How to Go Eff Without a Hitch: On Efficient Compilation from Eff to OCaml (Extended Abstract)[*]

Stien Vanderhallen[**], Georgios Karachalias, and Tom Schrijvers

KU Leuven, Department of Computer Science, Belgium

Abstract. In the light of algorithmic, as well as performance-related challenges in Eff-compilation, we introduce the implementation of an explicit type-and-effect system into the Eff-compiler, as described by Saleh et al. Thus, the way is cleared for a more efficient compilation to languages with a native effect system. We leverage this implementation to increase performance of compilation to languages without a native effect system as well. Karachalias et al. propose such a translation, which however does not seamlessly fit into the current state of the compiler. We describe the issues faced in integrating that translation, as well as a preliminary implementation.

Keywords: Algebraic effects and handlers · Efficient compilation · Eff · OCaml

Introduction

Given the growing use of algebraic effects and handlers, a need arises for efficient compilation of languages like Eff [1] supporting them. A stable theory [2, 4, 5] is in place for a performance-enhanced translation of Eff to Multicore OCaml, which supports effects as well. Compilation to backend languages having no native effect system, like OCaml, however remains in exploration [2], but without a well-established formalisation that is compatible with the Eff-compiler's current state.

1   Introducing the ExEff Language

1.1   Need for Explicit Effects

The Eff-compiler currently takes ImpEff as its core language, which is implicitly typed and resembles a desugared version of Eff. In practice, this approach yields a buggy and error-prone compilation process, as terms being typed implicitly leads to great intricacy in transforming them. Moreover, with compilation in

---

[*] Category: research.
[**] Stien Vanderhallen, the main author, is a student.

```
effect Decide : bool;;

handle
  let x = (if perform Decide then 10 else 20) in
  x-1
with
| effect Decide k -> k true;;
```

Fig. 1. Eff code snippet

itself presenting as a delicate challenge, optimisations to the process are hard to formulate, or implement adequately [3]. Facing both these algorithmic and performance-related issues, we introduced an explicit type-and-effect system into the Eff-compiler.

1.2   Formalisation and Implementation

The ExEff language, as presented by Saleh et al. [5], provides such an explicit type-and-effect system, and now extends the Eff-compiler's backend as an extra intermediate language. Type inference and elaboration from ImpEff to ExEff are implemented as proposed by Saleh et al. [5], and by extension enable translation from the Eff source language to ExEff through ImpEff. Further compilation from ExEff to Multicore OCaml is supported by introduction of SkelEff as a second intermediary language. In SkelEff, no coercions or explicit effect information are supported, and skeletons replace ExEff-(sub)types. As such, translation from ExEff to SkelEff requires no extra type inference, only erasure of ExEff's subtyping- and effect-information. SkelEff does provide an explicit term-level effect system, resulting in a strong resemblence to (a subset of) Multicore OCaml, which concludes full compilation from source Eff to Multicore OCaml via ExEff.

An illustrative example of this compilation is given by **Fig. 1** and **Fig. 2**. **Fig. 1** shows a simple Eff code snippet making use of effect handling, whose translation to Multicore OCaml as produced by the compiler is given by **Fig. 2**.

The left backend branch in **Fig. 4** illustrates how the intermediary languages introduced so far fit into the Eff-compiler, as well as their interactions. The ExEff-to-ExEff code optimisation depicted there, refers to performance-related transformations as explored by Serckx [6], which aim to reduce redundancy in further compilation, and to improve execution time.

2   Elaboration to Languages Without Native Effects

2.1   Ongoing and Future Research

To relieve the constraint of the Eff-compiler's target language needing an effect system, Karachalias et al. [2] introduce elaboration from ExEff to NoEff, which is a language without native support for effects. **Fig. 3** depicts the syntax of

```
fun _ ->
    (fun comp -> match comp with
                    | c -> c
                    | effect (Decide ()) k ->
                        continue (Obj.clone_continuation k) true)
    (fun _ ->
        (fun x -> x - 1)
          (match (perform Decide ()) with
              | true -> 10
              | false -> 20)) ;;
```

Fig. 2. Multicore OCaml code snippet

Terms

value $t ::= x \mid \texttt{unit} \mid \texttt{fun } x : A \mapsto t \mid t_1 \; t_2 \mid \Lambda\alpha.t \mid t \; A \mid \Lambda(\omega : \pi).t \mid t \; \gamma$
$\mid \quad t \rhd \gamma \mid \texttt{return } t \mid h \mid \texttt{let } x = t_1 \texttt{ in } t_2 \mid \texttt{Op } t_1 \; (y : B.t_2)$
$\mid \quad \texttt{do } x \leftarrow t_1; t_2 \mid \texttt{handle } t_c \texttt{ with } t_h$

handler $h ::= \{\texttt{return } (x : A) \mapsto t_r, [\texttt{Op } x \, k \mapsto t_{\texttt{Op}}]_{\texttt{Op} \in \mathcal{O}}\}$

Types

type $A, B ::= \alpha \mid \texttt{Unit} \mid A \to A \mid A \Rightarrow B \mid \pi \Rightarrow A \mid \texttt{Comp } A \mid \forall\alpha.A$
coercion type $\pi ::= A \leqslant B$

Coercions

$\gamma ::= \omega \mid \langle\texttt{Unit}\rangle \mid \langle\alpha\rangle \mid \gamma_1 \to \gamma_2 \mid \gamma_1 \Rightarrow \gamma_2 \mid \texttt{handToFun } \gamma_1 \; \gamma_2 \mid \texttt{funToHand } \gamma_1 \; \gamma_2$
$\mid \quad \forall\alpha.\gamma \mid \pi \Rightarrow \gamma \mid \texttt{Comp } \gamma \mid \texttt{return } \gamma \mid \texttt{unsafe } \gamma$

Fig. 3. NoEff syntax

NoEff. That elaboration however takes for its input an ExEff language with a syntax and operational semantics different from the one originally described by Saleh et al. [5]. The latter is implemented in the optimizing Eff-compiler, causing the elaboration of ExEff to NoEff to be non-applicable in its current form.

The two variants of ExEff are specifically distinguished by the different coercion forms they support. As the two however both can be elaborated to from the same ImpEff language, they exhibit the same level of expressivity for our purposes. A transformation from the original ExEff to the more recent version (hereafter referred to as inter-ExEff elaboration) thus should be possible, upon which elaboration to NoEff can be applied as already described.

This approach however has proven to be very complex. The interactions between the coercion forms in the original ExEff language and the ones in the ExEff variation used for elaboration to NoEff, introduce a combinatorial explosion of expression forms for which extra elaboration rules are to be formulated in order to retain a deterministic elaboration procedure. A different, more elegant approach thus is preferred, which might include a new definition of the expression forms that are the target of inter-ExEff-elaboration.

More specifically, careful consideration of the coercion forms that can appear in those "terminal" forms is to be done, as those determine which interactions with the original ExEff's coercion forms must actually be considered in elaboration. Further extending this approach of inter-ExEff-elaboration, the introduction of an intermediary level of expressivity in its syntax, outside of its terminal expression forms might reduce the amount of elaboration rules to formulate, as well as their complexity.

2.2   Preliminary Implementation of Eff-Compilation to OCaml

Though limited by the incompleteness of ExEff-to-NoEff elaboration, we still introduced NoEff into the compiler's backend. In order to steer clear of this limitation, we removed all polymorphism in skeletons, types and coercions from both ExEff and NoEff. Ongoing work includes implementation of a (rather straightforward) translation from NoEff to OCaml, which like NoEff has no native effects. As such, full compilation from source Eff to OCaml via ExEff is enabled for Eff-programs without polymorphism.

Fig. 4 in its right backend branch illustrates the introduction of NoEff into the Eff-compiler's backend, as well as the projected position of OCaml as a target language for the Eff-compiler.

References

1. Bauer, A., Pretnar, M.: Programming with algebraic effects and handlers. J. Log. Algebr. Meth. Program. **84**(1), 108–123 (2015)
2. Karachalias, G., Pretnar, M., Saleh, A.H., Schrijvers, T.: Explicit effect subtyping (2019), under consideration for publication in J. Functional Programming
3. Pretnar, M., Saleh, A.H.S., Faes, A., Schrijvers, T.: Efficient compilation of algebraic effects and handlers. Tech. Rep. CW 708, KU Leuven, Informatics Section, Department of Computer Science (2017)
4. Saleh, A.H.: Efficient Algebraic Effect Handlers. Ph.D. thesis, KU Leuven, Informatics Section, Department of Computer Science, Faculty of Engineering Science (2019)
5. Saleh, A.H., Karachalias, G., Pretnar, M., Schrijvers, T.: Explicit effect subtyping. In: 27th European Symposium on Programming (ESOP) (2018)
6. Serckx, B.: Optimalisaties in de Eff Compiler. Master's thesis, KU Leuven, Faculteit Ingenieurswetenschappen (2019)
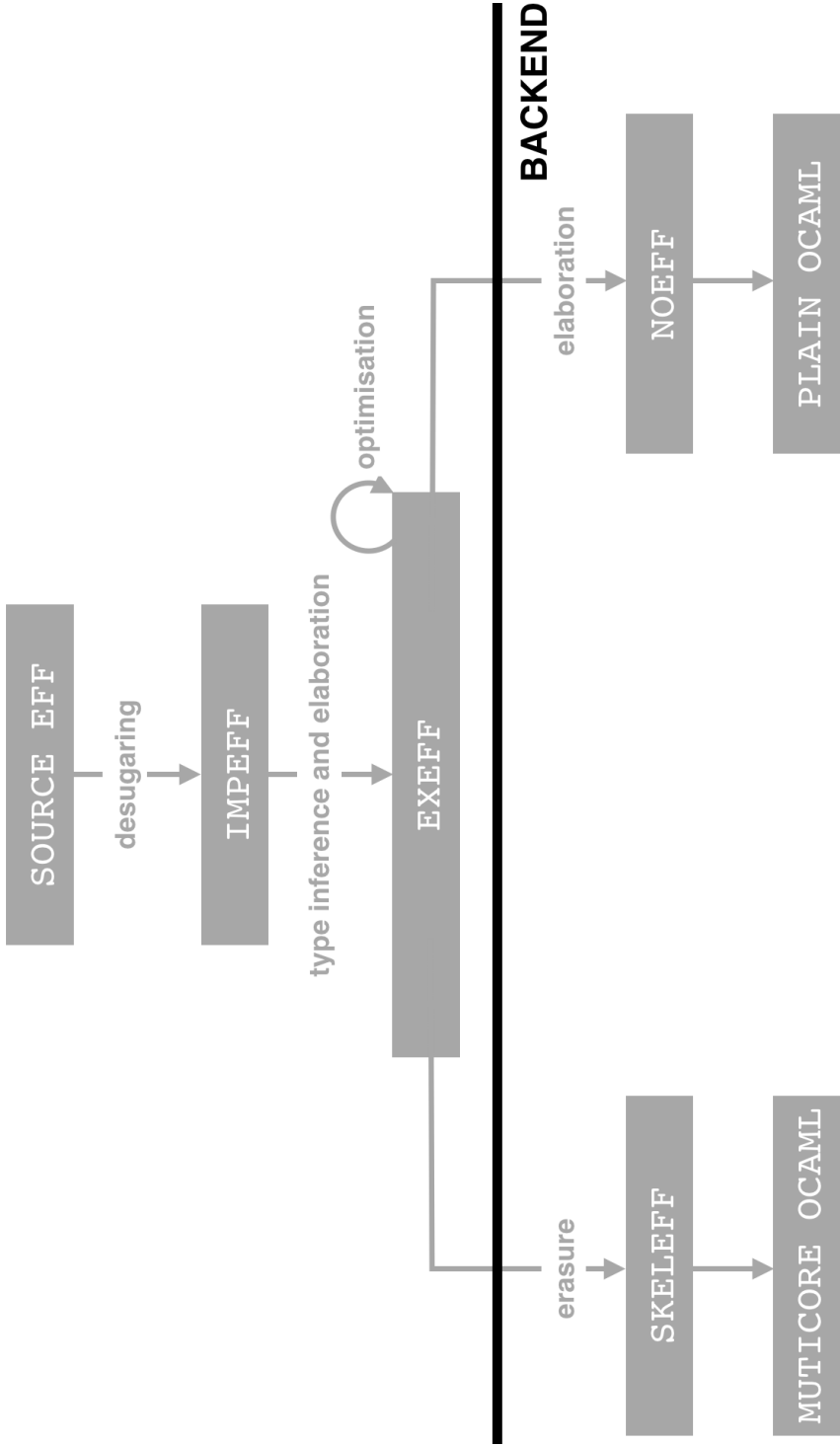
Fig. 4. Schematic overview of intermediary languages in the Eff-compiler. The left backend branch shows compilation to a language with native support for effects, the right branch targets a language without that support.