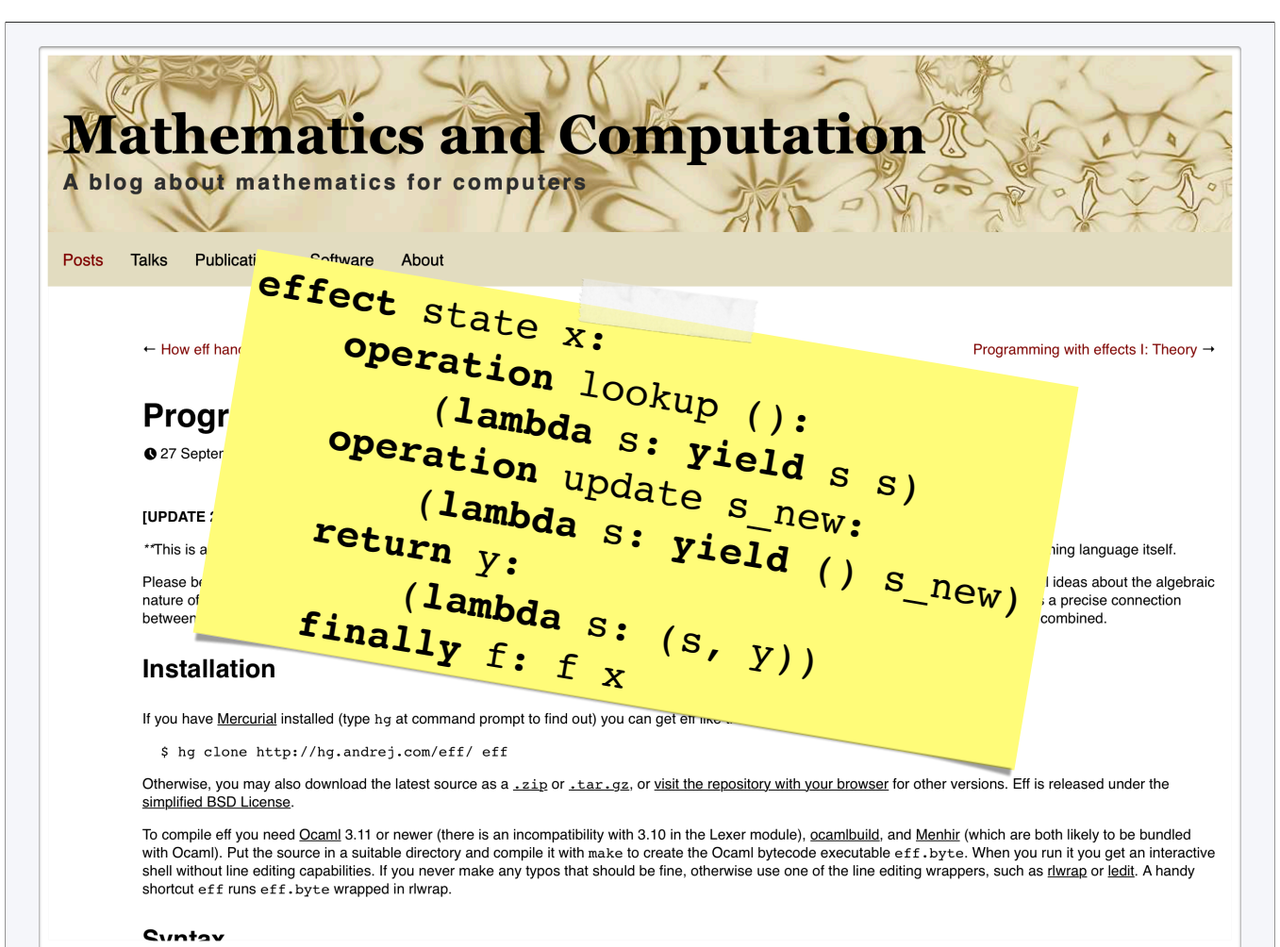
STATE OF EFF

Matija Pretnar
University of Ljubljana, Slovenia

2010

EFF 1.0

I was trying to reconstruct the history of Eff, and as far as I recall, the first version appeared in 2010.

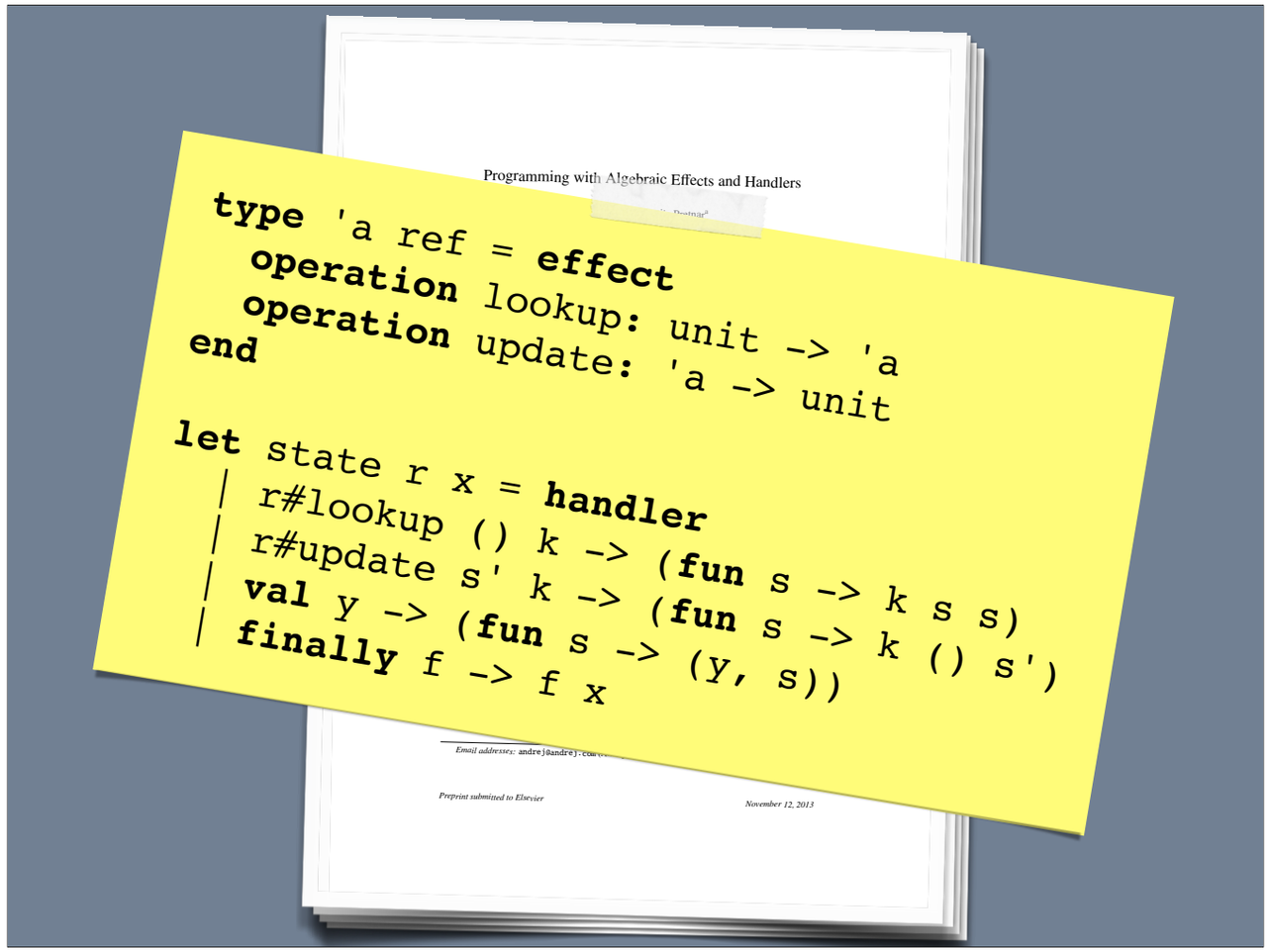


It waas announced on Andrej's blog, 13 years ago almost to a day. Two interesting notes: it's name was written in lower case, and it used a Python-like syntax.

2011

EFF 2.0

On our train ride back from the Domains workshop in Swansea, Andrej and I realised that Eff fits much better into an ML family.



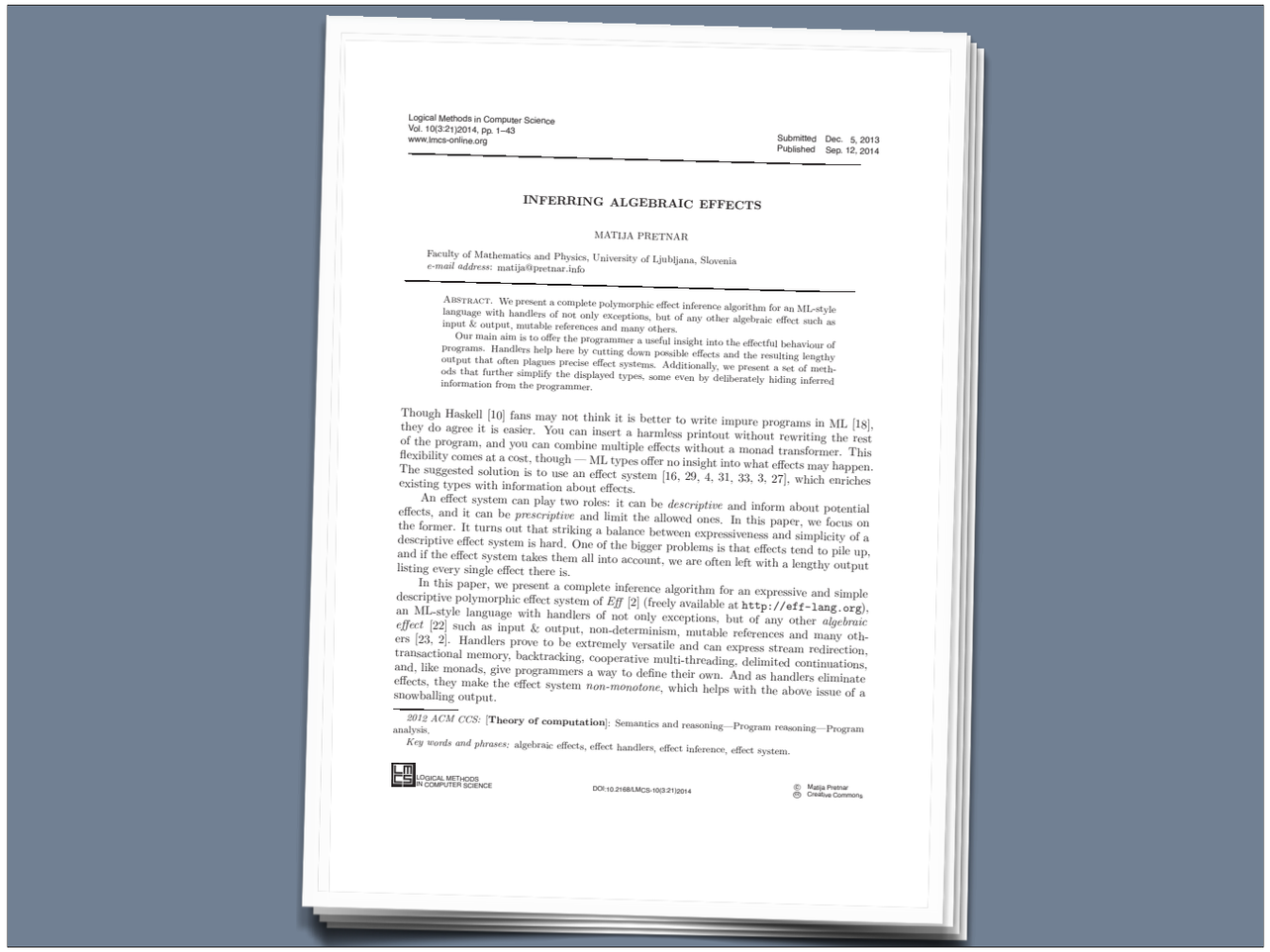
Eff 2.0 is what appeared in the “Programming with Algebraic Effects and Handlers” paper. At this point, Eff featured dynamic generation of effect instances, inspired by how references are created in OCaml.

After that point, the versioning becomes much more confusing. Version 3.0 appeared some time after that, but I do not know what the main difference from 2.0 was. So let's stick to chronological history from now onwards.

2013

EFFECT SYSTEM

In 2013, Eff got a subtyping-based effect system.



Logical Methods in Computer Science
Vol. 10(3:21)2014, pp. 1–43
www.lmcs-online.org

Submitted Dec. 5, 2013
Published Sep. 12, 2014

INFERRING ALGEBRAIC EFFECTS

MATIJA PRETNAR

Faculty of Mathematics and Physics, University of Ljubljana, Slovenia
e-mail address: matija@pretnar.info

Abstract. We present a complete polymorphic effect inference algorithm for an ML-style language with handlers of not only exceptions, but of any other algebraic effect such as input & output, mutable references and many others.

Our main aim is to offer the programmer a useful insight into the effectful behaviour of programs. Handlers help here by cutting down possible effects and the resulting lengthy output that often plagues precise effect systems. Additionally, we present a set of methods that further simplify the displayed types, some even by deliberately hiding inferred information from the programmer.

Though Haskell [10] fans may not think it is better to write impure programs in ML [18], they do agree it is easier. You can insert a harmless printout without rewriting the rest of the program, and you can combine multiple effects without a monad transformer. This flexibility comes at a cost, though — ML types offer no insight into what effects may happen. The suggested solution is to use an effect system [16, 29, 4, 31, 33, 3, 27], which enriches existing types with information about effects.

An effect system can play two roles: it can be *descriptive* and inform about potential effects, and it can be *prescriptive* and limit the allowed ones. In this paper, we focus on the former. It turns out that striking a balance between expressiveness and simplicity of a descriptive effect system is hard. One of the bigger problems is that effects tend to pile up, and if the effect system takes them all into account, we are often left with a lengthy output listing every single effect there is.

In this paper, we present a complete inference algorithm for an expressive and simple descriptive polymorphic effect system of *Eff* [2] (freely available at `http://eff-lang.org`), an ML-style language with handlers of not only exceptions, but of any other *algebraic effect* [22] such as input & output, non-determinism, mutable references and many others [23, 2]. Handlers prove to be extremely versatile and can express stream redirection, transactional memory, backtracking, cooperative multi-threading, delimited continuations, and, like monads, give programmers a way to define their own. And as handlers eliminate effects, they make the effect system *non-monotone*, which helps with the above issue of a snowballing output.

2012 ACM CCS: [**Theory of computation**]: Semantics and reasoning—Program reasoning—Program analysis.

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

LOGICAL METHODS IN COMPUTER SCIENCE

DOI:10.2168/LMCS-10(3:21)2014

Its inference algorithm was constraint based and used a variant of Francois Pottier's garbage collection.

2015-2016

EFFICIENT COMPILATION

In 2015, I visited Tom Schrijvers in Leuven, and we started looking at efficient evaluation of handlers.

```
effect Put: int -> unit
effect Get: unit -> int

let rec loop n =
  if n = 0 then () else
    perform (Put (perform (Get ()) + 1));
    loop (n - 1)

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

let main n =
  (with state_handler handle loop n) 0
```

```
let main n =
  let rec state_handler_loop m s =
    if m = 0 then s
             else state_handler_loop (m - 1) (s + 1)
  in
  state_handler_loop n 0
```

Tom's idea was to do source-level optimisations of Eff code and then to compile it down to OCaml using some sort of a monadic embedding (OCaml didn't have effects at that point yet).

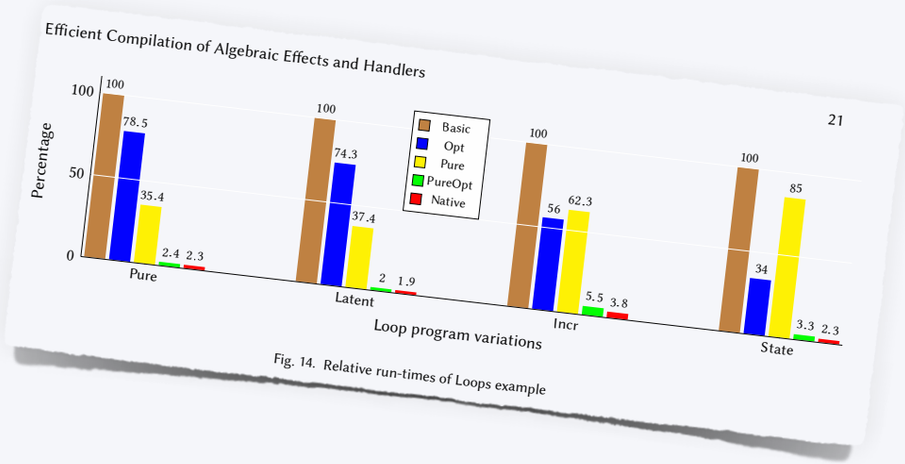
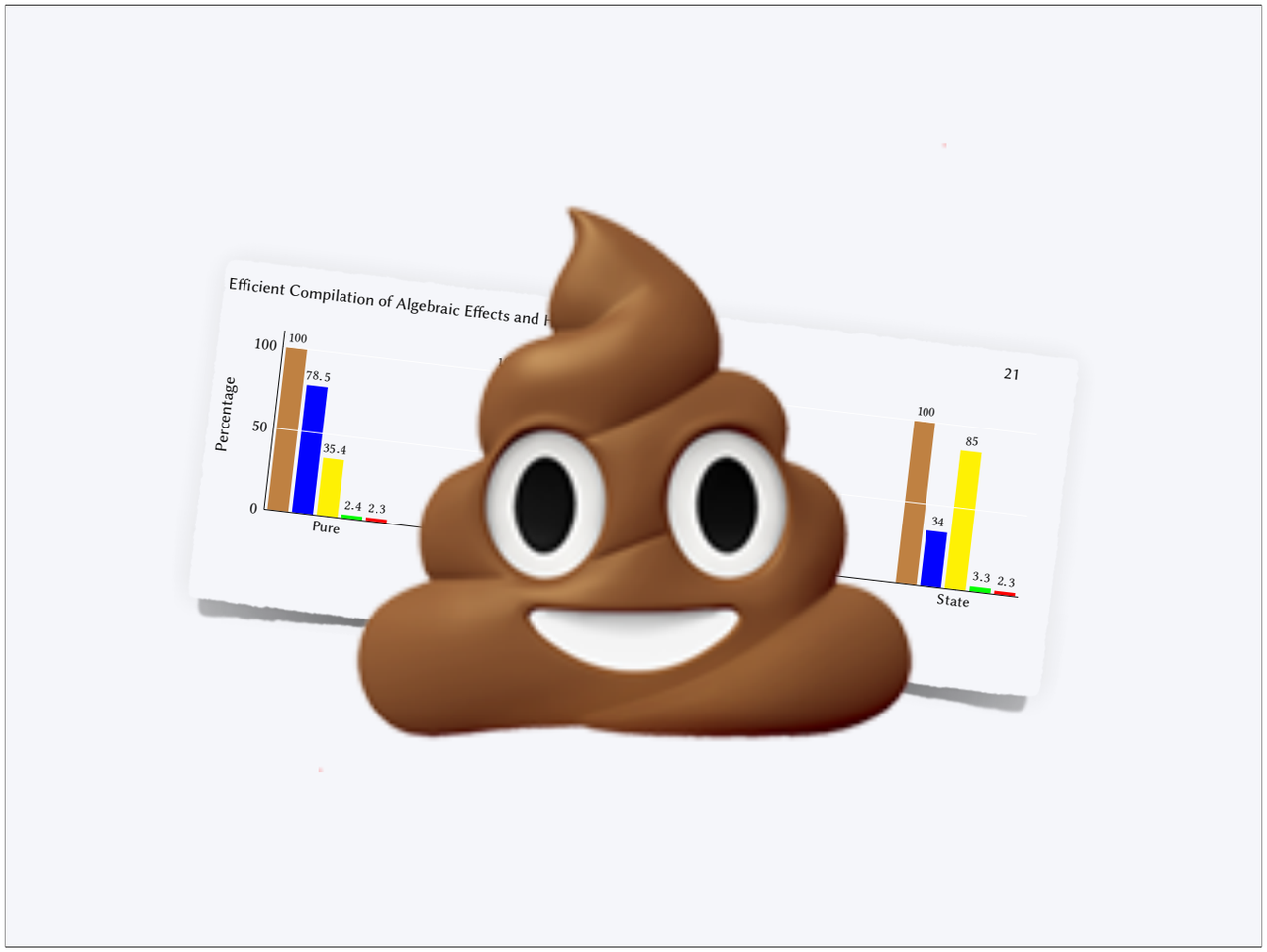
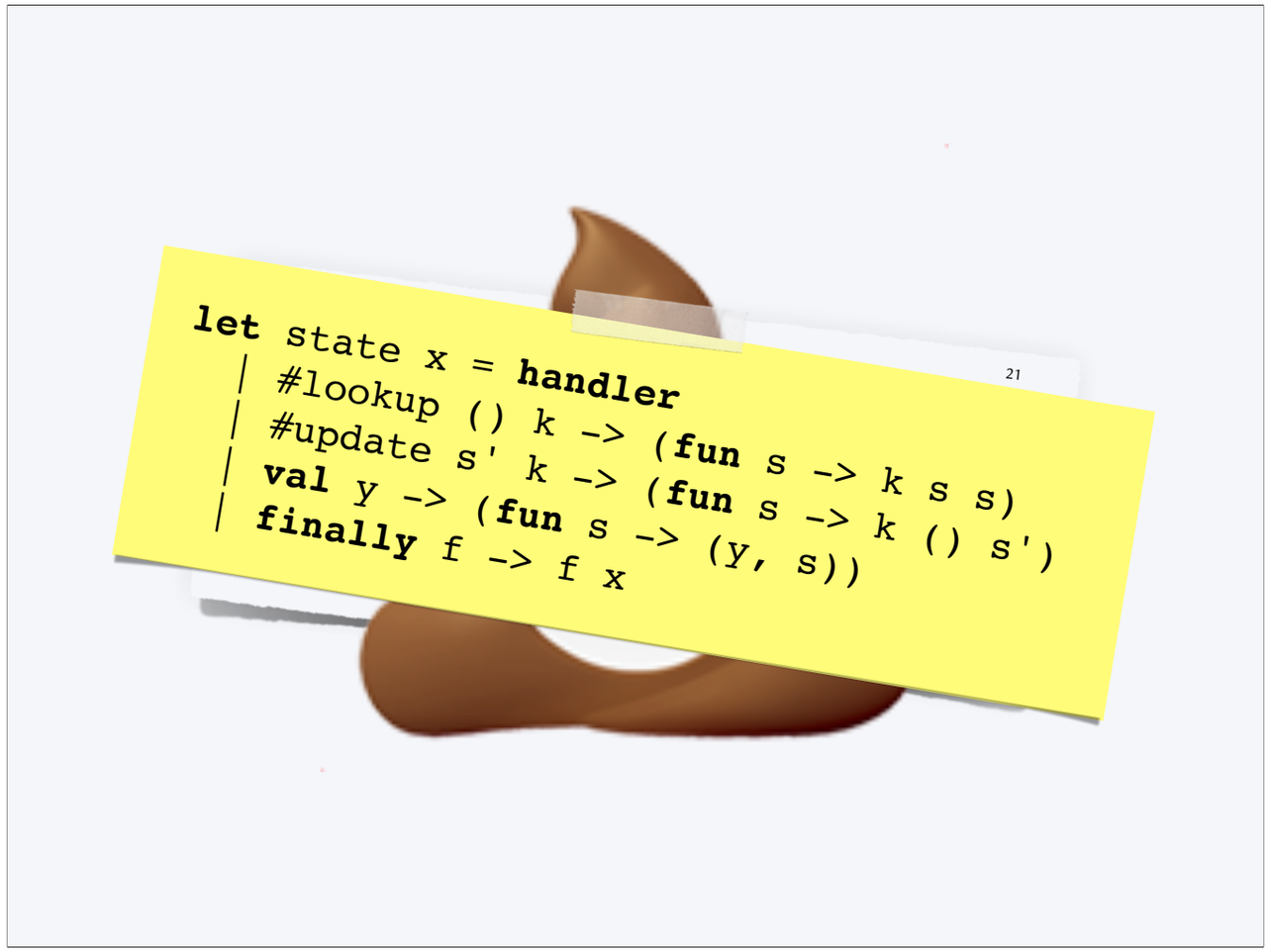


Fig. 14. Relative run-times of Loops example

Our initial results were promising, reaching performance of hand-written OCaml code.



But we ended up having too little examples for a thorough evaulation since the effect system was very fragile and every tweak either made our compilation too conservative and thus too slow, or too agressive and ill-typed.



The first step in resolving the mess was getting rid of instances (since one could use the generativity of modules (which Eff still doesn't have)), though even that did not help.

2017-2019

EXPLICIT SUBTYPING

The next step was to change all the implicit subtyping coercions in Eff into explicit ones, and this was done with the help of Tom Schrijvers and his students, mostly Amr Hany Saleh and Georgios Karachalias.

```
let apply_if p f x =
  if p x then
    f x
  else
    x
```

For example, take the following Eff function.

$$
\begin{array}{l}
\texttt{fun}\ p \mapsto \texttt{return}\ (\texttt{fun}\ f \mapsto \texttt{return}\ (\texttt{fun}\ x \mapsto (\\
\quad \texttt{do}\ b \leftarrow p\, x; \\
\quad \texttt{if}\ b\ \texttt{then}\ f\, x\ \texttt{else}\ \texttt{return}\ x \\
)))
\end{array}
$$

In fine-grain call-by-value, this gets elaborated to the following function with explicit binds and returns.

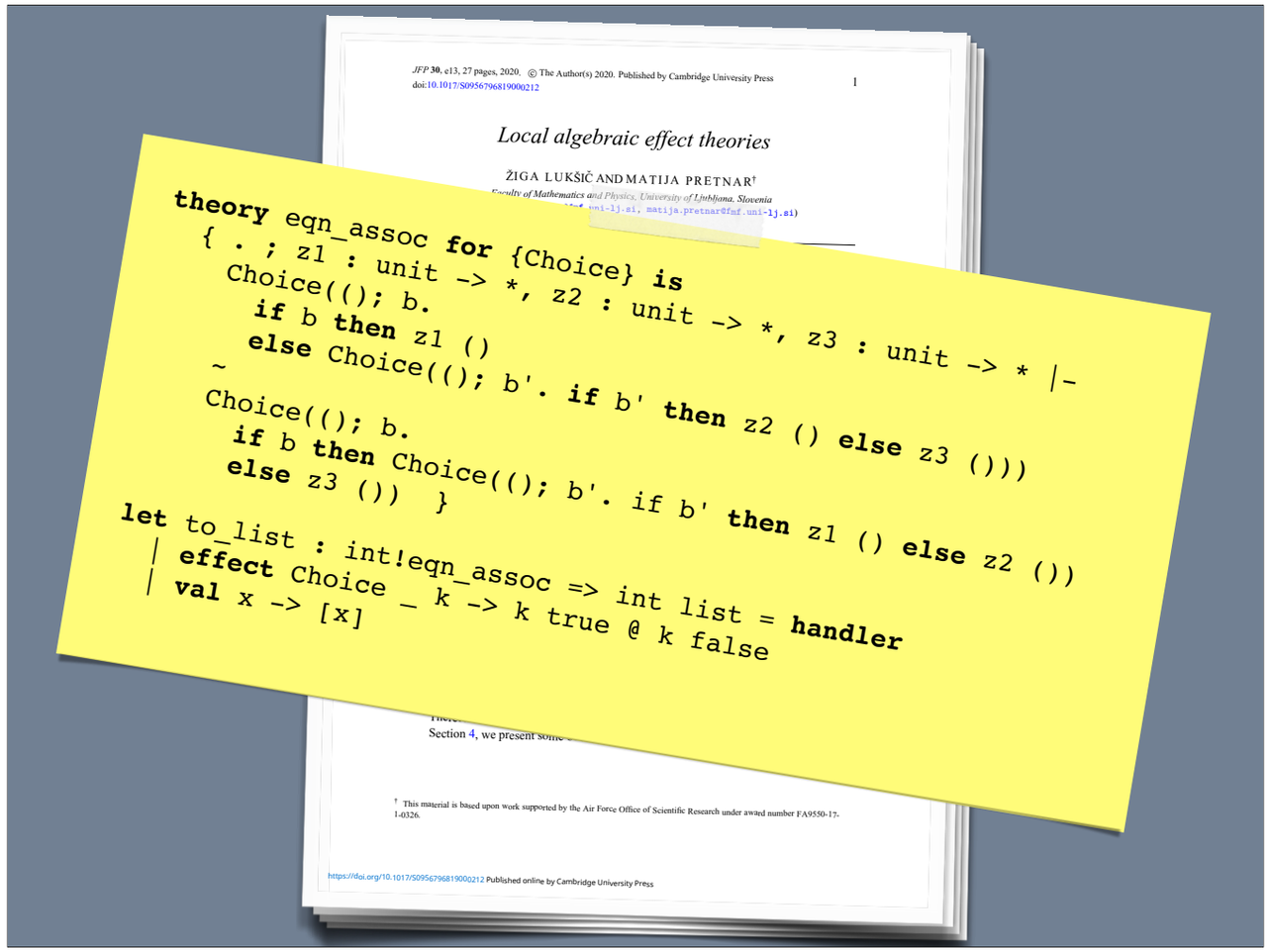
$$
\begin{array}{l}
\texttt{fun}\ (p : \alpha_1 \to \texttt{bool}) \mapsto \texttt{return}\ (\\
\quad \texttt{fun}\ (f : \alpha_2 \to \alpha_3) \mapsto \texttt{return}\ (\\
\quad\quad \texttt{fun}\ (x : \alpha_4) \mapsto (\\
\quad\quad\quad \texttt{do}\ b \leftarrow p\,(x \rhd \omega_1); \\
\quad\quad\quad \texttt{if}\ b\ \texttt{then} \\
\quad\quad\quad\quad (f\,(x \rhd \omega_2)) \rhd \omega_3 \\
\quad\quad\quad \texttt{else} \\
\quad\quad\quad\quad \texttt{return}\ (x \rhd \omega_4) \\
)))
\end{array}
$$

In the new explicitly annotated version, each variable is assigned a type, and types have to match, which is achieved through coercions. For example, the type α_4 of x doesn't need to match the argument type α_1 of f, but there must be an explicit coercion $\omega_1 : \alpha_1 \leq \alpha_4$ witnessing the subtyping. Now, all effect information can be read directly off the syntax.

2019-2020

EEFF

In parallel, a fork of Eff called EEFF appeared.



This was done by my PhD student Žiga Lukšič and was based on the work of extending the effect system with equations that the handlers need to satisfy. In EEFF, one can specify equations that have to hold, but the compiler does not do any checks along the lines of QuickCheck or SMT solving, just prints out the obligations to the user. If anyone is interested in helping continue this work, let me know!

2020

MAJOR CLEANUP

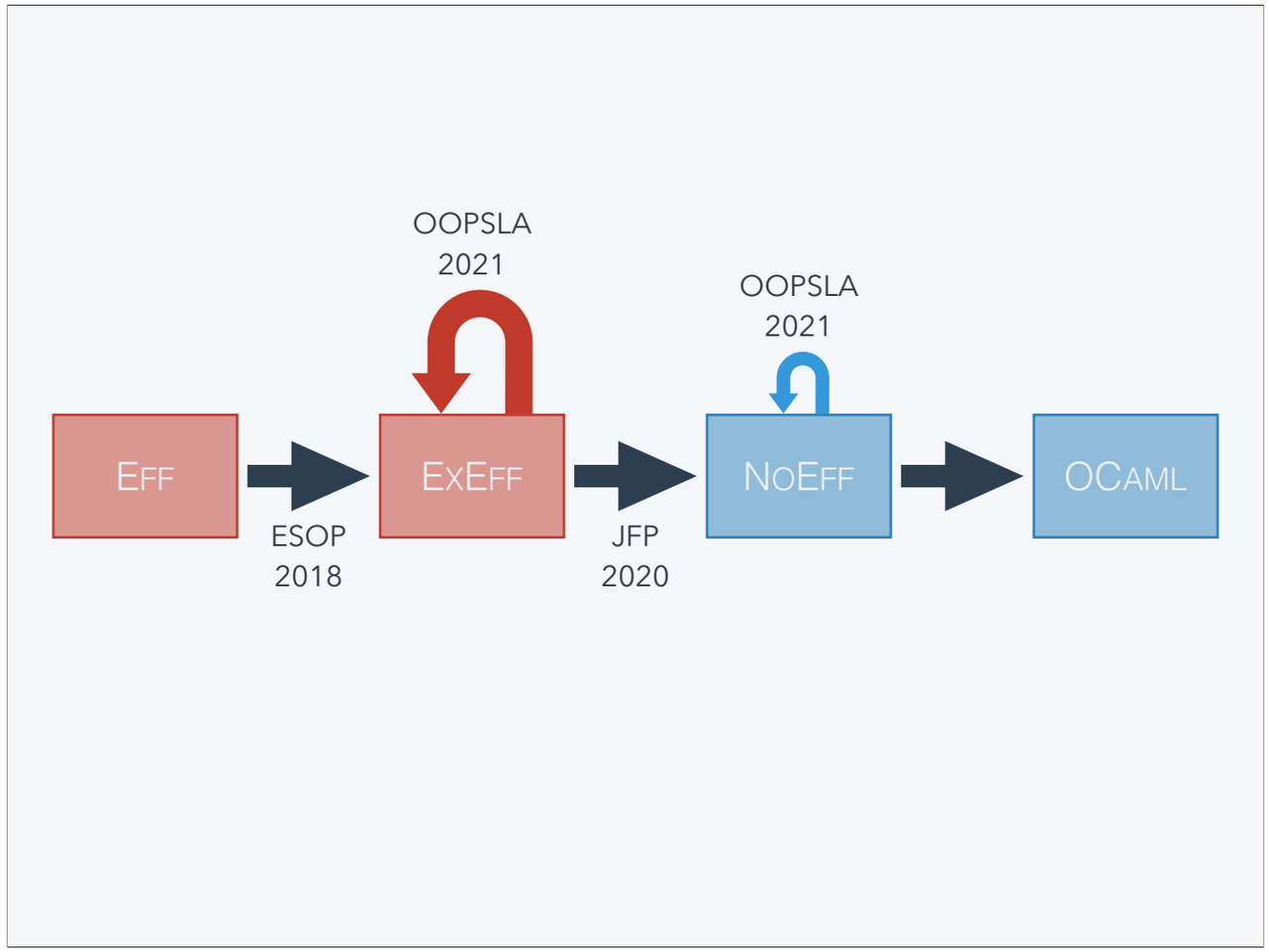
With the explicit subtyping sorted out, we decided to properly implement it, and this was done mostly in 2020 with the help of my student Filip Koprivec.

```
let apply (exp1, exp2) =
  match exp1.ty.term with
  | Type.Arrow (ty1, drty2) ->
      assert (Type.equal_ty exp2.ty ty1);
      { term = Apply (exp1, exp2); ty = drty2 }
  | _ -> assert false
```

One useful technique we used were smart-constructors of typed terms, which raised an assertion fault as soon as some types did not match. This caught countless bugs in source-level optimizations, which shuffle terms around a lot.

2021
OPTIMIZATIONS

With Eff cleaned up, it was time in 2021 to return to our initial work on optimizations.



This worked smoothly, and the current pipeline is as follows. After desugaring, Eff is first elaborated into an explicitly typed core language. Next, it is translated into an OCaml-like language that features no native effects, just their monadic reification. On both languages, mostly on the core one, we perform source-level optimizations that inline handlers, extract pure computations, …

2022-2023

SIMPLIFYING COERCIONS

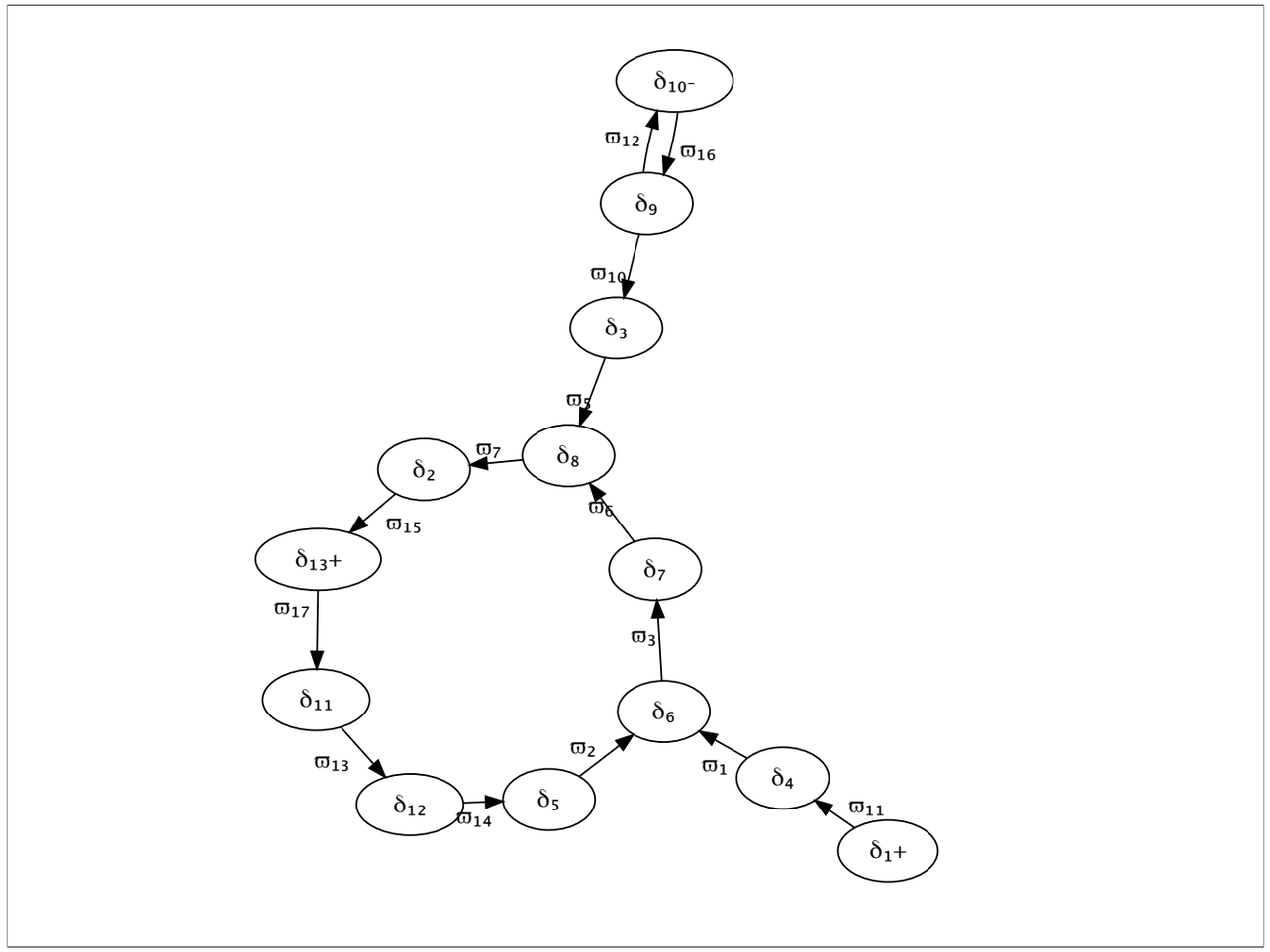
Our current work focuses on simplifying subtyping coercions.

```
fun (p : α1 → bool) ↦ return (
  fun (f : α2 → α3) ↦ return (
    fun (x : α4) ↦ (
      do b ← p (x ▷ ω1);
      if b then
        (f (x ▷ ω2)) ▷ ω3
      else
        return (x ▷ ω4)
)))
```

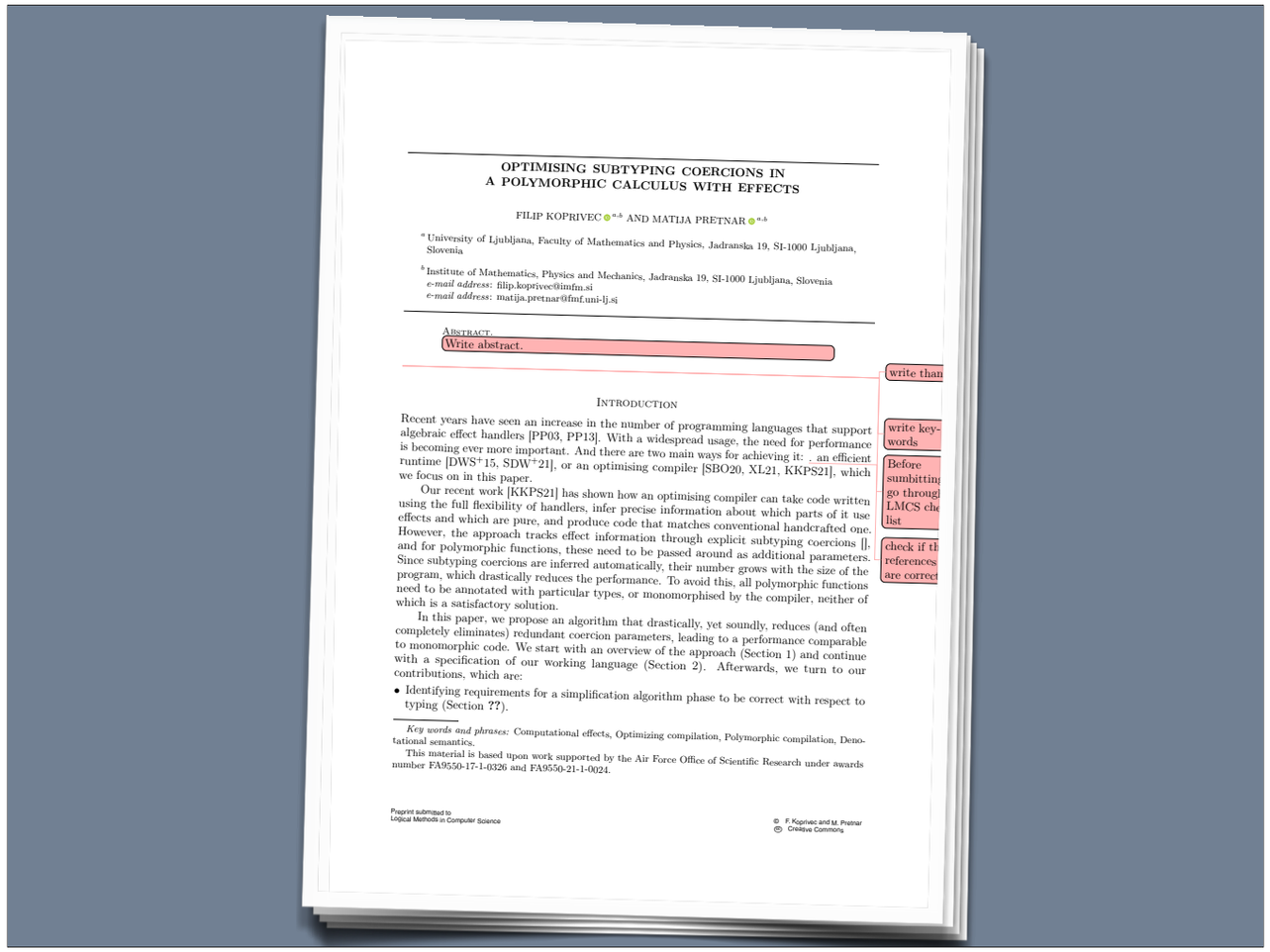
Recall that core language features explicit coercions in terms.

```
let apply_if w1 w2 w3 w4 p f x =
  p (x |> w1) >>= fun b ->
  if b then
    (f (x |> w2)) |> coer_comp w3
  else
    return (x |> w4)
```

When translating to OCaml, these coercions gain computational meaning (eg. they embed pure values into the monad), and any polymorphic coercion parameters get translated into additional function arguments. Very simple functions have a handful of such additional arguments, while a quick-sort implementation for example, already features a couple hundred of them, which is unacceptable.



Unfortunately, we cannot perform Francois Pottiers garbage collection because that relies on computational irrelevance of coercions and completely removes them. In our case, coercions form part of terms and have to be replaced rather than removed. Instead we rely on heuristics such as collapsing of cycles, collapsing coercions between parameters with singualr bounds, … We cannot get rid of all the coercions in general, but we can for example remove all of them in Eff's standard library or the quick-sort example.



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@imfm.si
e-mail address: matija.pretnar@fmf.uni-lj.si

Abstract.
Write abstract.

write than

write key-words

Before sumbitting go through LMCS che list

check if th references are correc

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.

Our recent work [KKPS21] has shown how an optimising compiler can take code written using the full flexibility of handlers, infer precise information about which parts of it use effects and which are pure, and produce code that matches conventional handcrafted one. However, the approach tracks effect information through explicit subtyping coercions ||, and for polymorphic functions, these need to be passed around as additional parameters. Since subtyping coercions are inferred automatically, their number grows with the size of the program, which drastically reduces the performance. To avoid this, all polymorphic functions need to be annotated with particular types, or monomorphised by the compiler, neither of which is a satisfactory solution.

In this paper, we propose an algorithm that drastically, yet soundly, reduces (and often completely eliminates) redundant coercion parameters, leading to a performance comparable to monomorphic code. We start with an overview of the approach (Section 1) and continue with a specification of our working language (Section 2). Afterwards, we turn to our contributions, which are:

- Identifying requirements for a simplification algorithm phase to be correct with respect to typing (Section ??).

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

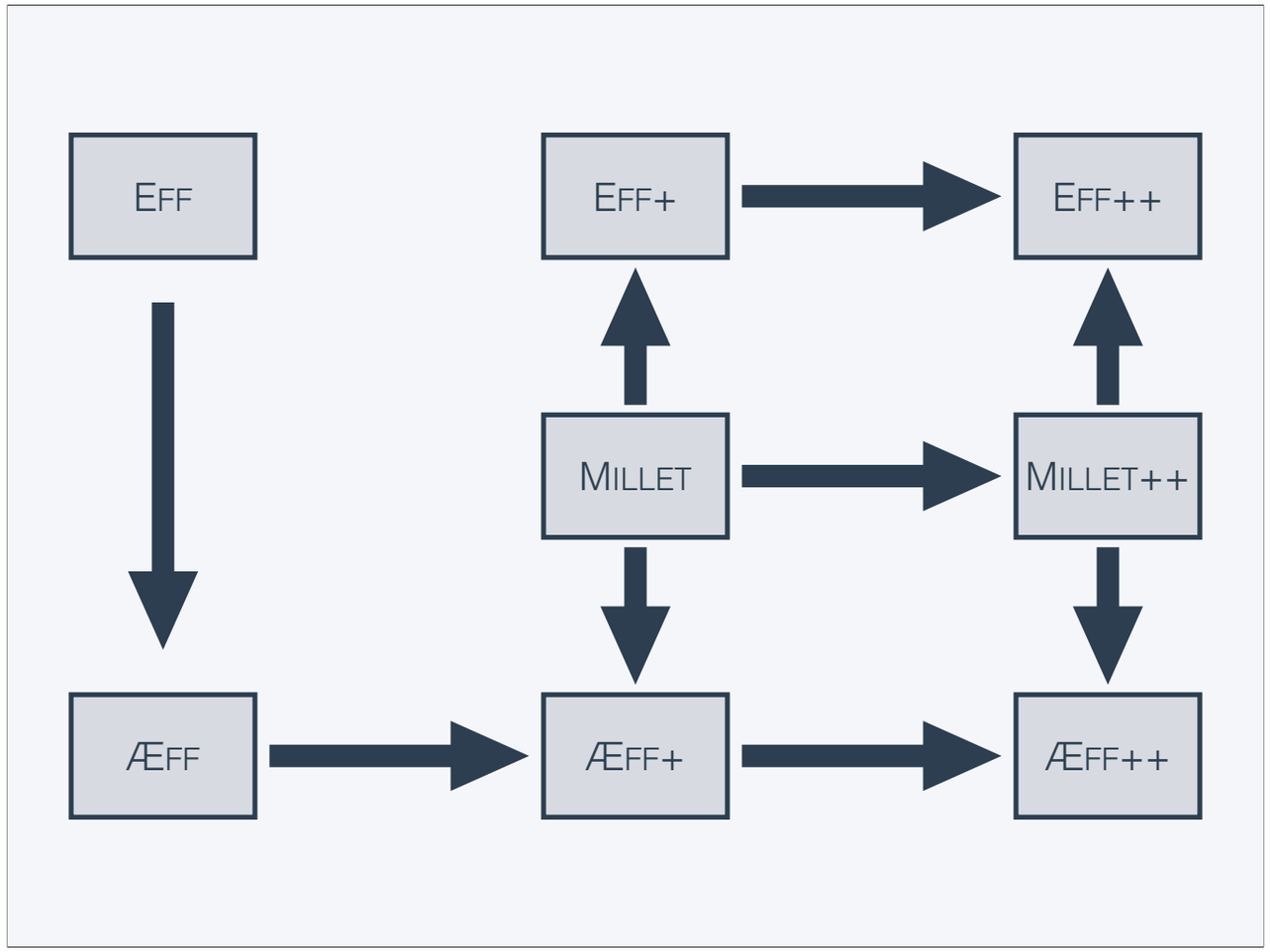
This material is based upon work supported by the Air Force Office of Scientific Research under awards number FA9550-17-1-0326 and FA9550-21-1-0024.

The correctness of this simplifications is established in a paper that Filip and I are submitting soon to LMCS.

2024-

What are the next steps for Eff?

The answer lies in Millet, a fine-grain call-by-value based ML-like language.



A few years ago, I worked with Danel Ahman on asynchronous effects. To test our ideas, I've developed a small prototype language called Æff. I got it by taking Eff, removing some stuff and adding some new one. Of course, during writing of Æff I corrected some mistakes present in Eff. How to port those changes back to get an improved version of Eff? What I started is Millet, a template language featuring all the boring but useful things in a language (recursive types, records, interactive loop, parser, lexer, desugarer, simple type-checker, interpreter, …) which one can fork and add only the interesting bits (handlers, asynchronous operations, …). Eff and Æff appeared before, and now I am slowly porting them to the point where they can be considered such forks. Now, if Millet gets an additional feature (right now, a couple of students of mine are developing a module system, a LSP, and a Wasm backend), one can "simply" merge those changes into forks. If you are researching a new feature and want to try it out, I encourage you to take a look at Millet.