Efficient Compilation of Algebraic Effect Handlers

GEORGIOS KARACHALIAS, Tweag, France
FILIP KOPRIVEC, University of Ljubljana, Slovenia and Institute of Mathematics, Physics and Mechanics, Slovenia
MATIJA PRETNAR, University of Ljubljana, Slovenia and Institute of Mathematics, Physics and Mechanics, Slovenia
TOM SCHRIJVERS, KU Leuven, Belgium

The popularity of algebraic effect handlers as a programming language feature for user-defined computational effects is steadily growing. Yet, even though efficient runtime representations have already been studied, most handler-based programs are still much slower than hand-written code.

This paper shows that the performance gap can be drastically narrowed (in some cases even closed) by means of type-and-effect directed optimising compilation. Our approach consists of source-to-source transformations in two phases of the compilation pipeline. Firstly, elementary rewrites, aided by judicious function specialisation, exploit the explicit type and effect information of the compiler's core language to aggressively reduce handler applications. Secondly, after erasing the effect information further rewrites in the backend of the compiler emit tight code.

This work comes with a practical implementation: an optimising compiler from Eff, an ML style language with algebraic effect handlers, to OCaml. Experimental evaluation with this implementation demonstrates that in a number of benchmarks, our approach eliminates much of the overhead of handlers, outperforms capability-passing style compilation and yields competitive performance compared to hand-written OCaml code as well Multicore OCaml's dedicated runtime support.

CCS Concepts: • **Software and its engineering → Functional languages**; **Compilers**; **Interpreters**.

Additional Key Words and Phrases: algebraic effect handlers, optimising compilation, OCaml

ACM Reference Format:
Georgios Karachalias, Filip Koprivec, Matija Pretnar, and Tom Schrijvers. 2021. Efficient Compilation of Algebraic Effect Handlers. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 102 (October 2021), 28 pages. https://doi.org/10.1145/3485479

1 INTRODUCTION

Algebraic effect handlers [Plotkin and Power 2003; Plotkin and Pretnar 2013] have quickly matured from a theoretical model of computational effects to a practical (functional) language feature for user-defined side-effects.

With a bevy of implementations available, runtime performance has become more and more of a concern. Much of the effort to improve performance has been directed towards improving the runtime representation of computations with handlers and associated operations [Dolan et al. 2017,

Authors' addresses: Georgios Karachalias, Tweag, Paris, France, georgios.karachalias@tweag.io; Filip Koprivec, University of Ljubljana, Ljubljana, Slovenia, Institute of Mathematics, Physics and Mechanics, Ljubljana, Slovenia, filip.koprivec@fmf.uni-lj.si; Matija Pretnar, University of Ljubljana, Ljubljana, Slovenia, Institute of Mathematics, Physics and Mechanics, Ljubljana, Slovenia, matija.pretnar@fmf.uni-lj.si; Tom Schrijvers, KU Leuven, Leuven, Belgium, tom.schrijvers@kuleuven.be.

2475-1421/2021/10-ART102
https://doi.org/10.1145/3485479

2015; Hillerström et al. 2020, 2016; Kiselyov and Ishii 2015]. Yet, in practice effect handlers still incur a performance overhead compared to hand-written code and native side-effects.

With an end-to-end overview of a compiler for the Koka language, Leijen [2017] has demonstrated that compilation is a valid alternative avenue for implementing algebraic effects and handlers. Indeed, existing purity-aware compilation approaches like that of Leijen [2017], and also that of Karachalias et al. [2020] in Eff, considerably reduce the runtime. Yet, algebraic effects and handlers are still often an order of magnitude slower than hand-written code. We believe that *optimising* compilation can further narrow this performance gap.

To substantiate this belief we present our approach to the optimised compilation in the context of the Eff language implementation [Bauer and Pretnar 2015]. Notably, we integrate optimisations into the compilation pipeline of Karachalias et al. [2020], which features explicit type and effect information that we exploit. This way we drastically reduce the runtimes and, in many cases, entirely close the performance gap with hand-written code.

In particular, our contributions are:

- Source-to-source optimisations of ExEff [Saleh et al. 2018], the core calculus of the Eff compiler, which include inlining of handlers, specialisation of recursive functions, and additional transformations that reorder code in a way that exposes more optimisation opportunities. (Section 3)
- Source-to-source optimisations of NoEff [Karachalias et al. 2020], a monadic calculus that serves as a stepping stone between ExEff and a language without support for algebraic effect handlers such as Haskell or OCaml. (Section 4)
- A prototype implementation of our optimising compiler as an extension of Eff, a basic functional language with support for algebraic effects and handlers [Bauer and Pretnar 2015]. The prototype is available online at https://zenodo.org/record/5497862. In addition to supplying the novel optimisations, it is also the first to implement and practically validate the ExEff–NoEff compiler pipeline described by Karachalias et al. [2020]. (Section 5)
- Experimental evaluation, which clearly demonstrates the effectiveness of this approach on a number of benchmarks. (Section 6).

Section 7 discusses related work and Section 8 concludes. We start with a short informal overview in Section 2.

2 OVERVIEW

This section motivates the need for optimised compilation on a small example. We explain the relevant concepts of algebraic effects and handlers, though we encourage the reader to look at [Pretnar 2015] for a more detailed introduction. As we are working with both OCaml and Eff, whose syntax closely follows OCaml, we use colours to distinguish between `OCaml code` and `Eff code`.

2.1 Programming with Algebraic Effect Handlers

Our running example is a simple loop that repeatedly increments an implicit integer state. We first declare two effectful operations to manipulate the state and then define a recursive function that increments the state a given number of times.

```
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))
```

Applying `loop` "as is" to any positive integer results in a runtime error similar to one of an uncaught exception, for we have so far given no meaning to `Get` and `Put`. To do so, we use handlers. Like exception handlers, which provide an interpretation for raised exceptions, algebraic effect handlers determine how to interpret operations when they appear in a computation (the *effect clauses*) and how to interpret the final result of a computation (the *return clause*). But in contrast to exceptions, which abort any further computation, operations can have *continuations*, i.e. the remainder of the computation waiting for the result of the operation, and the handler can access them.

A standard way of implementing stateful behaviour is to treat computations as functions from the initial to the final value of the state. This is achieved by the following handler:

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

Let us first take a look at the `Put` clause, which is defined in terms of the parameter `s'` (the new value of the state) and the continuation `k`. We handle `Put` as a function that accepts (though ignores) the initial state, and then resumes the continuation by passing it the expected result `() : unit`. The handlers we present in this paper are *deep* [Kammar et al. 2013], meaning that the handler implicitly continues handling further operations in the continuation. Thus `k ()` is itself a function of the initial state, and we pass it the new state `s'`.

The clause for `Get` is similar, except that we pass the initial state `s` to the continuation `k` twice: first as the result of the lookup, and second as the new (unmodified) initial state. Finally, the return clause ignores the final result of the handled computation and instead returns the current value of the state (though a variant which returns the final result is also possible). Note that by modifying the handled computation into a function, the handler changes the computation's type.

To tie everything together, we define the function `main`, which applies the handler to `loop` and provides the initial state `0` to the resulting function:

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

2.2 Basic Compilation to OCaml

The basic idea behind compiling algebraic effects to OCaml, which does not support them, is to represent Eff computations of type `'a` with the OCaml type `'a computation` that reifies the computation in a datastructure, the so-called free monad. We postpone the discussion of the exact implementation of this type to Section 5.

We build (representations of) effectful computations using the following constructors: a value embedding `return : 'a -> 'a computation` and basic operations `put : int -> unit computation` and `get : unit -> int computation`. The *monadic bind operator* `>>=` composes an effectful computation of type `'a computation` with a continuation of type `'a -> 'b computation` into a resulting `'b computation`.

It is important to note that Eff functions have effectful computations as bodies. Thus an Eff function of a type `'a -> 'b` is translated into a OCaml function of type `'a -> 'b computation`. This applies equally to (curried) multi-argument functions. So an Eff function `f : 'a -> 'b -> 'c` must be translated as `f : 'a -> ('b -> 'c computation) computation`, and an application `f x y` must be translated as `f x >>= fun g -> g y`.

With the above approach, basic compilation translates the `loop` function into the following code:

```
let rec loop n =
  equal n >>= fun f ->
  f 0     >>= fun b ->
  if b then return () else
    get ()  >>= fun s  ->
    plus s  >>= fun g  ->
    g 1     >>= fun s' ->
    put s'  >>= fun _  ->
    minus n >>= fun h  ->
    h 1     >>= fun n' ->
    loop n'
```

where `equal`, `plus` and `minus` are translations of Eff's arithmetic operations into predefined OCaml constants of the appropriate function type. For example, we define

```
let plus = fun x -> return (fun y -> return (x + y))
```

We can translate `state_handler` as

```
let state_handler = handler {
  put_clause = (fun s' k -> return (fun _ -> k () >>= fun f -> f s'));
  get_clause = (fun () k -> return (fun s -> k s >>= fun f -> f s));
  return_clause = (fun _ -> return (fun s -> return s));
}
```

where the record that specifies the handler clauses is of the predefined type

```
type ('a, 'b) handler_clauses = {
  put_clause : int -> (unit -> 'b computation) -> 'b computation;
  get_clause : unit -> (int -> 'b computation) -> 'b computation;
  return_clause : 'a -> 'b computation;
}
```

and the `handler : ('a, 'b) handler_clauses -> ('a computation -> 'b computation)` function takes these handler clauses and returns a function that uses them to transform computations.

Finally, the `main` function may be translated as

```
let main n =
  state_handler (loop n) >>= (fun f -> f 0)
```

While this basic approach is quite uniform and relatively easy to implement, it is not ideal in terms of performance. Indeed, the reification of computations as datastructures and the repeated composition and decomposition of these datastructures incurs a substantial overhead.

Various researchers have attempted to mitigate this overhead by using more efficient representations of the free monad datastructure. The most effective are those that leverage special support for delimited continuations in the target language's runtime system; unfortunately, these approaches are not suitable for target languages without that support.

The optimised compilation approach we present in this paper is complementary to the above datastructure and runtime improvements. It statically eliminates algebraic effects and handlers from the program where it can to avoid the associated runtime overhead altogether.

2.3 ExEff: Explicit Effect Subtyping

As we will see, statically available type-and-effect information is instrumental for this. For that reason, the Eff compiler is equipped with an explicitly typed core language called ExEff [Karachalias et al. 2020].

ExEff terms contain sufficient information for their types to be easily reconstructed. Notably, the types of computations take the form $A \,!\, \Delta$ where A is the type of value produced by the computation and Δ is the set of unhandled operations that may be invoked by the computation. For example, the type of the `loop` function is `int -> unit ! {Put,Get}` because it maps integers to the unit value and calls the operations `Put` and `Get` in the process.

The `state_handler` has type `(unit ! {Put,Get}) => ((int -> int ! {}) ! {})`. This type reveals that the handler maps stateful computations with `unit` value to pure computations that return a pure function from `int` to `int`. Hence, when `c` has type `unit ! {}`, we can simplify `with state_handler handle c`—based on the type information alone—to invoking the `return_clause` on `c`. This illustrates on a simple example how type-and-effect information is a driving force behind our optimised compilation.

The subtyping of Eff is a complicating factor in the transformation-based optimisation passes of the compiler. Indeed, in the above example, `c` is of type `unit ! {}` instead of the requisite type `unit ! {Put,Get}` expected by the handler. The reason the code is still valid is that the former is a subtype of the latter, and subtyping makes up for the difference. Eff's core language ExEff makes these appeals to subtyping explicit in the form of cast expressions. Hence, c actually takes the form $c' \rhd \gamma$ where c' has type `unit ! {}`, γ is a *coercion* that witnesses the subtyping and as result c has the appropriate type `unit ! {Put,Get}`. These explicit subtyping witnesses not only help easily reconstruct the type information for optimisations. They have also proven to be an indispensable sanity check for our rewrites that give us a measure of confidence in the correctness of the compiler. In contrast, an initial attempt without an explicitly typed core language turned out to be unmanageable due to many elusive typing bugs.

2.4 Purity-Aware Compilation

An immediate benefit of tracking explicit type-and-effect information is that it is easy to identify *pure* computations. Pure computations are of type $A \,!\, \emptyset$, i.e., their effect set is empty; this means that they do not call any algebraic effects. Hence, we do not need to reify pure computations in a datastructure and can instead generate regular, overhead-free OCaml code for them, as proposed by Leijen [2017]. For instance, since the arithmetic operators used in `loop` are pure, we can translate them directly into OCaml's arithmetic operations and bind their result with `let` rather than with the more expensive `>>=`.

```
let rec loop n =
  let f = (=) n in
  let b = f 0 in
  if b then return () else
    get () >>= fun s  ->
    let g = (+) s in
    let s' = g 1 in
    put s' >>= fun _  ->
    let h = (-) n in
    let n' = h 1 in
    loop n'
```

Table 1. Slowdown factors (lower is better) of different optimisation stages in comparison to a pure hand-written code.

	basic compilation	purity-aware compilation	source-to-source optimisations	fully optimised
simple pure loop	27.37	4.45	22.57	1.02
simple loop with a latent effect	41.53	5.96	39.43	1.64
stateful loop	80.17	64.28	38.14	0.98

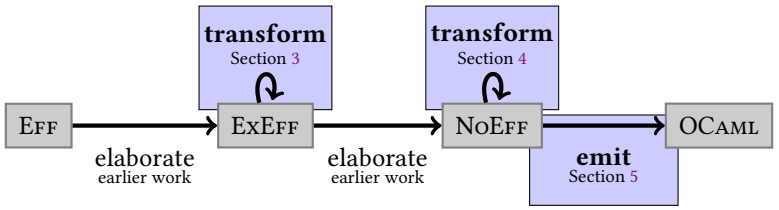

Fig. 1. The pipeline of our optimising compiler.

As we can see, translating pure Eff computations to plain OCaml computations avoids six `>>=` calls from the recursive loop body, and avoids the accompanying datastructures. Nevertheless, the backbone of the loop body still features two `>>=` calls that sequence the two effectful `get` and `put` operations. Additional optimisation techniques are needed to eliminate the overhead altogether.

2.5 Source-to-Source Optimisations

Our compiler provides further optimisations in the form of two source-to-source transformations at the level of ExEff. The first source-to-source transformation performs aggressive compile-time reductions on the algebraic effects features in the program to simplify and, if possible, eliminate them entirely. These reductions are aided by the second transformation, which specialises recursive functions for specific handlers.

Together the two transformations aggressively optimise the code by replacing operation calls in a handled operation with their corresponding operation clauses. The result is altogether much tighter code:

```
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
```

Here, all of the handler, the explicit operations `get` and `put`, and their explicit sequencing with `>>=` have been eliminated. The recursive function `loop` has been locally specialised for the particular interpretation of `state_handler`. The resulting code is essentially equivalent to the hand-written equivalent.

2.6 Summary

Figure 1 summarises the compiler pipeline, which extends that of earlier work [Karachalias et al. 2020] with new optimisation passes. After parsing Eff programs, the compiler infers their types

$$
\begin{array}{rl}
\text{value} & v ::= x \mid \texttt{unit} \mid \texttt{fun}\ (x : T) \mapsto c \mid h \mid v \triangleright \gamma \\
\text{handler} & h ::= \{\texttt{return}\ (x : T) \mapsto c_r, \mathsf{Op}_1\, x\, k \mapsto c_{\mathsf{Op}_1}, \ldots, \mathsf{Op}_n\, x\, k \mapsto c_{\mathsf{Op}_n}\} \\
\text{computation} & c ::= \texttt{return}\ v \mid \mathsf{Op}\ v\ (y : T.c) \mid \texttt{do}\ x \leftarrow c_1; c_2 \mid \texttt{handle}\ c\ \texttt{with}\ v \\
& \quad \mid\ v_1\ v_2 \mid \texttt{let}\ x = v\ \texttt{in}\ c \mid \texttt{let rec}\ f\ x = c_1\ \texttt{in}\ c_2 \mid c \triangleright \gamma \\
\text{value type} & T ::= \texttt{Unit} \mid T \longrightarrow \underline{C} \mid \underline{C}_1 \Rightarrow \underline{C}_2 \\
\text{computation type} & \underline{C} ::= T\ !\ \Delta \\
\text{dirt} & \Delta ::= \emptyset \mid \{\mathsf{Op}\} \cup \Delta \\
\text{coercion type} & \pi ::= T_1 \leqslant T_2 \mid \Delta_1 \leqslant \Delta_2 \mid \underline{C}_1 \leqslant \underline{C}_2 \\
\text{coercion} & \gamma ::= \langle \texttt{Unit} \rangle \mid \gamma_1 \longrightarrow \gamma_2 \mid \gamma_1 \Rightarrow \gamma_2 \mid \emptyset_\Delta \mid \{\mathsf{Op}\} \cup \gamma \mid \gamma_1\ !\ \gamma_2
\end{array}
$$

Fig. 2. ExEff Syntax

and elaborates them into the ExEff core language. This paper adds a novel source-to-source transformation phase after that, which optimises the original ExEff program (see Section 3). Next, ExEff is turned into NoEff, which is a representation of the target language (OCaml) without support for algebraic effects and handlers. Another optimisation pass removes spurious coercions from this representation (Section 4). Finally, the compiler emits actual OCaml code from the NoEff representation (see Section 5).

Several phases in the compiler make the generated code more effective, as is evident from the benchmarks in Table 1. The table compares the slowdown factor of several compiled variants of the above loop, each time with the equivalent hand-written pure OCaml code as the reference. The *simple pure loop* is a loop that does not feature any operations, the *simple loop with a latent effect* is a loop that features an operation which is dynamically not invoked because it sits in a dead branch, and the *stateful loop* is the running example of this section. The table shows that the basic monadic encoding is extremely inefficient and is an order of magnitude slower. The purity-aware compilation from ExEff to NoEff due to Karachalias et al. [2020] gets rid of unnecessary overhead, especially if the effects are never triggered. In contrast, our novel optimisations based on source-code rewriting of ExEff and of NoEff eliminate handlers and make the biggest impact in the presence of effects. With the synergy of both combined, we achieve runtimes often indistinguishable from those of hand-written code.

3 EXEFF OPTIMISATIONS

This section presents our optimisations of the explicitly-typed core language ExEff. The optimisations take the form of source-to-source transformations; they aim to statically reduce the application of handlers to algebraic effect operations. Before we explain these transformations (Sec. 3.2) and the complementary function specialisation (Sec. 3.3), we briefly summarise the ExEff language of Karachalias et al. [2020] (Sec. 3.1).

3.1 The ExEff Language

Figure 2 presents the syntax of ExEff.[1] ExEff is a fine-grained call-by-value calculus that segregates terms into values v and computations c. Values comprise, as usual, variables, unit values, and function abstractions, but also handlers h and type casts $v \triangleright \gamma$. Handlers h are essentially records with fields that explain how to interpret particular operations (the effect clauses) and how to

[1] In order to focus on the essence of the optimisations, we have omitted the polymorphic fragment of the language from the paper.

interpret the final result of a computation (the return clause). In the remainder of the paper we often abbreviate sets of handler clauses $\{\mathtt{Op}_1\, x\, k \mapsto c_{\mathtt{Op}_1}, \ldots, \mathtt{Op}_n\, x\, k \mapsto c_{\mathtt{Op}_n}\}$ as $[\mathtt{Op}\, x\, k \mapsto c_{\mathtt{Op}}]_{\mathtt{Op} \in O}$, and write O to denote the set of handled operations $\{\mathtt{Op}_1, \ldots, \mathtt{Op}_n\}$. We assume a given set of *operations* Op, including operations such as `Get` or `Put`. A type cast $v \rhd \gamma$ changes the type of a value v according with coercion γ (see below).

Computations include operations calls $\mathtt{Op}\ v\ (y : T.c)$, sequencing ($\mathtt{do}\ x \leftarrow c_1; c_2$), operation handling ($\mathtt{handle}\ c\ \mathtt{with}\ v$), term application ($v_1\ v_2$), non-recursive let-bindings ($\mathtt{let}\ x = v\ \mathtt{in}\ c$), recursive let-bindings ($\mathtt{let\ rec}\ f\ x = c_1\ \mathtt{in}\ c_2$), and type casting $c \rhd \gamma$. Form $\mathtt{return}\ v$ lifts a pure value into a computation,

Similarly to terms, we distinguish between two kinds of types: value types T and computation types $\underline{C}$. The former classify values, while the latter classify computations. There are three forms of value types: the unit type `Unit`, function/arrow types $T \to \underline{C}$, and handler types $\underline{C}_1 \Rightarrow \underline{C}_2$. Computation types $\underline{C}$ combine a value type and a dirt ($T\ !\ \Delta$). A dirt Δ overapproximates the set of operations that a computation may call; it can be either the empty set $\emptyset$ or non-empty set $\{\mathtt{Op}\} \cup \Delta$.

Subtyping coercions—denoted by γ—allow us to change the type of values and computations via explicit casting ($\rhd$). Hence, coercions are classified by the subtyping relation they witness, captured by coercion types π. Reflexivity for ground value types $\langle \mathtt{Unit} \rangle$ and subtyping between the empty dirt and any other $\emptyset_\Delta$ form the base cases. The remaining forms are simple congruences: $\gamma_1 \to \gamma_2$ for function types, $\gamma_1 \Rightarrow \gamma_2$ for handler types, $\{\mathtt{Op}\} \cup \gamma$ for non-empty dirt lists, and $\gamma_1\ !\ \gamma_2$ for computation types. Using the above congruences we can construct a reflexive coercion for any type T, which we denote as $\langle T \rangle$.

3.2 ExEff Source-to-Source Transformations

This section presents the ExEff source-to-source transformations at the heart of our optimisation approach. These transformations leverage the information of the type-and-effect system as well as the syntactic structure of terms to perform a number of optimisations that aim to remove handlers. We denote these transformations in terms of rewrite rules of the form $v_1 \rightsquigarrow v_2$ for values, and $c_1 \rightsquigarrow c_2$ for computations.

We have not formally proven the correctness of the rewrite rules, though we conjecture that they are both type-preserving and semantics-preserving as we have tested both of these properties on a test suite. We have naturally checked semantic preservation by verifying whether the same output is produced with and without the optimisations. More interestingly, the implementation uses assertions to ensure type preservation and employs *smart constructors* to type-check transformed ExEff programs. The latter are functions that build terms with the corresponding data constructors and locally re-typecheck.

We have classified the rules in three groups based on what they do: the cast rules, the normalization and β-rules, and the handler reduction rules. The last group contains the handler-specific optimisations, while the other two groups are mainly enablers: they simplify and re-arrange the program code to create opportunities for the handler reduction rules.

3.2.1 Cast Rules. The first set of rewrite rules deals with coercions and is given in Figure 3. The rules aim to expose rewrite opportunities for other rules by moving casts out of the way, by distributing them into subterms and possibly eliminating them altogether. We prefer this approach of many collaborating small rules over building additional complexity (for looking through casts) into a smaller set of rules.

The first two rules, Elim-Co-Val and Elim-Co-Comp, eliminate redundant casts on expressions and computations. By redundant we mean casts that do not alter the type of the value or computation. The other rules are so-called *push rules*, and they come directly from ExEff's operational

$$\frac{\gamma : T \leqslant T}{v \rhd \gamma \rightsquigarrow v}\ \textsc{Elim-Co-Val} \qquad \frac{\gamma : \underline{C} \leqslant \underline{C}}{c \rhd \gamma \rightsquigarrow c}\ \textsc{Elim-Co-Comp}$$

$$\frac{}{(\mathsf{Op}\ v\ (y : T.c)) \rhd \gamma \rightsquigarrow \mathsf{Op}\ v\ (y : T.(c \rhd \gamma))}\ \textsc{Push-Co-Op}$$

$$\frac{c_1 : T}{(\mathsf{do}\ x \leftarrow c_1; c_2) \rhd (\gamma_1\ !\ \gamma_2) \rightsquigarrow \mathsf{do}\ x \leftarrow (c_1 \rhd \langle T \rangle\ !\ \gamma_2); (c_2 \rhd \gamma_1\ !\ \gamma_2)}\ \textsc{Push-Co-Do}$$

$$\frac{}{(v_1 \rhd (\gamma_1 \to \gamma_2))\ v_2 \rightsquigarrow (v_1\ (v_2 \rhd \gamma_1)) \rhd \gamma_2}\ \textsc{Push-Co-App}$$

$$\frac{}{\mathtt{handle}\ c\ \mathtt{with}\ (v \rhd (\gamma_1 \Rightarrow \gamma_2)) \rightsquigarrow (\mathtt{handle}\ (c \rhd \gamma_1)\ \mathtt{with}\ v) \rhd \gamma_2}\ \textsc{Push-Co-Handle}$$

Fig. 3. ExEff optimisations: Push Rules and Elimination of Casts

$$\frac{}{(\mathtt{fun}\ (x : T) \mapsto c)\ v \rightsquigarrow c[v/x]}\ \textsc{App-Fun} \qquad \frac{}{\mathtt{let}\ x = v\ \mathtt{in}\ c \rightsquigarrow c[v/x]}\ \textsc{LetVal}$$

$$\frac{}{(\mathsf{do}\ x \leftarrow ((\mathtt{return}\ v) \rhd (\gamma_{v_1}\ !\ \gamma_{\Delta_1}) \rhd \cdots \rhd (\gamma_{v_n}\ !\ \gamma_{\Delta_n})); c) \rightsquigarrow c[(v \rhd \gamma_{v_1} \rhd \cdots \rhd \gamma_{v_n})/x]}\ \textsc{Do-Ret}$$

$$\frac{}{\mathsf{do}\ x \leftarrow (\mathsf{Op}\ v\ (y : T.c_1)); c_2 \rightsquigarrow \mathsf{Op}\ v\ (y : T.\mathsf{do}\ x \leftarrow c_1; c_2)}\ \textsc{Do-Op}$$

$$\frac{}{(\mathsf{do}\ x \leftarrow (\mathsf{do}\ y \leftarrow c_1; c_2); c_3) \rightsquigarrow (\mathsf{do}\ y \leftarrow c_1; (\mathsf{do}\ x \leftarrow c_2; c_3))}\ \textsc{Do-Do}$$

Fig. 4. ExEff optimisations: Normalization and β-Rules

semantics [Karachalias et al. 2020, Appendix B]. Their purpose is to distribute casts that block redexes into the subterms, thereby enabling reduction.

3.2.2 Normalization and β-Rules. The second set of rewrite rules, given in Figure 4, focuses on the simplifying and normalizing the structure of the program to expose opportunities for eliminating handlers.

The first four (App-Fun, LetVal, Do-Ret, and Do-Op) correspond to ExEff's β-rules and partially evaluate computations. To avoid code blow-up or increased runtime, the substitution $c[v/x]$ is performed selectively, for example when v is atomic or x appears at most once in c. The last rule reorders left-nested nested do-computations, utilizing the fact that do is associative, in order to normalise them to right-nested form. There is a clear synergy between rules Do-Op and Do-Do: an operation that is buried under nested do-computations can be brought to the surface via those two rules, so that it can potentially be reduced by a surrounding handler.

3.2.3 Handler Reduction Rules. The third set of rewrite rules, given in Figure 5, deals with reducing computations of the form (handle c with h). All rules except for the last one deal with different cases of c.

Rules With-LetVal and With-LetRec distribute the handler application with respect to plain and recursive let. The next two rules are essentially borrowed from ExEff's operational semantics.

In all the rules below, let $h = \{\texttt{return}\ x \mapsto c_r, [\texttt{Op}\ x\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in O}\}$

$$\frac{}{\texttt{handle}\ (\texttt{let}\ x = v\ \texttt{in}\ c)\ \texttt{with}\ h \rightsquigarrow \texttt{let}\ x = v\ \texttt{in}\ (\texttt{handle}\ c\ \texttt{with}\ h)}\ \textsc{With-LetVal}$$

$$\frac{}{\texttt{handle}\ (\texttt{let rec}\ f\ x = c_1\ \texttt{in}\ c_2)\ \texttt{with}\ h \rightsquigarrow \texttt{let rec}\ f\ x = c_1\ \texttt{in}\ (\texttt{handle}\ c_2\ \texttt{with}\ h)}\ \textsc{With-LetRec}$$

$$\textsc{With-Handled-Op}\quad \frac{\texttt{Op} \in O}{\texttt{handle}\ (\texttt{Op}\ v\ (y : T.c))\ \texttt{with}\ h \rightsquigarrow c_{\texttt{Op}}[v/x, (\texttt{fun}\ (y : T) \mapsto \texttt{handle}\ c\ \texttt{with}\ h)/k]}$$

$$\frac{\texttt{Op} \notin O}{\texttt{handle}\ (\texttt{Op}\ v\ (y : T.c))\ \texttt{with}\ h \rightsquigarrow \texttt{Op}\ v\ (y : T.\texttt{handle}\ c\ \texttt{with}\ h)}\ \textsc{With-Unhandled-Op}$$

$$\frac{h' = \{\texttt{return}\ y \mapsto (\texttt{handle}\ c_2\ \texttt{with}\ h), [\texttt{Op}\ x\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in O}\}}{\texttt{handle}\ (\texttt{do}\ y \leftarrow c_1; c_2)\ \texttt{with}\ h \rightsquigarrow \texttt{handle}\ c_1\ \texttt{with}\ h'}\ \textsc{With-Do}$$

$$\frac{\gamma_2 : O' \leqslant O \qquad h' = \{\texttt{return}\ y \mapsto (\texttt{let}\ x = y \rhd \gamma_1\ \texttt{in}\ c_r), [\texttt{Op}\ x\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in O'}\}}{\texttt{handle}\ c \rhd (\gamma_1\ !\ \gamma_2)\ \texttt{with}\ h \rightsquigarrow \texttt{handle}\ c\ \texttt{with}\ h'}\ \textsc{With-Cast}$$

$$\frac{h : T_i\ !\ \Delta_i \Rrightarrow T_o\ !\ \Delta_o \qquad c : T\ !\ \Delta \qquad \Delta \cap O = \emptyset}{\texttt{handle}\ c\ \texttt{with}\ h \rightsquigarrow \texttt{do}\ x \leftarrow (c \rhd \langle T \rangle\ !\ (\Delta \cup \emptyset_{(\Delta_o - \Delta)})); c_r}\ \textsc{With-Pure}$$

Fig. 5. ExEff optimisations: Handler Reduction Rules

Rule With-Handled-Op applies the appropriate effect clause, and Rule With-Unhandled-Op distributes the handler into the continuation of an operation that is not affected by the handler.

Rule With-Do turns the handling of a sequence of two computations into a form where the two computations are handled separately. The intuition is that any operations generated by c_1 are processed anyway by the handler h. However, returned values are first bound in c_2, and the resulting computation is further processed by h. The rewritten form accomplishes the same workflow with a single handler around c_1. The hope is that the handler around c_2 can be specialised independently from specialising the handler around c_1.

Similarly, rule With-Cast addresses the handling of a cast by introducing a new handler h' that casts the returned values in the appropriate clause. Additionally, the computation c calls fewer operations than h can handle (witnessed by γ_2), so we accordingly drop effect clauses in h'.

Finally, rule With-Pure ignores the syntactic shape of c and only considers its type-and-effect information. When the intersection of the operations that may be called by c with the operations handled by h is empty, we say that c is *pure relative to* h. In this case, only the handler's return clause is relevant. Hence, we insert it at the end of c, suitably casting it to c' whose dirt matches the dirt Δ_o emitted by h.

While the rewrite rules are all in all rather simple, they have a substantial synergy. For instance, the examples below illustrate how the rules collaborate to cover two interesting cases that at first sight might require additional rules.

Example 3.1. The set of handler reduction rules does not feature the following rule for `return` that is the static counterpart to the corresponding case in the operational semantics.

$$\frac{h = \{\texttt{return}\ x \mapsto c_r, [\texttt{Op}\ x\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op} \in O}\}}{\texttt{handle}\ ((\texttt{return}\ v) \rhd (\gamma_1 \mathbin{!} \gamma_1') \rhd \ldots \rhd (\gamma_n \mathbin{!} \gamma_n'))\ \texttt{with}\ h \rightsquigarrow c_r[(v \rhd \gamma_1 \rhd \ldots \rhd \gamma_n)/x]}\ \textsc{With-Ret}$$

The reason why this rule is not included is that it is not needed. Indeed, it can be derived from the other rules. Take a computation

$$\texttt{handle}\ (\texttt{return}\ v \rhd (\gamma \mathbin{!} \emptyset_{\Delta_1}))\ \texttt{with}\ h$$

where $\emptyset_{\Delta_1}$ is a coercion that suitably increases the pure dirt of the returned value to one expected by the handler $h : T_1 \mathbin{!} \Delta_1 \Longrightarrow T_2 \mathbin{!} \Delta_2$. We first get rid of the cast using With-Cast and get

$$\texttt{handle}\ (\texttt{return}\ v)\ \texttt{with}\ \{\texttt{return}\ y \mapsto (\texttt{let}\ x = y \rhd \gamma\ \texttt{in}\ c_r)\}$$

(note the lack of effect clauses). Now, since `return` v is pure, it is also pure relative to any handler, so we can employ With-Pure and get

$$\texttt{do}\ y \leftarrow (\texttt{return}\ v \rhd (\langle T_1 \rangle \mathbin{!} \emptyset_{\Delta_2})); (\texttt{let}\ x = y \rhd \gamma\ \texttt{in}\ c_r)$$

which simplifies as follows, first by Do-Ret, and then by Elim-Co-Val and LetVal:

$$\texttt{let}\ x = (v \rhd \langle T_1 \rangle) \rhd \gamma\ \texttt{in}\ c_r \quad \rightsquigarrow \quad \texttt{let}\ x = v \rhd \gamma\ \texttt{in}\ c_r \quad \rightsquigarrow \quad c_r[(v \rhd \gamma)/x]$$

A case with multiple coercions around `return` v proceeds similarly.

Example 3.2. A similar derived case is that of handling a bind $\texttt{do}\ y \leftarrow c_1; c_2$ where c_1 is pure relative to a handler h, in which case we can extract it from the handler as:

$$\texttt{handle}\ (\texttt{do}\ x \leftarrow c_1; c_2)\ \texttt{with}\ h \quad \rightsquigarrow \quad \texttt{do}\ y \leftarrow c_1'; (\texttt{handle}\ c_2\ \texttt{with}\ h)$$

If we employ With-Do, we get the reduction

$$\texttt{handle}\ (\texttt{do}\ x \leftarrow c_1; c_2)\ \texttt{with}\ h \quad \rightsquigarrow \quad \texttt{handle}\ c_1\ \texttt{with}\ h'$$

where h' is the same as h except for the return clause. Since c_1 was pure relative to h, it is also pure relative to h' because the two handle the same set of operations O. This means we can employ With-Pure and sequence c_1 with the return clause of h', but this is exactly h applied to c_2, meaning we get

$$\texttt{handle}\ c_1\ \texttt{with}\ h' \quad \rightsquigarrow \quad \texttt{do}\ y \leftarrow c_1'; (\texttt{handle}\ c_2\ \texttt{with}\ h)$$

In essence, we have pushed h inside the bind, appropriately adapting c_1 to c_1' with the increased dirt.

3.3 Function Specialisation

The rewrite rules above deal with most computations of the form (`handle` c `with` h) where h is a handler expression, either dropping the handler altogether or pushing it down in the subcomputations. However, one important case is not dealt with: the case where c is of the form $f\ v$ with f the name of a user-defined function.[2]

Consider this small example of the above situation:

```
let rec go n = go (perform (Next n)) in
handle (go 0) with
| x -> x
| effect (Next n) k -> if n > 100 then n else k (n * n + 1)
```

[2]If f is a function parameter of a higher-order function, we don't do anything.

The non-terminating recursive function `go` seems to diverge. Yet, with the provided handler, its argument steadily increases and evaluation eventually terminates when the argument exceeds `100`.

In order to optimise this situation, we create a specialised copy of the function that has the handler pushed into its body. In other words, for any recursive definition $\texttt{let rec}\ f\ x = c_f\ \texttt{in}\ c$, we perform the following general rewrite inside c:

$$\texttt{handle}\ f\ v\ \texttt{with}\ h \quad \rightsquigarrow \quad \texttt{let rec}\ f'\ x = \texttt{handle}\ c_f\ \texttt{with}\ h\ \texttt{in}\ f'\ v$$

The expectation is that, by exposing the handler to the body of the function (c_f), further optimisations succeed in eliminating the explicit handler. A critical step involved in the post-processing is to "tie the knot": after several rewrite steps in c_f, the handler is applied to the (original) recursive call, so we have a term of the form $\texttt{handle}\ f\ v'\ \texttt{with}\ h$, which we can replace by $f'\ v'$. This eliminates the handler entirely and turns the original example into

```
let rec go n = ... in
let rec go' n = if n > 100 then n else go' (n * n + 1) in
go' 0
```

Generalization to Varying Return Clauses. The above basic specialisation strategy only works when the handler has not changed in the recursive call. Yet, that is often not the case. Take for instance the following example.

```
let rec range n =
  match n with
  | 0 -> []
  | _ -> perform (Fetch ()) :: range (n - 1)
in
handle (range 5) with
| x -> x
| effect (Fetch _) k -> k 42
```

The function `range` creates a list of given length, filling it with elements obtained by the `Fetch` operation. To keep the example small we use a handler that always yields the value `42`.

With the basic specialisation strategy, further optimisation does not succeed in tying the knot. Instead, we obtain this partially optimised form:

```
let rec range n = ... in
let rec range' n =
  match n with
  | 0 -> []
  | _ -> handle (range (n - 1)) with
         | x -> 42 :: x
         | effect (Fetch _) k -> k 42
in
range' 5
```

In the tail position, the rewrite rule With-Do has kicked in to pull the call's continuation into the return clause of the handler. The resulting handler is wrapped around the recursive call, but differs from the original handler and prevents us from tying the knot.

We could create a second specialised function definition for this new handler, but the same problem would arise at its recursive call and so on, yielding an infinite sequence of specialised functions. Instead, we use generalisation to break out of this diverging process. Instead of specialising the

function for one specific handler in this diverging sequence, we specialise it for what they all have in common (the effect clauses) and parametrise it in what is different (the return clause).

This yields the following general rewrite rule: for any recursive definition $\texttt{let rec}\ f\ x = c_f\ \texttt{in}\ c$, we perform the following general rewrite inside c:

$$\texttt{handle}\ f\ v\ \texttt{with}\ \{\texttt{return}\ x \mapsto c_r, [\texttt{Op}\ x\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op}\in O}\}$$
$$\rightsquigarrow$$
$$\texttt{let rec}\ f'\ (x,k) = \texttt{handle}\ c_f\ \texttt{with}\ \{\texttt{return}\ x \mapsto k\ x, [\texttt{Op}\ x\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op}\in O}\}\ \texttt{in}\ f'\ (v, \texttt{fun}\ x \mapsto c_r)$$

and replace each handled recursive call $\texttt{handle}\ f\ v'\ \texttt{with}\ \{\texttt{return}\ x \mapsto c'_r, [\texttt{Op}\ x\ k \mapsto c_{\texttt{Op}}]_{\texttt{Op}\in O}\}$ with $f'\ (v', c'_r)$.

This strategy enables us to tie the knot in the `range` example and obtain this form

```
let rec range' (n, k) =
  match n with
  | 0 -> k []
  | _ -> range' (n - 1, (fun x -> k (42 :: x)))
in
range' (5, (fun x -> x))
```

Note that in effect this approach selectively CPS-transforms recursive functions to specialise them for a particular handler. Due to an explicit continuation argument, the resulting function is less efficient, so in practice, we always attempt the simpler specialisation and use the general one as a fallback in case it fails.

A careful reader may have observed that in the rule With-Pure, we change not only the return clause, but the effect clauses as well. However, the latter change only removes clauses that cannot be reached and thus has no impact on the specialisation. For this reason, varying the return clause is general enough to account for all cases that other optimisation rules can expose.

Termination. If left unchecked, function specialisation can diverge. This is illustrated by the following small example program:

```
let rec go n =
  if n = 0 then perform (Fail ())
    else if perform (Decide ()) then go (n-1) else go (n-2)
in handle (go m) with
   | effect (Decide _) k ->
       handle k true with
       | effect (Fail _) _ -> k false
```

After specialisation for the top-level handler, we obtain

```
let rec go n =
  if n = 0 then perform (Fail ())
    else if Decide then go (n-1) else go (n-2)
in let rec go1 n1 =
     if n1 = 0 then Fail
       else handle go1 (n1-1) with
            | effect (Fail _) _ -> go1 (n1-2)
   in go1 m
```

Note that the specialised function `go1` still contains the second handler, which is now applied to a recursive call. Hence, we can continue by specialising this handled call to obtain

$$
\begin{array}{rl}
\text{term} & t ::= x \mid \mathtt{unit} \mid \mathtt{fun}\ x : A \mapsto t \mid t_1\ t_2 \mid t \rhd \gamma \mid \mathtt{return}\ t \mid h \mid \mathtt{let}\ x = t_1\ \mathtt{in}\ t_2 \\
& \quad \mid \mathtt{let\ rec}\ f\ x = t_1\ \mathtt{in}\ t_2 \mid \mathtt{Op}\ t_1\ (y : B.t_2) \mid \mathtt{do}\ x \leftarrow t_1; t_2 \mid \mathtt{handle}\ t_c\ \mathtt{with}\ t_h \\
\text{handler} & h ::= \{\mathtt{return}\ (x : A) \mapsto t_r, [\mathtt{Op}\ x\ k \mapsto t_{\mathtt{Op}}]_{\mathtt{Op} \in O}\} \\
\text{type} & A, B ::= \mathtt{Unit} \mid A \to A \mid A \Rightarrow B \mid \mathtt{Comp}\ A \\
\text{coercion type} & \pi ::= A \leqslant B \\
\text{coercion} & \gamma ::= \langle \mathtt{Unit} \rangle \mid \gamma_1 \to \gamma_2 \mid \gamma_1 \Rightarrow \gamma_2 \mid \mathtt{comp}\ \gamma \mid \mathtt{return}\ \gamma \mid \ldots
\end{array}
$$

Fig. 6. NoEff Syntax

```
let rec go n =
  if n = 0 then perform (Fail ())
    else if perform (Decide ()) then go (n-1) else go (n-2)
in let rec go1 n1 =
     if n1 = 0 then perform (Fail ())
       else let rec go2 n2 =
              if n2 = 0 then go1 (n1-2)
                else handle (handle go1 (n2-1) with
                               | effect (Fail _) _ -> go1 (n2-2)) with
                     | effect (Fail _) _ -> go1 (n1-2)
            in go2 (n1-1)
   in go1 m
```

Now the resulting code contains two nested handlers around a recursive call. However, the inner of those two handlers is distinct from any of the previous handlers because it refers to the new variable `n2`. Hence, we can specialise again and again without end. This is non-termination is obviously undesirable and so we currently enforce termination by not re-specialising any already specialised function. We leave more sophisticated solutions, e.g., abstracting over the variation among the specialised handlers, to future work.

While we have focused on the interesting case of recursive functions, we apply the same specialisation to non-recursive functions.

4 NOEFF OPTIMISATIONS

This section presents the optimisations that happen in the backend of the compiler, when the ExEff core language has been elaborated into NoEff.

4.1 From ExEff to NoEff

4.1.1 The NoEff Language. The NoEff language is an intermediary between ExEff and general-purpose languages like OCaml without support for algebraic effects. Its syntax is given in Figure 6.

Though the syntax of NoEff is very close to that of ExEff, there are a few notable differences. First, whereas ExEff distinguishes between value and computation terms and types, NoEff has only one sort for terms and one for types. Second, and more importantly, while ExEff has a type-and-effect system that keeps track of *which* operations a computation may call, NoEff only keeps track of *whether* a term may call operations or not. This manifests itself in the lack of dirts: ExEff's fine-grained computation types ($T \,!\, \Delta$) are replaced by NoEff's computation types that do not mention dirts ($\mathtt{Comp}\ A$), and computation coercions ($\gamma_1 \,!\, \gamma_2$) are similarly replaced by the coarser ($\mathtt{comp}\ \gamma$). Similarly, coercions that cast pure computations into impure ones are replaced by the effect-agnostic form ($\mathtt{return}\ \gamma$).

$$\frac{\gamma : A \leqslant A}{t \rhd \texttt{return}\ \gamma \rightsquigarrow \texttt{return}\ t}\ \textsc{Elim-Ret-Co} \qquad \frac{\gamma : A \leqslant A}{t \rhd \gamma \rightsquigarrow t}\ \textsc{Elim-Co-Term}$$

$$\frac{}{\texttt{do}\ x \leftarrow (\texttt{return}\ t_1); t_2 \rightsquigarrow t_2[t_1/x]}\ \textsc{Do-Ret} \qquad \frac{}{\texttt{let}\ x = t_1\ \texttt{in}\ t_2 \rightsquigarrow t_2[t_1/x]}\ \textsc{LetVal}$$

Fig. 7. NoEff optimisations

Finally, NoEff features a few more coercion forms that facilitate the purity-aware translation of ExEff, which are out of the scope of this paper (denoted as ... in Figure 6). Interested readers can find the full definition of NoEff in the work of Karachalias et al. [2020]. What is relevant for performance is that, unlike in ExEff, the NoEff coercions between pure and impure types have computational content because they imply a change of representation.

4.1.2 Purity-aware Translation of ExEff to NoEff. The key idea behind the design of NoEff and the translation scheme of Karachalias et al. [2020, Section 7.4] that transforms ExEff programs into NoEff programs is the differentiation between pure and impure computations. This is nicely summarised in the more nuanced compilation of computation types that, conceptually, takes the following form (rephrased in terms of the translation function $[\![\cdot]\!]$ instead of the original inductive rules):

$$[\![T\ !\ \Delta]\!] = \begin{cases} [\![T]\!] & , \text{if } \Delta = \emptyset \\ \texttt{Comp}\ [\![T]\!] & , \text{if } \Delta \neq \emptyset \end{cases}$$

This distinction between pure and impure types also applies to the translation of the other well-typed terms (values, computations, coercion types, and coercions). For example, do-bindings are translated as follows:

$$[\![\Gamma \vdash (\texttt{do}\ x \leftarrow c_1; c2) : B\ !\ \Delta]\!] = \begin{cases} \texttt{let}\ x = [\![c_1]\!]\ \texttt{in}\ [\![c_2]\!] & , \text{if } \Delta = \emptyset \\ \texttt{do}\ x \leftarrow [\![c_1]\!]; [\![c_2]\!] & , \text{if } \Delta \neq \emptyset \end{cases}$$

If the do-computation is pure then it can be translated efficiently into a `let`-binding, otherwise, the translation falls back to the default behaviour and preserves the (more expensive) do-binding.

The remaining rules of Karachalias et al. [2020, Section 7.4.3] work in a similar fashion, generating more efficient NoEff code when computations are known to be pure and falling back to default translations for effectful computations.

4.2 Optimisations

Figure 7 presents the rewrite rules that seize the new optimisation opportunities that arise after ExEff is translated into NoEff.

The first pair of rules (Elim-Ret-Co and Elim-Co-Term) again eliminate redundant coercions, while the second pair of rules (Do-Ret and LetVal) (in practice selectively) β-reduce do-bindings and `let`-bindings.[3]

At first glance it might seem that these optimisations have no effect after similar ones have already been performed on the ExEff program. Yet, this is not true for several reasons. Firstly, by coalescing values and computations in NoEff, we can substitute variables for function calls

[3]Observe the substitution $t_2[t_1/x]$ may discard t_1 if x does not appear in t_2. While in ExEff only values can be discarded this way, in NoEff actual (pure) computations may be dropped. This may turn a non-terminating program into a terminating one. Like C, we consider this to be an acceptable optimisation. See https://en.cppreference.com/w/cpp/language/ub#Infinite_loop_without_side-effects.

in NoEff, which is not syntactically valid in ExEff. Secondly, because NoEff types only record whether types are pure or impure, ExEff coercions between distinct ExEff types may become NoEff coercions between indistinguishable NoEff types.

The following two examples illustrate these cases.

Example 4.1. Consider computation $g\ (f\ x)$, where $x : T_1$, $f : T_1 \to T_2\ !\ \emptyset$, and $g : T_2 \to T_3\ !\ \emptyset$. In ExEff, this seemingly innocuous computation must be translated into a do-computation, due to the repeated application, leading to the following ExEff program:

$$\mathsf{do}\ y \leftarrow f\ x; g\ y$$

Unfortunately, even though $(f\ x)$ is pure (its dirt is empty), it is still a computation, so fine-grained call-by-value syntax prevents us from substituting it for y, which is an expression. However, since all dirts involved are empty, the purity-aware translation to NoEff transforms the above computation into the following NoEff term:

$$\mathsf{let}\ y = f\ x\ \mathsf{in}\ g\ y$$

which we can further simplify into just $g\ (f\ x)$, using Rule LetVal.

Example 4.2. Similarly, if f has a pure type ($f : T_1 \to T_2\ !\ \emptyset$) but g does not ($g : T_2 \to T_3\ !\ \Delta$), the repeated application is translated into a slightly different do-computation:

$$\mathsf{do}\ y \leftarrow (f\ x) \triangleright \gamma; g\ y$$

where $\gamma : T_2\ !\ \emptyset \to T_2\ !\ \Delta$. Again, this program cannot be further simplified, but is translated into the following NoEff program:

$$\mathsf{do}\ y \leftarrow (f\ x) \triangleright \mathsf{return}\ \gamma'; g\ y$$

Now the synergy between the rules comes into play, allowing us to first eliminate the cast (via Rule Elim-Ret-Co)

$$\mathsf{do}\ y \leftarrow (f\ x) \triangleright \mathsf{return}\ \gamma'; g\ y \quad \rightsquigarrow \quad \mathsf{do}\ y \leftarrow \mathsf{return}\ (f\ x); g\ y$$

and then inline $(f\ x)$ (via Rule Do-Ret)

$$\mathsf{do}\ y \leftarrow \mathsf{return}\ (f\ x); g\ y \quad \rightsquigarrow \quad g\ (f\ x)$$

5 IMPLEMENTATION

To test the presented ideas in practice, we have implemented an optimising compiler for Eff, a prototype functional programming language with algebraic effects and handlers. The actual source syntax of Eff is based on OCaml's and features only a single syntactic sort of terms, which lumps together values and computations. Desugaring and type inference result in a core syntax, which is very close to ExEff. The implementation supports standard features such as datatype declarations and control structures as well as standard optimisations of these features which we have omitted from ExEff to avoid the clutter.

After optimisations, terms of the core syntax are elaborated into an extended version of NoEff, which straightforwardly translates into OCaml using a monadic encoding. One could consider multiple implementations of the monad [Kiselyov and Sivaramakrishnan 2016], though currently we use free monads as they are the simplest and turn out to be sufficiently performant for our purposes, as optimisations remove most of effects anyway. For example, the following effectful code with multiple nested handlers

```
let test_generator n =
  let rec generate (l, u) =
    if l > u then ()
```

```
    else (
      perform (Yield l);
      generate (l + 1, u)
    )
  in
  (handle
    handle
      generate (perform (Get ()), n)
    with
    | effect (Yield e) k ->
        (perform (Put (perform (Get ()) + e))); k ()
  with
  | x -> fun s -> s
  | effect (Put s') k -> fun s ->  k () s'
  | effect (Get _) k -> fun s -> k s s
 ) 0
```

is successfully transformed into the following (up to renaming) pure code with the store being tracked in the additional argument `x` of `handled_generate`:

```
let test_generator (n : int) =
  let rec handled_generate (l, u) (x : int) =
    if l > u then x
    else handled_generate (l + 1, u) (x + l)
  in
  handled_generate (0, n) 0
```

Representing computation types. Turning back to the monadic encoding, most NoEff types map directly onto their OCaml counterparts. The computation type `Comp` A is mapped to a predefined `comp` type. For a fixed signature, this type could be defined as:

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

where `Return x` represents returned values while `Put (x, k)` and `Get (x, k)` represent operation calls with argument `x` and continuation `k`. Note that the translation erases the dirt Δ from computation types $A \,!\, \Delta$, for lack of a convenient way to represent it in OCaml. The algebraic effect handlers implementation of Multicore OCaml [Dolan et al. 2015] has made a similar choice not to reflect the set of possible operations in the type.

Since Eff enables users to declare their own operations through declarations such as

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

we use a more involved encoding. First, we define an initially empty extensible variant type

```
type ('arg, 'res) effect = ..
```

parametrised by a type of operation argument and a type of value expected by the continuation. Declarations as the one above then get translated in type extensions such as

```
type (_, _) effect += Get : (unit, int) effect
```

while the free monad is defined as

```
type 'a comp =
  | Return : 'a -> 'a comp
  | Call : ('arg, 'res) effect * 'arg * ('res -> 'a comp) -> 'a comp
```

where instead of multiple operation constructors, we have a single one that takes the called operation, its argument and its continuation, with GADTs ensuring that their types agree. Monadic bind used in the translation of do $x \leftarrow t_1; t_2$ is defined recursively as

```
let rec ( >>= ) (c : 'a comp) (f : 'a -> 'b comp) =
  match c with
  | Return x -> f x
  | Call (eff, arg, k) -> Call (eff, arg, fun y -> k y >>= f)
```

Representing handlers. Next, to represent handlers, we first define an auxiliary type capturing all the handler clauses in a record:

```
type ('a, 'b) handler_clauses = {
  return_clause : 'a -> 'b;
  effect_clauses :
    'arg 'res. ('arg, 'res) effect -> 'arg -> ('res -> 'b) -> 'b
}
```

The type is parameterised by the type of incoming and outgoing computation type of the handler. The return clause is a simple function while effect clauses are represented by a function taking an operation, its argument, and handled continuation and performs the corresponding behaviour a handled computation, with GADTs again ensuring that all types match. Finally, we have a function turning the record of handler clauses into a function recursively mapping from one computation type to another:

```
let handler (hc : ('a, 'b) handler_clauses) : 'a comp -> 'b =
  let rec h = function
    | Return x -> hc.return_clause x
    | Call (eff, arg, k) -> hc.effect_clauses eff arg (fun y -> h (k y))
  in
  h
```

For example, the read-only state handler that provides a constant value to memory lookups and aborts on an update, which is written as

```
let read_only_state = handler
  | effect (Get ()) k -> k 42
  | effect (Put _) k -> None
  | x -> Some x
```

gets translated as

```
{
  return_clause = (fun (x : int) -> Return (Some x));
  effect_clauses =
    (fun (type arg res) (eff : (arg, res) effect) :
      (arg -> (res -> _) -> _) ->
      match eff with
      | Get -> fun () k -> k 42
      | Put -> fun _ _ -> Return None
```

```
      | eff' -> fun arg k -> Call (eff', arg, k));
}
```

Note that the translation automatically adds a default case that propagates all unhandled operations outwards. The type annotations on the effect clauses function are required to help OCaml compiler determine the most general type.

Representing coercions. Finally, a coercion between two NoEff types is represented by a function between the corresponding OCaml types. For example, coercions $\gamma_1 \longrightarrow \gamma_2$ and comp γ are represented by

```
let rec coer_comp (coer : 'a -> 'b) : 'a comp -> 'b comp =
  function
  | Return t -> Return (coer t)
  | Call (eff, arg, k) -> Call (eff, arg, fun x -> coer_comp coer (k x))

let coer_arrow (coer1 : 'a2 -> 'a1) (coer2 : 'b1 -> 'b2) :
  ('a1 -> 'a2) -> 'b1 -> 'b2 =
  fun f x -> coer2 (f (coer1 x))
```

A NoEff cast $t \triangleright \gamma$ is then simply function application.

6 EVALUATION

We evaluate the effectiveness of the Eff compiler by comparing the execution speed of the generated code to other OCaml based implementations:

- hand-written pure OCaml code, which serves as the baseline of benchmarks,
- hand-written OCaml code using the algebraic effect support of Multicore OCaml [Dolan et al. 2015],
- code generated by a recently developed capability-passing style compiler of effect handlers [Schuster et al. 2020], where provided.

We have also tested the effectiveness of the OCaml code written using the *Handlers in Action* [Kammar et al. 2013] and *Eff Directly in OCaml* implementations [Kiselyov and Sivaramakrishnan 2016], which are based on OCaml's DelimCC library for delimited control [Kiselyov 2012], but it was significantly slower so we do not include the results. The latter implementation also comes with a Multicore OCaml alternative, whose efficiency is roughly comparable to hand-written Multicore OCaml [Kiselyov and Sivaramakrishnan 2016]. All benchmarks were run on a MacBook Pro with an 2.9 GHz Intel Core i7 processor and 16 GB 2133 MHz LPDDR3 RAM using the Bechamel micro-benchmarking tool running on Multicore OCaml 4.10.0. The benchmarks were also repeated on Intel Xeon W-2145 3.7 GHz processor on Ubuntu 20.04.1 using 5.4.72-microsoft-standard-WSL2. The produced results did not show noticeable deviations so we omit them from further analysis.

We only compare the performance of generated code with the systems targeting the same (OCaml) backend. This enables easier and more meaningful comparison, since the differences in performance of different backends does not adversely affect the overall analysis. Specifically Xie and Leijen [2021] targets C and Javascript backends, which means that comparison with OCaml will inadvertently include the comparison of performance between OCaml and C or Javascript. To make a future comparison between Eff and other systems more available, we have included our compiler in the recently created benchmark suite for systems implementing effect handlers.[4]

[4] https://github.com/effect-handlers/effect-handlers-bench

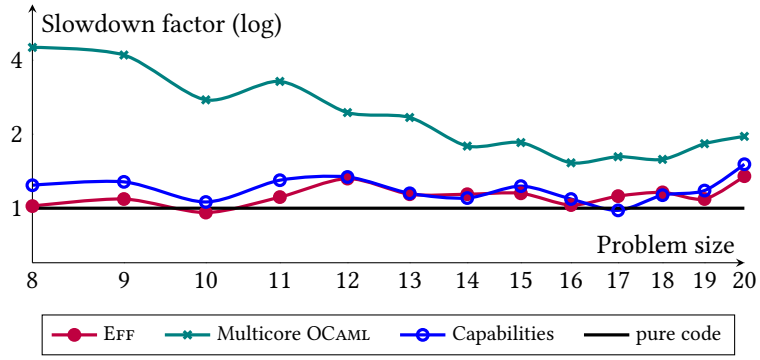

Fig. 8. Slowdown factors (lower is better) of finding a single solution of the n-queens problem in comparison to a pure hand-written code.

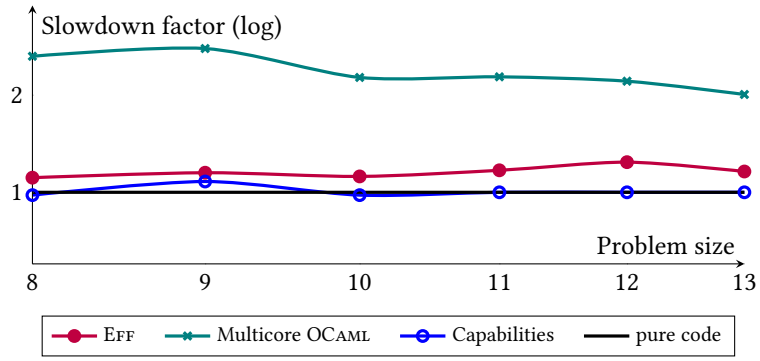

Fig. 9. Slowdown factors (lower is better) of finding a list of all solutions of the n-queens problem in comparison to a pure hand-written code.

Our first set of benchmarks feature the well-known n-queens problem where the underlying program explores the combinatorial space with the effects `Decide : bool` and `Fail : empty` for non-determinism. The first variant tries to find one solution (Figure 8), while the second variant returns the list of all solutions (Figure 9). For the implementations based on handlers, the difference between the two variants amounts to picking a different handler while the hand-written code requires one to rewrite the whole program around the new backtracking strategy. The results clearly show that in both benchmarks, both compiled versions (ours and one by Schuster et al. [2020], where available) are significantly faster than running the handlers in Multicore OCaml and are competitive with hand-written OCaml code.

We repeated the comparison on a number of additional benchmarks, including a few provided by Schuster et al. [2020]. Since the results do not vary too much with the problem size, we present only the largest instance of each in Table 2. Full graphs together with a description of executed programs can be found in Appendix A.

The code generated by our compiler is consistently among the fastest ones, sometimes outperforming even hand-written pure code. The only benchmark in which Multicore OCaml effects are faster is the arithmetic expression interpreter, in which exceptions are triggered rarely. Since effects are expensive only to trigger, but not to install, this outperforms the CPS variant that our compiler produces. A hand-written variant using OCaml exceptions is 1.65× faster (a slowdown

Table 2. Slowdown factors (lower is better) of additional benchmarks in comparison to a pure hand-written code.

	Eff	Multicore OCaml	Capabilities
one solution of n-queens (Figure 8)	**1.35**	1.96	1.51
all solutions of n-queens (Figure 9)	1.16	2.01	**1.00**
stateful counter (Figure 10)	**1.01**	60.90	5.56
list of generator values (Figure 11)	**1.85**	3.14	1.89
stateful sum of generator values (Figure 12)	**1.93**	86.95	5.59
exceptional arithmetic (Figure 13)	1.45	**0.92**	1.48
stateful arithmetic (Figure 14)	1.40	2.81	**1.38**
pure tree traversal (Figure 15)	0.88	4.22	**0.73**
reader tree traversal (Figure 16)	2.21	4.08	**1.08**
stateful tree traversal (Figure 17)	2.49	3.67	**1.05**

factor of 0.57). Generated code for pure tree traversal similarly transforms direct code into CPS and is thus faster than the hand-written code which one would usually write in direct style.

The last two examples of a reader and stateful tree traversal are particularly interesting because they combine two handlers: one for non-determinism and one for state. In this case, where handlers are nested, our compilation scheme does not specialise already specialised functions. The generated code is therefore not fully optimised. It still outperforms Multicore OCaml, but the difference is not as high as in other benchmarks and is outperformed by capability passing style compilation.

Since the two handlers are orthogonal to each other, one may flatten them into a single handler with all the clauses. The code generated from this flattened handler in the last example turns out to be 1.11× faster than hand-written code (a slowdown factor of 0.90), showing the full potential of additional source-to-source optimisations.

We have also tested the effect of individual optimisations on the performance of the generated code. Based on their similarity, optimisations were put into five groups: function specialisation (Section 3.3), coercion eliminations (Figure 7), coercion push rules (Figure 3), handler reductions (Figure 5), and purity-aware compilation to NoEff (Section 4.1.2). Full benchmarks were compiled with each of the optimisation groups independently disabled to analyze direct and indirect effect on performance. Detailed statistics of disabling individual optimisations for each benchmark case are presented in Table 3 where each column corresponds to the slowdown when the optimisation was disabled. Disabled optimisations mostly incur a constant slowdown except with handler-heavy programs, where the slowdowns increase rapidly with increased problem size. The results and manual inspection of generated code show that the handler reduction rules are almost always necessary to unblock the code for function specialisation.

7 RELATED WORK

Leijen [2017] presents a type-directed compilation approach for the Koka language. His setting differs from ours in several ways: Firstly, Koka features row-typing rather than effect subtyping. Secondly, Koka's compiler directly targets a CPS backend, rather than an intermediate language. The only featured optimisation is that of selective CPS: pure computations are translated to direct-style expressions. Unfortunately, no experimental evaluation is provided to establish the significance of this optimisation. More recently, Xie and Leijen [2021] have started developing efficient compilation techniques based on evidence passing. It would be interesting to examine the relationship between

Table 3. Slowdown factors (lower is better) of disabling individual groups of optimisations in comparison to hand-written code.

	w/o func. spec.	w/o coer. elim.	w/o coer. push rules	w/o hand. reduct.	w/o purity-aware compilation	fully optimised
one *n*-queens	1.54	4.04	1.51	1.18	16.57	1.35
all *n*-queens	1.21	3.70	1.31	1.21	13.52	1.16
stateful counter	41.84	2.15	1.05	42.00	33.14	1.01
generator list	7564.59	4.65	2.44	7560.49	7.65	1.85
generator sum	52.66	13.90	101.22	52.54	42.80	1.93
exception arithmetic	2.67	1.50	1.52	2.72	3.34	1.45
stateful arithmetic	6537.50	1.61	1.69	6536.86	4.45	1.40
pure tree	4.10	1.38	0.97	4.12	6.03	0.88
reader tree	4.54	4.35	2.97	4.69	7.00	2.21
stateful tree	6.70	5.67	4.39	6.86	8.10	2.49

their and our approach, especially as we believe that our source-to-source transformations can be easily inserted as an additional stage in the Koka compiler and adapted to draw on the row-typing information.

The Multicore OCaml backend [Dolan et al. 2015] provides supports for algebraic effects in terms of the multicore *fibers* to efficiently represent delimited continuations at runtime. These come both in a cheaper one-shot and more expensive multi-shot form. Several works [Hillerström et al. 2016; Kiselyov and Sivaramakrishnan 2016] have shown that this provides an effective compilation target for algebraic effects. Yet, as far as we know, no existing works performs optimising compilation in this setting.

Kammar et al. [2013] compare the performance of a number of different encodings of effect handlers in Haskell. Inspired by this comparison, Wu and Schrijvers [2015] show how effect handlers can be fused and inlined when programs are represented with the codensity monad. They explain that, with a careful setup based on type classes, the GHC Haskell compiler automatically carries out this optimisation as part of (constrained) polymorphic function specialisation; benchmarks illustrate the effectiveness of this approach. While the result of this approach is similar to our function specialisation, the type-class approach does not readily carry over to other compilers and languages.

The work on capability-passing style Schuster et al. [2020] takes a compilation approach that shares much with the library-based approach of Kammar et al. [2013] and Wu and Schrijvers [2015]. It uses a Church-encoding-like representation of the free monad that is obtained by passing around the handlers to the call sites of operations, rather than the other way around, and using an iterated continuation-passing style transformation to make the delimited continuations explicit. By means of a two-level lambda-calculus they can statically beta-reduce handler applications. A big difference with the library-based approach is that their type system is aware of effects, and, e.g., guarantees static reductions for a subset of programs. Because our approach does not systematically convert programs to continuation-passing style, it tends to allocate fewer function closures; this typically results in better performance.

Kiselyov and others [2015; 2013; 2014] investigate a number of different implementations of the free monad that exhibit good runtime performance and/or algorithmic time complexity. There are several differences between their setting and ours. They consider a library in the lazily evaluated

language Haskell, while we compile to eagerly evaluated OCaml. They support so-called *shallow handlers* and explicit manipulation of the free monad structure in the source language, while we do not. Hence, we did not adopt their dequeue-based implementation of the free monad's bind. Nevertheless we share the same functor construction based on the co-Yoneda lemma and our free monad is specialised in the same way for this construction.

Saleh and Schrijvers [2016] present a term-rewriting-based approach for optimising an embedding of algebraic effects and handlers in Prolog. They obtain good speed-ups, but their setting is simpler and less general. Firstly, Prolog is essentially a procedure-oriented rather than expression-oriented language. Hence, the rewrites related to returning values are not relevant in their setting. Moreover, they do not support higher-order programming and feature only a crude effect system. Also, their handlers may only contain lexically visible calls to the continuation. In addition, they do not perform selective CPS to deal with non-tail recursion. Finally, they do not perform purity-aware code generation but instead require native support of (Prolog-style) delimited control [Schrijvers et al. 2013].

Kammar and Plotkin [2012] develop a theory of optimisations valid for effectful programs that satisfy a certain algebraic theory. For example, if we assume symmetry of non-deterministic choice, we may safely exchange the order in which non-deterministic computations are executed. Bauer and Pretnar [2014] consider a subset of these optimisations for computations in an absolutely free algebra, i.e., with a trivial equational theory, but under particular handlers, and Luksic and Pretnar [2020] generalise their approach to general algebraic theories that may vary through with program terms. Some of the optimisations are already subsumed by our rewriting optimisations, but there is still ample untapped potential for exploiting effect information in specialised optimisations through sophisticated reasoning.

Our function specialisation is closely related to the fixed point promotion of Ohori and Sasano [2007] that specialises the composition of two recursive functions $f \circ g$. While they consider a pure calculus we have an effectful setting where the handler plays the role of the recursive function f. Fixed point promotion requires that f is strict, which is indeed the case for handlers. A difference is that we more aggressively inline the handler than fixed point promotion, which unfolds f only once.

8 CONCLUSION

This paper has presented a two-step approach for the optimised compilation of algebraic effects and handlers. First we perform a number of source-to-source transformations on the purity-aware ExEff core language, including the specialisation of recursive function definitions for particular handlers. Then, after purity-aware elaboration into the effect-agnostic backend language NoEff, we perform further rewrites that are possible when effect information no longer needs to be preserved.

Our experimental evaluation shows that this approach is effective at eliminating handlers from a number of benchmarks and obtaining performance that is competitive with hand-written code. Moreover, the benchmarks demonstrate that our approach mostly outperforms other non-optimising (Multicore OCaml) and optimising (capability-passing style) implementations of algebraic effects and handlers.

In future work we would like to investigate how to best optimise complex patterns like nested handlers and how to extend our work to the polymorphic setting [Karachalias et al. 2020], which will enable us to experiment with meaningful large programs. An especially interesting aspect of polymorphism is translating polymorphic higher-order functions: in general, their arguments can be impure and introduce computation types, but one could specialise some of their applications to use more efficient pure variants. Combining our optimisations with an efficient runtime system like that of Multicore OCaml seems also promising.

Also on our agenda is investigating the compatibility of our optimisations with other type-and-effect systems for algebraic effects and handlers like those based on row-polymorphism [Leijen 2014, 2017] or with implicit effect polymorphism [Lindley et al. 2017].

ACKNOWLEDGMENTS

We are indebted to Amr Hany Saleh, Brecht Serckx and Philipp Schuster for valuable contributions and support. We are grateful to Nicolas Wu, Mauro Jaskelioff, Ruben Pieters, Alexander Vandenbroucke, Klara Mardirosian, and the members of IFIP WG 2.1 as well as the anonymous reviewers of ICFP 2017 and OOPSLA 2021 for their helpful comments. We also appreciate the feedback from the participants of Dagstuhl seminars 16112, 18172, 21292 and Shonan meeting 146. This material is partly based upon work supported by the Air Force Office of Scientific Research under award numbers FA9550-14-1-0096, FA9550-17-1-0326 and FA9550-21-1-0024, and partly funded by the Flemish Fund for Scientific Research (FWO).

A BENCHMARK GRAPHS

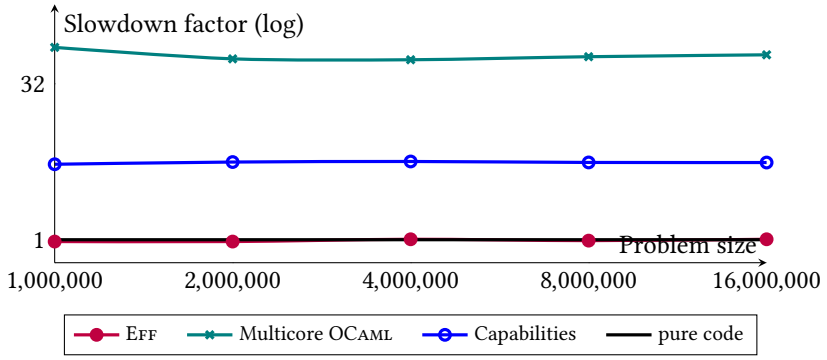

Fig. 10. Slowdown factors (lower is better) of a countdown loop using state for its counter [Schuster et al. 2020] in comparison to a pure hand-written code.

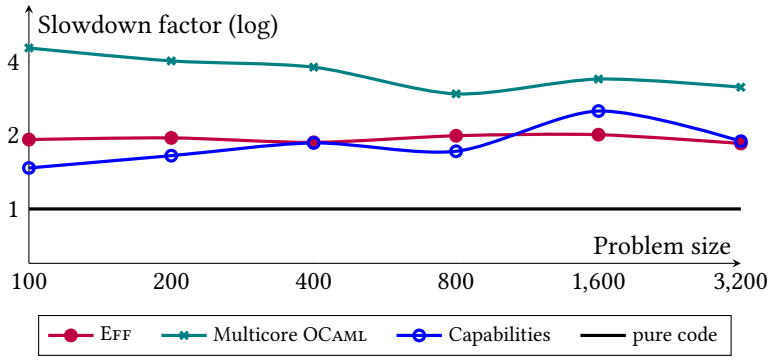

Fig. 11. Slowdown factors (lower is better) of a computation producing a list of numbers yielded from a generator in comparison to a pure hand-written code.

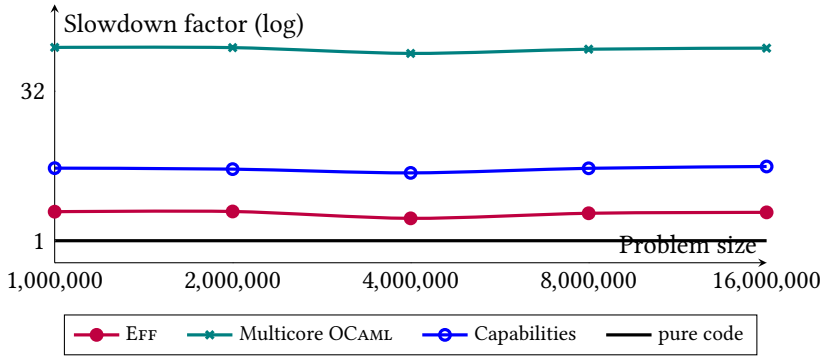

Fig. 12. Slowdown factors (lower is better) of a computation using state to sum numbers yielded from a generator [Schuster et al. 2020] in comparison to a pure hand-written code.

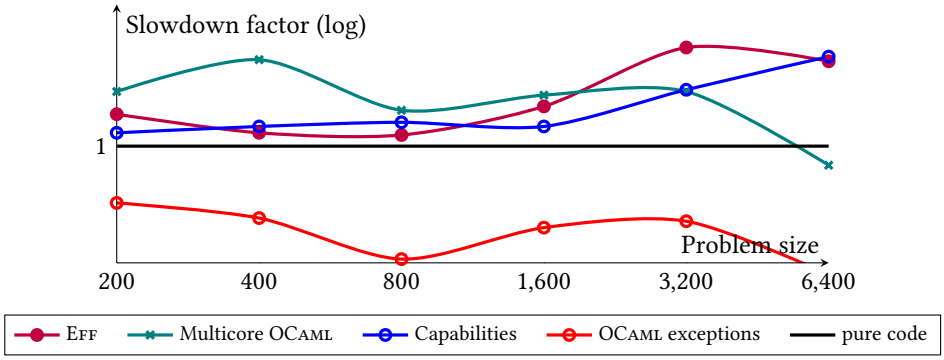

Fig. 13. Slowdown factors (lower is better) of an evaluation of arithmetic expressions potentially raising division-by-zero exceptions in comparison to a pure hand-written code.

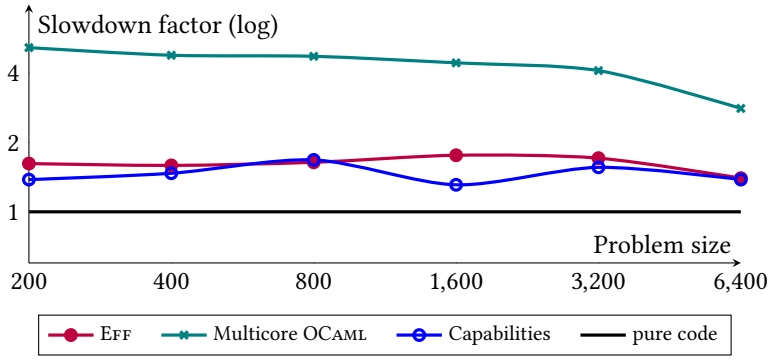

Fig. 14. Slowdown factors (lower is better) of a stateful evaluation of arithmetic expressions whilst modifying state in comparison to a pure hand-written code.

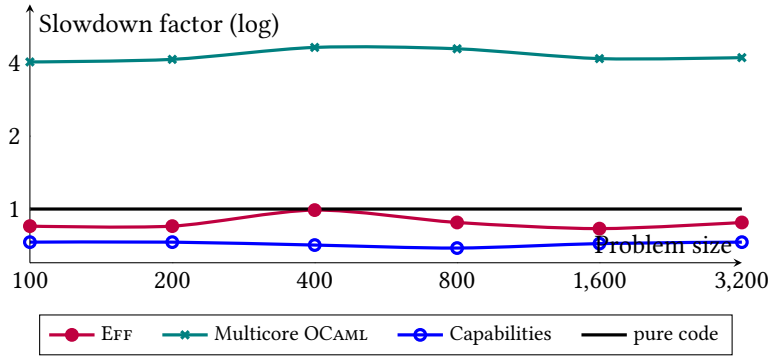

Fig. 15. Slowdown factors (lower is better) of a non-deterministic tree exploration in comparison to a pure hand-written code.

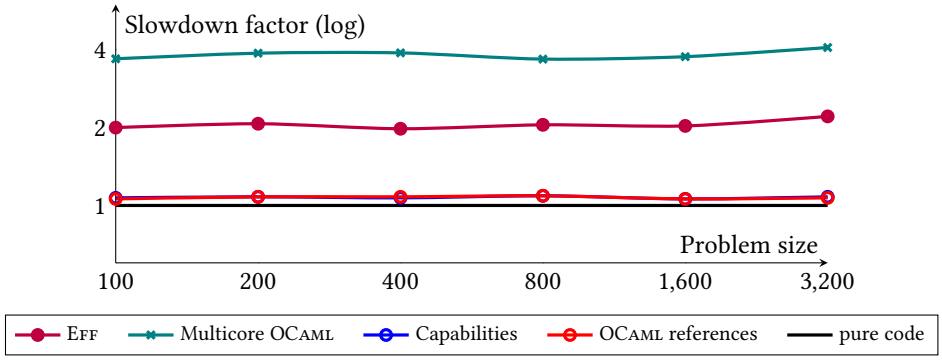

Fig. 16. Slowdown factors (lower is better) of a non-deterministic tree exploration using memory lookups in comparison to a pure hand-written code.

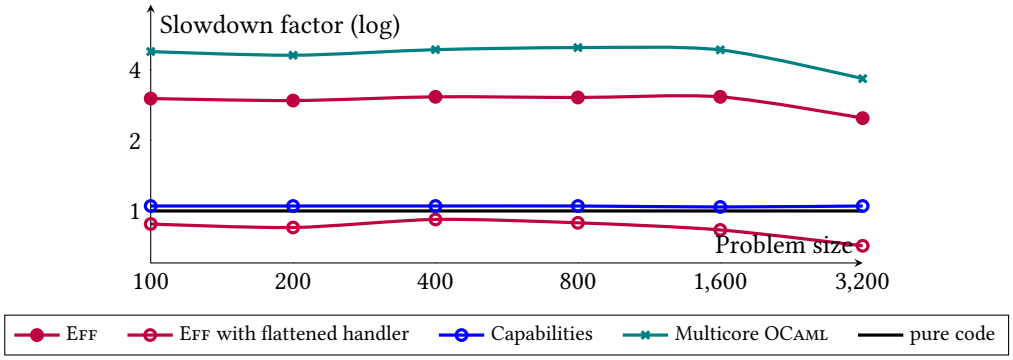

Fig. 17. Slowdown factors (lower is better) of a non-deterministic tree exploration using memory lookups and updates in comparison to a pure hand-written code.

REFERENCES

Andrej Bauer and Matija Pretnar. 2014. An Effect System for Algebraic Effects and Handlers. *Logical Methods in Computer Science* 10, 4 (2014). https://doi.org/10.2168/LMCS-10(4:9)2014

Andrej Bauer and Matija Pretnar. 2015. Programming with Algebraic Effects and Handlers. *Journal of Logic and Algebraic Programming* 84, 1 (2015), 108–123. https://doi.org/10.1016/j.jlamp.2014.02.001

Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, K. C. Sivaramakrishnan, and Leo White. 2017. Concurrent System Programming with Effect Handlers. In *Trends in Functional Programming - 18th International Symposium, TFP 2017, Canterbury, UK, June 19-21, 2017, Revised Selected Papers (Lecture Notes in Computer Science)*, Meng Wang and Scott Owens (Eds.), Vol. 10788. Springer, 98–117. https://doi.org/10.1007/978-3-319-89719-6_6

Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy. 2015. Effective Concurrency through Algebraic Effects. In *OCaml Workshop*.

Daniel Hillerström, Sam Lindley, and Robert Atkey. 2020. Effect handlers via generalised continuations. *J. Funct. Program.* 30 (2020), e5. https://doi.org/10.1017/S0956796820000040

Daniel Hillerström, Sam Lindley, and KC Sivaramakrishnan. 2016. Compiling Links Effect Handlers to the OCaml Backend. In *OCaml Workshop*.

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional programming (ICFP '13)*. ACM, 145–158. https://doi.org/10.1145/2500365.2500590

Ohad Kammar and Gordon D. Plotkin. 2012. Algebraic foundations for effect-dependent optimisations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 349–360. https://doi.org/10.1145/2103621.2103698

Georgios Karachalias, Matija Pretnar, Amr Hany Saleh, Stien Vanderhallen, and Tom Schrijvers. 2020. Explicit effect subtyping. *Journal of Functional Programming* 30 (2020). https://doi.org/10.1017/S0956796820000131

Oleg Kiselyov. 2012. Delimited control in OCaml, abstractly and concretely. *Theorical Computer Science* 435 (2012), 56–76. https://doi.org/10.1016/j.tcs.2012.02.025

Oleg Kiselyov and Hiromi Ishii. 2015. Freer monads, more extensible effects. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, Ben Lippmeier (Ed.). ACM, 94–105. https://doi.org/10.1145/2887747.2804319

Oleg Kiselyov, Amr Sabry, and Cameron Swords. 2013. Extensible effects: an alternative to monad transformers. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013*, Chung-chieh Shan (Ed.). 59–70. https://doi.org/10.1145/2578854.2503791

Oleg Kiselyov and K. C. Sivaramakrishnan. 2016. Eff Directly in OCaml. In *Proceedings ML Family Workshop / OCaml Users and Developers workshops, ML/OCAML 2016, Nara, Japan, September 22-23, 2016 (EPTCS)*, Kenichi Asai and Mark R. Shinwell (Eds.), Vol. 285. 23–58. https://doi.org/10.4204/EPTCS.285.2

Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014. (EPTCS)*, Paul Levy and Neel Krishnaswami (Eds.), Vol. 153. 100–126. https://doi.org/10.4204/EPTCS.153.8

Daan Leijen. 2017. Type directed compilation of row-typed algebraic effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 486–499. https://doi.org/10.1145/3093333.3009872

Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do be do be do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 500–514. https://doi.org/10.1145/3009837.3009897

Ziga Luksic and Matija Pretnar. 2020. Local algebraic effect theories. *J. Funct. Program.* 30 (2020), e13. https://doi.org/10.1017/S0956796819000212

Atsushi Ohori and Isao Sasano. 2007. Lightweight fusion by fixed point promotion. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, Martin Hofmann and Matthias Felleisen (Eds.). ACM, 143–154. https://doi.org/10.1145/1190216.1190241

Gordon D. Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11, 1 (2003), 69–94. https://doi.org/10.1023/A:1023064908962

Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4 (2013). https://doi.org/10.2168/LMCS-9(4:23)2013

Matija Pretnar. 2015. An introduction to algebraic effects and handlers, invited tutorial. *Electronic Notes in Theoretical Computer Science* 319 (2015), 19–35. https://doi.org/10.1016/j.entcs.2015.12.003

Amr Hany Saleh, Georgios Karachalias, Matija Pretnar, and Tom Schrijvers. 2018. Explicit Effect Subtyping. In *ESOP (Lecture Notes in Computer Science)*, Vol. 10801. Springer, 327–354. https://doi.org/10.1007/978-3-319-89884-1_12

Amr Hany Saleh and Tom Schrijvers. 2016. Efficient algebraic effect handlers for Prolog. *Theory and Practice of Logic Programming* 16, 5-6 (2016), 884–898. https://doi.org/10.1017/S147106841600034X

Tom Schrijvers, Bart Demoen, Benoit Desouter, and Jan Wielemaker. 2013. Delimited continuations for Prolog. *Theory and Practice of Logic Programming* 13, 4-5 (2013), 533–546. https://doi.org/10.1017/S1471068413000331

Philipp Schuster, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2020. Compiling Effect Handlers in Capability-Passing Style. *Proc. ACM Program. Lang.* 4, ICFP, Article 93 (Aug. 2020), 28 pages. https://doi.org/10.1145/3408975

Atze van der Ploeg and Oleg Kiselyov. 2014. Reflection without remorse: revealing a hidden sequence to speed up monadic reflection. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, Wouter Swierstra (Ed.). ACM, 133–144. https://doi.org/10.1145/2633357.2633360

Nicolas Wu and Tom Schrijvers. 2015. Fusion for Free - Efficient Algebraic Effect Handlers. In *Mathematics of Program Construction (LNCS)*, Vol. 9129. Springer, 302–322. https://doi.org/10.1007/978-3-319-19797-5_15

Ningning Xie and Daan Leijen. 2021. Generalized evidence passing for effect handlers: efficient compilation of effect handlers to C. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30. https://doi.org/10.1145/3473576