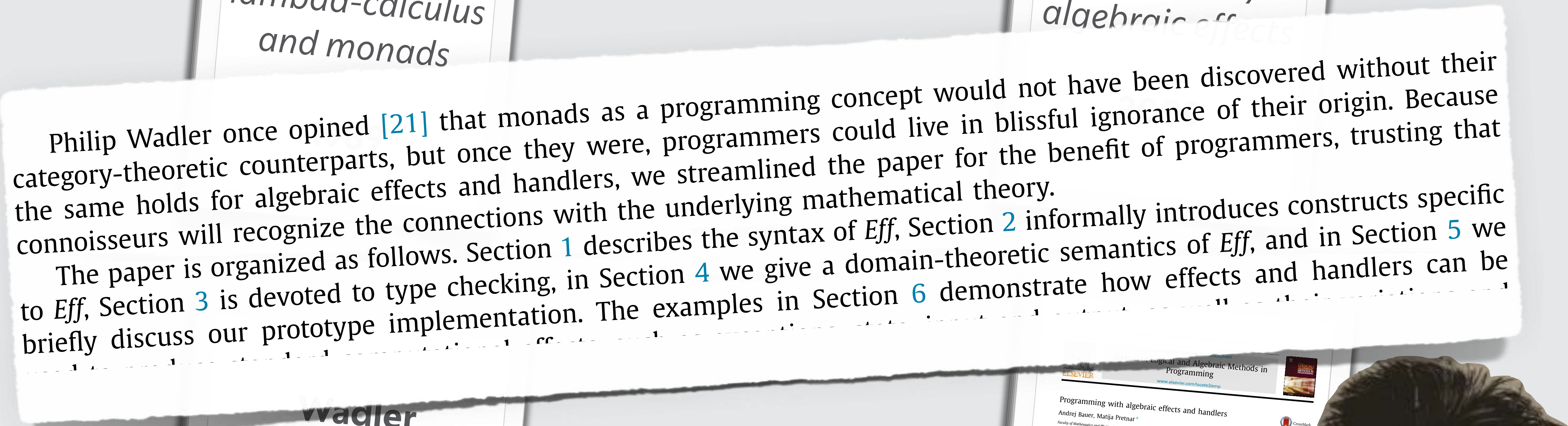
The new version of Eff also had an accompanying **research paper**

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

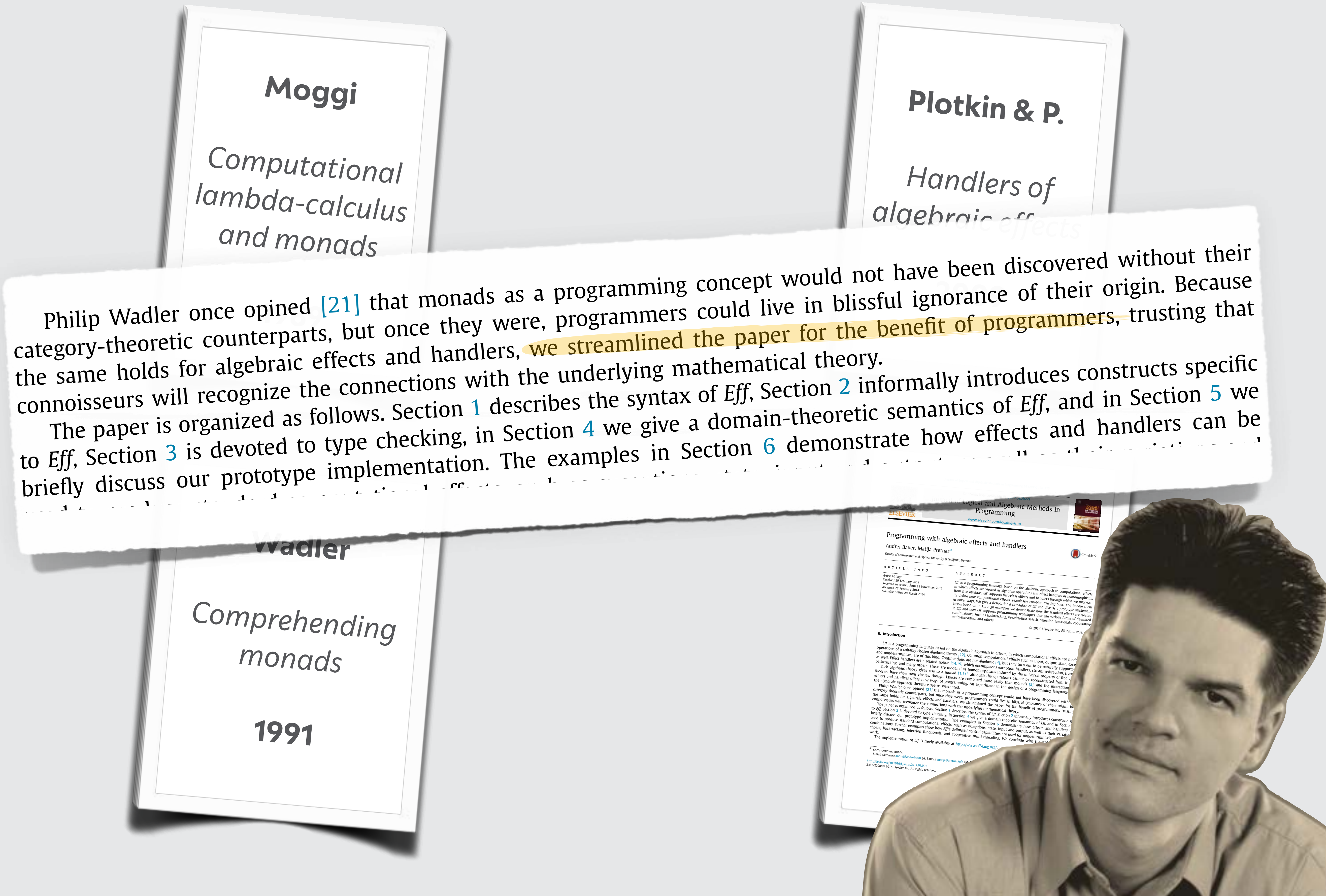
1991

Logical and Algebraic Methods in Programming

Programming with algebraic effects and handlers

Andrej Bauer, Matija Pretnar

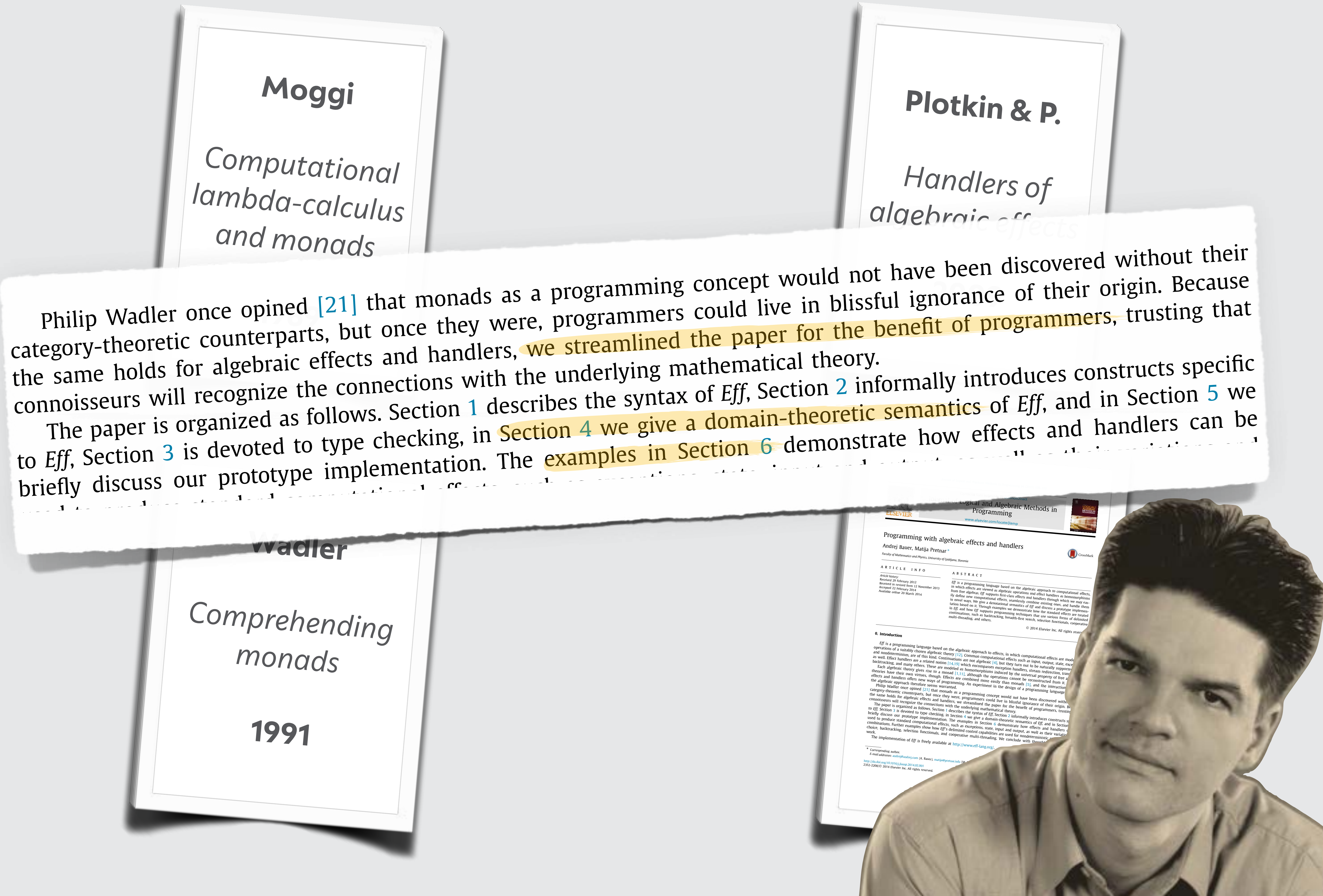
The new version of Eff also had an accompanying **research paper**


Moggi
Computational lambda-calculus and monads
Plotkin & P.
Handlers of algebraic effects
Wadler
Comprehending monads
1991
ELSEVIER
Logical and Algebraic Methods in Programming
www.elsevier.com/locate/jlamp
Programming with algebraic effects and handlers
Andrej Bauer, Matija Pretnar *
Faculty of Mathematics and Physics, University of Ljubljana, Slovenia
ARTICLE INFO
ABSTRACT
0. Introduction

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

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 used to produce standard computational effects, such as exceptions, state, input and output, as well as their variations and

Wadler

Comprehending monads

1991

ELSEVIER

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Programming with algebraic effects and handlers

Andrej Bauer, Matija Pretnar *

Faculty of Mathematics and Physics, University of Ljubljana, Slovenia

CrossMark

ARTICLE INFO

Article history:
Received 29 February 2012
Received in revised form 12 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 from free algebras. *Eff* supports first-class effects and handlers through which we may easily define new computational effects, seamlessly combine existing ones, and handle them in novel ways. We give a denotational semantics of *Eff* and discuss a prototype implementation based on it. Through examples we demonstrate how the standard effects are treated in *Eff*, and how *Eff* supports programming techniques that use various forms of delimited continuations, such as backtracking, breadth-first search, selection functionals, cooperative multi-threading, and others.

© 2014 Elsevier Inc. All rights reser

0. Introduction

Eff is a programming language based on the algebraic approach to effects, in which computational effects are mode operations of a suitably chosen algebraic theory [12]. Common computational effects such as input, output, state, exce and nondeterminism, are of this kind. Continuations are not algebraic [4], but they turn out to be naturally supporte as well. Effect handlers are a related notion [14,19] which encompasses exception handlers, stream redirection, tran backtracking, and many others. These are modeled as homomorphisms induced by the universal property of free a

Each algebraic theory gives rise to a monad [1,11], although the operations cannot be reconstructed from it. theories have their own virtues, though. Effects are combined more easily than monads [5], and the interaction effects and handlers offers new ways of programming. An experiment in the design of a programming language the algebraic approach therefore seems warranted.

Philip Wadler once opined [21] that monads as a programming concept would not have been discovered witho category-theoretic counterparts, but once they were, programmers could live in blissful ignorance of their origin. Be the same holds for algebraic effects and handlers, we streamlined the paper for the benefit of programmers, trustin 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 s to *Eff*, Section 3 is devoted to type checking, in Section 4 we give a domain-theoretic semantics of *Eff*, and in Section briefly discuss our prototype implementation. The examples in Section 6 demonstrate how effects and handlers c used to produce standard computational effects, such as exceptions, state, input and output, as well as their variatio combinations. Further examples show how *Eff*'s delimited control capabilities are used for nondeterministic a choice, backtracking, selection functionals, and cooperative multi-threading. We conclude with though work.

The implementation of *Eff* is freely available at http://www.eff-lang.org/.

* Corresponding author.
E-mail addresses: andrej@andrej.com (A. Bauer), matija@pretnar.info (M.

http://dx.doi.org/10.1016/j.jlamp.2014.02.001
2352-2208/© 2014 Elsevier Inc. All rights reserved.

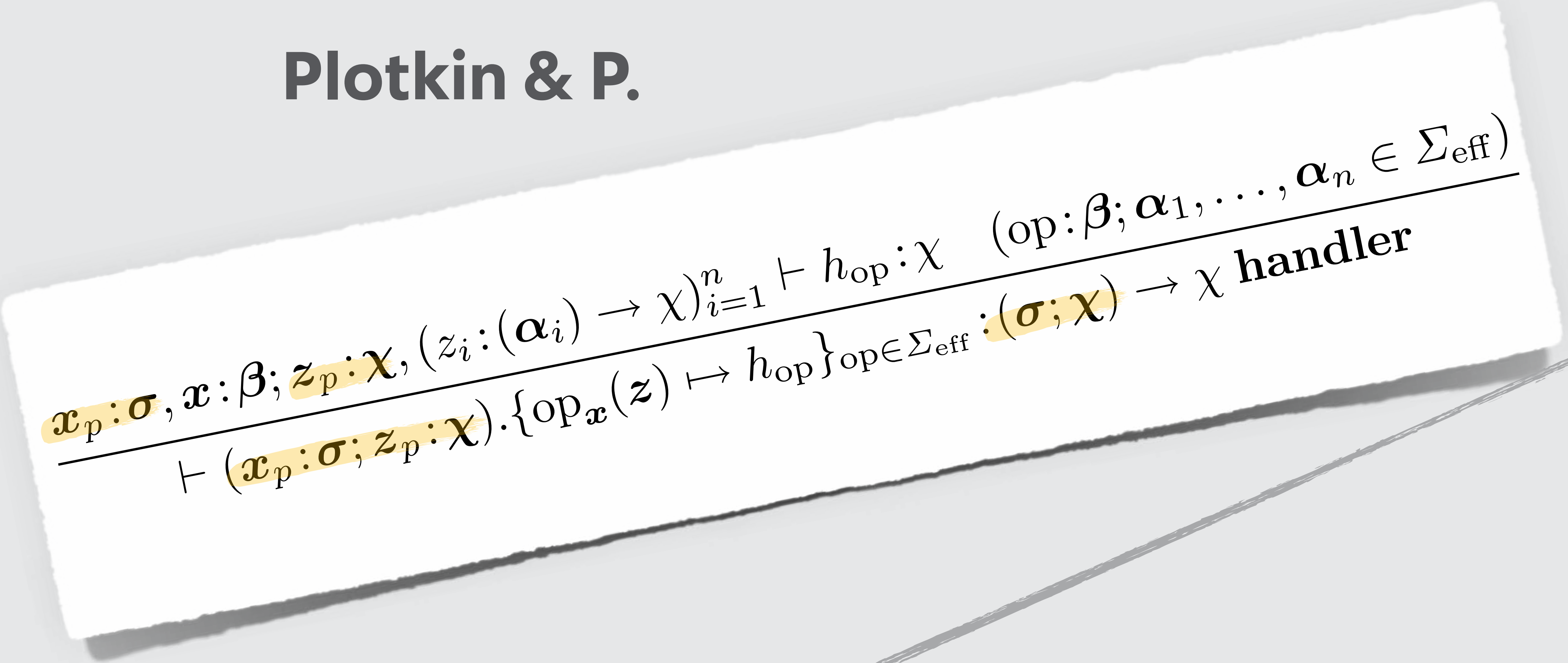
Moving from **mathematics to programming** gave **extra flexibility**

Plotkin & P.

Bauer & P.

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

Plotkin & P.

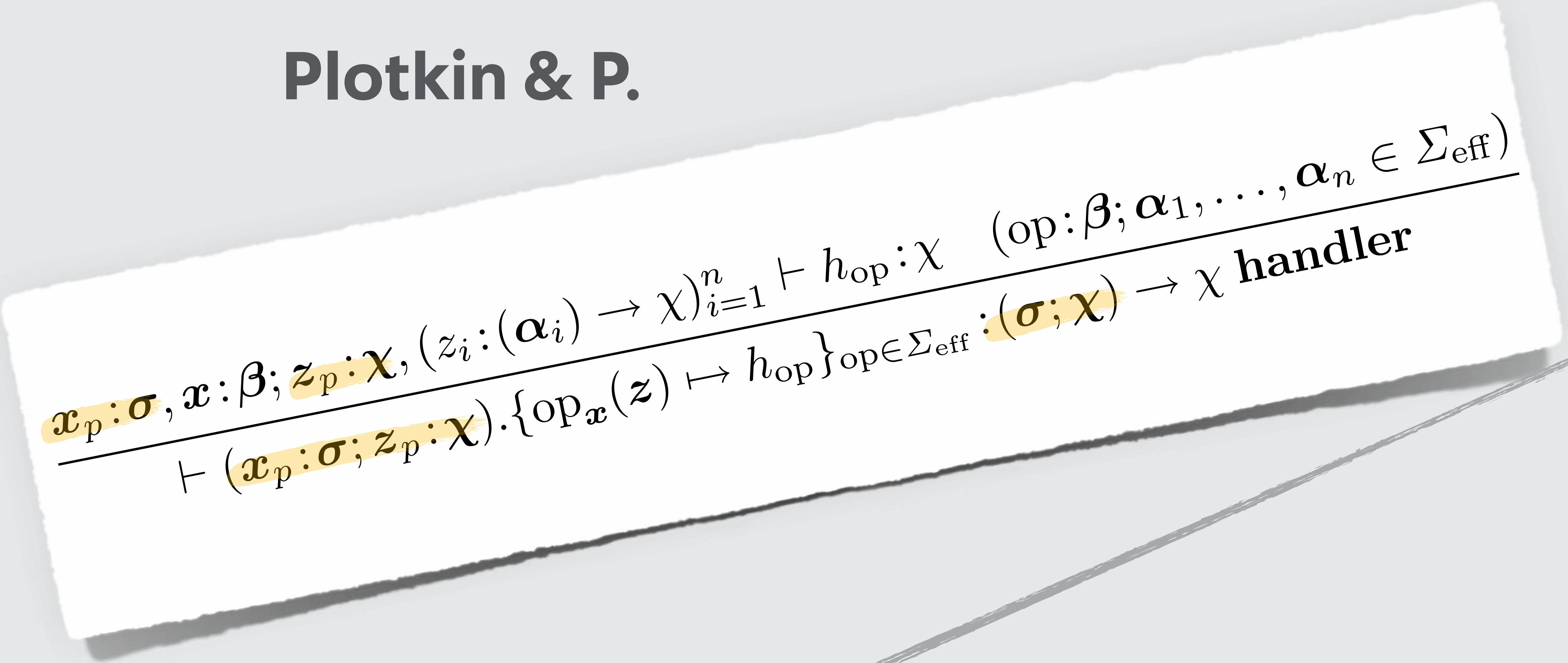


$$\frac{\boldsymbol{x}_p : \boldsymbol{\sigma}, \boldsymbol{x} : \boldsymbol{\beta}; \boldsymbol{z}_p : \boldsymbol{\chi}, (z_i : (\boldsymbol{\alpha}_i) \to \chi)_{i=1}^{n} \vdash h_{\mathrm{op}} : \chi \quad (\mathrm{op} : \boldsymbol{\beta}; \boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_n \in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

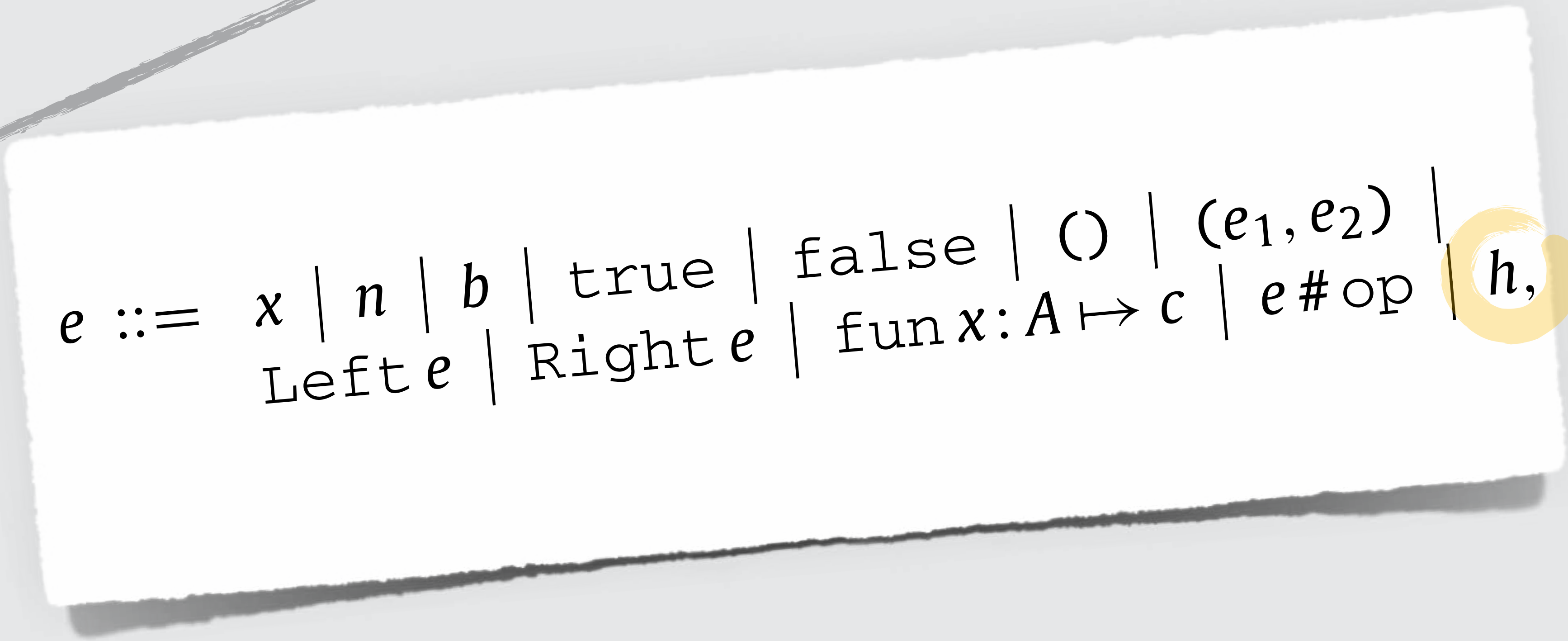
Bauer & P.

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

Plotkin & P.



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



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

Bauer & P.

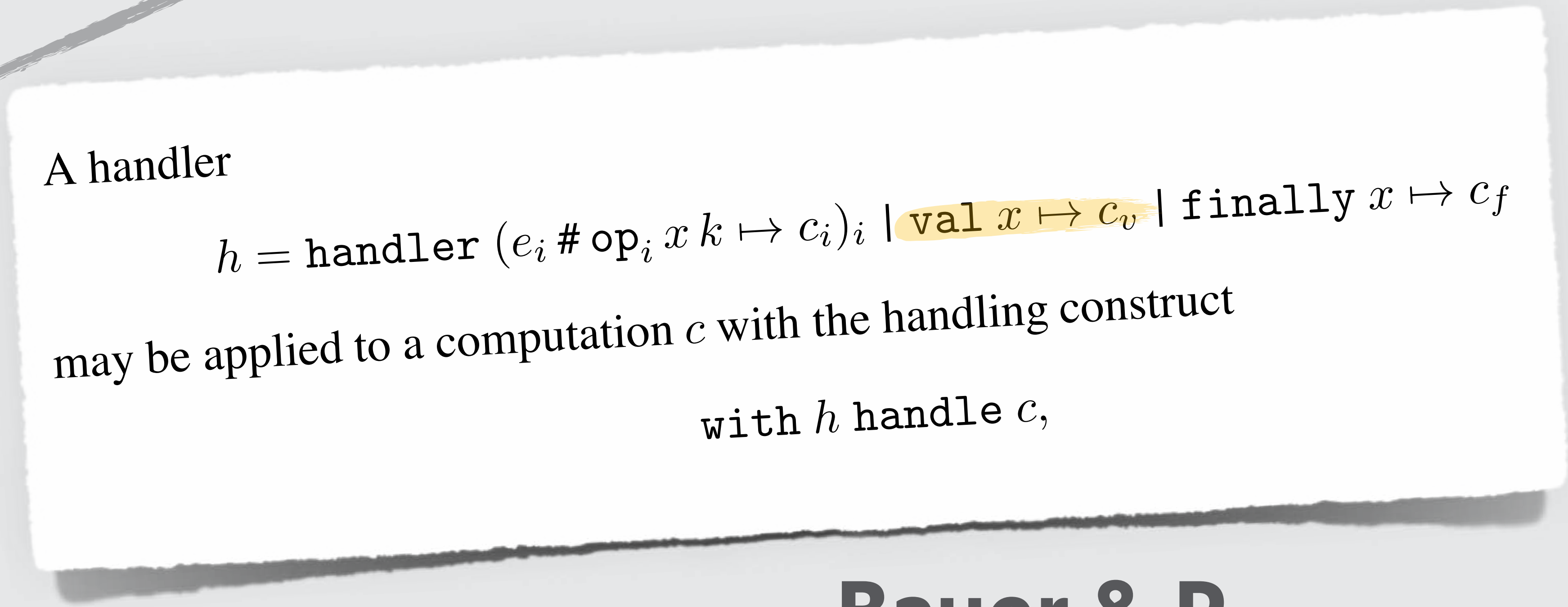
Notion of models **got absorbed** in homomorphisms

Plotkin & P.

Bauer & P.

Notion of models **got absorbed** in homomorphisms

Plotkin & P.



A handler

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

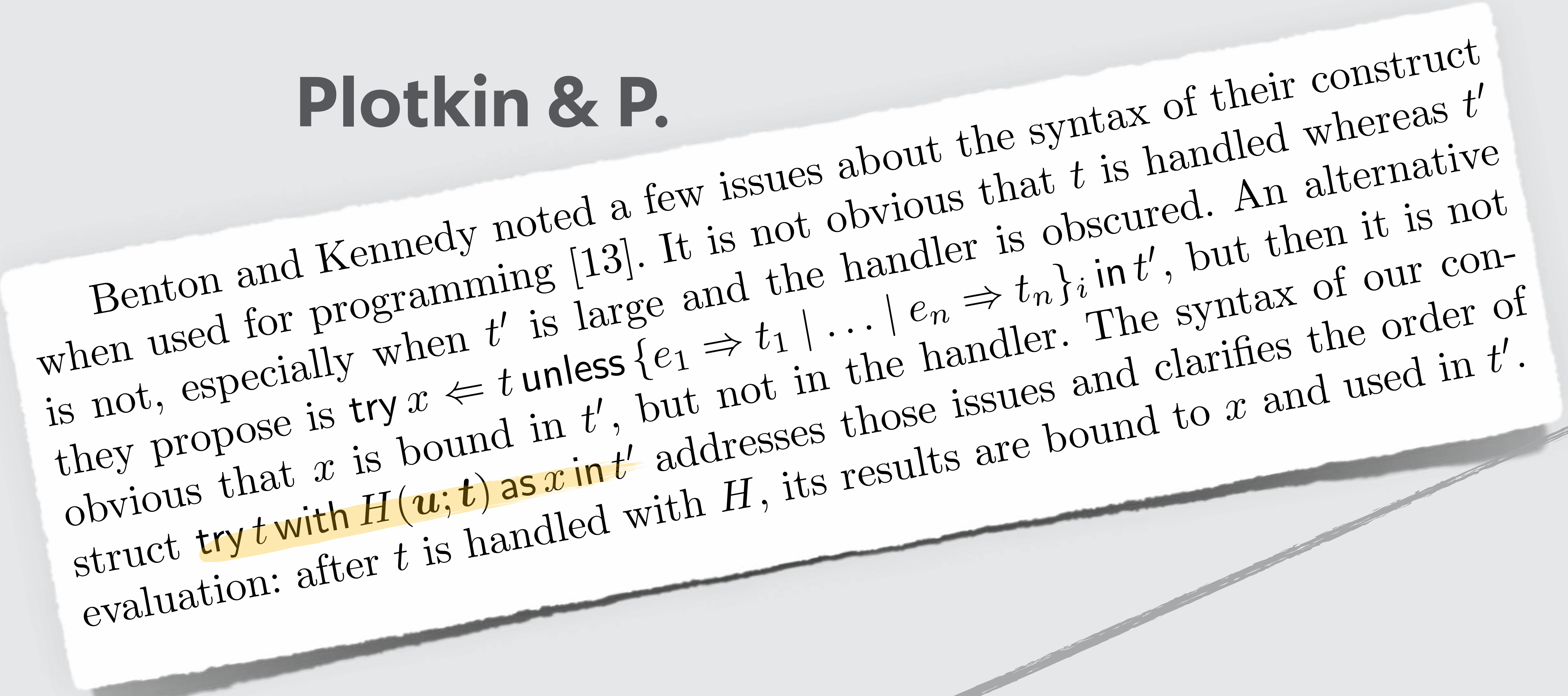
may be applied to a computation c with the handling construct

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

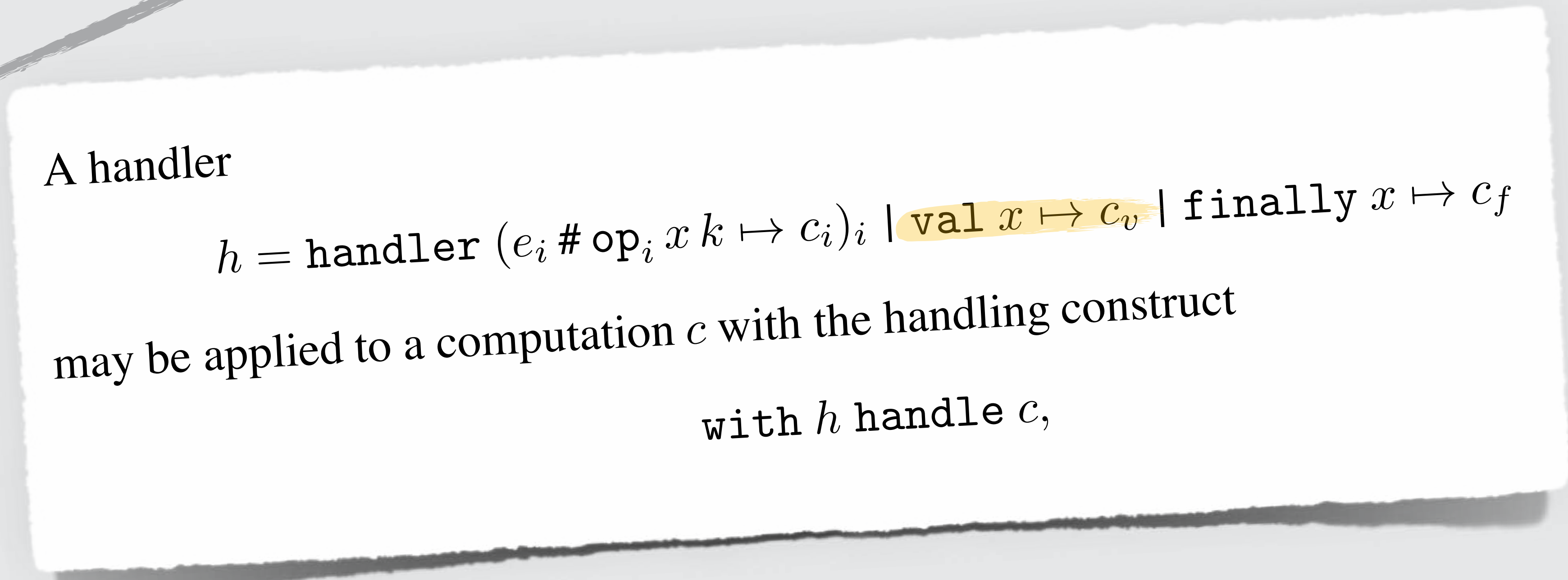
Bauer & P.

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

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

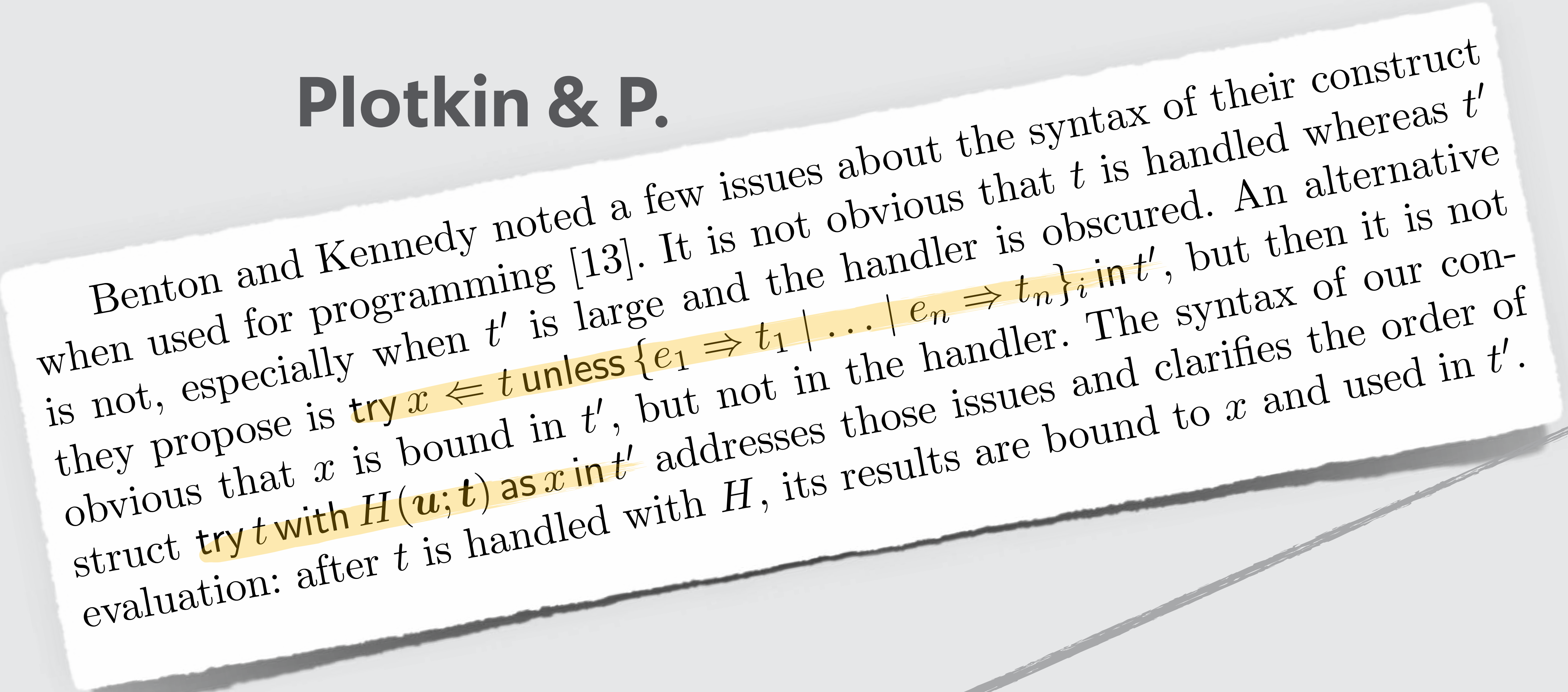
may be applied to a computation c with the handling construct

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

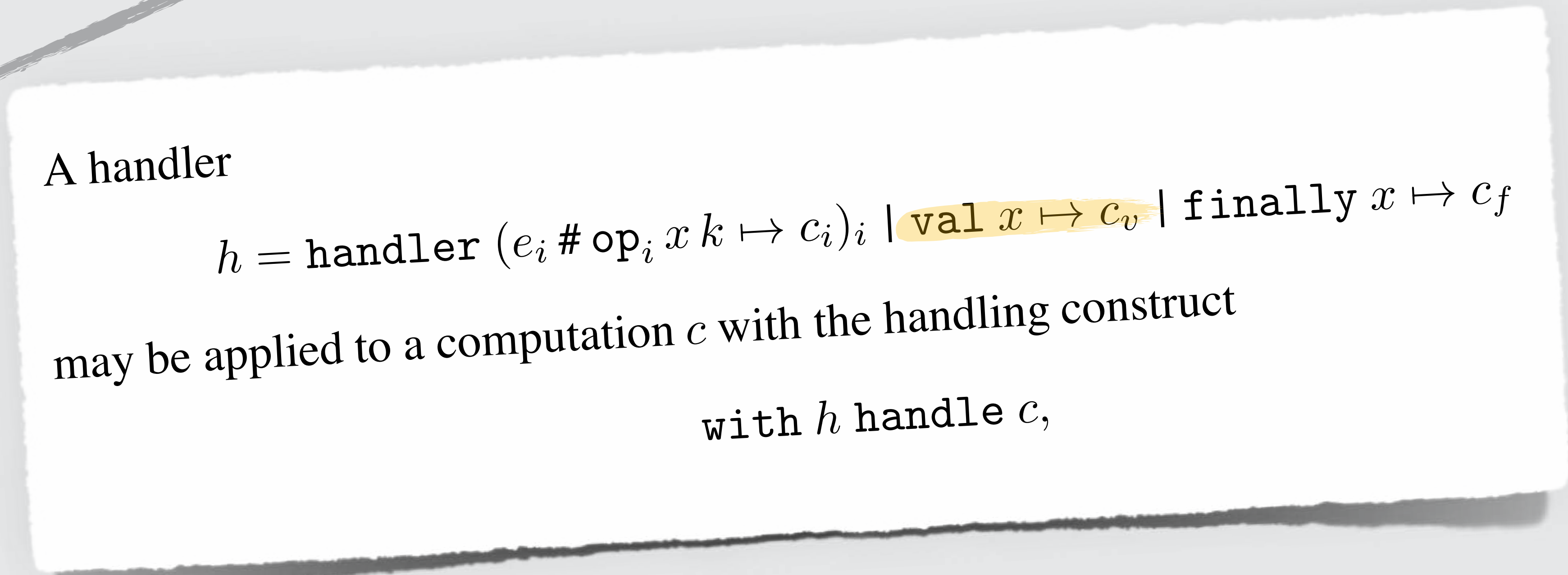
Bauer & P.

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.

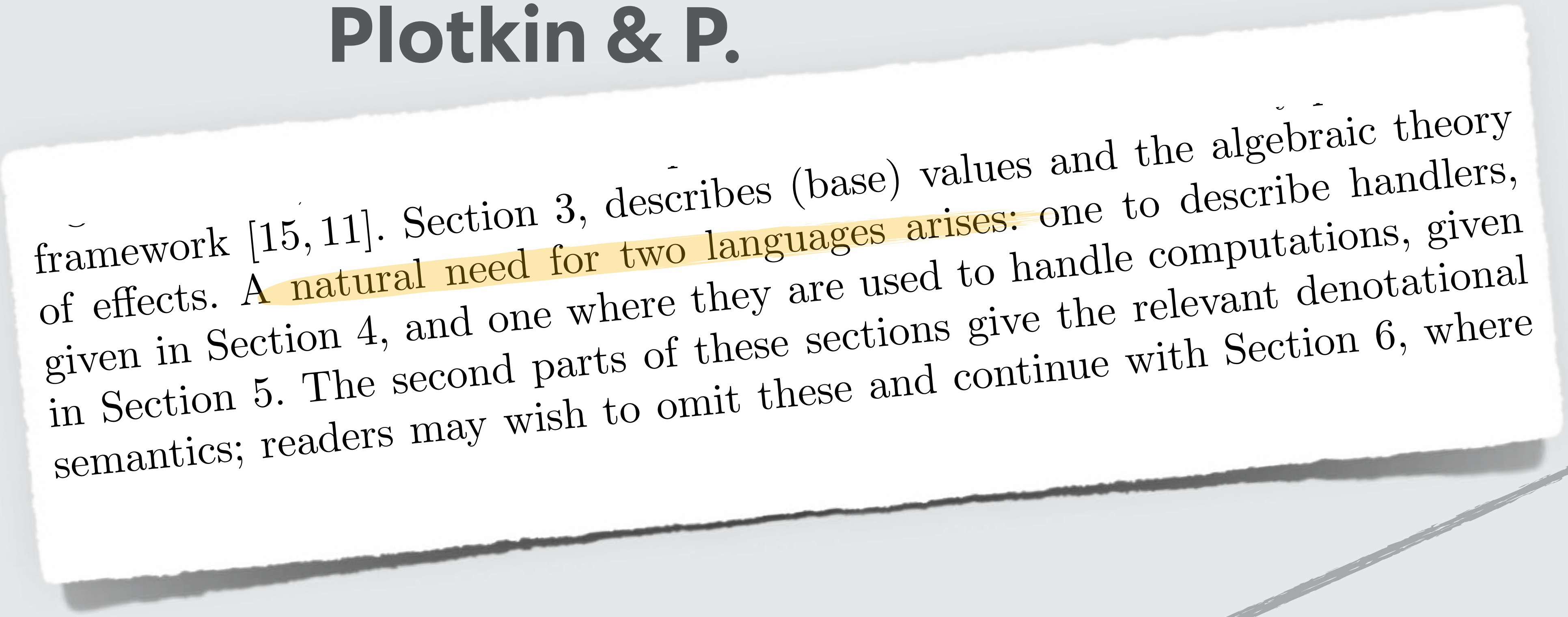
Equations disappeared

Plotkin & P.

Bauer & P.

Equations disappeared

Plotkin & P.

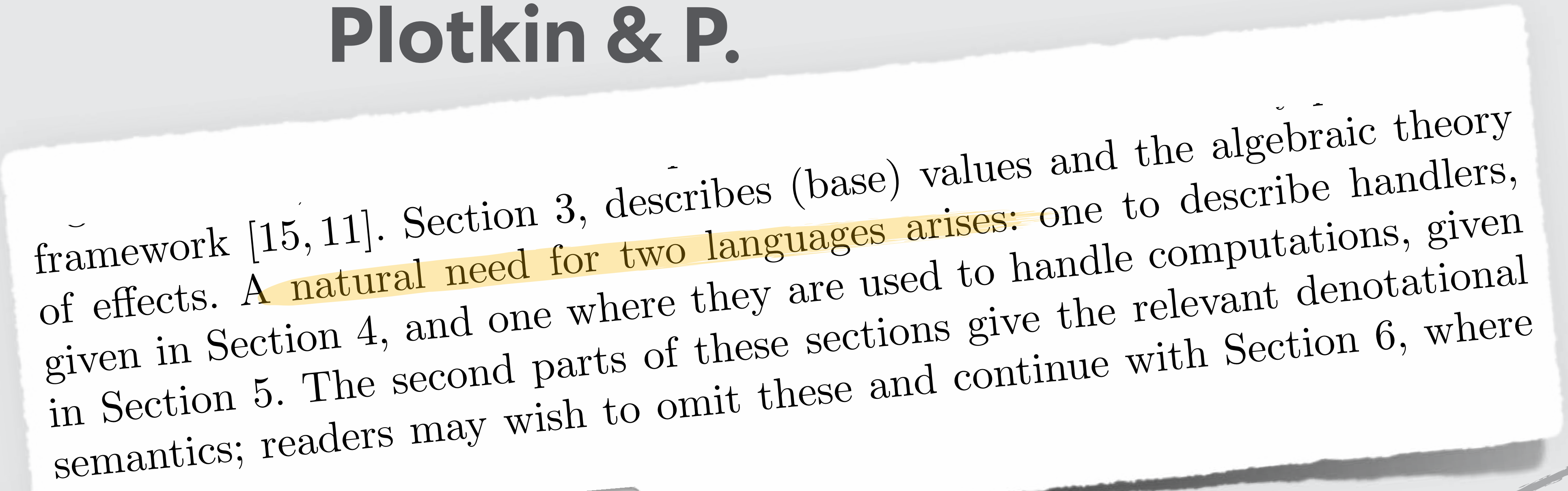


framework [15, 11]. Section 3, describes (base) values and the algebraic theory of effects. A natural need for two languages arises: one to describe handlers, given in Section 4, and one where they are used to handle computations, given in Section 5. The second parts of these sections give the relevant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

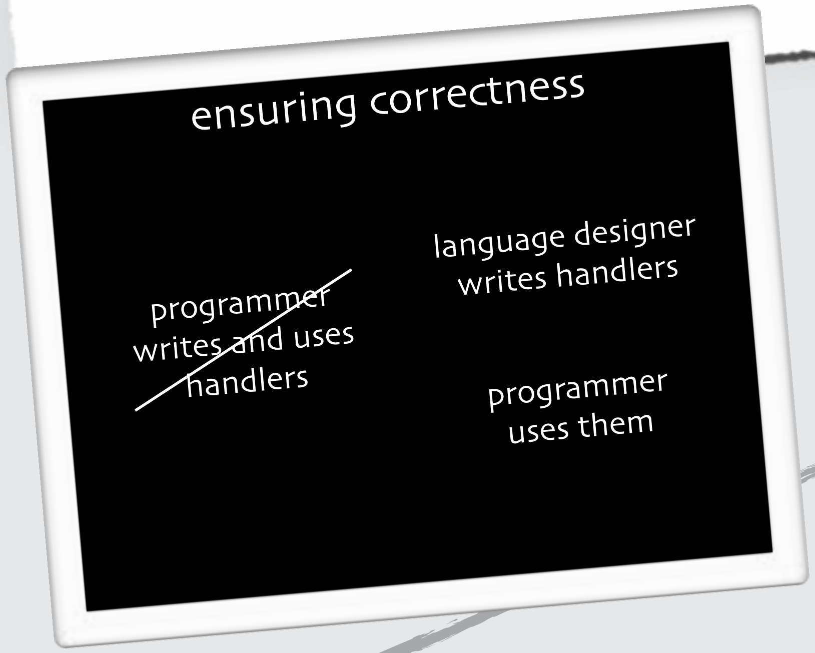
Bauer & P.

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 omit these and continue with Section 6, where



ensuring correctness

~~programmer writes and uses handlers~~

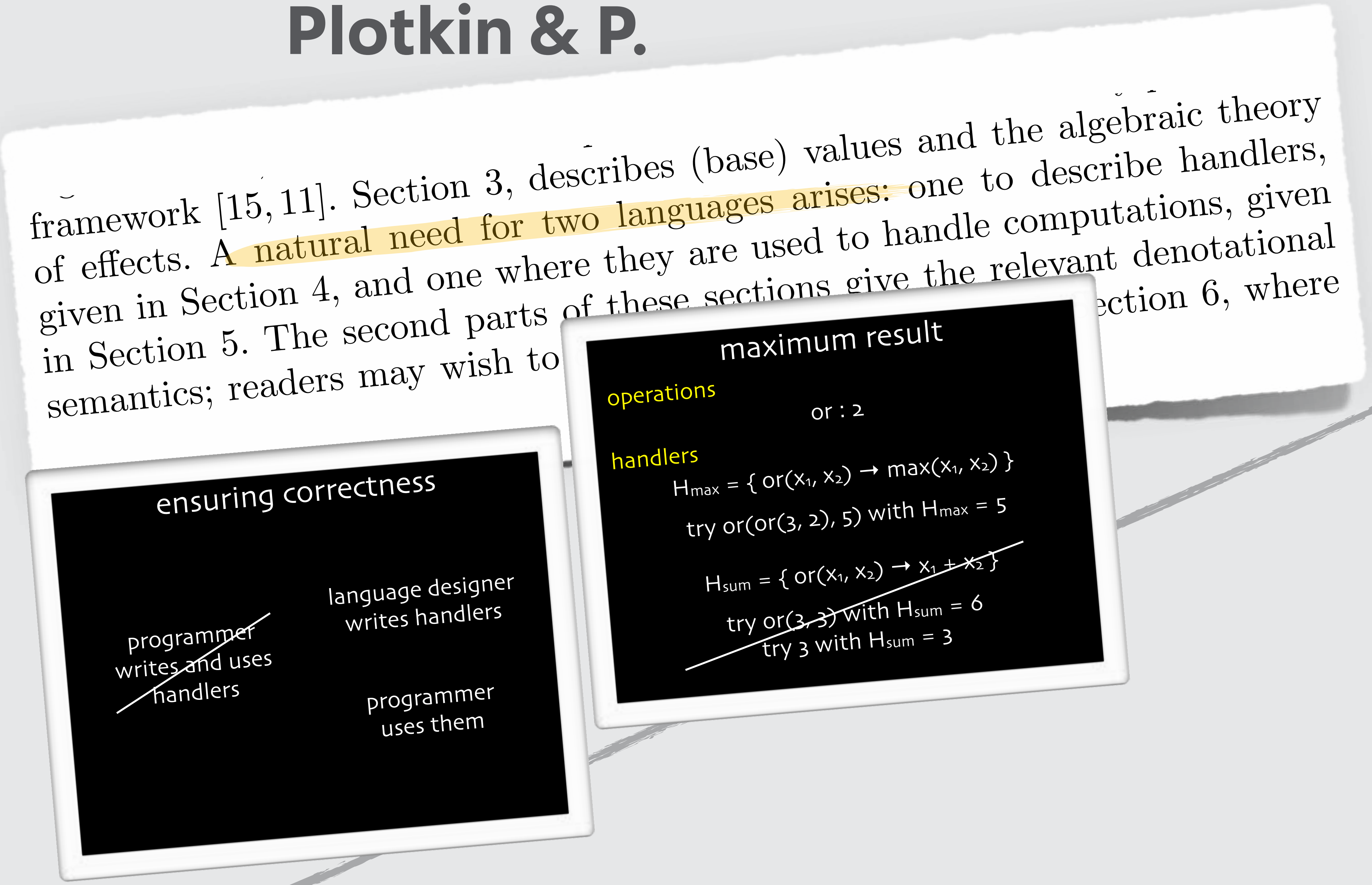
language designer writes handlers

programmer uses them

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

ensuring correctness

~~programmer writes and uses handlers~~

language designer writes handlers

programmer uses them

maximum result

operations

or : 2

handlers

H_{max} = { or(x_1, x_2) → max(x_1, x_2) }

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

~~H_{sum} = { or(x_1, x_2) → x_1 + x_2 }~~

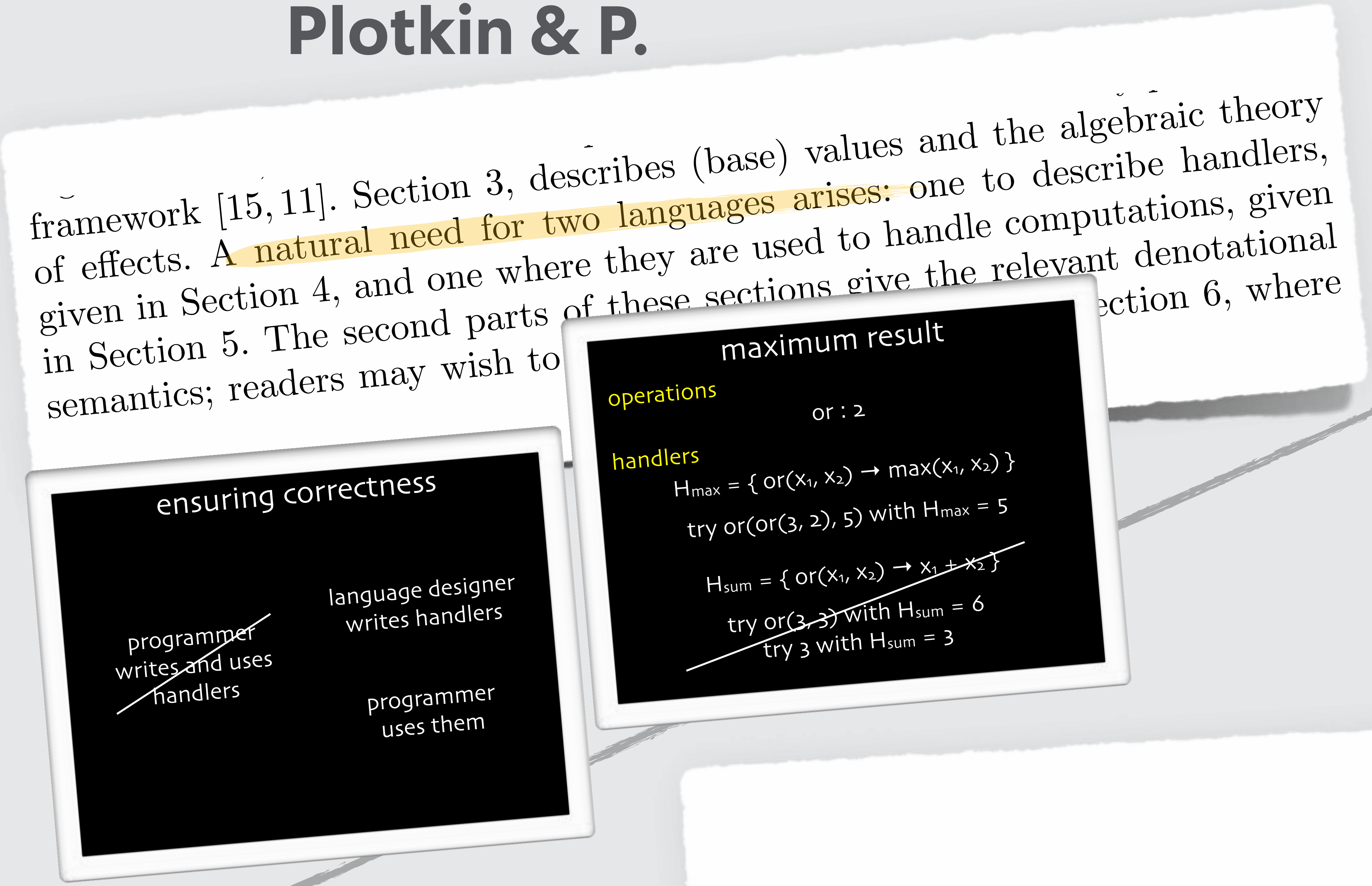
~~try or(3, 3) with H_{sum} = 6~~

~~try 3 with H_{sum} = 3~~

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

ensuring correctness

~~programmer writes and uses handlers~~

language designer writes handlers

programmer uses them

maximum result

operations

or : 2

handlers

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

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

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

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

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

Bauer & P.

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

Handlers in Action

Ohad Kammar
University of Cambridge
ohad.kammar@cl.cam.ac.uk

Sam Lindley
University of Strathclyde
Sam.Lindley@ed.ac.uk

Nicolas Oury
nicolas.oury@gmail.com

Abstract

Plotkin and Pretnar's handlers for algebraic effects occupy a sweet spot in the design space of abstractions for effectful computation. By separating effect signatures from their implementation, algebraic effects provide a high degree of modularity, allowing programmers to express effectful programs independently of the concrete interpretation of their effects. A handler is an interpretation of the effects of an algebraic computation. The handler abstraction adapts well to multiple settings: pure or impure, strict or lazy, static types or dynamic types.

This is a position paper whose main aim is to popularise the handler abstraction. We give a gentle introduction to its use, a collection of illustrative examples, and a straightforward operational semantics. We describe our Haskell implementation of handlers in detail, outline the ideas behind our OCaml, SML, and Racket implementations, and present experimental results comparing handlers with existing code.

Categories and Subject Descriptors D.1.1 [*Applicative (Functional) Programming*]; D.3.1 [*Formal Definitions and Theory*]; D.3.2 [*Language Classifications*]: Applicative (functional) languages; D.3.3 [*Language Constructs and Features*]; F.3.2 [*Semantics of Programming Languages*]: Operational semantics

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

1. Introduction

Monads have proven remarkably successf
tion over effectful computations [4, 30
programming language primitive v
lation principle: program to an *in*

Modular programs are cons
building blocks. This is *modu*
an *abstract* interface, we ins
tation. Given a composite i
dependently instantiated w
This is *modular instantiatio*

The monadic approach t
crete implementation rathe
For instance, in Haskell

ICFP '13, September 25–27, 2013, Bost
Copyright is held by the owner/author(s). P
ACM 978-1-4503-2326-0/13/09…$15.00.
http://dx.doi.org/10.1145/2500365.2500590

```
newtype State s a = State {runState :: s → (a, s)}
instance Monad (State s) where
  return x = State (λs → (x, s))
  m ≫= f  = State (λs → let (x, s′) = runState m s in
                                runState (f x) s′)
```

This definition says nothing about the intended use of *State s a* as the type of computations that read and write state. Worse, it breaks abstraction as consumers of state are exposed to its concrete implementation as a function of type $s \to (a, s)$. We can of course define the natural *get* and *put* operations on state, but their implementations are fixed.

Jones [18] advocates modular abstraction for monads in Haskell using type classes. For instance, we can define the following interface to abstract state computation[1]:

```
class Monad m ⇒ MonadState s m | m → s where
  get ::      m s
  put :: s → m ()
```

The *MonadState* interface can be smoothly combined with other interfaces, taking advantage of Haskell's type class mechanism to represent type-level sets of effects.

Monad transformers [25] provide a form of modular instanti
tion for abstract monadic computations. For instance, state can be
handled in the presence of other effects by incorpo
monad transformer within a monad transformer stac

A fundamental problem with monad transform
ticular abstract effect is instanti
comes concrete, and it becc
ns through the stack. Tamir
research area [16,
own monad
ply adding
odular abst
act operati
pose
atio

med

alg
er handler in ple

Library [12].

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

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

Handlers in Action

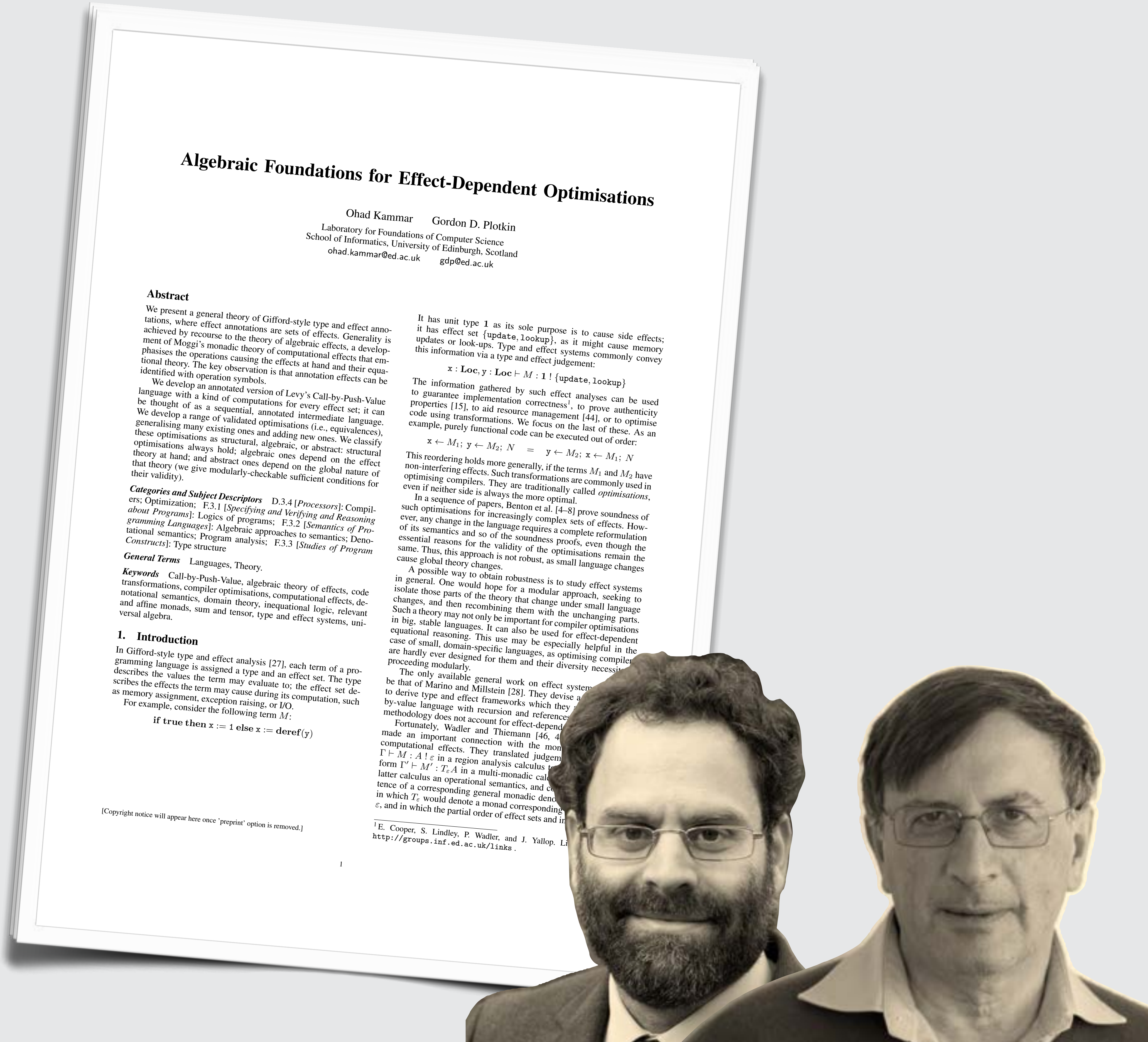
Ohad Kammar

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.

HANDLERS

HANDLERS

Equations not only describe effects, but entail additional laws



Algebraic Foundations for Effect-Dependent Optimisations

Ohad Kammar Gordon D. Plotkin

Laboratory for Foundations of Computer Science
School of Informatics, University of Edinburgh, Scotland
ohad.kammar@ed.ac.uk gdp@ed.ac.uk

Abstract

We present a general theory of Gifford-style type and effect annotations, where effect annotations are sets of effects. Generality is achieved by recourse to the theory of algebraic effects, a development of Moggi's monadic theory of computational effects that emphasises the operations causing the effects at hand and their equational theory. The key observation is that annotation effects can be identified with operation symbols.

We develop an annotated version of Levy's Call-by-Push-Value language with a kind of computations for every effect set; it can be thought of as a sequential, annotated intermediate language. We develop a range of validated optimisations (i.e., equivalences), generalising many existing ones and adding new ones. We classify these optimisations as structural, algebraic, or abstract: structural optimisations always hold; algebraic ones depend on the effect theory at hand; and abstract ones depend on the global nature of that theory (we give modularly-checkable sufficient conditions for their validity).

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers; Optimization; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Logics of programs; F.3.2 [*Semantics of Programming Languages*]: Algebraic approaches to semantics; Denotational semantics; Program analysis; F.3.3 [*Studies of Program Constructs*]: Type structure

General Terms Languages, Theory.

Keywords Call-by-Push-Value, algebraic theory of effects, code transformations, compiler optimisations, computational effects, denotational semantics, domain theory, inequational logic, relevant and affine monads, sum and tensor, type and effect systems, universal algebra.

1. Introduction

In Gifford-style type and effect analysis [27], each term of a programming language is assigned a type and an effect set. The type describes the values the term may evaluate to; the effect set describes the effects the term may cause during its computation, such as memory assignment, exception raising, or I/O.

For example, consider the following term M:

$$\textbf{if true then } \mathtt{x} := 1 \textbf{ else } \mathtt{x} := \textbf{deref}(\mathtt{y})$$

It has unit type $\mathbf{1}$ as its sole purpose is to cause side effects; it has effect set $\{\mathtt{update}, \mathtt{lookup}\}$, as it might cause memory updates or look-ups. Type and effect systems commonly convey this information via a type and effect judgement:

$$\mathtt{x} : \mathbf{Loc}, \mathtt{y} : \mathbf{Loc} \vdash M : \mathbf{1} \,!\, \{\mathtt{update}, \mathtt{lookup}\}$$

The information gathered by such effect analyses can be used to guarantee implementation correctness[1], to prove authenticity properties [15], to aid resource management [44], or to optimise code using transformations. We focus on the last of these. As an example, purely functional code can be executed out of order:

$$\mathtt{x} \leftarrow M_1;\ \mathtt{y} \leftarrow M_2;\ N \quad = \quad \mathtt{y} \leftarrow M_2;\ \mathtt{x} \leftarrow M_1;\ N$$

This reordering holds more generally, if the terms M_1 and M_2 have non-interfering effects. Such transformations are commonly used in optimising compilers. They are traditionally called *optimisations*, even if neither side is always the more optimal.

In a sequence of papers, Benton et al. [4–8] prove soundness of such optimisations for increasingly complex sets of effects. However, any change in the language requires a complete reformulation of its semantics and so of the soundness proofs, even though the essential reasons for the validity of the optimisations remain the same. Thus, this approach is not robust, as small language changes cause global theory changes.

A possible way to obtain robustness is to study effect systems in general. One would hope for a modular approach, seeking to isolate those parts of the theory that change under small language changes, and then recombining them with the unchanging parts. Such a theory may not only be important for compiler optimisations in big, stable languages. It can also be used for effect-dependent equational reasoning. This use may be especially helpful in the case of small, domain-specific languages, as optimising compiler
are hardly ever designed for them and their diversity necessit
proceeding modularly.

The only available general work on effect system
be that of Marino and Millstein [28]. They devise a
to derive type and effect frameworks which they a
by-value language with recursion and references
methodology does not account for effect-depend

Fortunately, Wadler and Thiemann [46, 4
made an important connection with the mon
computational effects. They translated judgem
$\Gamma \vdash M : A \,!\, \varepsilon$ in a region analysis calculus t
form $\Gamma' \vdash M' : T_\varepsilon A$ in a multi-monadic calc
latter calculus an operational semantics, and c
tence of a corresponding general monadic deno
in which T_ε would denote a monad corresponding
ε, and in which the partial order of effect sets and in

[1] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Li
`http://groups.inf.ed.ac.uk/links` .

Dropping equations due to handlers weakens the results of the logic

operations

$$\mathtt{fail} : 0$$

$$\mathtt{choose} : 2$$

equations

$$\mathtt{choose}\,(\mathtt{choose}\,M\,N)\,P = \mathtt{choose}\,M\,(\mathtt{choose}\,N\,P)$$

$$\mathtt{choose}\,M\,N = \mathtt{choose}\,N\,M$$

$$\mathtt{choose}\,M\,M = M$$

$$\mathtt{choose}\,\mathtt{fail}\,M = M = \mathtt{choose}\,M\,\mathtt{fail}$$

algebraicity

$$\mathtt{do}\,x \Leftarrow (\mathtt{choose}\,M\,N)\,\mathtt{in}\,P = \mathtt{choose}\,(\mathtt{do}\,x \Leftarrow M\,\mathtt{in}\,P)\,(\mathtt{do}\,x \Leftarrow N\,\mathtt{in}\,P)$$

Dropping equations due to handlers weakens the results of the logic

operations

$$\begin{aligned} \texttt{fail} &: 0 \\ \texttt{choose} &: 2 \end{aligned}$$

equations

$$\begin{aligned} \texttt{choose}\,(\texttt{choose}\,M\,N)\,P &= \texttt{choose}\,M\,(\texttt{choose}\,N\,P) \\ \texttt{choose}\,M\,N &= \texttt{choose}\,N\,M \\ \texttt{choose}\,M\,M &= M \\ \texttt{choose}\,\texttt{fail}\,M &= M = \texttt{choose}\,M\,\texttt{fail} \end{aligned}$$

algebraicity

$$\texttt{do}\,x \Leftarrow (\texttt{choose}\,M\,N)\,\texttt{in}\,P = \texttt{choose}\,(\texttt{do}\,x \Leftarrow M\,\texttt{in}\,P)\,(\texttt{do}\,x \Leftarrow N\,\texttt{in}\,P)$$

nondeterministic laws

$$\begin{aligned} \texttt{choose}\,(\texttt{do}\,x \Leftarrow M\,\texttt{in}\,N)\,(\texttt{do}\,x \Leftarrow M\,\texttt{in}\,P) &= \texttt{do}\,x \Leftarrow M\,\texttt{in}\,(\texttt{choose}\,N\,P) \\ \texttt{do}\,x \Leftarrow M\,\texttt{in}\,(\texttt{do}\,y \Leftarrow N\,\texttt{in}\,P) &= \texttt{do}\,y \Leftarrow N\,\texttt{in}\,(\texttt{do}\,x \Leftarrow M\,\texttt{in}\,P) \end{aligned}$$

Dropping equations due to handlers **weakens the results** of the logic

operations

$$\begin{aligned}\mathtt{fail} &: 0\\ \mathtt{choose} &: 2\end{aligned}$$

algebraicity

$$\mathtt{do}\,x \Leftarrow (\mathtt{choose}\,M\,N)\,\mathtt{in}\,P = \mathtt{choose}\,(\mathtt{do}\,x \Leftarrow M\,\mathtt{in}\,P)\,(\mathtt{do}\,x \Leftarrow N\,\mathtt{in}\,P)$$

nondeterministic laws

$$\begin{aligned}\mathtt{choose}\,(\mathtt{do}\,x \Leftarrow M\,\mathtt{in}\,N)\,(\mathtt{do}\,x \Leftarrow M\,\mathtt{in}\,P) &= \mathtt{do}\,x \Leftarrow M\,\mathtt{in}\,(\mathtt{choose}\,N\,P)\\ \mathtt{do}\,x \Leftarrow M\,\mathtt{in}\,(\mathtt{do}\,y \Leftarrow N\,\mathtt{in}\,P) &= \mathtt{do}\,y \Leftarrow N\,\mathtt{in}\,(\mathtt{do}\,x \Leftarrow M\,\mathtt{in}\,P)\end{aligned}$$

Dropping equations due to handlers **weakens the results** of the logic

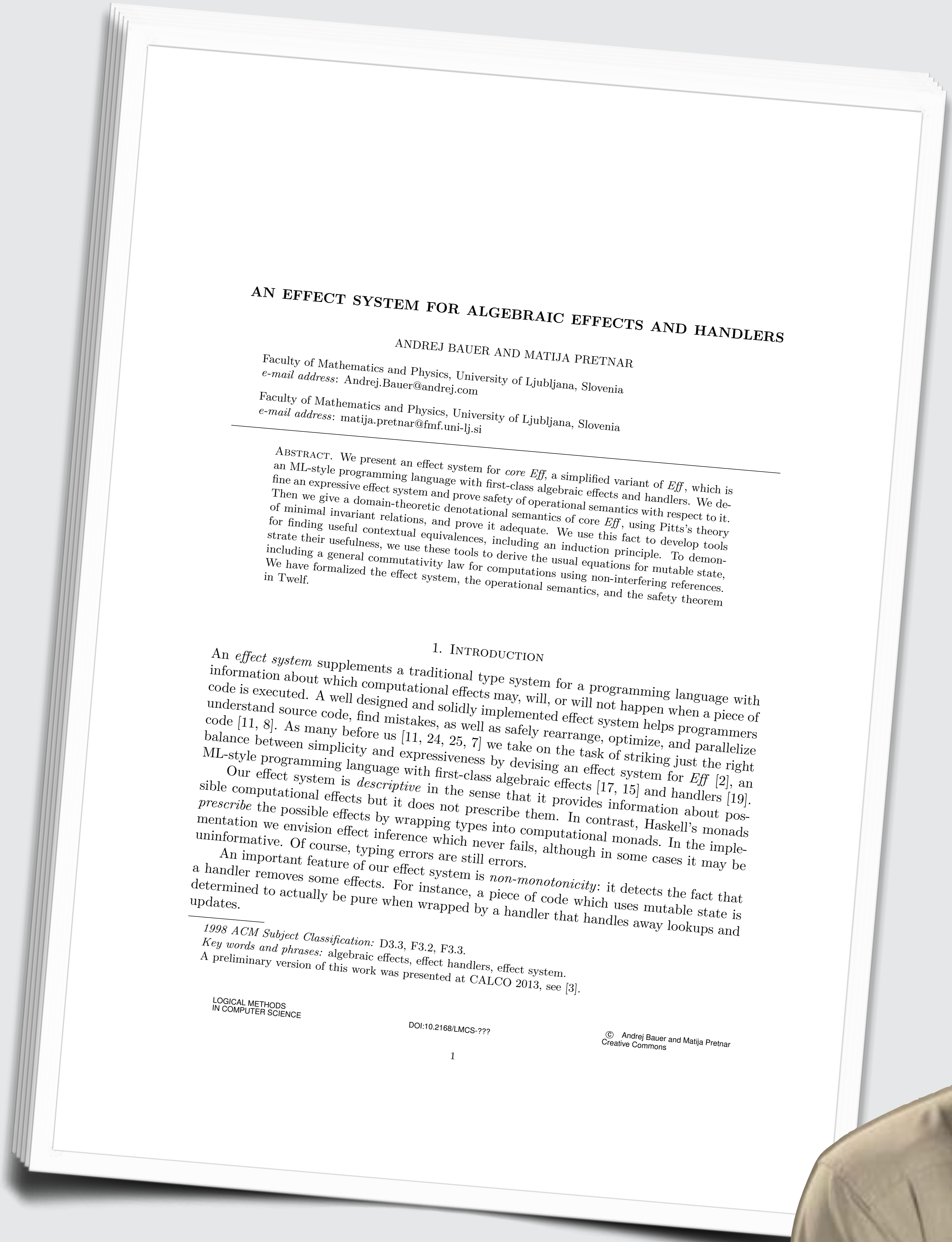
operations

$$\begin{aligned} \mathtt{fail} &: 0 \\ \mathtt{choose} &: 2 \end{aligned}$$

algebraicity

$$\mathtt{do}\, x \Leftarrow (\mathtt{choose}\, M\, N)\, \mathtt{in}\, P = \mathtt{choose}\, (\mathtt{do}\, x \Leftarrow M\, \mathtt{in}\, P)\, (\mathtt{do}\, x \Leftarrow N\, \mathtt{in}\, P)$$

Certain laws can be reconstructed **under a particular handler**



AN EFFECT SYSTEM FOR ALGEBRAIC EFFECTS AND HANDLERS

ANDREJ BAUER AND MATIJA PRETNAR

Faculty of Mathematics and Physics, University of Ljubljana, Slovenia
e-mail address: Andrej.Bauer@andrej.com

Faculty of Mathematics and Physics, University of Ljubljana, Slovenia
e-mail address: matija.pretnar@fmf.uni-lj.si

Abstract. We present an effect system for *core Eff*, a simplified variant of *Eff*, which is an ML-style programming language with first-class algebraic effects and handlers. We define an expressive effect system and prove safety of operational semantics with respect to it. Then we give a domain-theoretic denotational semantics of core *Eff*, using Pitts's theory of minimal invariant relations, and prove it adequate. We use this fact to develop tools for finding useful contextual equivalences, including an induction principle. To demonstrate their usefulness, we use these tools to derive the usual equations for mutable state, including a general commutativity law for computations using non-interfering references. We have formalized the effect system, the operational semantics, and the safety theorem in Twelf.

1. Introduction

An *effect system* supplements a traditional type system for a programming language with information about which computational effects may, will, or will not happen when a piece of code is executed. A well designed and solidly implemented effect system helps programmers understand source code, find mistakes, as well as safely rearrange, optimize, and parallelize code [11, 8]. As many before us [11, 24, 25, 7] we take on the task of striking just the right balance between simplicity and expressiveness by devising an effect system for *Eff* [2], an ML-style programming language with first-class algebraic effects [17, 15] and handlers [19].

Our effect system is *descriptive* in the sense that it provides information about possible computational effects but it does not prescribe them. In contrast, Haskell's monads *prescribe* the possible effects by wrapping types into computational monads. In the implementation we envision effect inference which never fails, although in some cases it may be uninformative. Of course, typing errors are still errors.

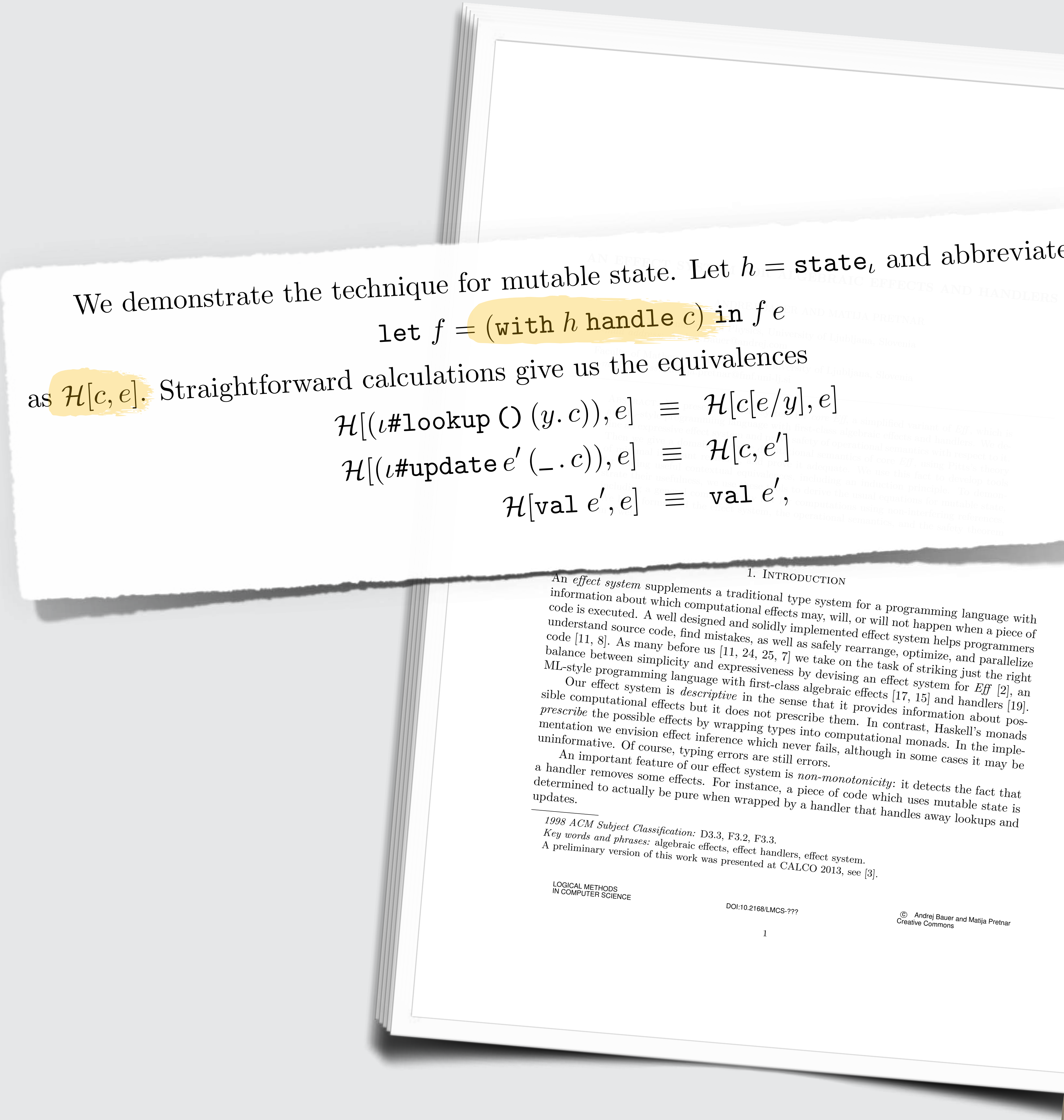
An important feature of our effect system is *non-monotonicity*: it detects the fact that a handler removes some effects. For instance, a piece of code which uses mutable state is determined to actually be pure when wrapped by a handler that handles away lookups and updates.

1998 ACM Subject Classification: D3.3, F3.2, F3.3.
Key words and phrases: algebraic effects, effect handlers, effect system.

A preliminary version of this work was presented at CALCO 2013, see [3].

DOI:10.2168/LMCS-???

Certain laws can be reconstructed **under a particular handler**



We demonstrate the technique for mutable state. Let $h = \mathtt{state}_\iota$ and abbreviate

$$\mathtt{let}\ f = (\mathtt{with}\ h\ \mathtt{handle}\ c)\ \mathtt{in}\ f\ e$$

as $\mathcal{H}[c, e]$. Straightforward calculations give us the equivalences

$$\begin{aligned}
\mathcal{H}[(\iota\#\mathtt{lookup}\ ()\ (y.\,c)), e] &\equiv \mathcal{H}[c[e/y], e] \\
\mathcal{H}[(\iota\#\mathtt{update}\ e'\ (\_.\,c)), e] &\equiv \mathcal{H}[c, e'] \\
\mathcal{H}[\mathtt{val}\ e', e] &\equiv \mathtt{val}\ e',
\end{aligned}$$

1. INTRODUCTION

An *effect system* supplements a traditional type system for a programming language with information about which computational effects may, will, or will not happen when a piece of code is executed. A well designed and solidly implemented effect system helps programmers understand source code, find mistakes, as well as safely rearrange, optimize, and parallelize code [11, 8]. As many before us [11, 24, 25, 7] we take on the task of striking just the right balance between simplicity and expressiveness by devising an effect system for *Eff* [2], an ML-style programming language with first-class algebraic effects [17, 15] and handlers [19].

Our effect system is *descriptive* in the sense that it provides information about possible computational effects but it does not prescribe them. In contrast, Haskell's monads *prescribe* the possible effects by wrapping types into computational monads. In the implementation we envision effect inference which never fails, although in some cases it may be uninformative. Of course, typing errors are still errors.

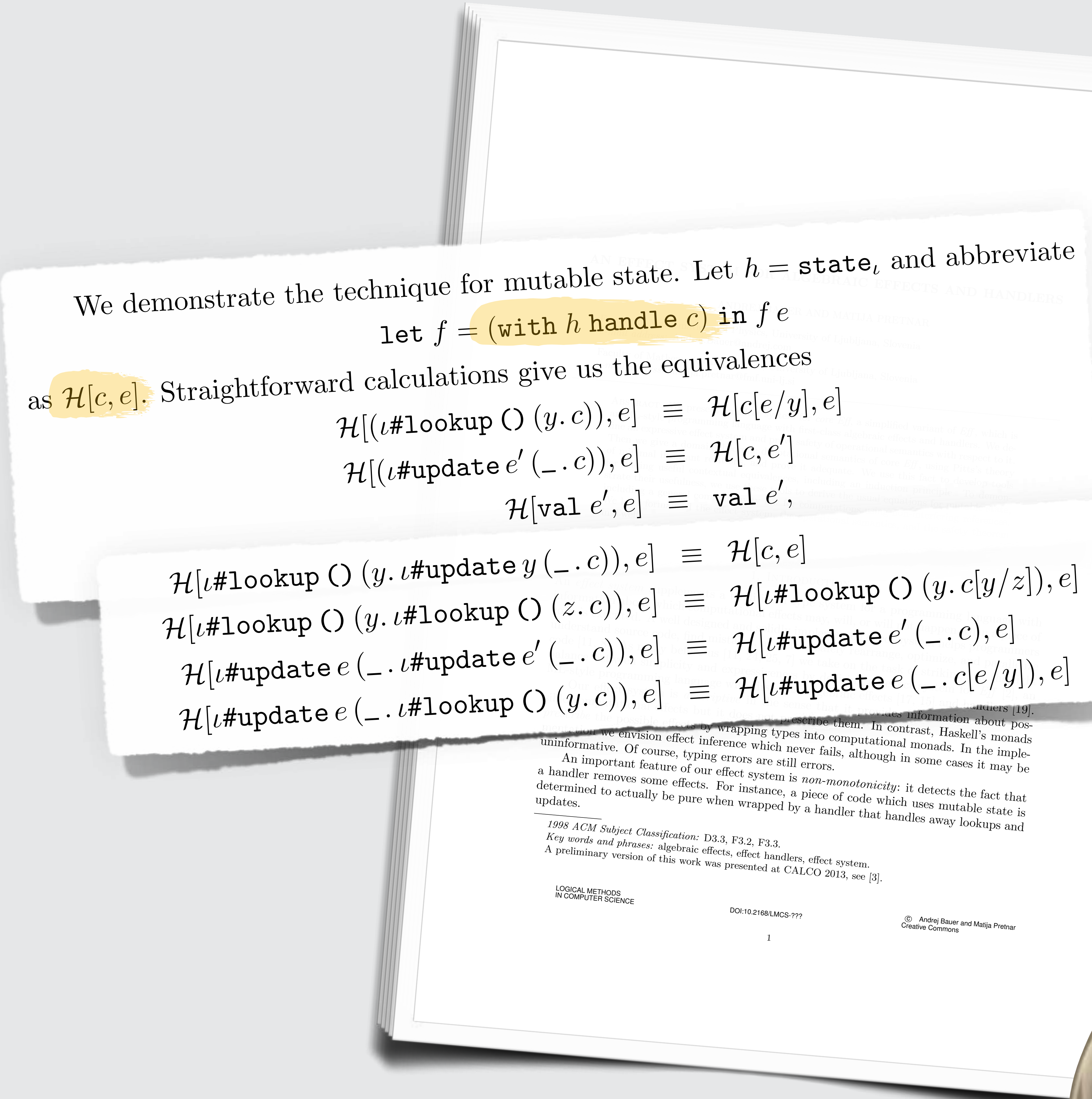
An important feature of our effect system is *non-monotonicity*: it detects the fact that a handler removes some effects. For instance, a piece of code which uses mutable state is determined to actually be pure when wrapped by a handler that handles away lookups and updates.

1998 ACM Subject Classification: D3.3, F3.2, F3.3.
Key words and phrases: algebraic effects, effect handlers, effect system.
A preliminary version of this work was presented at CALCO 2013, see [3].

LOGICAL METHODS
IN COMPUTER SCIENCE

DOI:10.2168/LMCS-???

Certain laws can be reconstructed **under a particular handler**



We demonstrate the technique for mutable state. Let $h = \texttt{state}_\iota$ and abbreviate

$$\texttt{let } f = (\texttt{with } h \texttt{ handle } c) \texttt{ in } f\, e$$

as $\mathcal{H}[c, e]$. Straightforward calculations give us the equivalences

$$\begin{aligned}
\mathcal{H}[(\iota\#\texttt{lookup}\,()\,(y.\,c)), e] &\equiv \mathcal{H}[c[e/y], e]\\
\mathcal{H}[(\iota\#\texttt{update}\,e'\,(\_.\,c)), e] &\equiv \mathcal{H}[c, e']\\
\mathcal{H}[\texttt{val}\ e', e] &\equiv \texttt{val}\ e',
\end{aligned}$$

$$\begin{aligned}
\mathcal{H}[\iota\#\texttt{lookup}\,()\,(y.\,\iota\#\texttt{update}\,y\,(\_.\,c)), e] &\equiv \mathcal{H}[c, e]\\
\mathcal{H}[\iota\#\texttt{lookup}\,()\,(y.\,\iota\#\texttt{lookup}\,()\,(z.\,c)), e] &\equiv \mathcal{H}[\iota\#\texttt{lookup}\,()\,(y.\,c[y/z]), e]\\
\mathcal{H}[\iota\#\texttt{update}\,e\,(\_.\,\iota\#\texttt{update}\,e'\,(\_.\,c)), e] &\equiv \mathcal{H}[\iota\#\texttt{update}\,e'\,(\_.\,c), e]\\
\mathcal{H}[\iota\#\texttt{update}\,e\,(\_.\,\iota\#\texttt{lookup}\,()\,(y.\,c)), e] &\equiv \mathcal{H}[\iota\#\texttt{update}\,e\,(\_.\,c[e/y]), e]
\end{aligned}$$

handlers [19].

provides information about possible computational effects but it does not prescribe them. In contrast, Haskell's monads by wrapping types into computational monads. In the implementation we envision effect inference which never fails, although in some cases it may be uninformative. Of course, typing errors are still errors.

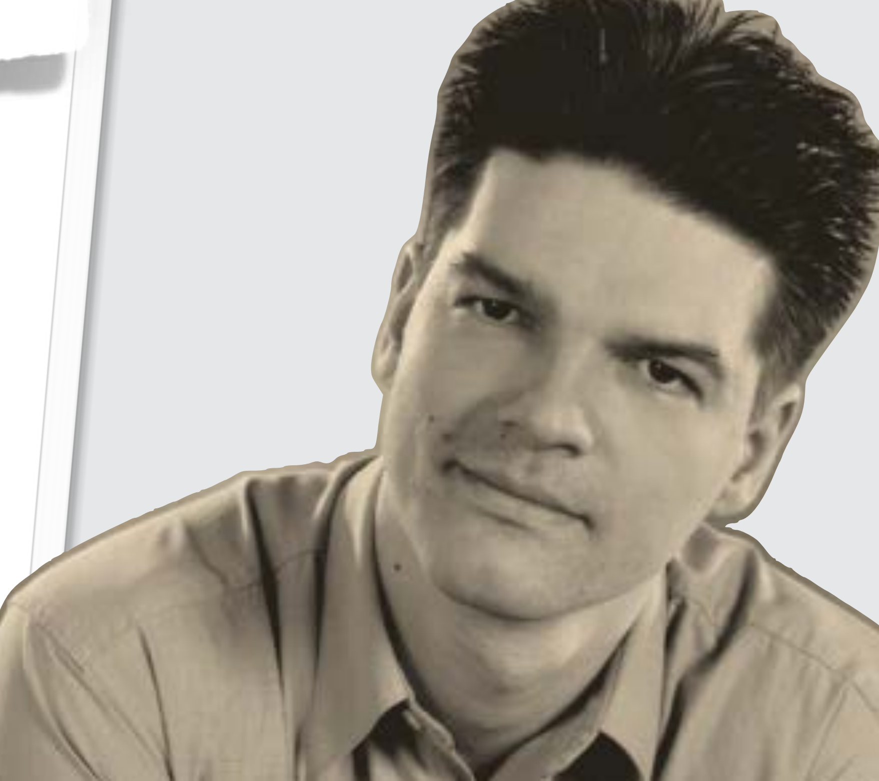
An important feature of our effect system is *non-monotonicity*: it detects the fact that a handler removes some effects. For instance, a piece of code which uses mutable state is determined to actually be pure when wrapped by a handler that handles away lookups and updates.

1998 ACM Subject Classification: D3.3, F3.2, F3.3.

Key words and phrases: algebraic effects, effect handlers, effect system.

A preliminary version of this work was presented at CALCO 2013, see [3].

DOI:10.2168/LMCS-???



The technique can and **has to be repeated** for any concrete handler

The technique can and **has to be repeated** for any concrete handler

$$
\begin{aligned}
\mathscr{H}_{\mathrm{max}}[M] &= \mathtt{with}\, H_{\mathrm{max}}\, \mathtt{handle}\, M \\
\mathscr{H}_{\mathrm{max}}\big[\mathtt{choose}\,(\mathtt{do}\, x \Leftarrow M \,\mathtt{in}\, N)\,(\mathtt{do}\, x \Leftarrow M \,\mathtt{in}\, P)\big] &= \mathscr{H}_{\mathrm{max}}\big[\mathtt{do}\, x \Leftarrow M \,\mathtt{in}\,(\mathtt{choose}\, N\, P)\big] \\
\mathscr{H}_{\mathrm{max}}\big[\mathtt{do}\, x \Leftarrow M \,\mathtt{in}\,(\mathtt{do}\, y \Leftarrow N \,\mathtt{in}\, P)\big] &= \mathscr{H}_{\mathrm{max}}\big[\mathtt{do}\, y \Leftarrow N \,\mathtt{in}\,(\mathtt{do}\, x \Leftarrow M \,\mathtt{in}\, P)\big]
\end{aligned}
$$

The technique can and **has to be repeated** for any concrete handler

$$\begin{aligned}
\mathscr{H}_{\mathrm{max}}[M] &= \texttt{with}\, H_{\mathrm{max}}\, \texttt{handle}\, M \\
\mathscr{H}_{\mathrm{max}}\big[\texttt{choose}\,(\texttt{do}\, x \Leftarrow M \,\texttt{in}\, N)\,(\texttt{do}\, x \Leftarrow M \,\texttt{in}\, P)\big] &= \mathscr{H}_{\mathrm{max}}\big[\texttt{do}\, x \Leftarrow M \,\texttt{in}\,(\texttt{choose}\, N\, P)\big] \\
\mathscr{H}_{\mathrm{max}}\big[\texttt{do}\, x \Leftarrow M \,\texttt{in}\,(\texttt{do}\, y \Leftarrow N \,\texttt{in}\, P)\big] &= \mathscr{H}_{\mathrm{max}}\big[\texttt{do}\, y \Leftarrow N \,\texttt{in}\,(\texttt{do}\, x \Leftarrow M \,\texttt{in}\, P)\big]
\end{aligned}$$

$$\begin{aligned}
\mathscr{H}_{\mathrm{sum}}[M] &= \texttt{with}\, H_{\mathrm{sum}}\, \texttt{handle}\, M \\
\mathscr{H}_{\mathrm{sum}}\big[\texttt{choose}\,(\texttt{do}\, x \Leftarrow M \,\texttt{in}\, N)\,(\texttt{do}\, x \Leftarrow M \,\texttt{in}\, P)\big] &= \mathscr{H}_{\mathrm{sum}}\big[\texttt{do}\, x \Leftarrow M \,\texttt{in}\,(\texttt{choose}\, N\, P)\big] \\
\mathscr{H}_{\mathrm{sum}}\big[\texttt{do}\, x \Leftarrow M \,\texttt{in}\,(\texttt{do}\, y \Leftarrow N \,\texttt{in}\, P)\big] &= \mathscr{H}_{\mathrm{sum}}\big[\texttt{do}\, y \Leftarrow N \,\texttt{in}\,(\texttt{do}\, x \Leftarrow M \,\texttt{in}\, P)\big]
\end{aligned}$$

The technique can and **has to be repeated** for any concrete handler

$$\begin{aligned}
\mathscr{H}_{\text{max}}[M] &= \mathtt{with}\ H_{\text{max}}\ \mathtt{handle}\ M \\
\mathscr{H}_{\text{max}}\big[\mathtt{choose}\,(\mathtt{do}\ x \Leftarrow M\ \mathtt{in}\ N)\,(\mathtt{do}\ x \Leftarrow M\ \mathtt{in}\ P)\big] &= \mathscr{H}_{\text{max}}\big[\mathtt{do}\ x \Leftarrow M\ \mathtt{in}\,(\mathtt{choose}\ N\ P)\big] \\
\mathscr{H}_{\text{max}}\big[\mathtt{do}\ x \Leftarrow M\ \mathtt{in}\,(\mathtt{do}\ y \Leftarrow N\ \mathtt{in}\ P)\big] &= \mathscr{H}_{\text{max}}\big[\mathtt{do}\ y \Leftarrow N\ \mathtt{in}\,(\mathtt{do}\ x \Leftarrow M\ \mathtt{in}\ P)\big]
\end{aligned}$$

$$\begin{aligned}
\mathscr{H}_{\text{sum}}[M] &= \mathtt{with}\ H_{\text{sum}}\ \mathtt{handle}\ M \\
\mathscr{H}_{\text{sum}}\big[\mathtt{choose}\,(\mathtt{do}\ x \Leftarrow M\ \mathtt{in}\ N)\,(\mathtt{do}\ x \Leftarrow M\ \mathtt{in}\ P)\big] &= \mathscr{H}_{\text{sum}}\big[\mathtt{do}\ x \Leftarrow M\ \mathtt{in}\,(\mathtt{choose}\ N\ P)\big] \\
\mathscr{H}_{\text{sum}}\big[\mathtt{do}\ x \Leftarrow M\ \mathtt{in}\,(\mathtt{do}\ y \Leftarrow N\ \mathtt{in}\ P)\big] &= \mathscr{H}_{\text{sum}}\big[\mathtt{do}\ y \Leftarrow N\ \mathtt{in}\,(\mathtt{do}\ x \Leftarrow M\ \mathtt{in}\ P)\big]
\end{aligned}$$

$$\begin{aligned}
\mathscr{H}_{\text{list}}[M] &= \mathtt{with}\ H_{\text{list}}\ \mathtt{handle}\ M \\
\mathscr{H}_{\text{list}}\big[\mathtt{choose}\,(\mathtt{do}\ x \Leftarrow M\ \mathtt{in}\ N)\,(\mathtt{do}\ x \Leftarrow M\ \mathtt{in}\ P)\big] &= \mathscr{H}_{\text{list}}\big[\mathtt{do}\ x \Leftarrow M\ \mathtt{in}\,(\mathtt{choose}\ N\ P)\big] \\
\mathscr{H}_{\text{list}}\big[\mathtt{do}\ x \Leftarrow M\ \mathtt{in}\,(\mathtt{do}\ y \Leftarrow N\ \mathtt{in}\ P)\big] &\neq \mathscr{H}_{\text{list}}\big[\mathtt{do}\ y \Leftarrow N\ \mathtt{in}\,(\mathtt{do}\ x \Leftarrow M\ \mathtt{in}\ P)\big]
\end{aligned}$$

Different handlers satisfy **varying equations**

				left	
	max	sum	list	right	swap
assoc	✓	✓	✓	✓	✓
comm	✓	✓	✗	✗	✓
idem	✓	✗	✗	✓	✓
unit	✓	✓	✓	✗	✓

Could equations be tracked with an **effect system**?

Could equations be tracked with an **effect system**?

$$[\![\sigma!\varphi]\!] = T_\varphi[\![\sigma]\!]$$

$$[\![\sigma!\varphi/\mathscr{E}]\!] = T_\varphi[\![\sigma]\!]/\sim_{\mathscr{E}}$$

Could equations be tracked with an **effect system**?

$$[\![\sigma!\varphi]\!] = \boxed{T_\varphi}[\![\sigma]\!]$$

(absolutely) free model

$$[\![\sigma!\varphi/\mathscr{E}]\!] = T_\varphi[\![\sigma]\!]/\sim_{\mathscr{E}}$$

Could equations be tracked with an **effect system**?

$$[\![\sigma!\varphi]\!] = \boxed{T_\varphi}[\![\sigma]\!]$$

equivalence relation

(absolutely) free model

$$[\![\sigma!\varphi/\mathscr{E}]\!] = T_\varphi[\![\sigma]\!]/\boxed{\sim_{\mathscr{E}}}$$

Could equations be tracked with an **effect system**?

$$[\![\sigma!\varphi]\!] = \boxed{T_\varphi}[\![\sigma]\!]$$

equivalence relation

(absolutely) free model

$$[\![\sigma!\varphi/\mathscr{E}]\!] = T_\varphi[\![\sigma]\!]/\boxed{\sim_{\mathscr{E}}}$$

$$\Gamma \vdash M : \sigma!\varphi$$

Could equations be tracked with an **effect system**?

$$[\![\sigma!\varphi]\!] = T_\varphi[\![\sigma]\!]$$

(absolutely) free model

equivalence relation

$$[\![\sigma!\varphi/\mathcal{E}]\!] = T_\varphi[\![\sigma]\!]/{\sim_{\mathcal{E}}}$$

operations

$$\Gamma \vdash M : \sigma!\varphi$$

Could equations be tracked with an **effect system**?

$$[\![\sigma ! \varphi]\!] = T_{\varphi}[\![\sigma]\!]$$

(absolutely) free model

equivalence relation

$$[\![\sigma ! \varphi / \mathscr{E}]\!] = T_{\varphi}[\![\sigma]\!] / \sim_{\mathscr{E}}$$

operations

$$\Gamma \vdash M : \sigma ! \varphi / \mathscr{E}$$

Could equations be tracked with an **effect system**?

$$[\![\sigma!\varphi]\!] = \boxed{T_\varphi}[\![\sigma]\!]$$

equivalence relation

(absolutely) free model

$$[\![\sigma!\varphi/\mathcal{E}]\!] = T_\varphi[\![\sigma]\!]/\boxed{\sim_{\mathcal{E}}}$$

operations

equations

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

Previous reasoning can be factored into **two parts**

Previous reasoning can be factored into **two parts**

handlers respect equations

$$
\begin{aligned}
H_{\text{max}} &: \texttt{int}!\{\texttt{choose}, \texttt{fail}\}/\overbrace{\{\texttt{assoc}, \texttt{comm}, \texttt{idem}, \texttt{unit}\}}^{\mathscr{E}} \Rightarrow \texttt{int}!\varnothing/\varnothing \\
H_{\text{sum}} &: \texttt{int}!\{\texttt{choose}, \texttt{fail}\}/\mathscr{E} \Rightarrow \texttt{int}!\varnothing/\varnothing \\
H_{\text{list}} &: \texttt{int}!\{\texttt{choose}, \texttt{fail}\}/\{\texttt{assoc}, \texttt{unit}\} \Rightarrow \texttt{int}\ \texttt{list}!\varnothing/\varnothing \\
H_{\text{left}} &: \tau!\{\texttt{choose}, \texttt{fail}\}/\{\texttt{assoc}, \texttt{idem}, \texttt{unit}\} \Rightarrow \tau!\{\texttt{fail}\}/\varnothing \\
H_{\text{swap}} &: \tau!\{\texttt{choose}, \texttt{fail}\}/\{\texttt{assoc}, \texttt{unit}\} \Rightarrow \tau!\{\texttt{choose}, \texttt{fail}\}/\{\texttt{assoc}, \texttt{unit}\} \\
H_{\text{swap}} &: \tau!\{\texttt{choose}, \texttt{fail}\}/\mathscr{E} \Rightarrow \tau!\{\texttt{choose}, \texttt{fail}\}/\mathscr{E}
\end{aligned}
$$

Previous reasoning can be factored into **two parts**

handlers respect equations

$$
\begin{aligned}
H_{\text{max}} &: \mathtt{int}!\{\mathtt{choose}, \mathtt{fail}\}/\overbrace{\{\mathtt{assoc}, \mathtt{comm}, \mathtt{idem}, \mathtt{unit}\}}^{\mathscr{E}} \Rightarrow \mathtt{int}!\varnothing/\varnothing \\
H_{\text{sum}} &: \mathtt{int}!\{\mathtt{choose}, \mathtt{fail}\}/\mathscr{E} \Rightarrow \mathtt{int}!\varnothing/\varnothing \\
H_{\text{list}} &: \mathtt{int}!\{\mathtt{choose}, \mathtt{fail}\}/\{\mathtt{assoc}, \mathtt{unit}\} \Rightarrow \mathtt{int}\ \mathtt{list}!\varnothing/\varnothing \\
H_{\text{left}} &: \tau!\{\mathtt{choose}, \mathtt{fail}\}/\{\mathtt{assoc}, \mathtt{idem}, \mathtt{unit}\} \Rightarrow \tau!\{\mathtt{fail}\}/\varnothing \\
H_{\text{swap}} &: \tau!\{\mathtt{choose}, \mathtt{fail}\}/\{\mathtt{assoc}, \mathtt{unit}\} \Rightarrow \tau!\{\mathtt{choose}, \mathtt{fail}\}/\{\mathtt{assoc}, \mathtt{unit}\} \\
H_{\text{swap}} &: \tau!\{\mathtt{choose}, \mathtt{fail}\}/\mathscr{E} \Rightarrow \tau!\{\mathtt{choose}, \mathtt{fail}\}/\mathscr{E}
\end{aligned}
$$

Previous reasoning can be factored into **two parts**

handlers respect equations

$$H_{\mathrm{max}} : \texttt{int!\{choose, fail\}/}\overbrace{\texttt{\{assoc, comm, idem, unit\}}}^{\mathscr{E}} \Rightarrow \texttt{int!}\varnothing/\varnothing$$

$$H_{\mathrm{sum}} : \texttt{int!\{choose, fail\}}/\mathscr{E} \Rightarrow \texttt{int!}\varnothing/\varnothing$$

$$H_{\mathrm{list}} : \texttt{int!\{choose, fail\}/\{assoc, unit\}} \Rightarrow \texttt{int list!}\varnothing/\varnothing$$

$$H_{\mathrm{left}} : \tau\texttt{!\{choose, fail\}/\{assoc, idem, unit\}} \Rightarrow \tau\texttt{!\{fail\}/}\varnothing$$

$$H_{\mathrm{swap}} : \tau\texttt{!\{choose, fail\}/\{assoc, unit\}} \Rightarrow \tau\texttt{!\{choose, fail\}/\{assoc, unit\}}$$

$$H_{\mathrm{swap}} : \tau\texttt{!\{choose, fail\}}/\mathscr{E} \Rightarrow \tau\texttt{!\{choose, fail\}}/\mathscr{E}$$

equations imply properties

$$\texttt{choose}\,(\texttt{do}\,x \Leftarrow M\,\texttt{in}\,N)\,(\texttt{do}\,x \Leftarrow M\,\texttt{in}\,P) =_{\mathscr{E}} \texttt{do}\,x \Leftarrow M\,\texttt{in}\,(\texttt{choose}\,N\,P)$$

$$\texttt{do}\,x \Leftarrow M\,\texttt{in}\,(\texttt{do}\,y \Leftarrow N\,\texttt{in}\,P) =_{\mathscr{E}} \texttt{do}\,y \Leftarrow N\,\texttt{in}\,(\texttt{do}\,x \Leftarrow M\,\texttt{in}\,P)$$

Typing rules for **monadic constructs**

$$\frac{\Gamma \vdash V : \sigma}{\Gamma \vdash \mathtt{val}\, V : \sigma ! \varphi / \mathscr{E}}$$

$$\frac{\Gamma \vdash M : \sigma ! \varphi / \mathscr{E} \qquad \Gamma, x : \sigma \vdash N : \tau ! \varphi / \mathscr{E}}{\Gamma \vdash \mathtt{do}\, x \Leftarrow M \,\mathtt{in}\, P : \tau ! \varphi / \mathscr{E}}$$

$$\frac{\Gamma \vdash V : \sigma \qquad (F : \sigma \twoheadrightarrow \tau) \in \varphi}{\Gamma \vdash \mathtt{perform}\, F\, V : \tau ! \varphi / \mathscr{E}}$$

Handling and **subsumption** typing rules

$$\frac{\Gamma \vdash H : \sigma!\varphi/\mathscr{E} \Rightarrow \tau!\varphi'/\mathscr{E}' \qquad \Gamma \vdash M : \sigma!\varphi/\mathscr{E}}{\Gamma \vdash \mathtt{with}\, H\, \mathtt{handle}\, M : \tau!\varphi'/\mathscr{E}'}$$

$$\frac{\Gamma \vdash M : \sigma!\varphi/\mathscr{E} \qquad \sigma <: \sigma' \quad \varphi \subseteq \varphi' \quad \mathscr{E}' \vDash \mathscr{E}}{\Gamma \vdash M : \sigma'!\varphi'/\mathscr{E}'}$$

Handling and subsumption typing rules

$$\frac{\Gamma \vdash H : \sigma ! \varphi / \mathscr{E} \Rightarrow \tau ! \varphi' / \mathscr{E}' \qquad \Gamma \vdash M : \sigma ! \varphi / \mathscr{E}}{\Gamma \vdash \mathtt{with}\, H\, \mathtt{handle}\, M : \tau ! \varphi' / \mathscr{E}'}$$

$$\frac{\Gamma \vdash M : \sigma ! \varphi / \mathscr{E} \qquad \sigma <: \sigma' \quad \varphi \subseteq \varphi' \quad \mathscr{E}' \vDash \mathscr{E}}{\Gamma \vdash M : \sigma' ! \varphi' / \mathscr{E}'}$$

Handler typing rule and **instantiation** of theory **equations**

$$\frac{\Gamma \vdash H{:}\sigma!\varphi \rightsquigarrow \tau!\varphi'/\mathscr{E}' \qquad \left[\Gamma \vdash H[T_1] = H[T_2]{:}\tau!\varphi'/\mathscr{E}'\right]_{(T_1=T_2)\in\mathscr{E}}}{\Gamma \vdash H : \sigma!\varphi/\mathscr{E} \Rightarrow \tau!\varphi'/\mathscr{E}'}$$

$$\frac{(T_1 = T_2) \in \mathscr{E} \qquad [\Gamma \vdash M_i : \sigma!\varphi/\mathscr{E}]_i}{\Gamma \vdash T_1(M_1, \ldots, M_n) = T_2(M_1, \ldots, M_n) : \sigma!\varphi/\mathscr{E}}$$

Handler typing rule and **instantiation** of theory **equations**

well-typed handling clauses

$$\frac{\boxed{\Gamma \vdash H{:}\sigma!\varphi \rightsquigarrow \tau!\varphi'/\mathscr{E}'} \qquad \left[\Gamma \vdash H[T_1] = H[T_2]{:}\tau!\varphi'/\mathscr{E}'\right]_{(T_1=T_2)\in\mathscr{E}}}{\Gamma \vdash H : \sigma!\varphi/\mathscr{E} \Rightarrow \tau!\varphi'/\mathscr{E}'}$$

$$\frac{(T_1 = T_2) \in \mathscr{E} \qquad [\Gamma \vdash M_i : \sigma!\varphi/\mathscr{E}]_i}{\Gamma \vdash T_1(M_1, \ldots, M_n) = T_2(M_1, \ldots, M_n) : \sigma!\varphi/\mathscr{E}}$$

Handler typing rule and **instantiation** of theory **equations**

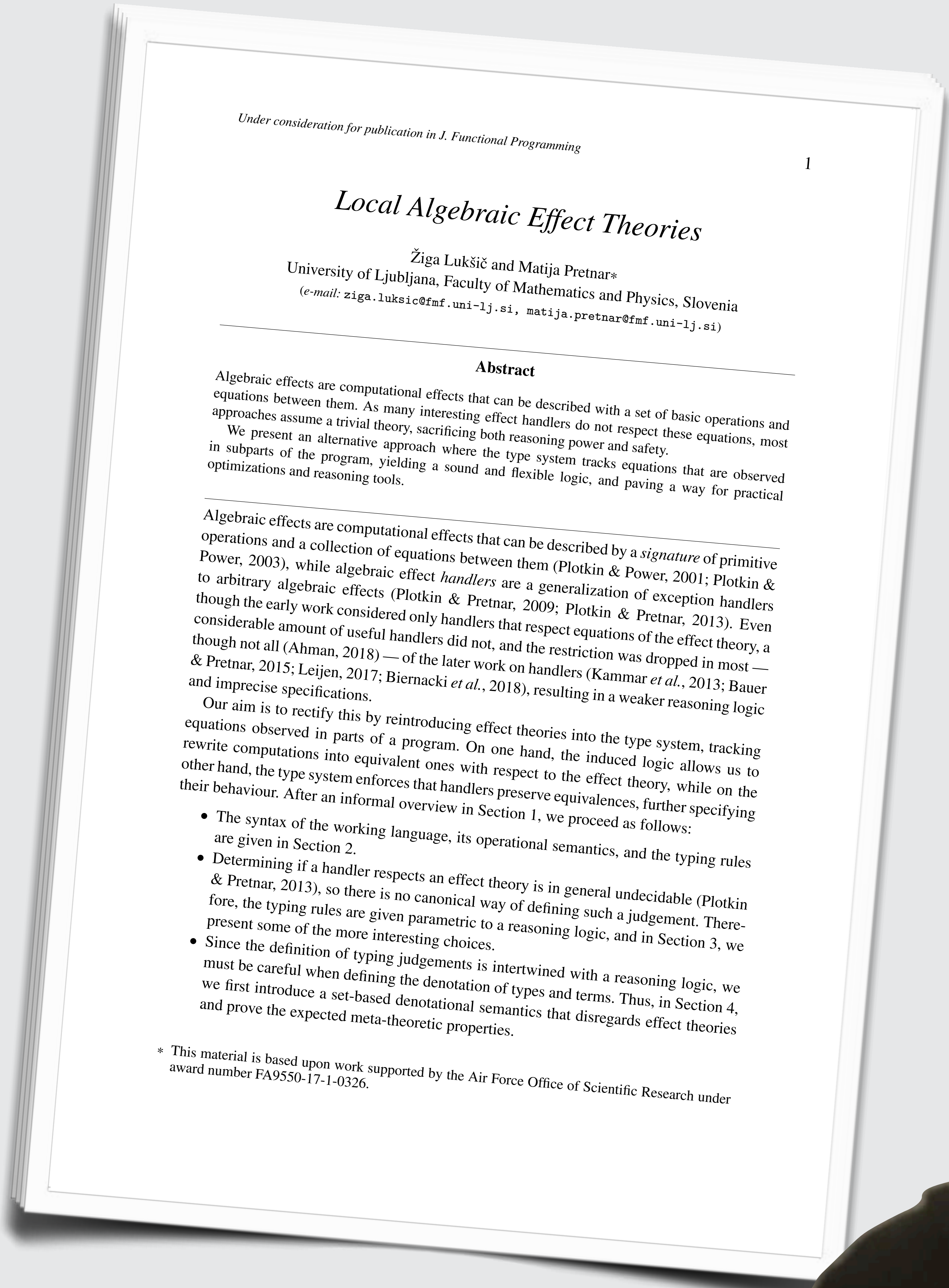
well-typed handling clauses

$$\frac{\begin{array}{c}\boxed{\Gamma \vdash H{:}\sigma!\varphi \rightsquigarrow \tau!\varphi'/\mathscr{E}'} \\ \boxed{\left[\Gamma \vdash H[T_1] = H[T_2]{:}\tau!\varphi'/\mathscr{E}'\right]_{(T_1=T_2)\in\mathscr{E}}}\end{array}}{\Gamma \vdash H : \sigma!\varphi/\mathscr{E} \Rightarrow \tau!\varphi'/\mathscr{E}'}$$

respecting equations

$$\frac{(T_1 = T_2) \in \mathscr{E} \qquad [\Gamma \vdash M_i : \sigma!\varphi/\mathscr{E}]_i}{\Gamma \vdash T_1(M_1, \ldots, M_n) = T_2(M_1, \ldots, M_n) : \sigma!\varphi/\mathscr{E}}$$

This work has only **partly** been **put into practice**



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. As many interesting effect handlers do not respect these equations, most approaches assume a trivial theory, sacrificing both reasoning power and safety.

We present an alternative approach where the type system tracks equations that are observed in subparts of the program, yielding a sound and flexible logic, and paving a way for practical optimizations and reasoning tools.

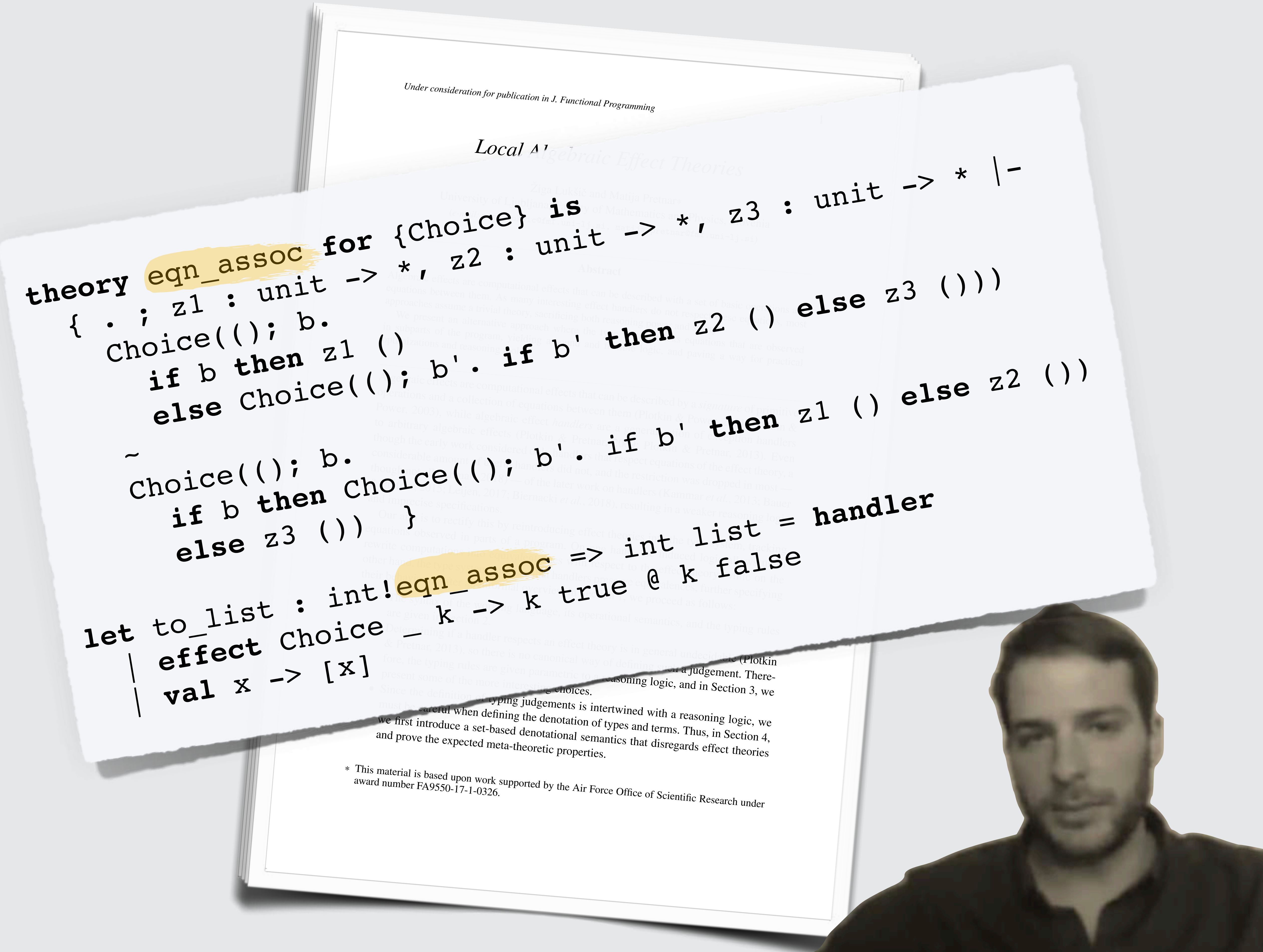
Algebraic effects are computational effects that can be described by a *signature* of primitive operations and a collection of equations between them (Plotkin & Power, 2001; Plotkin & Power, 2003), while algebraic effect *handlers* are a generalization of exception handlers to arbitrary algebraic effects (Plotkin & Pretnar, 2009; Plotkin & Pretnar, 2013). Even though the early work considered only handlers that respect equations of the effect theory, a considerable amount of useful handlers did not, and the restriction was dropped in most — though not all (Ahman, 2018) — of the later work on handlers (Kammar *et al.*, 2013; Bauer & Pretnar, 2015; Leijen, 2017; Biernacki *et al.*, 2018), resulting in a weaker reasoning logic and imprecise specifications.

Our aim is to rectify this by reintroducing effect theories into the type system, tracking equations observed in parts of a program. On one hand, the induced logic allows us to rewrite computations into equivalent ones with respect to the effect theory, while on the other hand, the type system enforces that handlers preserve equivalences, further specifying their behaviour. After an informal overview in Section 1, we proceed as follows:

- The syntax of the working language, its operational semantics, and the typing rules are given in Section 2.
- Determining if a handler respects an effect theory is in general undecidable (Plotkin & Pretnar, 2013), so there is no canonical way of defining such a judgement. Therefore, the typing rules are given parametric to a reasoning logic, and in Section 3, we present some of the more interesting choices.
- Since the definition of typing judgements is intertwined with a reasoning logic, we must be careful when defining the denotation of types and terms. Thus, in Section 4, we first introduce a set-based denotational semantics that disregards effect theories and prove the expected meta-theoretic properties.

* This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0326.

This work has only **partly** been **put into practice**



Under consideration for publication in J. Functional Programming

Local Algebraic Effect Theories

```
theory eqn_assoc for {Choice} is
  { . ; z1 : unit -> *, z2 : unit -> *, z3 : unit -> * |-
    Choice((); b.
      if b then z1 ()
      else Choice((); b'. if b' then z2 () else z3 ()))
  ~
    Choice((); b.
      if b then Choice((); b'. if b' then z1 () else z2 ())
      else z3 ()) }

let to_list : int!eqn_assoc => int list = handler
  | effect Choice _ k -> k true @ k false
  | val x -> [x]
```

(Plotkin … a judgement. Therefore, … reasoning logic, and in Section 3, we … choices.

… typing judgements is intertwined with a reasoning logic, we … careful when defining the denotation of types and terms. Thus, in Section 4, we first introduce a set-based denotational semantics that disregards effect theories and prove the expected meta-theoretic properties.

* This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0326.

HANDLERS

HANDLERS

The **simplest** way of efficiently executing Eff was **through OCaml**

Eff

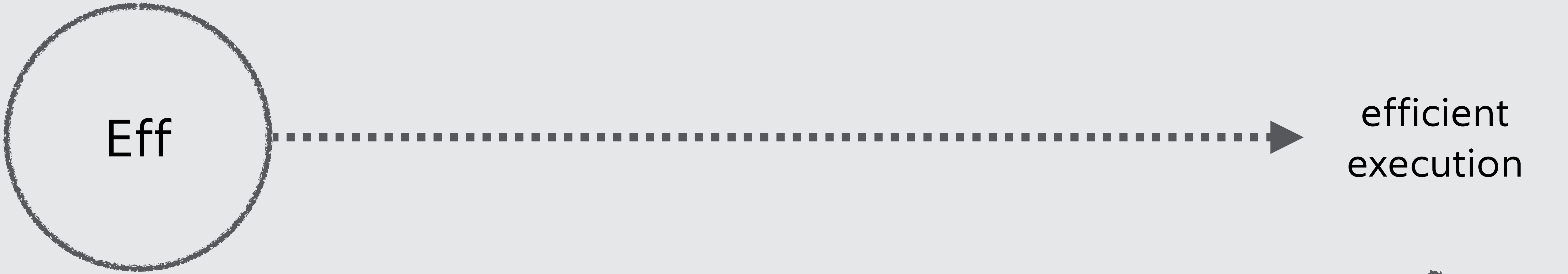
efficient
execution

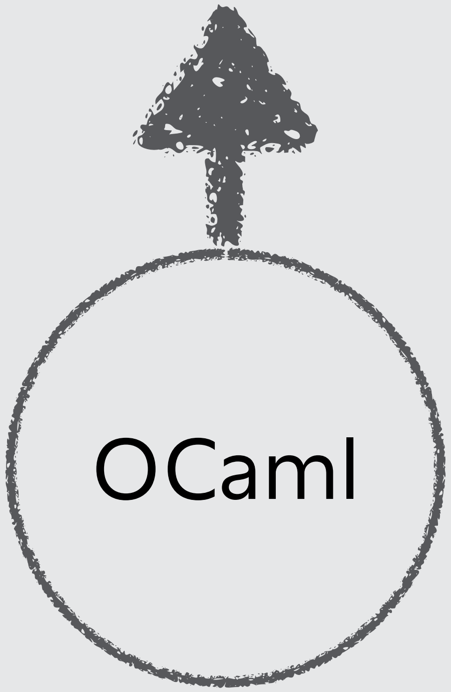
The **simplest** way of efficiently executing Eff was **through OCaml**

Eff

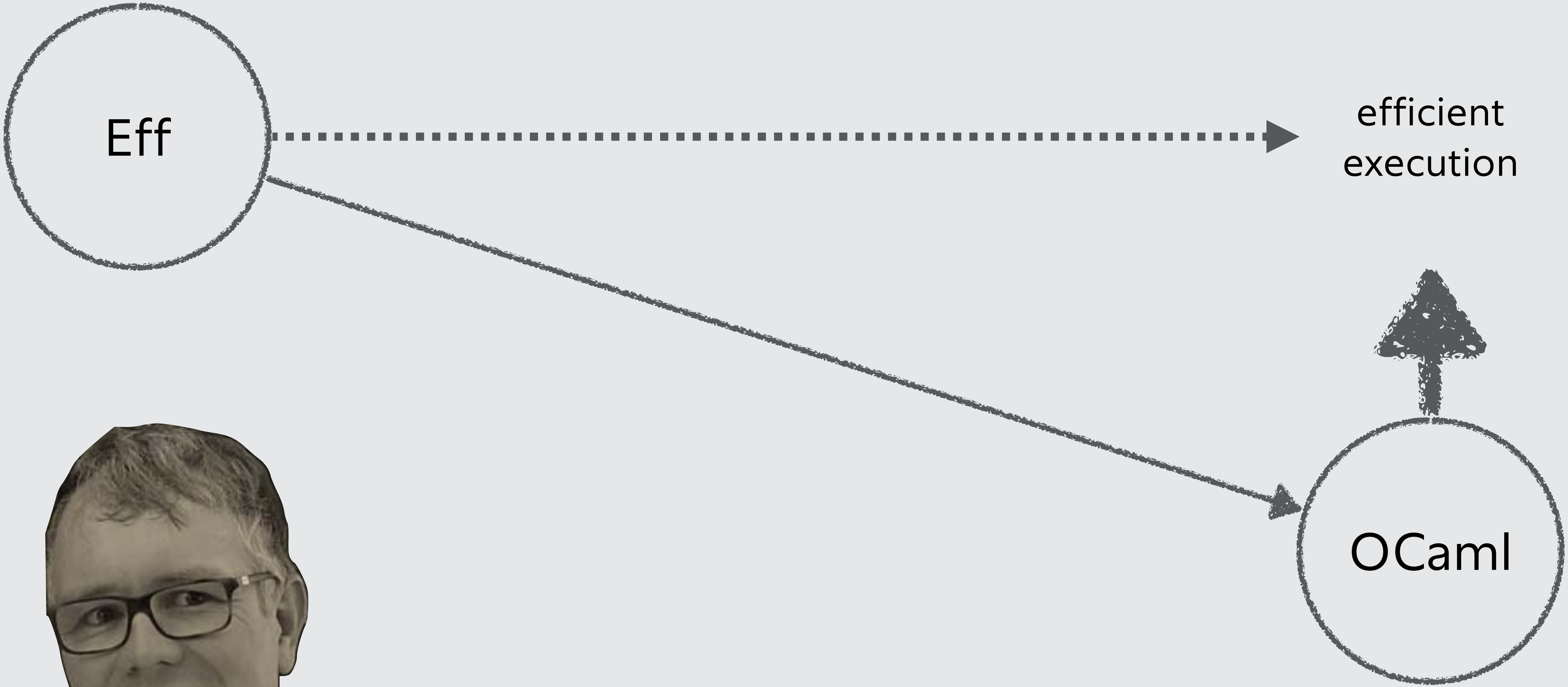
efficient execution

The **simplest** way of efficiently executing Eff was **through OCaml**





The **simplest** way of efficiently executing Eff was **through OCaml**



Eff

efficient execution

OCaml

Since OCaml did **not have handlers**, we did **monadic embedding**

Since OCaml did **not have handlers**, we did **monadic embedding**

free monad

```
type ‘a comp =
   | Return of ‘a
   | Get of unit * (int -> ‘a comp)
   | Set of int * (unit -> ‘a comp)

type (‘a, ‘b) handler = { (* handler clauses *) }
```

Since OCaml did **not have handlers**, we did **monadic embedding**

free monad

```
type ‘a comp =
  | Return of ‘a
  | Get of unit * (int -> ‘a comp)
  | Set of int * (unit -> ‘a comp)

type (‘a, ‘b) handler = { (* handler clauses *) }
```

operations

```
val return : ‘a -> ‘a comp
val (>>=) : ‘a comp -> (‘a -> ‘b comp) -> ‘b comp
val map : (‘a -> ‘b) -> ‘a comp -> ‘b comp

val get : unit -> int comp
val set : int -> unit comp

val handle : (‘a, ‘b) handler -> ‘a comp -> ‘b comp
```

Since OCaml did **not have handlers**, we did **monadic embedding**

free monad

```
type ‘a comp =
   | Return of ‘a
   | Get of unit * (int -> ‘a comp)
   | Set of int * (unit -> ‘a comp)

type (‘a, ‘b) handler = { (* handler clauses *) }
```

operations

```
val return : ‘a -> ‘a comp
val (>>=) : ‘a comp -> (‘a -> ‘b comp) -> ‘b comp
val map : (‘a -> ‘b) -> ‘a comp -> ‘b comp

val get : unit -> int comp
val set : int -> unit comp

val handle : (‘a, ‘b) handler -> ‘a comp -> ‘b comp
```

First component was a **purity-aware** translation

source Eff

```
let y = perform Get in
perform (Set (y + 1));
loop (n - 1)
```

First component was a **purity-aware** translation

source Eff

```
let y = perform Get in
perform (Set (y + 1));
loop (n - 1)
```

desired OCaml

```
get () >>= fun y ->
set (y+1) >>= fun _ ->
loop (n - 1)
```

First component was a **purity-aware** translation

source Eff

```
let y = perform Get in
perform (Set (y + 1));
loop (n - 1)
```

generated OCaml

```
get () >>= fun y ->
(+) y >>= fun f ->
f 1 >>= z ->
set y >>= fun _ ->
(-) n >>= fun g ->
g 1 >>= m
loop m
```

desired OCaml

```
get () >>= fun y ->
set (y+1) >>= fun _ ->
loop (n - 1)
```

First component was a **purity-aware** translation

source Eff

```
let y = perform Get in
perform (Set (y + 1));
loop (n - 1)
```

generated OCaml

```
get () >>= fun y ->
(+) y >>= fun f ->
f 1 >>= z ->
set y >>= fun _ ->
(-) n >>= fun g ->
g 1 >>= m
loop m
```

desired OCaml

```
get () >>= fun y ->
set (y+1) >>= fun _ ->
loop (n - 1)
```

First component was a purity-aware translation

source Eff

```
let y = perform Get in
perform (Set (y + 1));
loop (n - 1)
```

purity-aware translation

```
get () >>= fun y ->
let f = (+) y in
let z = f 1 in
set y >>= fun _ ->
let g = (-) n in
let m = g 1 in
loop m
```

generated OCaml

```
get () >>= fun y ->
(+) y >>= fun f ->
f 1 >>= z ->
set y >>= fun _ ->
(-) n >>= fun g ->
g 1 >>= m
loop m
```

desired OCaml

```
get () >>= fun y ->
set (y+1) >>= fun _ ->
loop (n - 1)
```

First component was a purity-aware translation

source Eff

```
let y = perform Get in
perform (Set (y + 1));
loop (n - 1)
```

generated OCaml

```
get () >>= fun y ->
(+) y >>= fun f ->
f 1 >>= z ->
set y >>= fun _ ->
(-) n >>= fun g ->
g 1 >>= m
loop m
```

$$\mathscr{C}(\sigma!\varphi) = \begin{cases} \mathscr{C}(\sigma) & \varphi = \emptyset \\ \mathscr{C}(\sigma)\,\mathtt{comp} & \varphi \neq \emptyset \end{cases}$$

purity-aware translation

```
get () >>= fun y ->
let f = (+) y in
let z = f 1 in
set y >>= fun _ ->
let g = (-) n in
let m = g 1 in
loop m
```

desired OCaml

```
get () >>= fun y ->
set (y+1) >>= fun _ ->
loop (n - 1)
```

Second component was **inlining handler** definitions

Second component was **inlining handler** definitions

stateful function

```
let rec loop n =
  if n = 0 then () else
    let y = perform Get () in
    perform (Set (y + 1));
    loop (n - 1)
```

Second component was **inlining handler** definitions

stateful function

```
let rec loop n =
  if n = 0 then () else
    let y = perform Get () in
    perform (Set (y + 1));
    loop (n - 1)
```

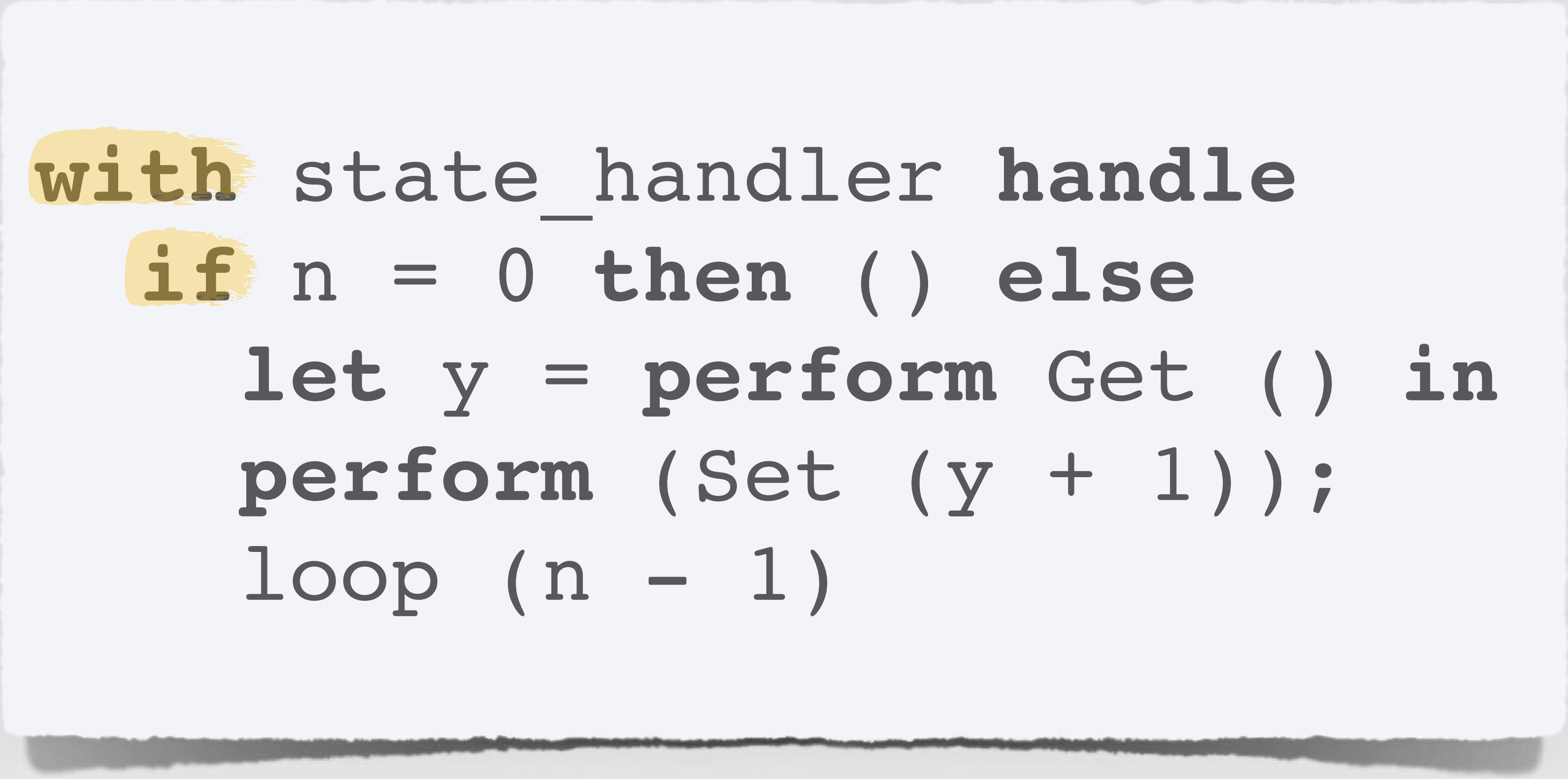
handler for state

```
let state_handler = handler
  | effect (Get ()) k -> (fun s -> k s s)
  | effect (Set s') k -> (fun _ -> k () s')
  | _ -> (fun s -> s)
```

Handlers get **pushed inside** other constructs

```
with state_handler handle
  if n = 0 then () else
    let y = perform Get () in
    perform (Set (y + 1));
    loop (n - 1)
```

Handlers get **pushed inside** other constructs



```
with state_handler handle
  if n = 0 then () else
    let y = perform Get () in
    perform (Set (y + 1));
    loop (n - 1)
```

Handlers get **pushed inside** other constructs

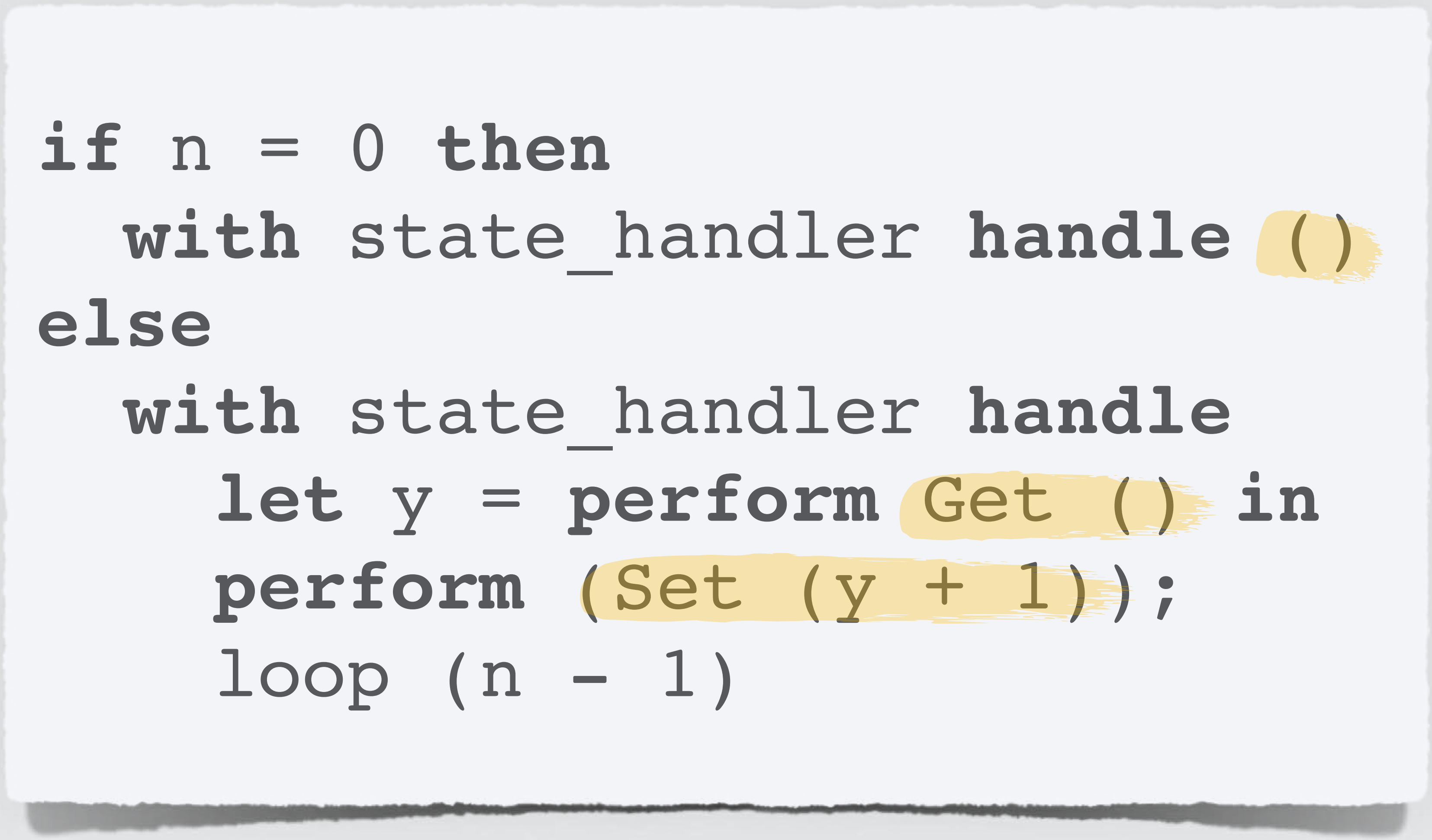
```
with state_handler handle
  if n = 0 then () else
    let y = perform Get () in
    perform (Set (y + 1));
    loop (n - 1)
```

```
if n = 0 then
  with state_handler handle ()
else
  with state_handler handle
    let y = perform Get () in
    perform (Set (y + 1));
    loop (n - 1)
```

We **unfold the handler** once encountering a value or operation

```
if n = 0 then
  with state_handler handle ()
else
  with state_handler handle
    let y = perform Get () in
    perform (Set (y + 1));
    loop (n - 1)
```

We **unfold the handler** once encountering a value or operation



```
if n = 0 then
  with state_handler handle ()
else
  with state_handler handle
    let y = perform Get () in
    perform (Set (y + 1));
    loop (n - 1)
```

We **unfold the handler** once encountering a value or operation

```
if n = 0 then
  with state_handler handle ()
else
  with state_handler handle
    let y = perform Get () in
    perform (Set (y + 1));
    loop (n - 1)
```

```
if n = 0 then (fun s -> s) else
  fun s -> with state_handler handle
      loop (n - 1)
    ) (s + 1)
```

We **unfold the handler** once encountering a value or operation

```
if n = 0 then
  with state_handler handle ()
else
  with state_handler handle
    let y = perform Get () in
    perform (Set (y + 1));
    loop (n - 1)
```

```
if n = 0 then (fun s -> s) else
  fun s -> with state_handler handle
      loop (n - 1)
    ) (s + 1)
```

For functions, we use function **specialisation** & **fusion**

For functions, we use function **specialisation** & **fusion**

```
let rec loop’ n =
  with state_handler handle loop n
```

For functions, we use function **specialisation** & **fusion**

```
let rec loop’ n =
  with state_handler handle loop n
```

```
let rec loop’ n =
  with state_handler handle (* …loop body… *)
```

For functions, we use function **specialisation** & **fusion**

```
let rec loop' n =
  with state_handler handle loop n
```

```
let rec loop' n =
  with state_handler handle (* ...loop body... *)
```

```
let rec loop' n s =
  if n = 0 then s else
   (with state_handler handle loop (n - 1))
     (s + 1)
```

For functions, we use function **specialisation** & **fusion**

```
let rec loop’ n =
  with state_handler handle loop n
```

```
let rec loop’ n =
  with state_handler handle (* …loop body… *)
```

```
let rec loop’ n s =
  if n = 0 then s else
   (with state_handler handle loop (n - 1))
     (s + 1)
```

For functions, we use function **specialisation** & **fusion**

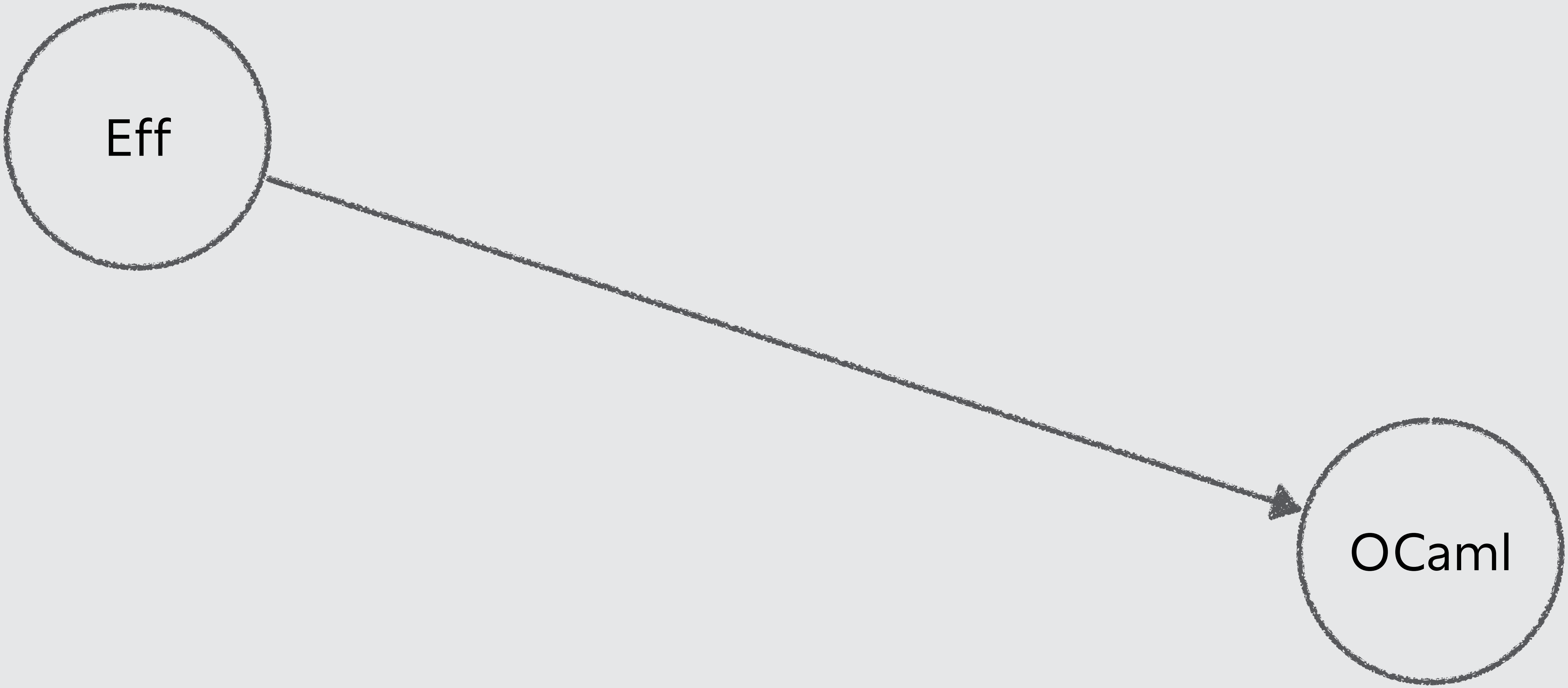
```
let rec loop’ n =
  with state_handler handle loop n
```

```
let rec loop’ n =
  with state_handler handle (* …loop body… *)
```

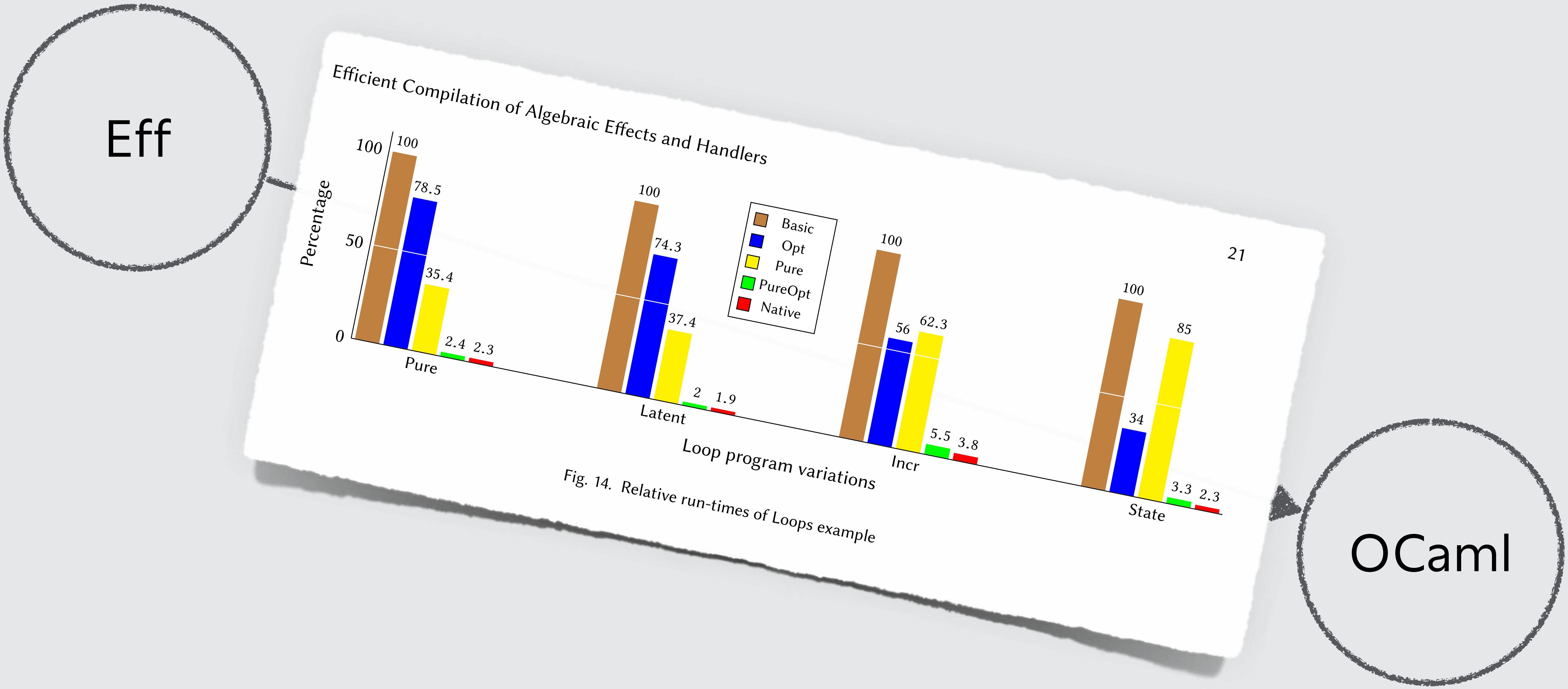
```
let rec loop’ n s =
  if n = 0 then s else
   (with state_handler handle loop (n - 1))
    (s + 1)
```

```
let rec loop’ n s =
  if n = 0 then s else loop’ (n - 1) (s + 1)
```

Purity-aware translation proved **tricky** due to **implicit typing** information



Purity-aware translation proved **tricky** due to **implicit typing** information

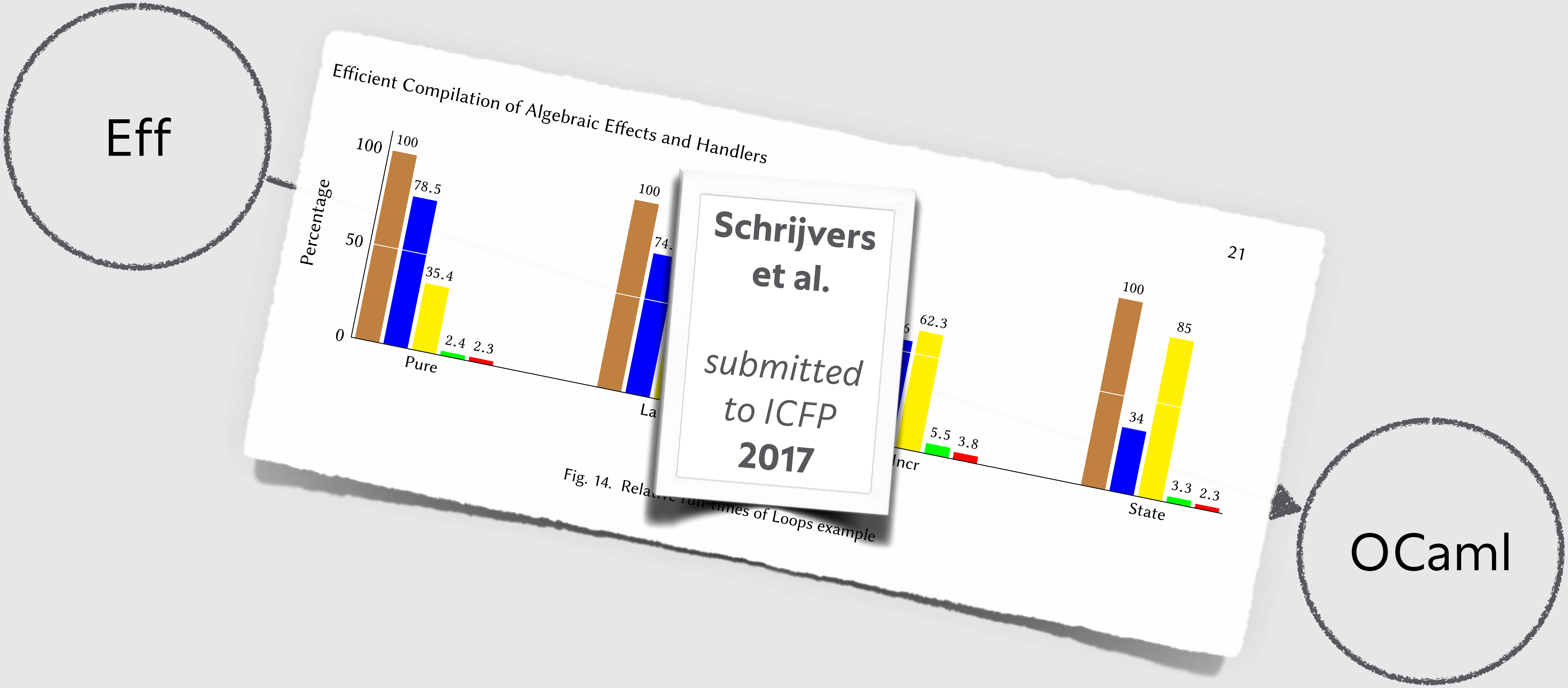


Efficient Compilation of Algebraic Effects and Handlers

21

Fig. 14. Relative run-times of Loops example

Purity-aware translation proved **tricky** due to **implicit typing** information



Efficient Compilation of Algebraic Effects and Handlers

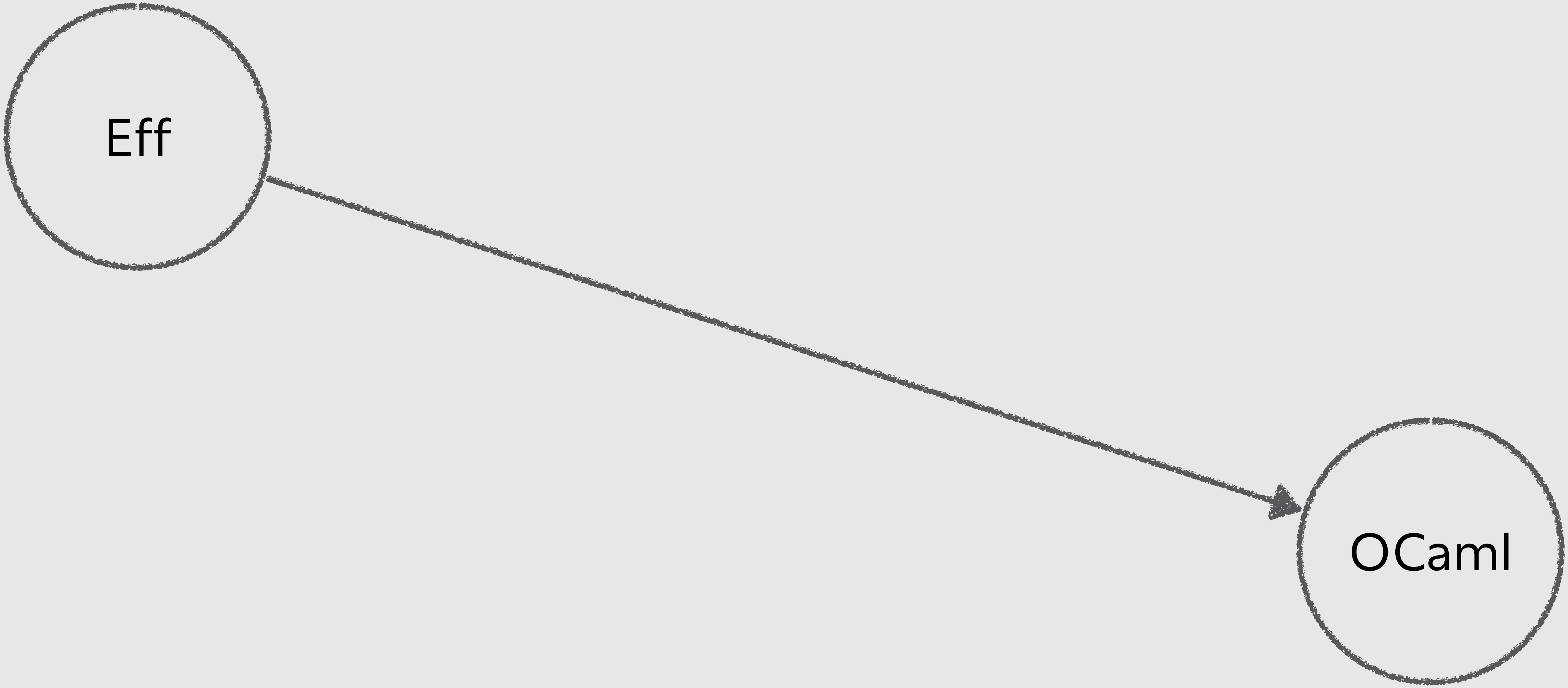
21

Fig. 14. Relative run-times of Loops example

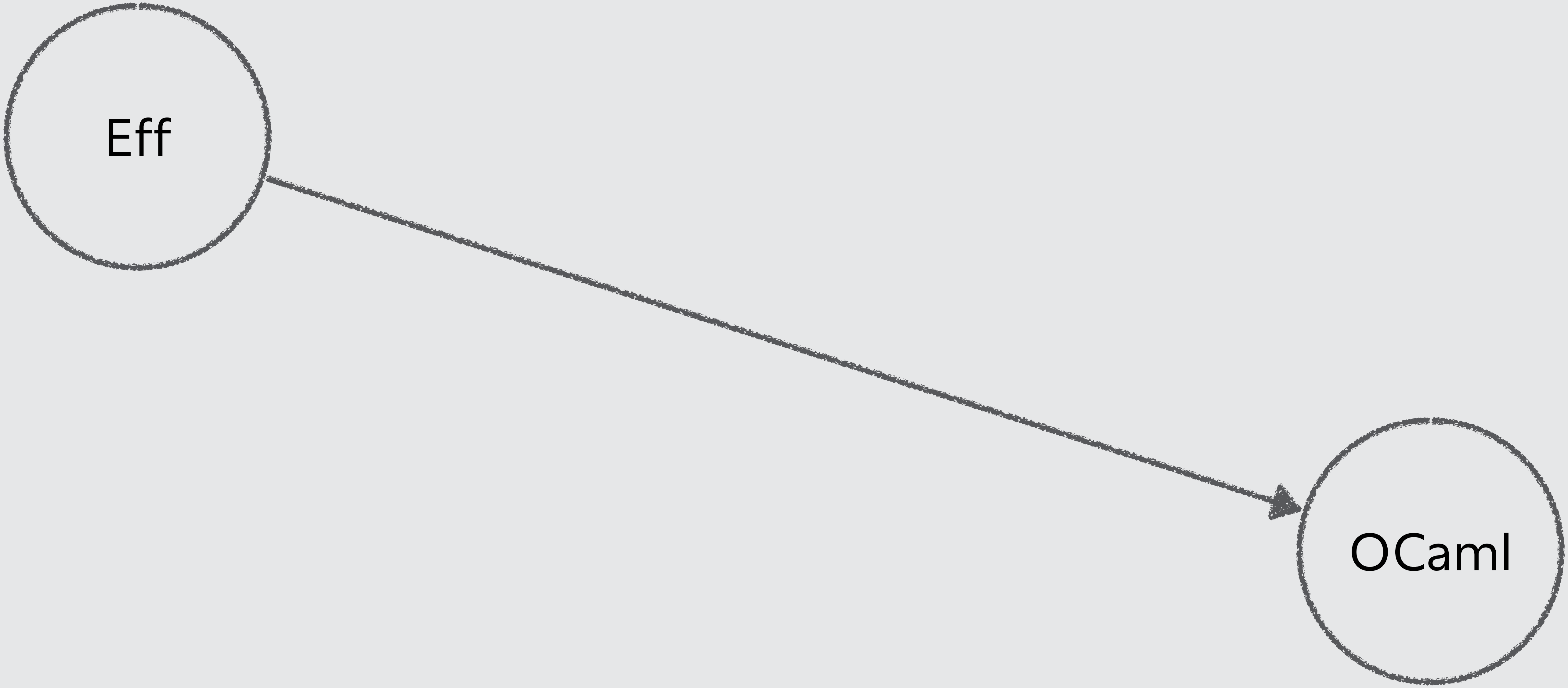
Schrijvers et al.

submitted to ICFP **2017**

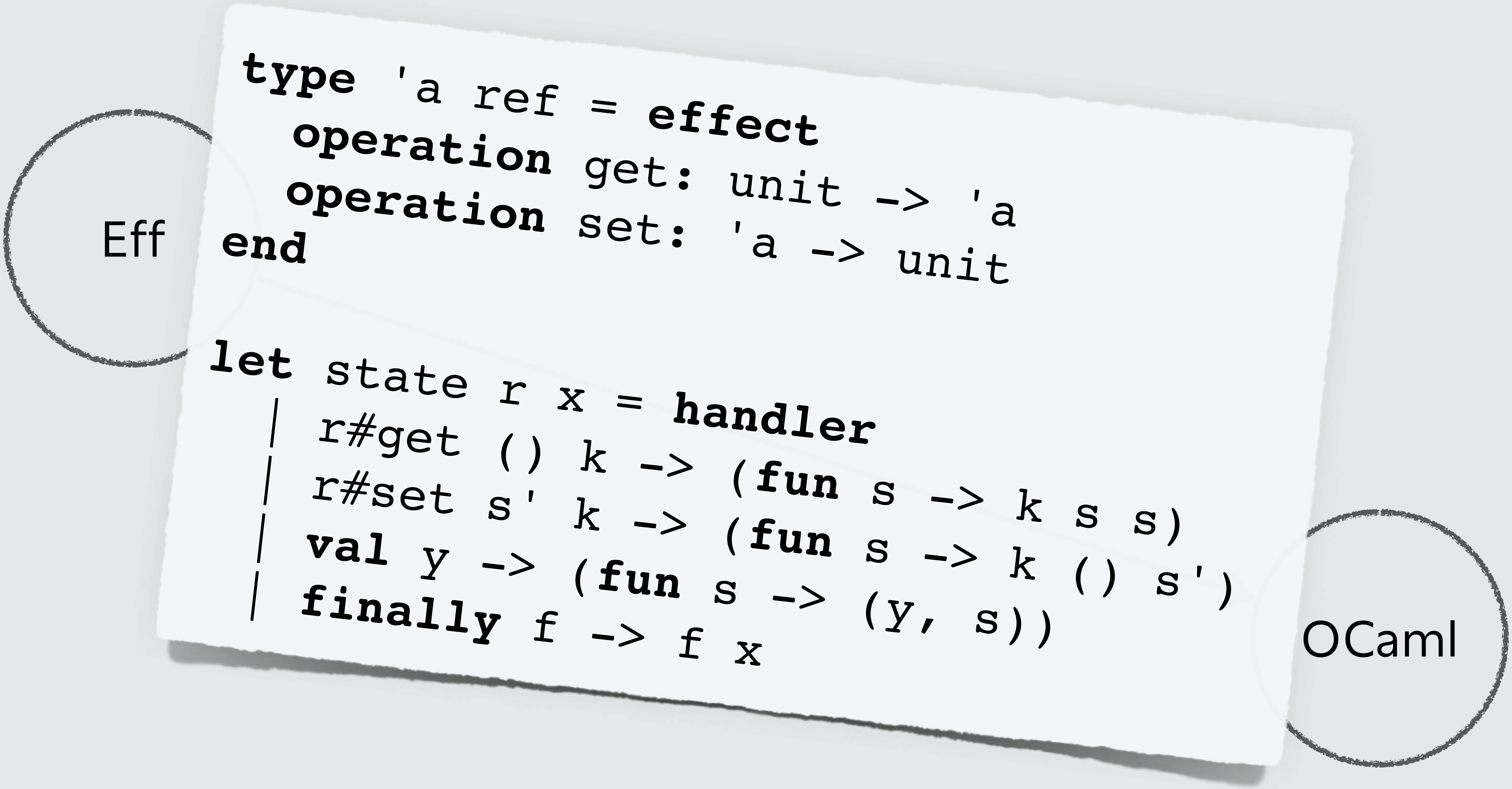
Purity-aware translation proved **tricky** due to **implicit typing** information



One thing removed for simplicity were **effect instances**



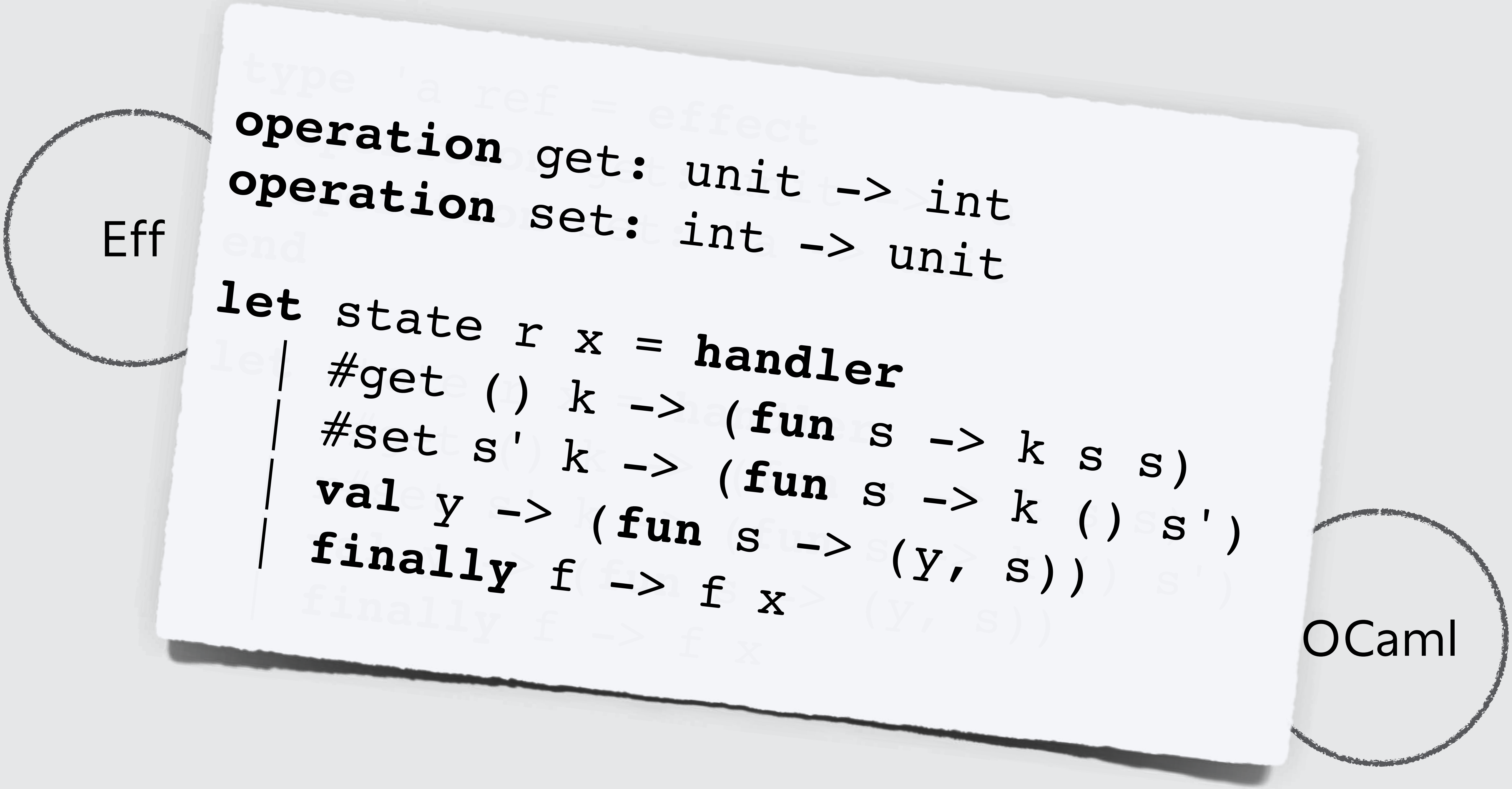
One thing removed for simplicity were **effect instances**



```
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
```

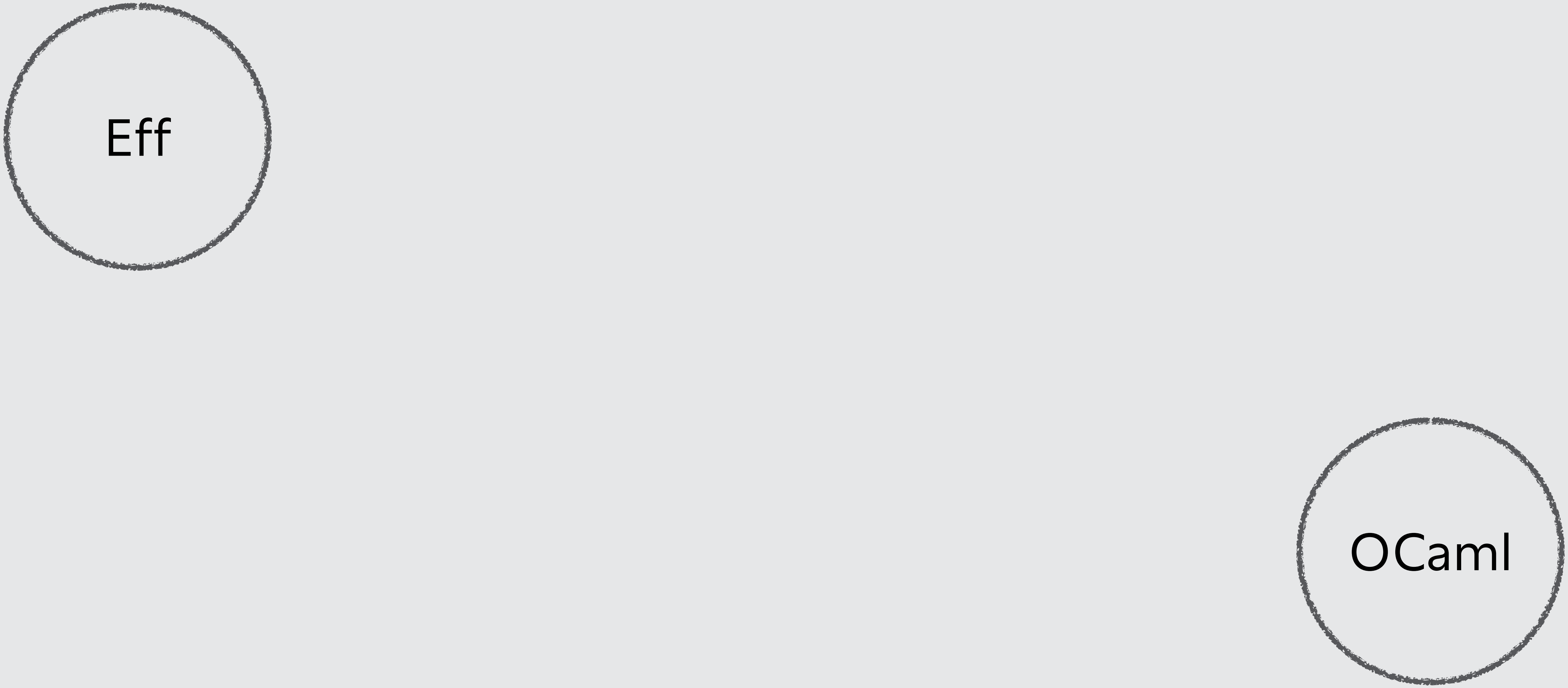
One thing removed for simplicity were **effect instances**



```
operation get: unit -> int
operation set: int -> unit

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

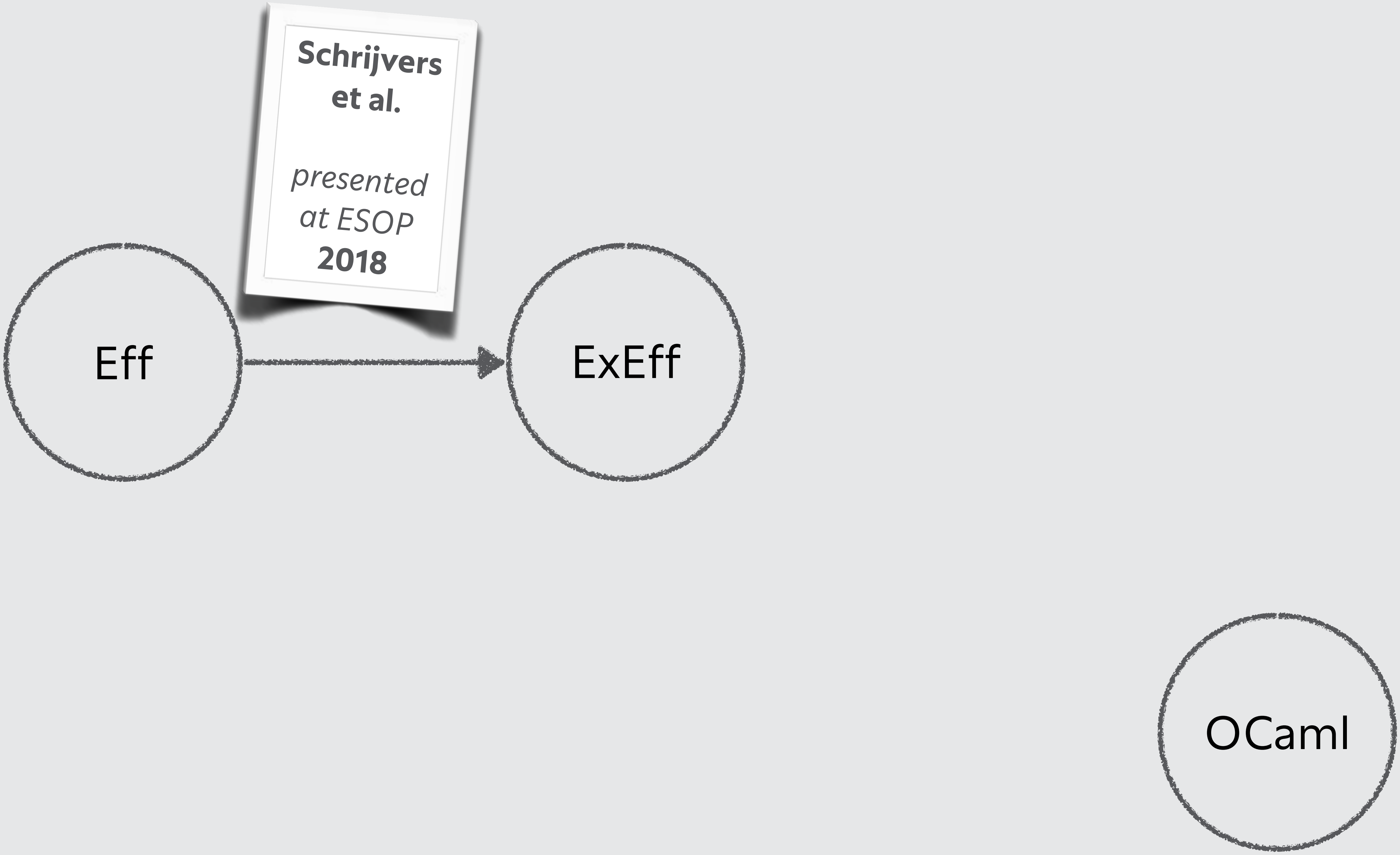
The main step was adding **coercions** as witnesses of **subtyping**



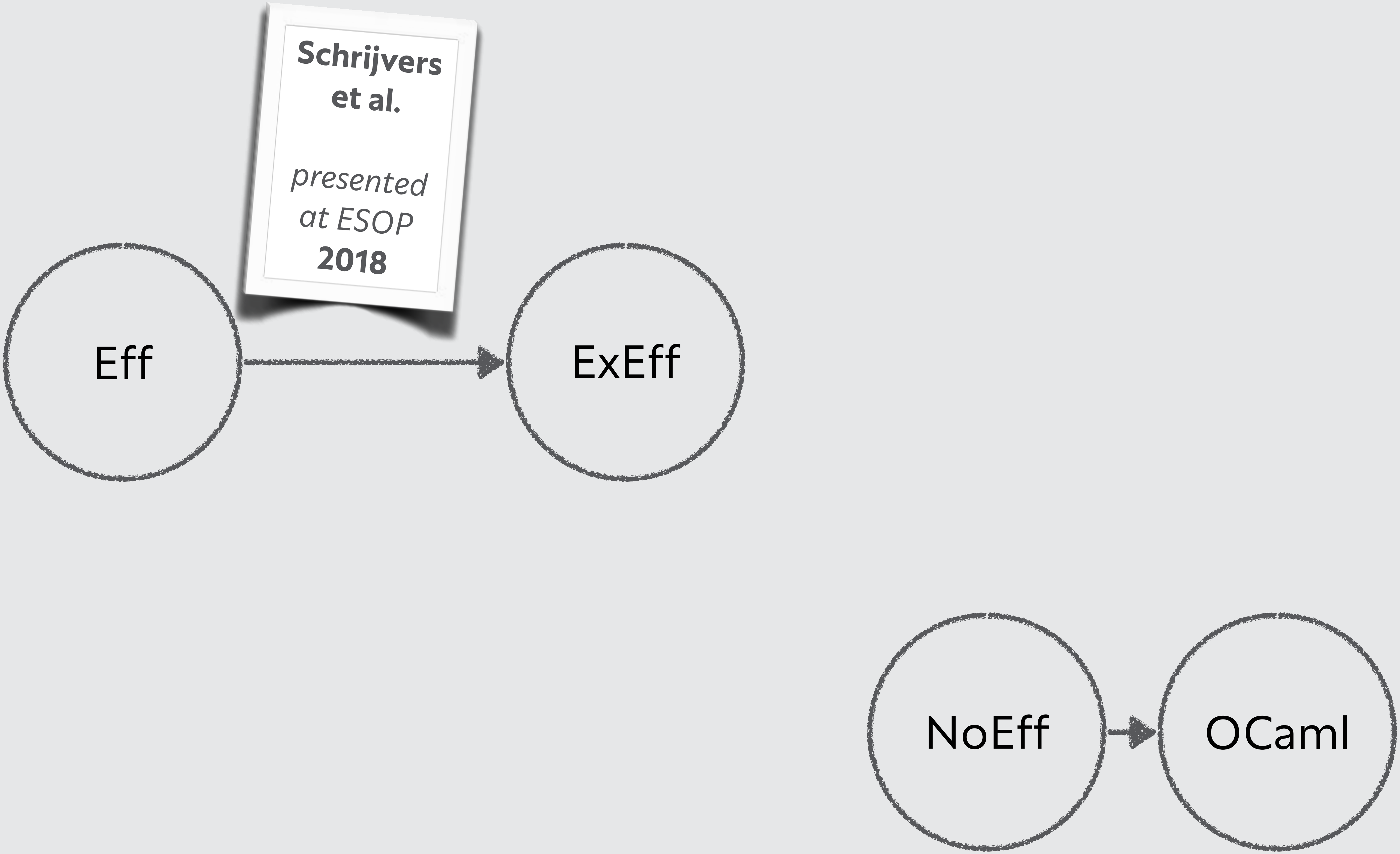
Eff

OCaml

The main step was adding **coercions** as witnesses of **subtyping**



The main step was adding **coercions** as witnesses of **subtyping**



Schrijvers et al.

presented at ESOP **2018**

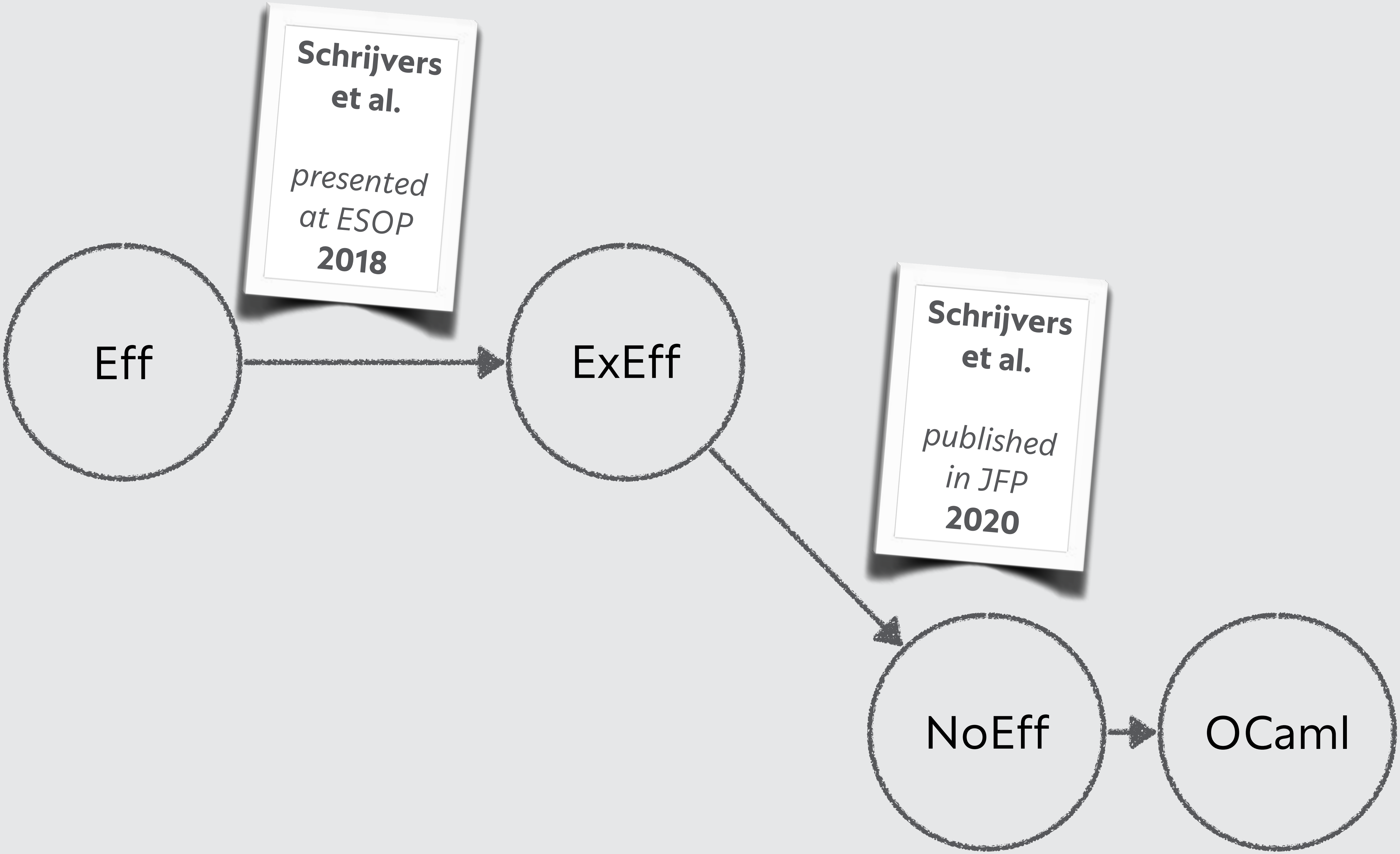
Eff

ExEff

NoEff

OCaml

The main step was adding **coercions** as witnesses of **subtyping**



Eff

Schrijvers et al.

presented at ESOP **2018**

ExEff

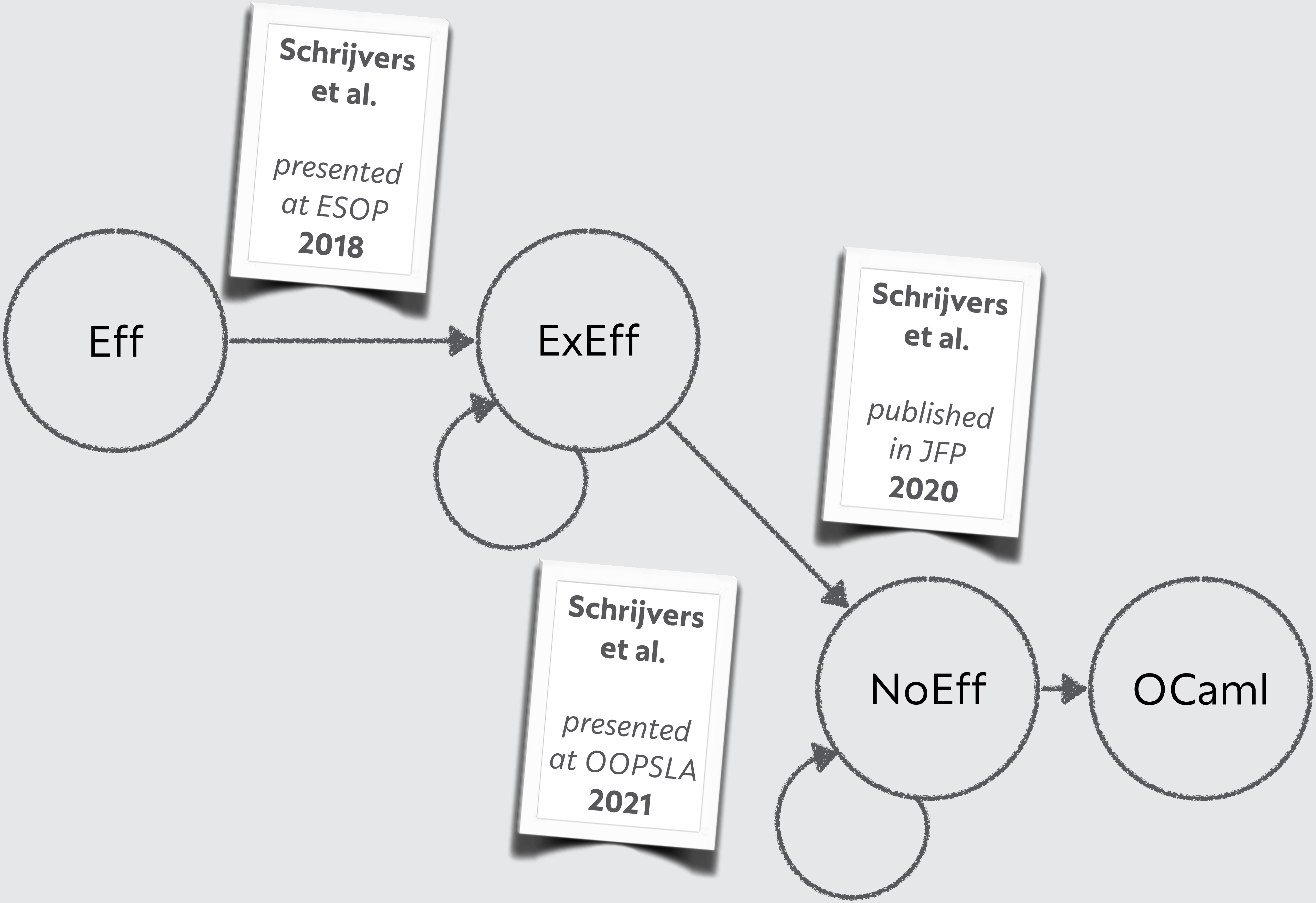
Schrijvers et al.

published in JFP **2020**

NoEff

OCaml

The main step was adding **coercions** as witnesses of **subtyping**



Eff

ExEff

NoEff

OCaml

Schrijvers et al.

presented at ESOP
2018

Schrijvers et al.

published in JFP
2020

Schrijvers et al.

presented at OOPSLA
2021

The syntax of **types** and well-formedness rules for **coercions**

types

$$\sigma ::= b \mid \alpha \mid \sigma \to \underline{\tau} \qquad \underline{\tau} ::= \sigma!\varphi$$

coercions

$$\frac{}{\Xi \vdash \langle\sigma\rangle : (\sigma <: \sigma)} \qquad \frac{\omega : (\sigma <: \sigma') \in \Xi}{\Xi \vdash \omega : (\sigma <: \sigma')}$$

$$\frac{\Xi \vdash \omega_v : (\sigma' <: \sigma) \qquad \Xi \vdash \omega_c : (\underline{\tau} <: \underline{\tau}')}{\Xi \vdash \omega_v \to \omega_c : \big((\sigma \to \underline{\tau}) <: (\sigma \to \underline{\tau}')\big)}$$

$$\frac{\Xi \vdash \omega_v : (\sigma <: \sigma') \qquad \Xi \vdash \varpi : (\varphi <: \varphi')}{\Xi \vdash \omega_v!\varpi : (\sigma!\varphi <: \sigma'!\varphi')}$$

When compiling to **OCaml**, coercions are mapped into **functions**

$$\begin{aligned}
\mathscr{C}(\langle\sigma\rangle) &= \texttt{id}\\
\mathscr{C}(\omega_i) &= \texttt{w\_i}\\
\mathscr{C}(\omega_v \to \omega_c) &= \texttt{fun}\, f \mapsto x \mapsto (f(x \triangleright \mathscr{C}(\omega_v)) \triangleright \mathscr{C}(\omega_c)
\end{aligned}$$

$$\mathscr{C}(\omega_v ! \varpi) = \begin{cases} \mathscr{C}(\omega_v) & \varpi : \emptyset \subseteq \emptyset \\ \texttt{return} \circ \mathscr{C}(\omega_v) & \varpi : \emptyset \subseteq \varphi \\ \texttt{map}\, \mathscr{C}(\omega_v) & \varpi : \varphi \subseteq \varphi' \end{cases}$$

When compiling to **OCaml**, coercions are mapped into **functions**

$$\mathscr{C}(\langle\sigma\rangle) = \texttt{id}$$

$$\mathscr{C}(\omega_i) = \texttt{w\_i}$$

$$\mathscr{C}(\omega_v \to \omega_c) = \texttt{fun}\, f \mapsto x \mapsto (f(x \triangleright \mathscr{C}(\omega_v)) \triangleright \mathscr{C}(\omega_c)$$

$$\mathscr{C}(\omega_v ! \varpi) = \begin{cases} \mathscr{C}(\omega_v) & \varpi : \emptyset \subseteq \emptyset \\ \texttt{return} \circ \mathscr{C}(\omega_v) & \varpi : \emptyset \subseteq \varphi \\ \texttt{map}\, \mathscr{C}(\omega_v) & \varpi : \varphi \subseteq \varphi' \end{cases}$$

Translating a **polymorphic** function incurs **additional parameters**

Translating a **polymorphic** function incurs **additional parameters**

Eff source

```
let apply_zero f = f 0 in
apply_zero cos
```

Translating a **polymorphic** function incurs **additional parameters**

Eff source

```
let apply_zero f = f 0 in
apply_zero cos
```

Internal representation

$$\texttt{let}\ \mathrm{applyZero}_{\alpha,\beta,(\omega:\texttt{int}<:\alpha)}\,(f:\alpha\to\beta) = f(0 \triangleright \omega)\ \texttt{in}$$

$$\mathrm{applyZero}_{\texttt{float},\texttt{float},\texttt{int2float}}\,\texttt{cos}$$

Translating a **polymorphic** function incurs **additional parameters**

Eff source

```
let apply_zero f = f 0 in
apply_zero cos
```

Internal representation

$$\texttt{let}\ \mathrm{applyZero}_{\alpha,\beta,(\omega:\texttt{int}<:\alpha)}\,(f : \alpha \to \beta) = f(0 \triangleright \omega)\ \texttt{in}$$

$$\mathrm{applyZero}_{\texttt{float},\texttt{float},\texttt{int2float}}\,\texttt{cos}$$

OCaml translation

```
let apply_zero w f = f (w 0) in
apply_zero float_of_int cos
```

Translating a **polymorphic** function incurs **additional parameters**

Eff source

```
let apply_zero f = f 0 in
apply_zero cos
```

Internal representation

$$\texttt{let}\ \mathrm{applyZero}_{\alpha,\beta,(\omega:\texttt{int}<:\alpha)}\,(f:\alpha\to\beta) = f(0 \triangleright \omega)\ \texttt{in}$$

$$\mathrm{applyZero}_{\texttt{float},\texttt{float},\texttt{int2float}}\,\texttt{cos}$$

OCaml translation

```
let apply_zero w f = f (w 0) in
apply_zero float_of_int cos
```

Translating a **polymorphic** function incurs **additional parameters**

Eff source

```
let apply_zero f = f 0 in
apply_zero cos
```

Internal representation

$$\texttt{let}\ \mathrm{applyZero}_{\alpha,\beta,(\omega:\texttt{int}<:\alpha)}\,(f : \alpha \to \beta) = f(0 \triangleright \omega)\ \texttt{in}$$

$$\mathrm{applyZero}_{\texttt{float},\texttt{float},\texttt{int2float}}\,\texttt{cos}$$

OCaml translation

```
let apply_zero w f = f (w 0) in
apply_zero float_of_int cos
```

Eff standard library ~450 coercion parameters

quicksort ~200 coercion parameters

Translating a **polymorphic** function incurs **additional parameters**

Eff source

```
let apply_zero f = f 0 in
apply_zero cos
```

Internal representation

$$\texttt{let}\ \mathrm{applyZero}_{\alpha,\beta,(\omega:\texttt{int}<:\alpha)}\,(f : \alpha \to \beta) = f(0 \rhd \omega)\ \texttt{in}$$

$$\mathrm{applyZero}_{\texttt{float},\texttt{float},\texttt{int2float}}\,\texttt{cos}$$

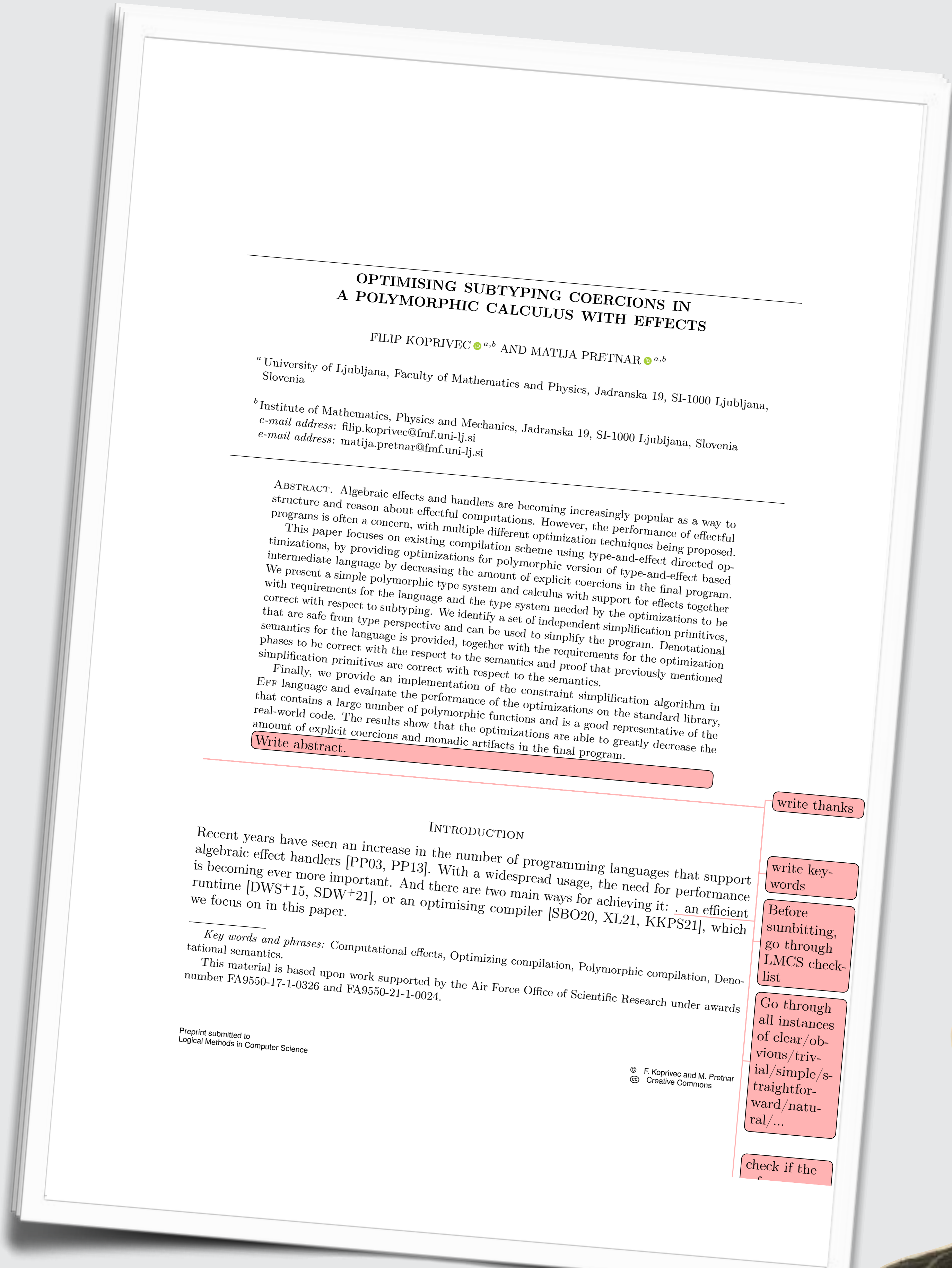
OCaml translation

```
let apply_zero w f = f (w 0) in
apply_zero float_of_int cos
```

Eff standard library ~450 coercion parameters

quicksort ~200 coercion parameters

Coercions can be **replaced without impacting** the semantics



OPTIMISING SUBTYPING COERCIONS IN A POLYMORPHIC CALCULUS WITH EFFECTS

FILIP KOPRIVEC [a,b] AND MATIJA PRETNAR [a,b]

[a] University of Ljubljana, Faculty of Mathematics and Physics, Jadranska 19, SI-1000 Ljubljana, Slovenia

[b] Institute of Mathematics, Physics and Mechanics, Jadranska 19, SI-1000 Ljubljana, Slovenia
e-mail address: filip.koprivec@fmf.uni-lj.si
e-mail address: matija.pretnar@fmf.uni-lj.si

Abstract. Algebraic effects and handlers are becoming increasingly popular as a way to structure and reason about effectful computations. However, the performance of effectful programs is often a concern, with multiple different optimization techniques being proposed.

This paper focuses on existing compilation scheme using type-and-effect directed optimizations, by providing optimizations for polymorphic version of type-and-effect based intermediate language by decreasing the amount of explicit coercions in the final program. We present a simple polymorphic type system and calculus with support for effects together with requirements for the language and the type system needed by the optimizations to be correct with respect to subtyping. We identify a set of independent simplification primitives, that are safe from type perspective and can be used to simplify the program. Denotational semantics for the language is provided, together with the requirements for the optimization phases to be correct with the respect to the semantics and proof that previously mentioned simplification primitives are correct with respect to the semantics.

Finally, we provide an implementation of the constraint simplification algorithm in Eff language and evaluate the performance of the optimizations on the standard library, that contains a large number of polymorphic functions and is a good representative of the real-world code. The results show that the optimizations are able to greatly decrease the amount of explicit coercions and monadic artifacts in the final program.

Write abstract.

write thanks

Introduction

Recent years have seen an increase in the number of programming languages that support algebraic effect handlers [PP03, PP13]. With a widespread usage, the need for performance is becoming ever more important. And there are two main ways for achieving it: . an efficient runtime [DWS+15, SDW+21], or an optimising compiler [SBO20, XL21, KKPS21], which we focus on in this paper.

write keywords

Before sumbitting, go through LMCS checklist

Go through all instances of clear/obvious/trivial/simple/straightforward/natural/...

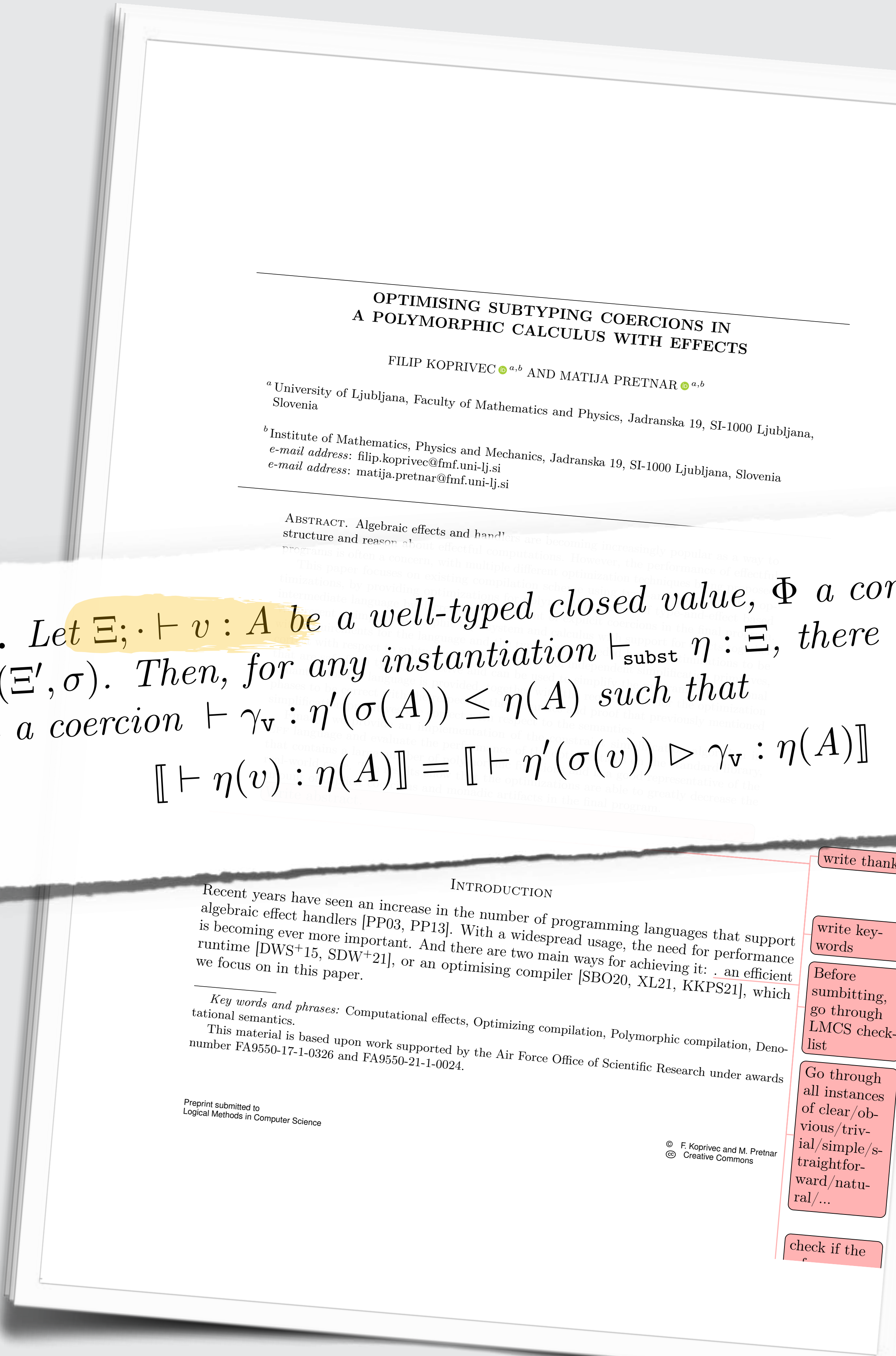
check if the

Key words and phrases: Computational effects, Optimizing compilation, Polymorphic compilation, Denotational semantics.

This material is based upon work supported by the Air Force Office of Scientific Research under awards number FA9550-17-1-0326 and FA9550-21-1-0024.

Preprint submitted to
Logical Methods in Computer Science

Coercions can be **replaced without impacting** the semantics



OPTIMISING SUBTYPING COERCIONS IN A POLYMORPHIC CALCULUS WITH EFFECTS

FILIP KOPRIVEC [a,b] AND MATIJA PRETNAR [a,b]

[a] University of Ljubljana, Faculty of Mathematics and Physics, Jadranska 19, SI-1000 Ljubljana, Slovenia

[b] Institute of Mathematics, Physics and Mechanics, Jadranska 19, SI-1000 Ljubljana, Slovenia
e-mail address: filip.koprivec@fmf.uni-lj.si
e-mail address: matija.pretnar@fmf.uni-lj.si

ABSTRACT. Algebraic effects and hand
structure and reason a

Corollary 5.4. *Let $\Xi; \cdot \vdash v : A$ be a well-typed closed value, Φ a complete phase such that $\Phi(\Xi, \mathtt{fp}(A)) = (\Xi', \sigma)$. Then, for any instantiation $\vdash_{\mathtt{subst}} \eta : \Xi$, there exists an instantiation $\vdash_{\mathtt{subst}} \eta' : \Xi'$ and a coercion $\vdash \gamma_{\mathtt{v}} : \eta'(\sigma(A)) \leq \eta(A)$ such that*

$$[\![\vdash \eta(v) : \eta(A)]\!] = [\![\vdash \eta'(\sigma(v)) \rhd \gamma_{\mathtt{v}} : \eta(A)]\!]$$

write thanks

write keywords

Before sumbitting, go through LMCS checklist

Go through all instances of clear/obvious/trivial/simple/straightforward/natural/...

check if the f

INTRODUCTION

Recent years have seen an increase in the number of programming languages that support algebraic effect handlers [PP03, PP13]. With a widespread usage, the need for performance is becoming ever more important. And there are two main ways for achieving it: . an efficient runtime [DWS+15, SDW+21], or an optimising compiler [SBO20, XL21, KKPS21], which we focus on in this paper.

Key words and phrases: Computational effects, Optimizing compilation, Polymorphic compilation, Denotational semantics.

This material is based upon work supported by the Air Force Office of Scientific Research under awards number FA9550-17-1-0326 and FA9550-21-1-0024.

Preprint submitted to
Logical Methods in Computer Science

Coercions can be **replaced without impacting** the semantics

OPTIMISING SUBTYPING COERCIONS IN A POLYMORPHIC CALCULUS WITH EFFECTS

FILIP KOPRIVEC[a,b] AND MATIJA PRETNAR[a,b]

[a] University of Ljubljana, Faculty of Mathematics and Physics, Jadranska 19, SI-1000 Ljubljana, Slovenia

[b] Institute of Mathematics, Physics and Mechanics, Jadranska 19, SI-1000 Ljubljana, Slovenia
e-mail address: filip.koprivec@fmf.uni-lj.si
e-mail address: matija.pretnar@fmf.uni-lj.si

Abstract. Algebraic effects and handl
structure and reason ab

Corollary 5.4. *Let $\Xi; \cdot \vdash v : A$ be a well-typed closed value, Φ a complete phase such that $\Phi(\Xi, \mathtt{fp}(A)) = (\Xi', \sigma)$. Then, for any instantiation $\vdash_{\mathtt{subst}} \eta : \Xi$, there exists an instantiation $\vdash_{\mathtt{subst}} \eta' : \Xi'$ and a coercion $\vdash \gamma_{\mathtt{v}} : \eta'(\sigma(A)) \leq \eta(A)$ such that*

$$[\![\vdash \eta(v) : \eta(A)]\!] = [\![\vdash \eta'(\sigma(v)) \rhd \gamma_{\mathtt{v}} : \eta(A)]\!]$$

write thanks

Introduction

Recent years have seen an increase in the number of programming languages that support algebraic effect handlers [PP03, PP13]. With a widespread usage, the need for performance is becoming ever more important. And there are two main ways for achieving it: . an efficient runtime [DWS+15, SDW+21], or an optimising compiler [SBO20, XL21, KKPS21], which we focus on in this paper.

write key-words

Before sumbitting, go through LMCS check-list

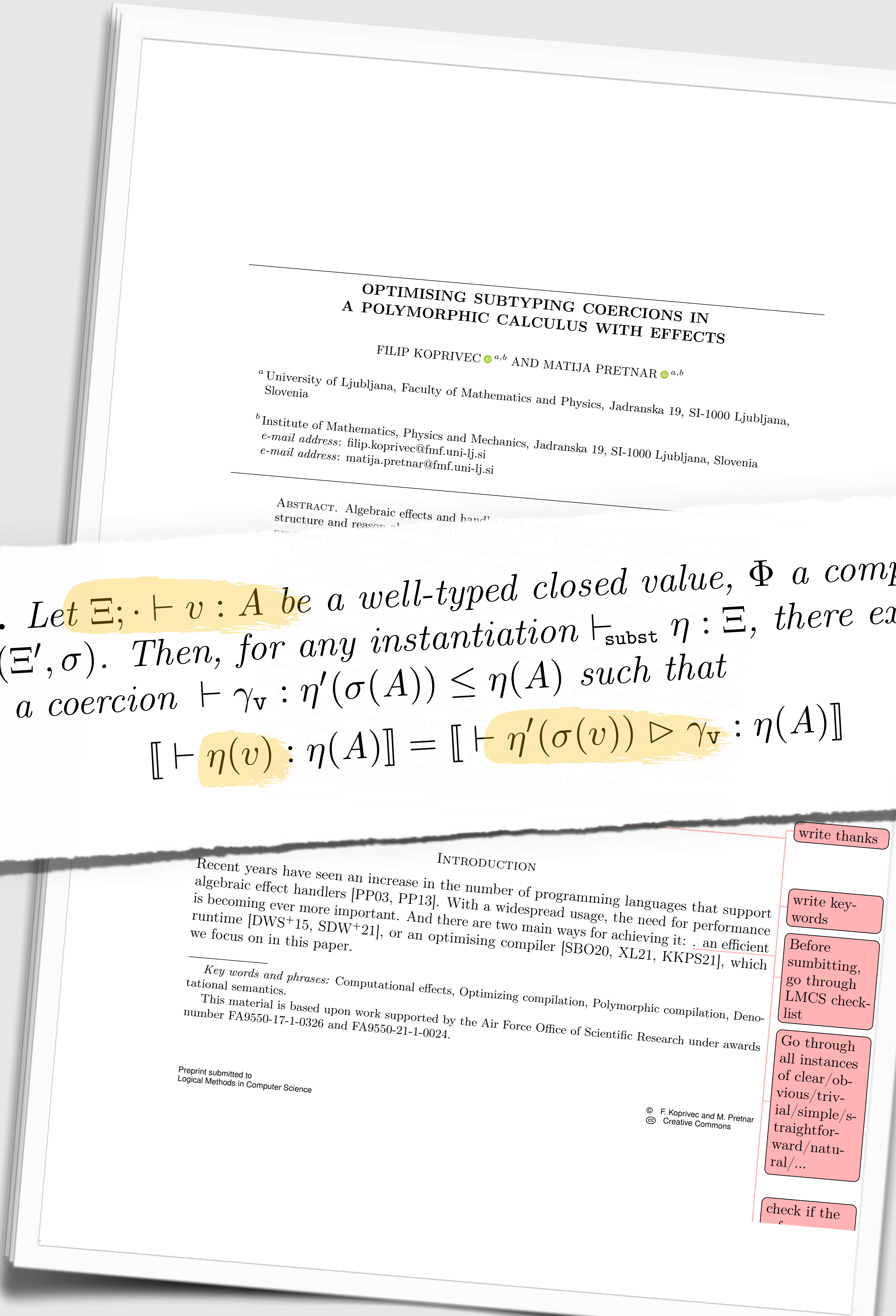
Go through all instances of clear/obvious/trivial/simple/straightforward/natural/...

check if the

Key words and phrases: Computational effects, Optimizing compilation, Polymorphic compilation, Denotational semantics.
This material is based upon work supported by the Air Force Office of Scientific Research under awards number FA9550-17-1-0326 and FA9550-21-1-0024.

Preprint submitted to
Logical Methods in Computer Science

Coercions can be **replaced without impacting** the semantics



OPTIMISING SUBTYPING COERCIONS IN
A POLYMORPHIC CALCULUS WITH EFFECTS

FILIP KOPRIVEC[a,b] AND MATIJA PRETNAR[a,b]

[a] University of Ljubljana, Faculty of Mathematics and Physics, Jadranska 19, SI-1000 Ljubljana, Slovenia

[b] Institute of Mathematics, Physics and Mechanics, Jadranska 19, SI-1000 Ljubljana, Slovenia
e-mail address: filip.koprivec@fmf.uni-lj.si
e-mail address: matija.pretnar@fmf.uni-lj.si

Abstract. Algebraic effects and hand
structure and reason

Corollary 5.4. *Let $\Xi; \cdot \vdash v : A$ be a well-typed closed value, Φ a complete phase such that $\Phi(\Xi, \mathtt{fp}(A)) = (\Xi', \sigma)$. Then, for any instantiation $\vdash_{\mathtt{subst}} \eta : \Xi$, there exists an instantiation $\vdash_{\mathtt{subst}} \eta' : \Xi'$ and a coercion $\vdash \gamma_{\mathtt{v}} : \eta'(\sigma(A)) \le \eta(A)$ such that*

$$[\![\vdash \eta(v) : \eta(A)]\!] = [\![\vdash \eta'(\sigma(v)) \rhd \gamma_{\mathtt{v}} : \eta(A)]\!]$$

write thanks

Introduction

Recent years have seen an increase in the number of programming languages that support algebraic effect handlers [PP03, PP13]. With a widespread usage, the need for performance is becoming ever more important. And there are two main ways for achieving it: . an efficient runtime [DWS+15, SDW+21], or an optimising compiler [SBO20, XL21, KKPS21], which we focus on in this paper.

write keywords

Before sumbitting, go through LMCS checklist

Go through all instances of clear/obvious/trivial/simple/straightforward/natural/...

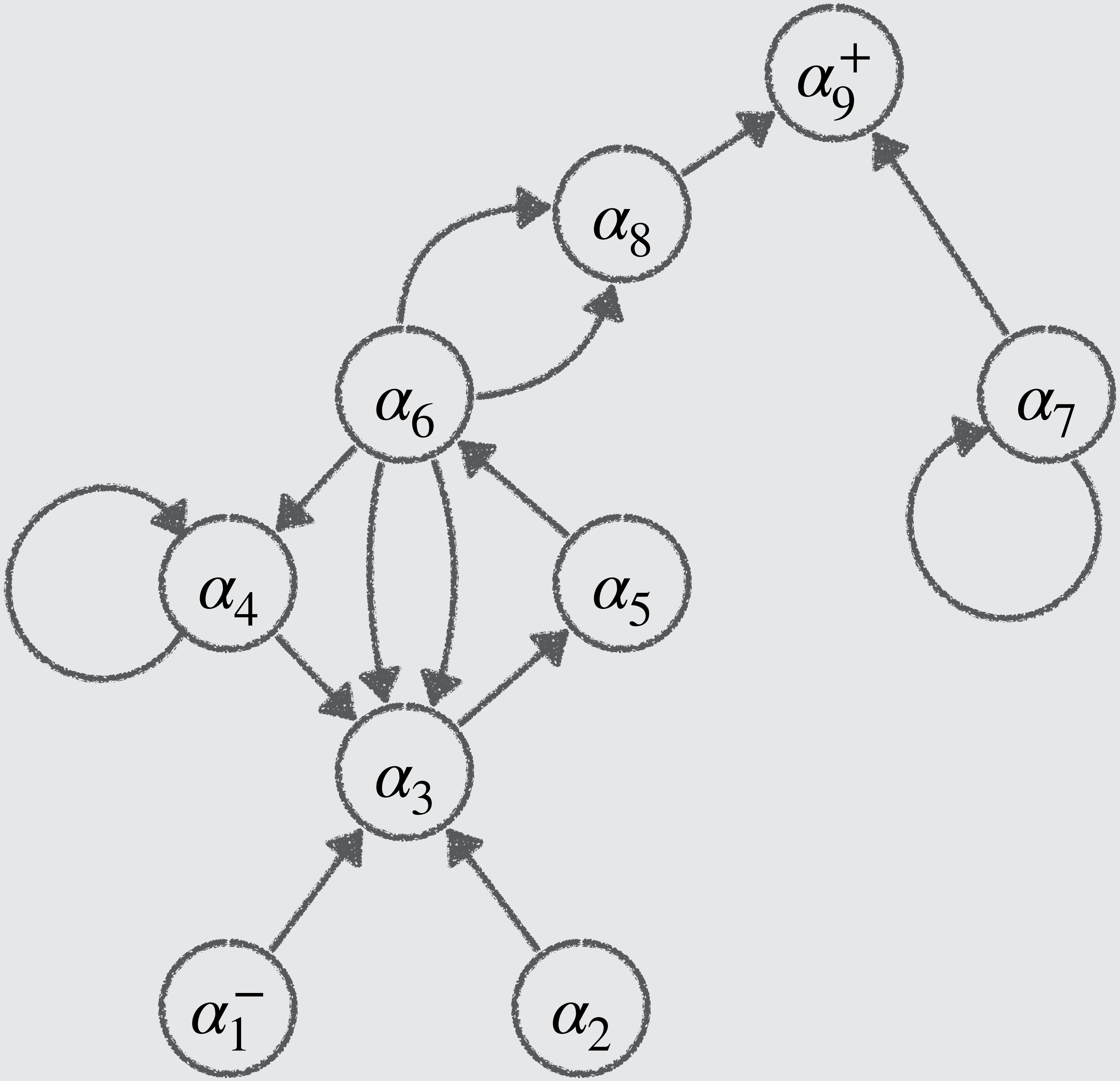
check if the

Key words and phrases: Computational effects, Optimizing compilation, Polymorphic compilation, Denotational semantics.

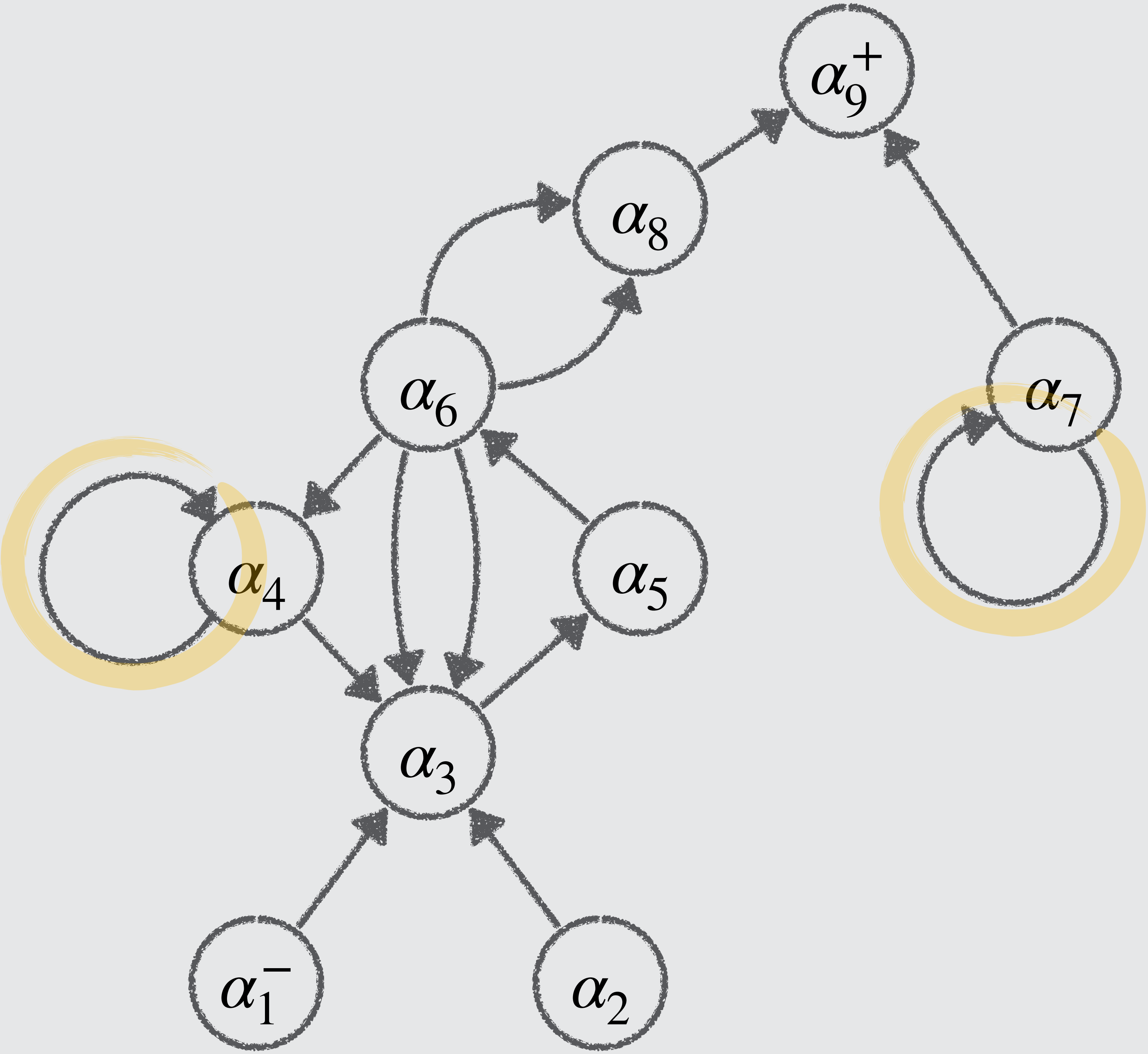
This material is based upon work supported by the Air Force Office of Scientific Research under awards number FA9550-17-1-0326 and FA9550-21-1-0024.

Preprint submitted to
Logical Methods in Computer Science

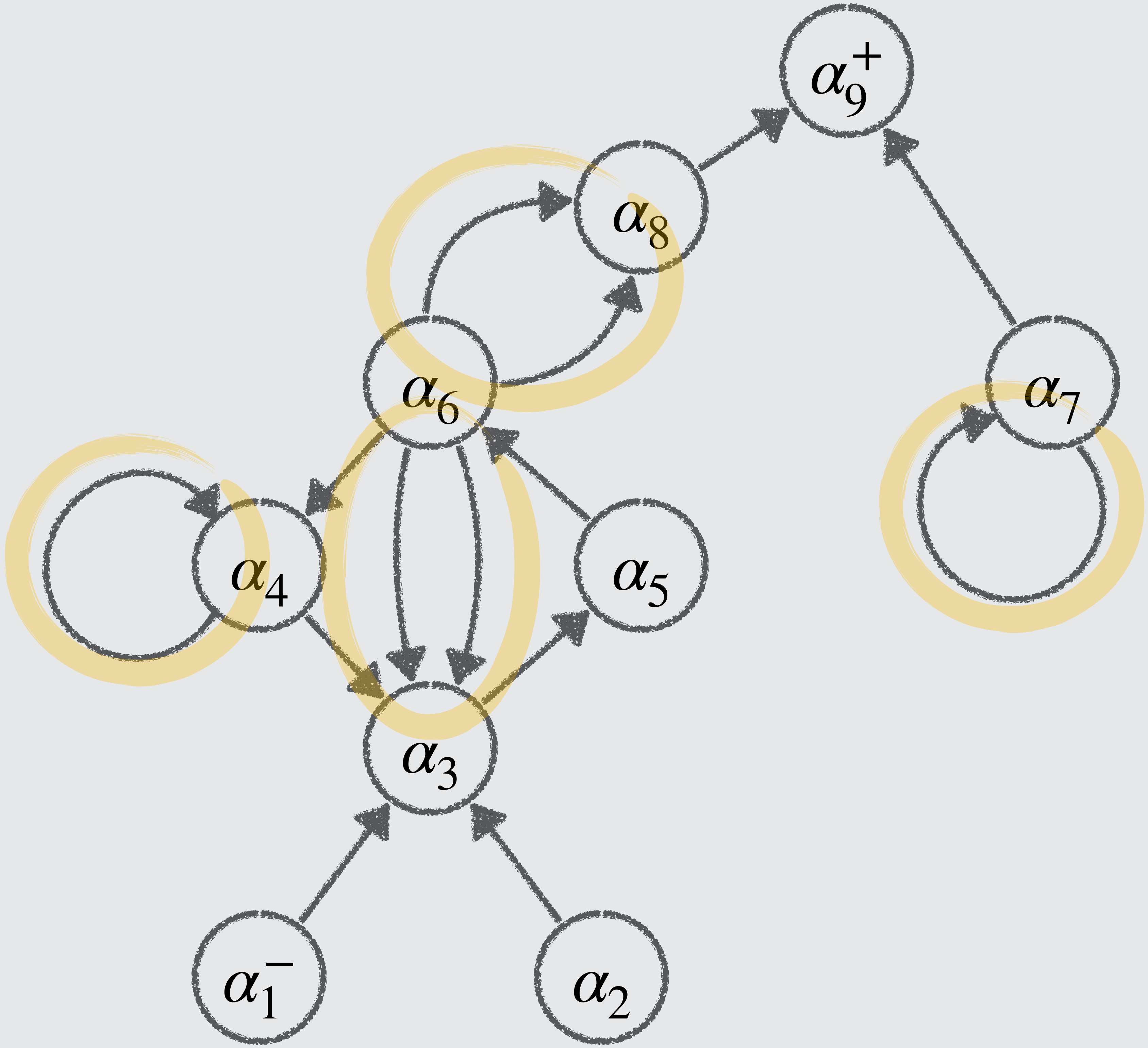
Constraints can be represented with **directed graphs**



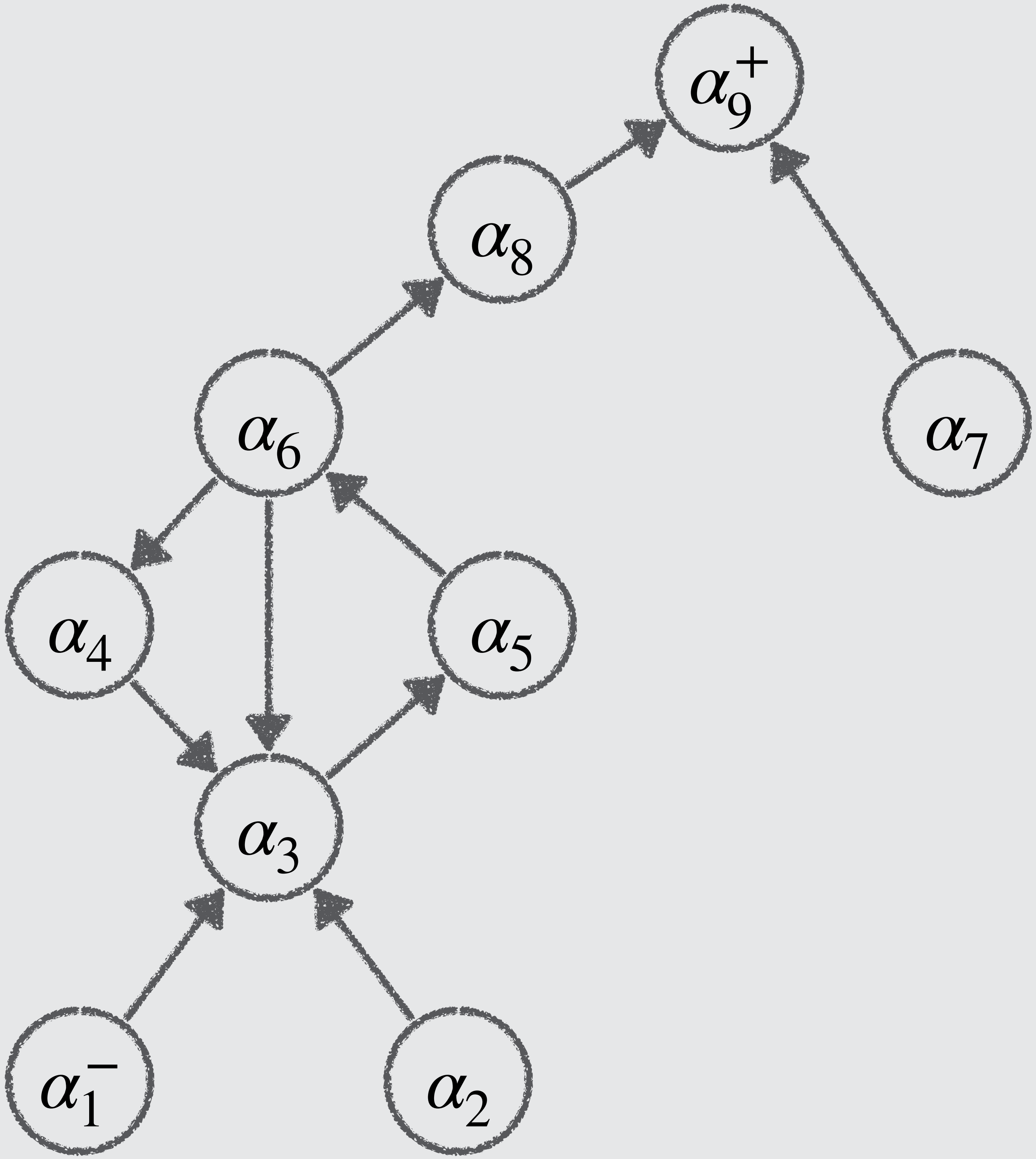
Constraints can be represented with **directed graphs**



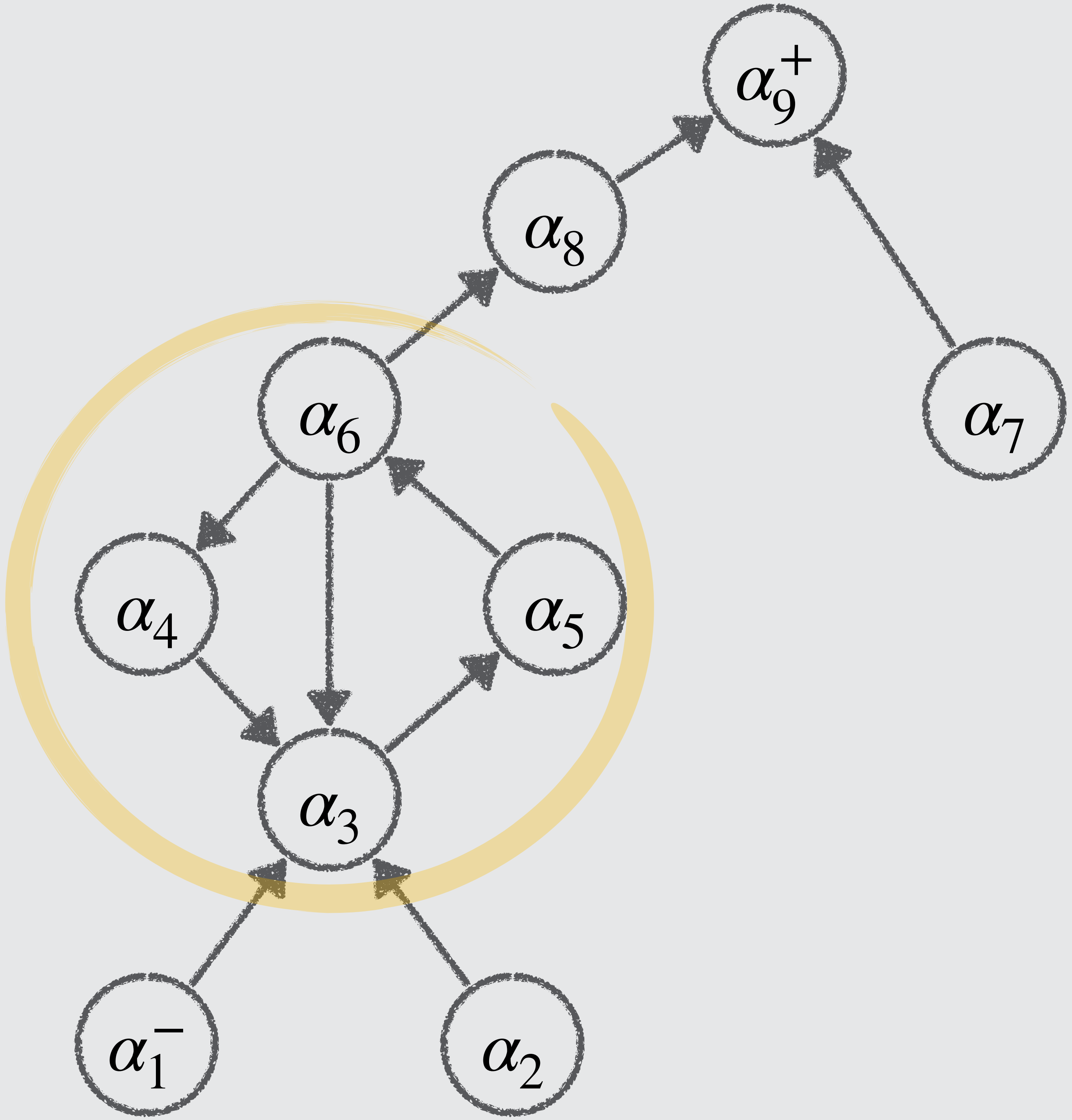
Constraints can be represented with **directed graphs**



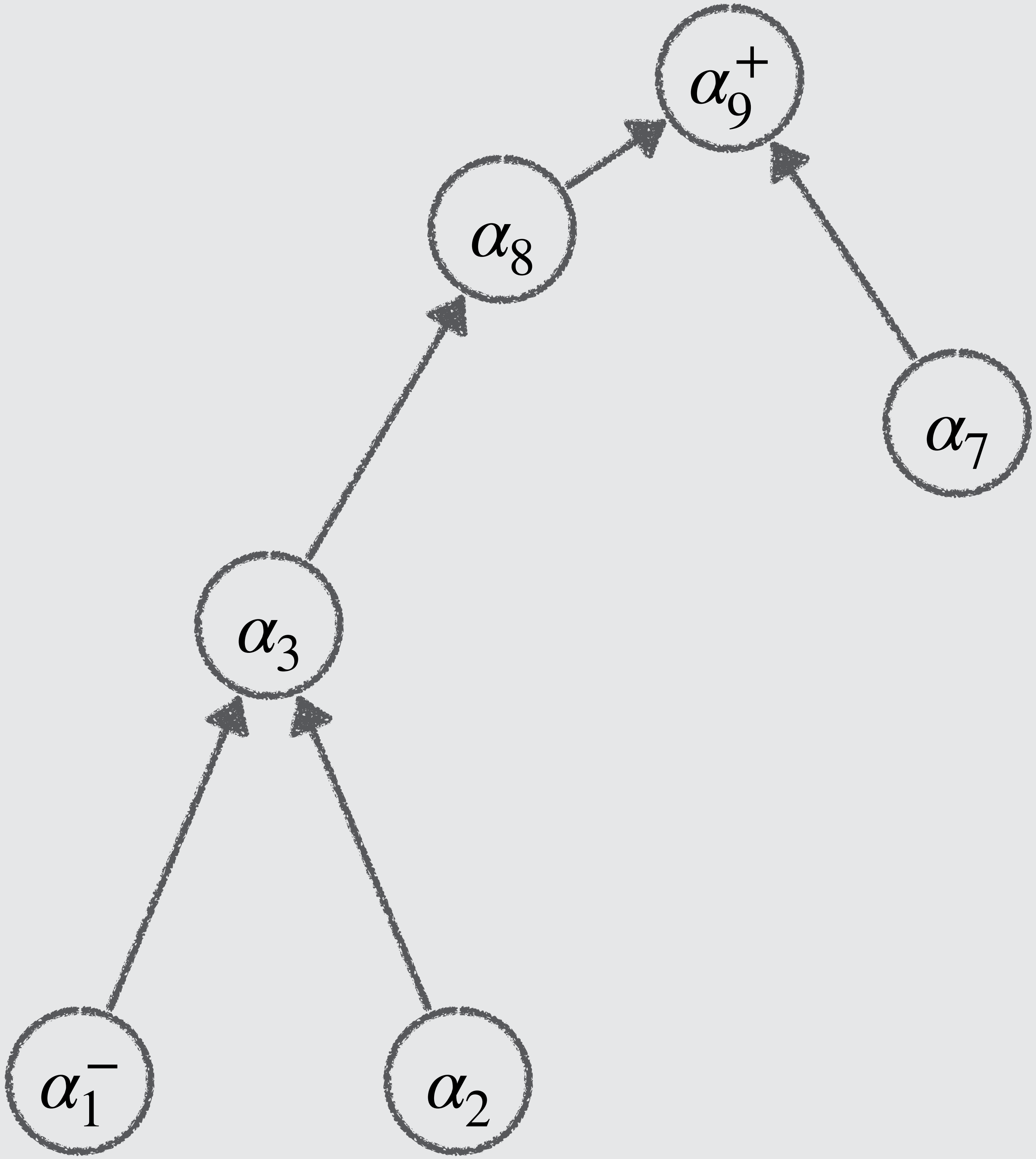
We can **remove** self **loops** and **parallel** edges to get a simple graph



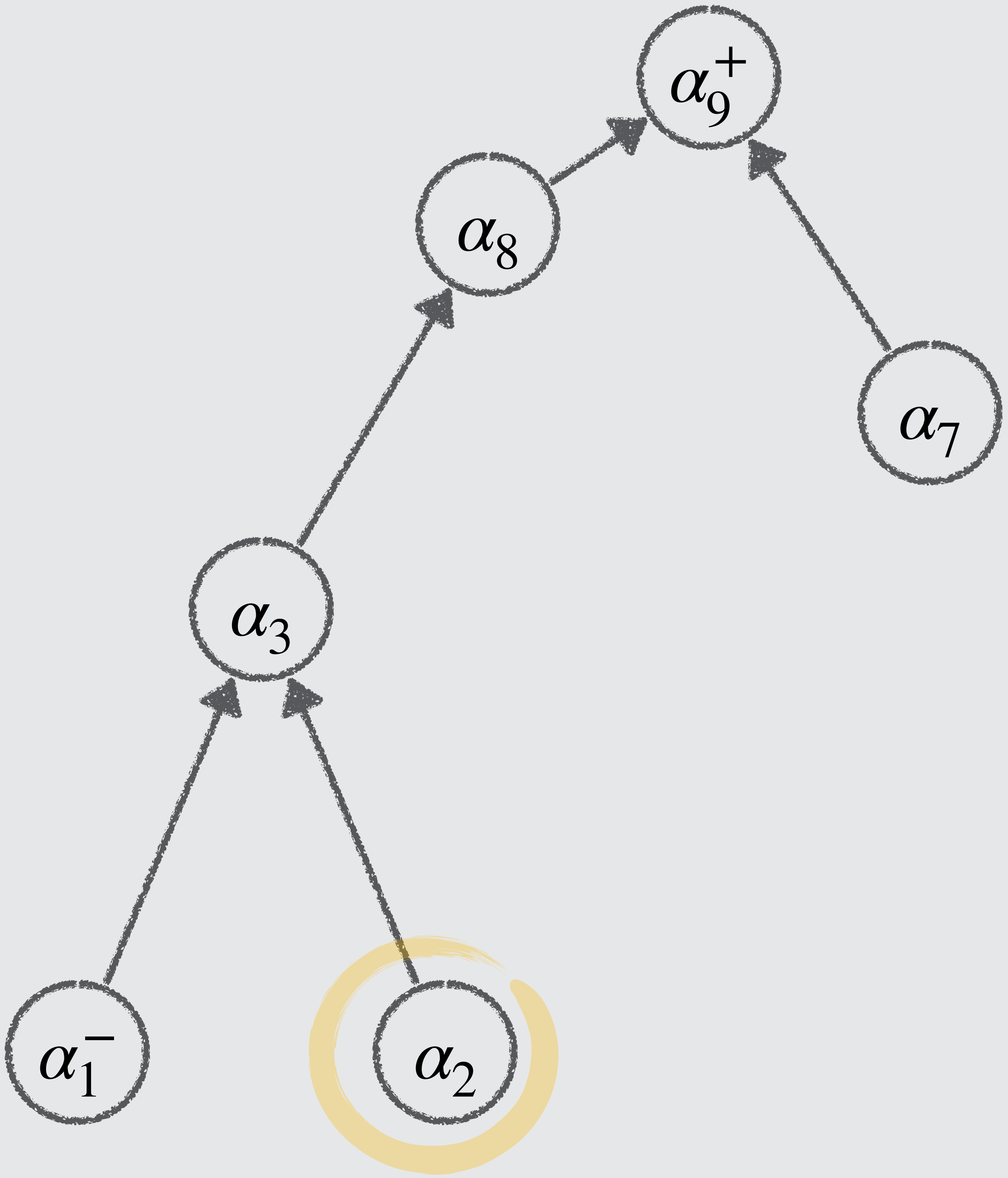
We can **remove** self **loops** and **parallel** edges to get a simple graph



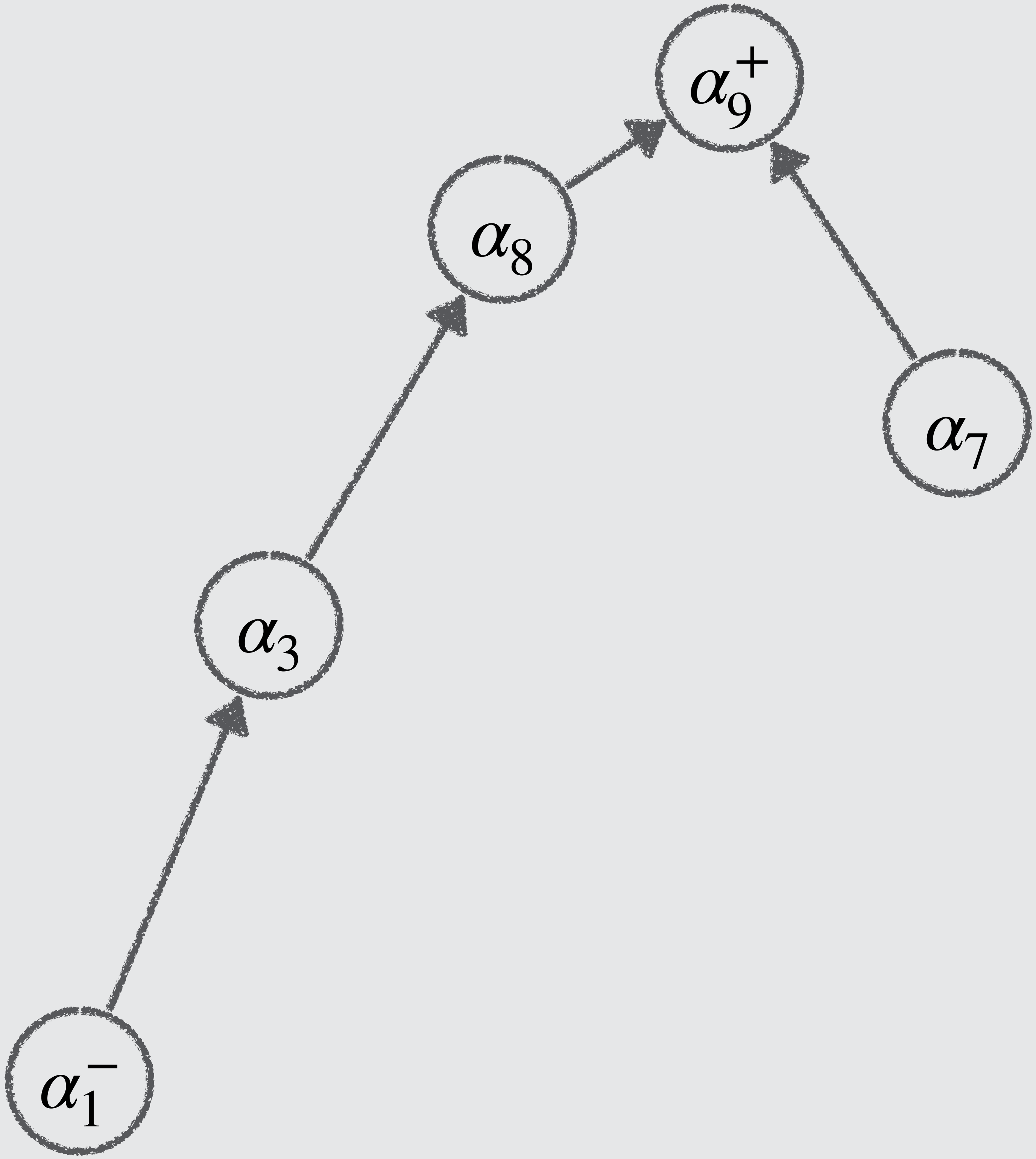
We can **collapse** strongly **connected** components to get a DAG



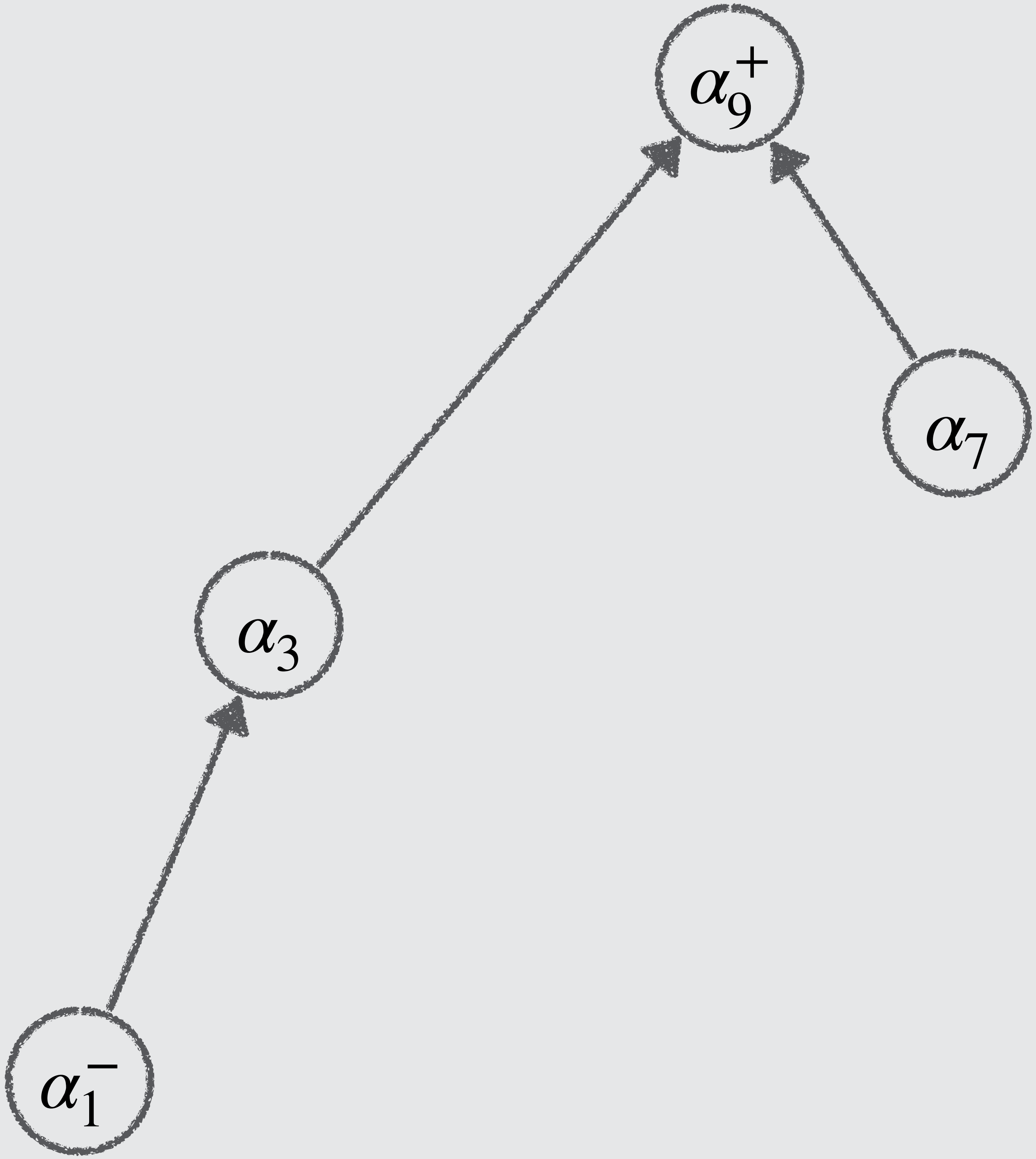
We can **collapse** strongly **connected** components to get a DAG



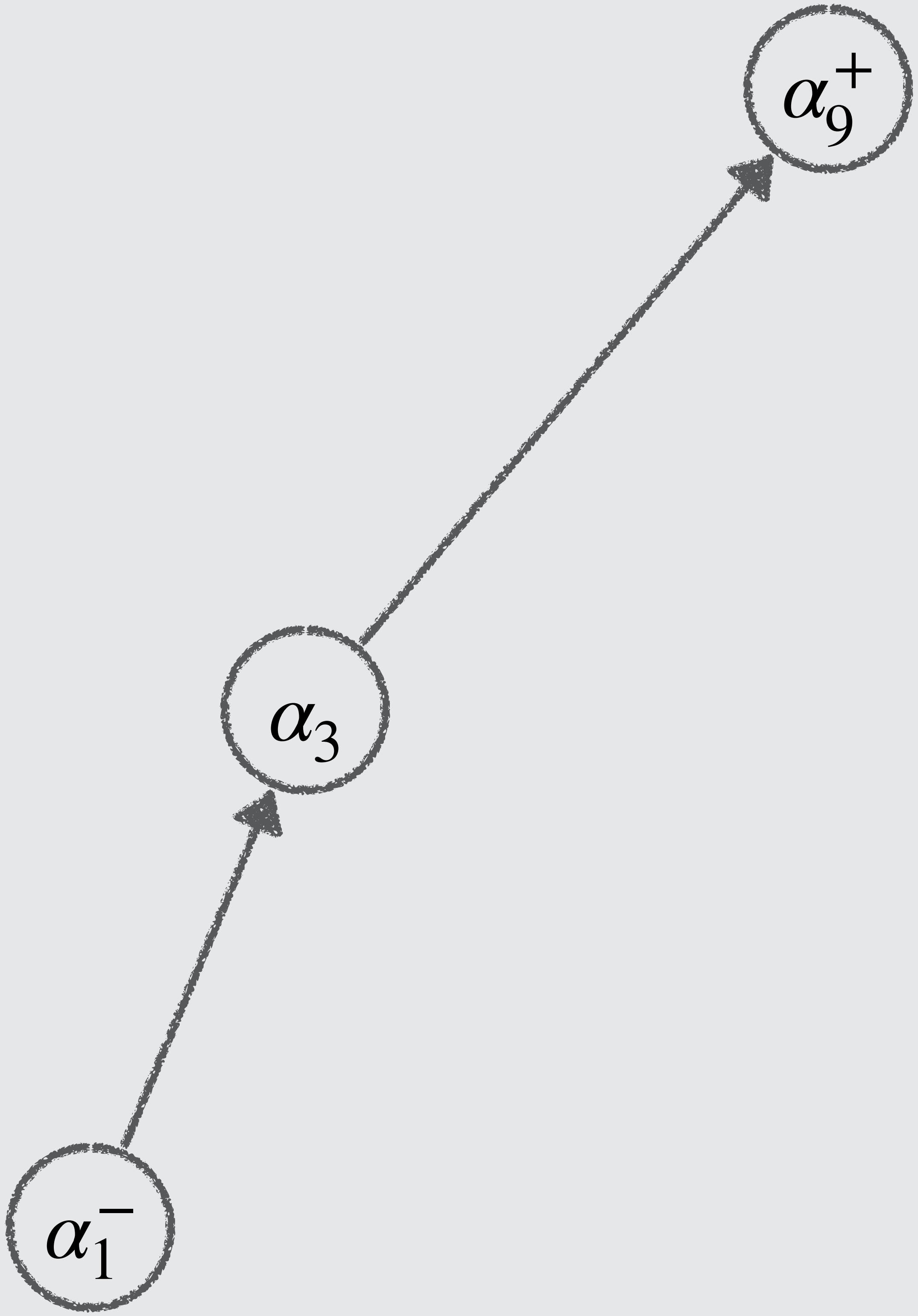
We can **collapse** bridges, if **polarity** allows



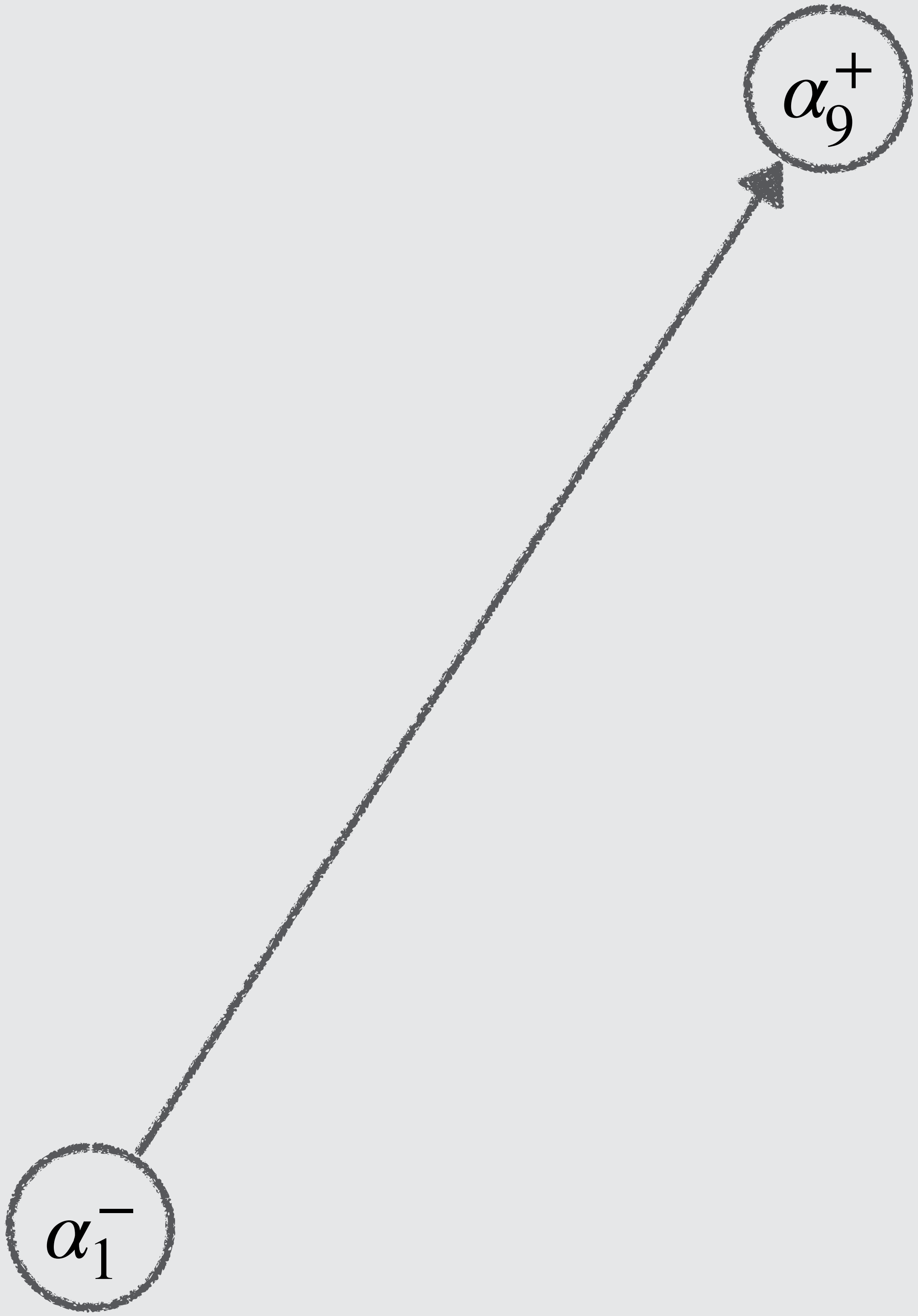
We can **collapse** bridges, if **polarity** allows



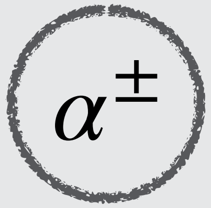
We can **collapse** bridges, if **polarity** allows



We can **collapse** bridges, if **polarity** allows



We can **collapse** bridges, if **polarity** allows



$\alpha^{\pm}$

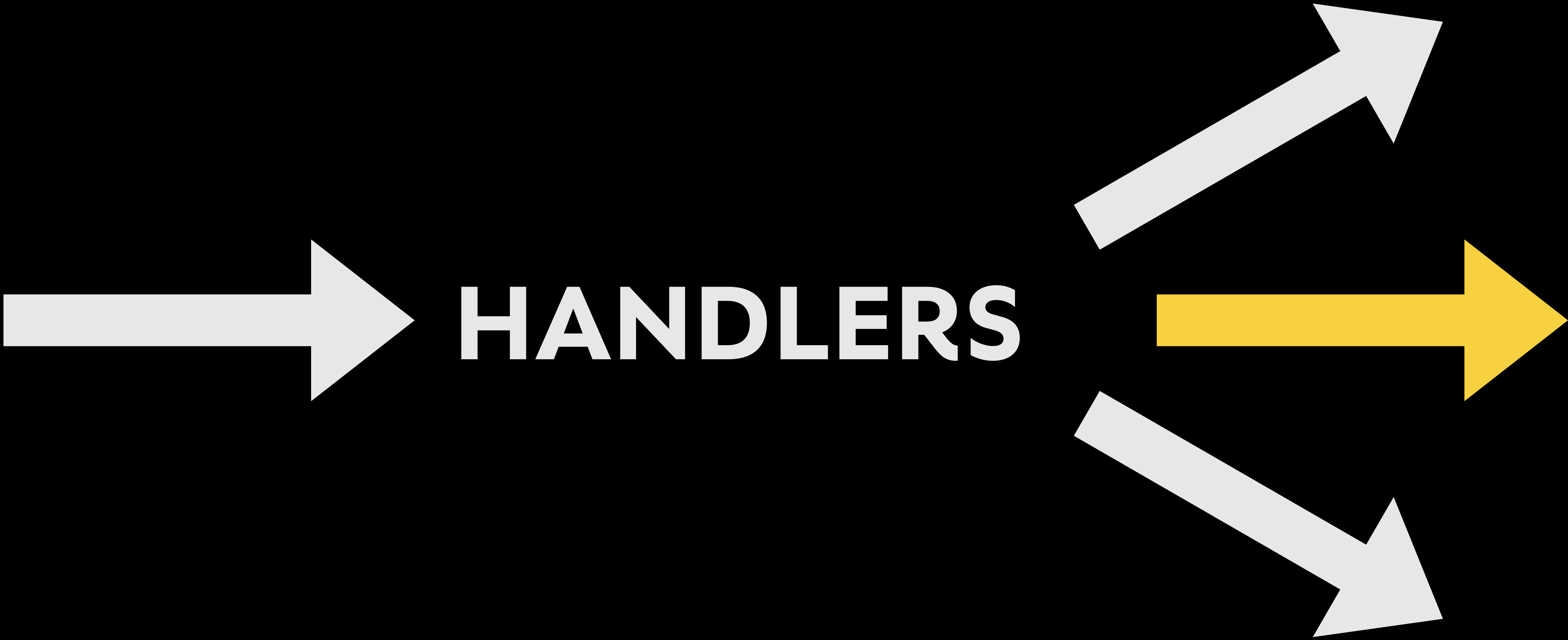
Even though not **canonical**, we significantly simplify compiled **Eff stdlib**

	before	**after**
type coercions	447	0
dirt coercions	644	0
`return`	78	31
`>>=`	29	17

Even though not **canonical**, we significantly simplify compiled **Eff stdlib**

	before	**after**
type coercions	447	0
dirt coercions	644	0
`return`	78	31
`>>=`	29	17

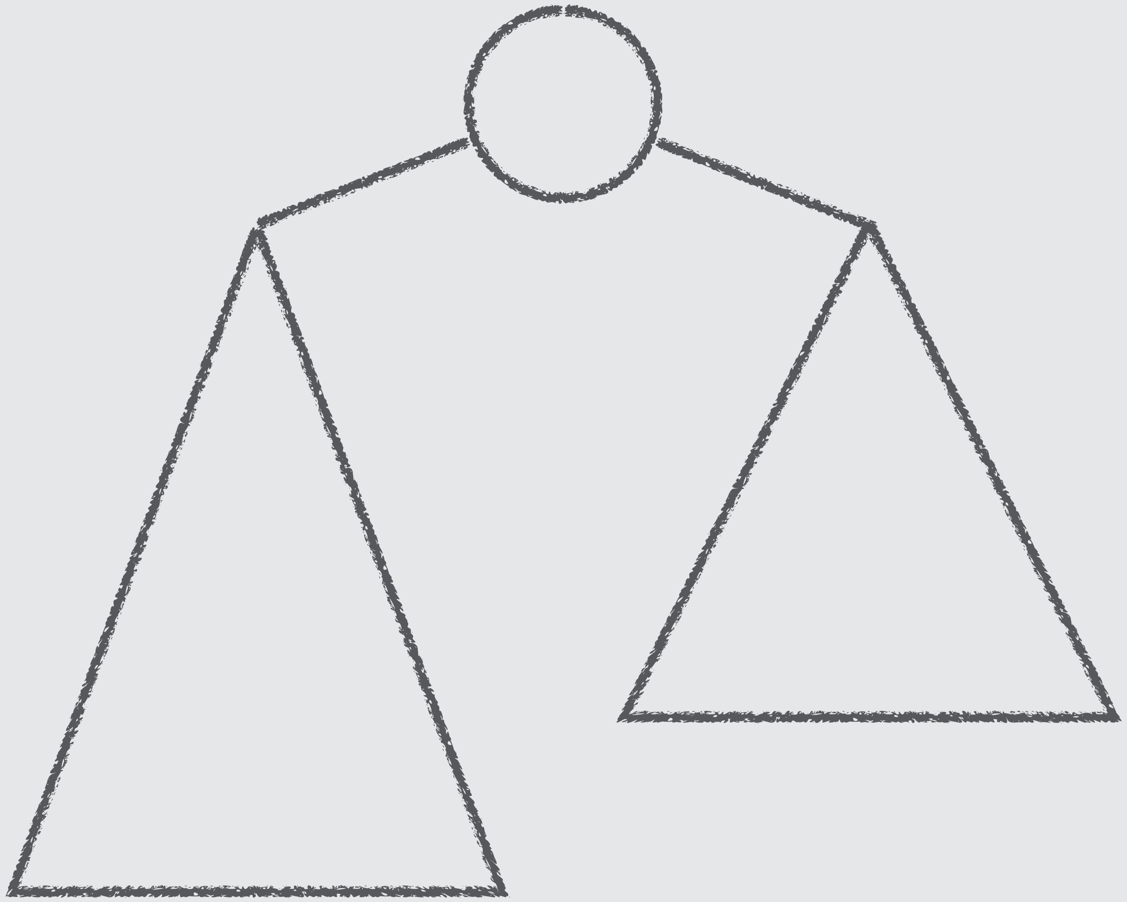
HANDLERS

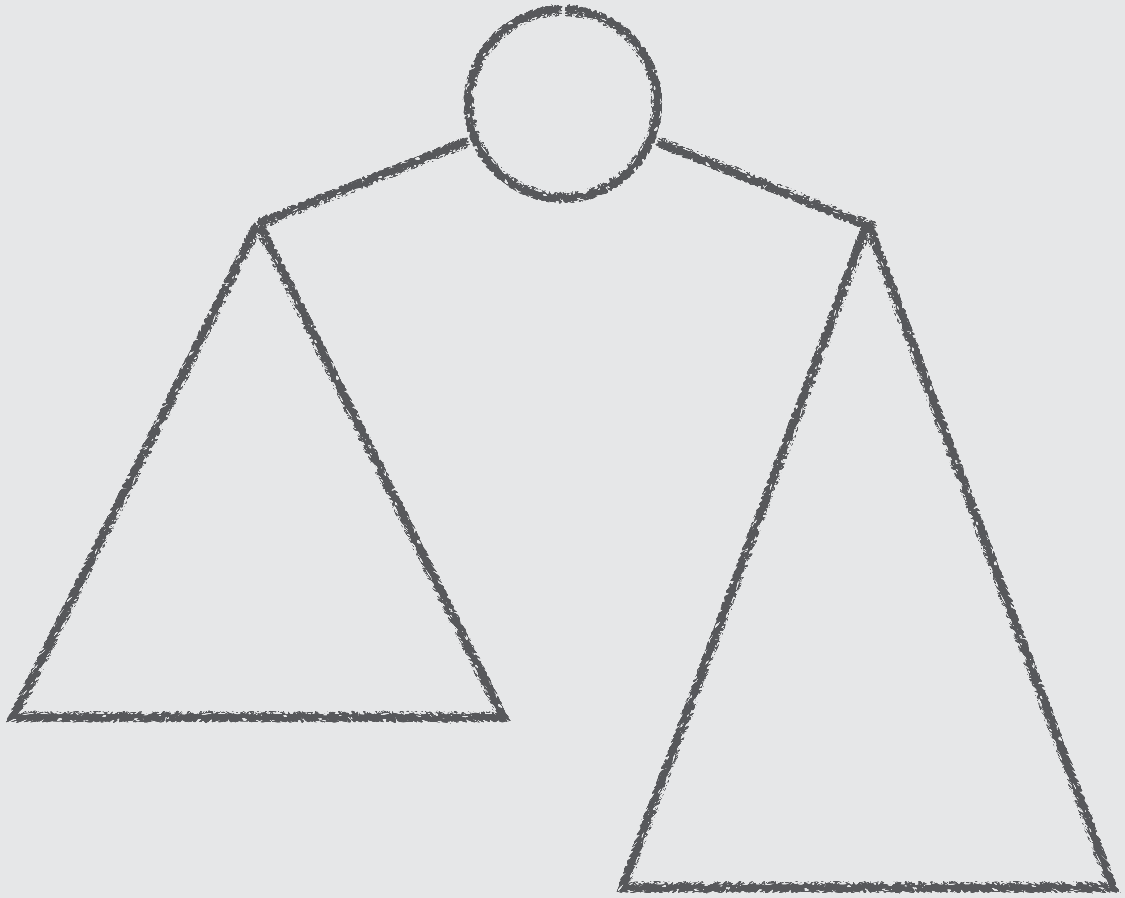


HANDLERS

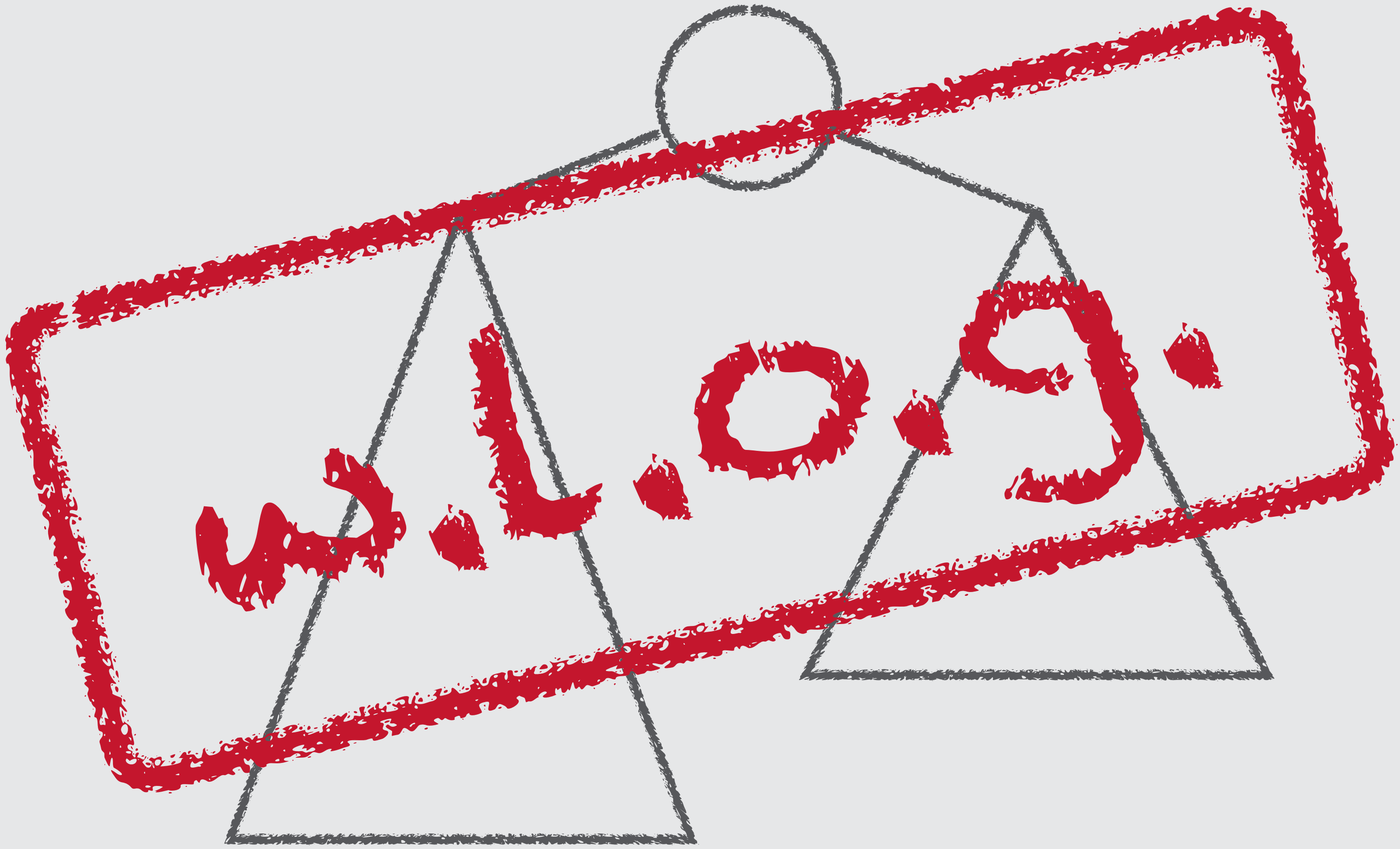
Symmetry is present, but not used in **programming**/**proving**

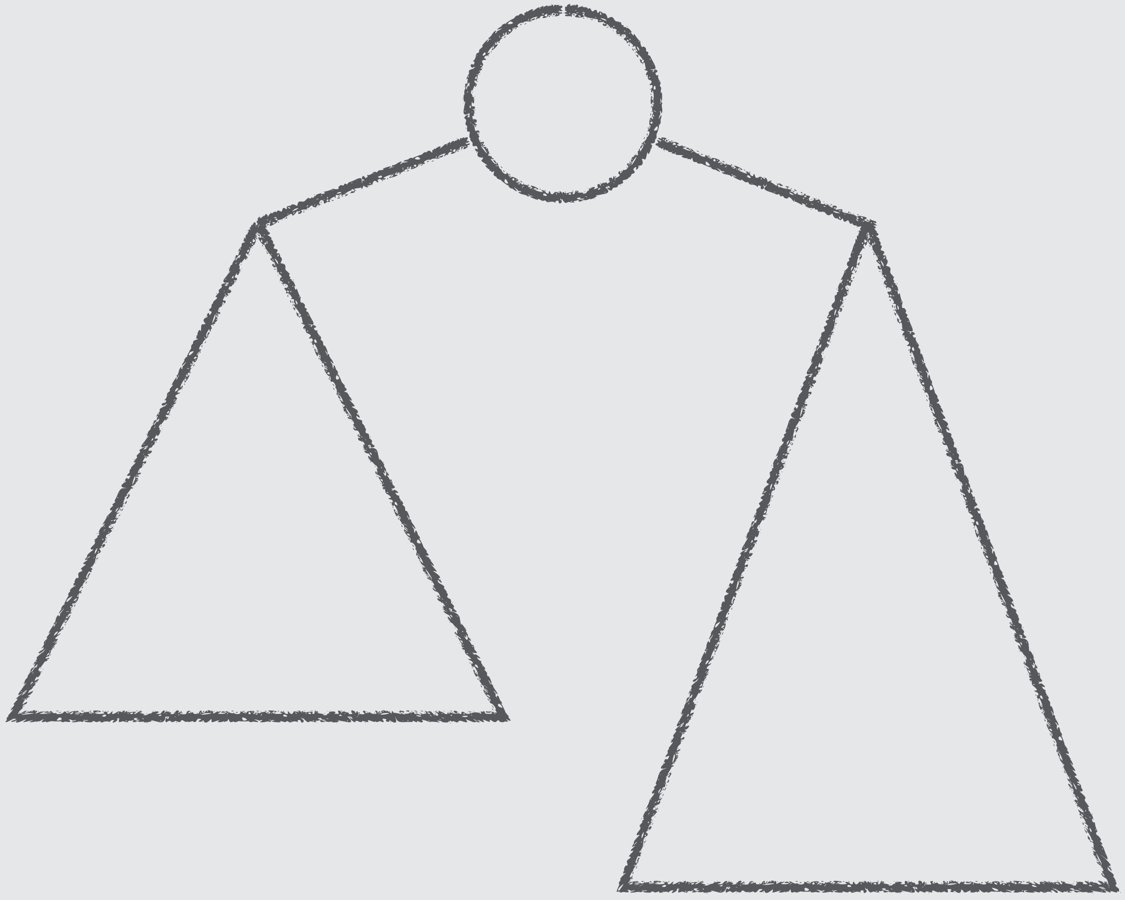
Symmetry is present, but not used in **programming**/**proving**





Symmetry is present, but not used in **programming**/**proving**





QUESTIONS?

THANK YOU!