A core language with row-based effects for optimised compilation

Axel Faes
student : undergraduate
ACM Student Member: 2461936
Department of Computer Science
KU Leuven
axel.faes@student.kuleuven.be

Tom Schrijvers
advisor
Department of Computer Science
KU Leuven
tom.schrijvers@kuleuven.be

Abstract

Algebraic effects and handlers are a very active area of research. An important aspect is the development of an optimising compiler. Eff is an ML-style language with support for effects and forms the testbed for the optimising compiler. However, Eff does not offer explicit typing, which makes it easy for type checking bugs to be introduced during the construction of optimised compilation. This work presents a new core language with row-based effects. The core language is explicitly typed in order to reduce bugs in the optimised compilation.

Keywords algebraic effect handler, row based effect, optimised compilation

1 Introduction

Algebraic effect handling is a very active area of research. Implementations of algebraic effect handlers are becoming available. Because of this, improving performance is becoming the focus of research. A lot of research focusses on speeding up the runtime performance. However, a runtime penalty still occurs. This happens since handlers or continuations need to be repeatedly copied on the heap. Due to this, we are looking towards type-directed optimised compilation of algebraic effect handlers. We want to remove the handlers such that no copying is required and thus no runtime penalty occurs.

In our ongoing research towards type-directed optimised compilation, term rewrite rules and purity aware compilation optimise away most handlers. Term rewrite rules use information of the type-&-effect system. Term rewrite rules perform two types of actions. They remove handlers and apply effects such that eventually the program does not contain any more handlers. Term rewrite rules can also change the syntactic structure in order to expose more possibilities for optimisations. Purity aware compilation identifies computations that are effectively pure and purifies them.

Eff, an ML-style language, is being used to develop an optimised compiler for algebraic effect handlers. Eff uses a type system based on subtyping [1]. As explained by Bauer and Pretnar in [2], terms in Eff do not contain any information about computational effects. This information has to be inferred using type inference algorithms. The lack of explicit type information makes source-to-source transformations much more error-prone. Additionally, ensuring that a transformation does not break typeability becomes a time-consuming task, since we need to reconstruct types after each optimisation pass.

The current type system with subtyping becomes impractical since the typing information is not explicitly contained in each term. There are several solutions to make the type system more practical. It is possible to keep subtyping, but use a unification based algorithm [3]. Implicit effect polymorphism can also be used [7]. The option that is explored in this work, is to use a simple type-&-effect system based on row-polymorphism [4–6].

In this work, we present a simple explicitly-typed language that can serve as an intermediate language during compilation of Eff, and allows for the development of type-preserving core-to-core transformations. Optimisation and term rewriting is done using this core language. This approach will ease the development of an optimised compiler since typechecking becomes linear due to the explicit typing.

Below is an example program:

```
effect Op : unit -> int;;
let rec x () = #Op ();;
let result =
  handle (x ()) with
    | #Op () k -> k 1
```

This program will be optimised into the code below after applying a function specialization optimization. The function 'x' is specialized and the handler is brought inside the function. Implementing this optimization requires indepth knowledge of the typing system that is used. The subtyping-based approach for the type system is difficult to use and does not contain explicit typing information. This makes source-to-source transformations error prone and ensuring transformations do not break typability is time consuming.

```
effect Op : unit -> int;;
let rec x () = #Op ();;
let result =
  let rec x_spec () =
    handle (#Op ()) with
    | #Op () k -> k 1
  in
  x_spec ()
```

2 Background (Eff)

The type-&-effect system that is used in Eff is based on subtyping and dirty types [1].

2.1 Types and terms

Terms Figure 1 shows the two types of terms in Eff. There are values v and computations c. Computations are terms that can contain effects. Effects are denoted as operations Op which can be called.

Types Figure 2 shows the types of Eff. There are two main sorts of types. There are (pure) types A, B and dirty types $\underline{C}, \underline{D}$. A dirty type is a pure type A tagged with a finite set of operations Δ, which we call dirt, that can be called. This finite set Δ is an over-approximation of the operations that are actually called. The type $\underline{C} \Rrightarrow \underline{D}$ is used for handlers because a handler takes an input computation $\underline{C}$, handles the effects in this computation and outputs computation $\underline{D}$ as the result.

value v	::=	x	variable
	|	k	constant
	|	fun $x \mapsto c$	function
	|	{	handler
		return $x \mapsto c_r$,	return case
		$[\mathsf{Op}\, y\, k \mapsto c_{\mathsf{Op}}]_{\mathsf{Op} \in O}$	operation cases
		}	
comp c	::=	$v_1\, v_2$	application
	|	let rec $f\, x = c_1$ in c_2	rec definition
	|	return v	returned val
	|	Op v	operation call
	|	do $x \leftarrow c_1$; c_2	sequencing
	|	handle c with v	handling

Figure 1. Terms of Eff

2.2 Type System

2.2.1 Subtyping

The dirty type $A \,!\, \Delta$ is assigned to a computation returning values of type A and potentially calling operations from the set Δ. This set Δ is always an over-approximation of the actually called operations, and may safely be increased,

(pure) type A, B	::=	bool | int	basic types
	|	$A \to \underline{C}$	function type
	|	$\underline{C} \Rrightarrow \underline{D}$	handler type
dirty type $\underline{C}, \underline{D}$	::=	$A \,!\, \Delta$	
dirt Δ	::=	$\{\mathsf{Op}_1, \ldots, \mathsf{Op}_n\}$	

Figure 2. Types of Eff

Subtyping

Sub-bool
$$\frac{}{\mathsf{bool} \leqslant \mathsf{bool}}$$

Sub-int
$$\frac{}{\mathsf{int} \leqslant \mathsf{int}}$$

Sub-$\to$
$$\frac{A' \leqslant A \qquad \underline{C} \leqslant \underline{C}'}{A \to \underline{C} \leqslant A' \to \underline{C}'}$$

Sub-$\Rrightarrow$
$$\frac{\underline{C}' \leqslant \underline{C} \qquad \underline{D} \leqslant \underline{D}'}{\underline{C} \Rrightarrow \underline{D} \leqslant \underline{C}' \Rrightarrow \underline{D}'}$$

Sub-!
$$\frac{A \leqslant A' \qquad \Delta \subseteq \Delta'}{A \,!\, \Delta \leqslant A' \,!\, \Delta'}$$

Figure 3. Subtyping for pure and dirty types of Eff

inducing a natural subtyping judgement $A \,!\, \Delta \leq A \,!\, \Delta'$ on dirty types. As dirty types can occur inside pure types, we also get a derived subtyping judgement on pure types. Both judgements are defined in Figure 3. Observe that, as usual, subtyping is contravariant in the argument types of functions and handlers, and covariant in their return types.

2.2.2 Typing rules

Figure 4 defines the typing judgements for values and computations with respect to a standard typing context Γ.

Values The rules for subtyping, variables, and functions are entirely standard. For constants we assume a signature Σ that assigns a type A to each constant k, which we write as $(\mathsf{k} : A) \in \Sigma$.

A handler expression has type $A \,!\, \Delta \cup O \Rrightarrow B \,!\, \Delta$ iff all branches (both the operation cases and the return case) have dirty type $B \,!\, \Delta$ and the operation cases cover the set of operations O. Note that the intersection $\Delta \cap O$ is not necessarily empty. The handler deals with the operations O, but in the process may re-issue some of them (i.e., $\Delta \cap O$).

When typing operation cases, the given signature for the operation $(\mathsf{Op} : A_{\mathsf{Op}} \to B_{\mathsf{Op}}) \in \Sigma$ determines the type A_{Op} of the parameter x and the domain B_{Op} of the continuation k. As our handlers are deep, the codomain of k should be the same as the type $B \,!\, \Delta$ of the cases.

Computations With the following exceptions, the typing judgement $\Gamma \vdash c : \underline{C}$ has a straightforward definition. The return construct renders a value v as a pure computation, i.e., with empty dirt. An operation invocation $\mathsf{Op}\, v$ is typed typing contexts $\Gamma ::= \epsilon \mid \Gamma, x : A$

Expressions

$$\textsc{SubVal}\ \frac{\Gamma \vdash v : A \qquad A \leqslant A'}{\Gamma \vdash v : A'} \qquad \textsc{Var}\ \frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{Const}\ \frac{(\mathsf{k} : A) \in \Sigma}{\Gamma \vdash \mathsf{k} : A}$$

$$\textsc{Fun}\ \frac{\Gamma, x : A \vdash c : \underline{C}}{\Gamma \vdash \mathsf{fun}\ x \mapsto c : A \to \underline{C}}$$

$$\textsc{Hand}\ \frac{\Gamma, x : A \vdash c_r : B \,!\, \Delta \qquad \left[\begin{array}{l}(\mathsf{Op} : A_{\mathsf{Op}} \to B_{\mathsf{Op}}) \in \Sigma \\ \Gamma, x : A_{\mathsf{Op}}, k : B_{\mathsf{Op}} \to B \,!\, \Delta \vdash c_{\mathsf{Op}} : B \,!\, \Delta\end{array}\right]_{\mathsf{Op} \in O}}{\begin{array}{c}\Gamma \vdash \{\mathtt{return}\ x \mapsto c_r, [\mathsf{Op}\, y\, k \mapsto c_{\mathsf{Op}}]_{\mathsf{Op} \in O}\} : \\ A \,!\, \Delta \cup O \Rightarrow B \,!\, \Delta\end{array}}$$

Computations

$$\textsc{SubComp}\ \frac{\Gamma \vdash c : \underline{C} \qquad \underline{C} \leqslant \underline{C}'}{\Gamma \vdash c : \underline{C}'} \qquad \textsc{App}\ \frac{\Gamma \vdash v_1 : A \to \underline{C} \qquad \Gamma \vdash v_2 : A}{\Gamma \vdash v_1\, v_2 : \underline{C}}$$

$$\textsc{LetRec}\ \frac{\Gamma, f : A \to \underline{C}, x : A \vdash c_1 : \underline{C} \qquad \Gamma, f : A \to \underline{C} \vdash c_2 : \underline{D}}{\Gamma \vdash \mathtt{let\ rec}\ f\ x = c_1\ \mathtt{in}\ c_2 : \underline{D}}$$

$$\textsc{Ret}\ \frac{\Gamma \vdash v : A}{\Gamma \vdash \mathtt{return}\ v : A \,!\, \emptyset} \qquad \textsc{Op}\ \frac{(\mathsf{Op} : A \to B) \in \Sigma \qquad \Gamma \vdash v : A}{\Gamma \vdash \mathsf{Op}\, v : B \,!\, \{\mathsf{Op}\}}$$

$$\textsc{Do}\ \frac{\Gamma \vdash c_1 : A \,!\, \Delta \qquad \Gamma, x : A \vdash c_2 : B \,!\, \Delta}{\Gamma \vdash \mathsf{do}\ x \leftarrow c_1 \,;\, c_2 : B \,!\, \Delta}$$

$$\textsc{With}\ \frac{\Gamma \vdash v : \underline{C} \Rightarrow \underline{D} \qquad \Gamma \vdash c : \underline{C}}{\Gamma \vdash \mathsf{handle}\ c\ \mathtt{with}\ v : \underline{D}}$$

Figure 4. Typing of Eff

according to the operation's signature, with the operation itself as its only operation. Finally, rule With shows that a handler with type $\underline{C} \Rightarrow \underline{D}$ transforms a computation with type $\underline{C}$ into a computation with type $\underline{D}$.

3 Core language

The core language with row-based effects is based on the explicitly typed language used in Links [4]. Links uses a row polymorphic type-&-effect system . The design of their calculus is partially based on the type system used by Pretnar which makes it a suitable candidate for our core language [8]. The terms of the core language are seen in Figure 5, the types are seen in the Figure 6.

3.1 Types and terms

The proposed type system uses type application and type abstractions in order to attain explicit typing information. The subtyping approach is replaced by polymorphism. Aside from this, the type system remains as close as possible to the source language of Eff in order to maximise compatability.

value v ::=	x	variable
|	k	constant
|	$\lambda(x : A).c$	**function**
|	$\Lambda\alpha.c$	**type abstraction**
|	{	handler
	$\mathtt{return}\ x \mapsto c_r,$	return case
	$[\mathsf{Op}\, y\, k \mapsto c_{\mathsf{Op}}]_{\mathsf{Op} \in O}$	operation cases
	}	
comp c ::=	$v_1\, v_2$	application
|	$v\, A$	**type application**
|	$\mathtt{let\ rec}\ f\ x = c_1\ \mathtt{in}\ c_2$	rec definition
|	$\mathtt{return}\ v$	returned val
|	$\mathsf{Op}\, v$	operation call
|	$\mathsf{do}\ x \leftarrow c_1 \,;\, c_2$	sequencing
|	$\mathsf{handle}\ c\ \mathtt{with}\ v$	handling

Figure 5. Terms of the explicitly typed core language

3.2 Typing rules

There are no surprises in the typing rules either. The typing rules are standard rules. It is important to note the row polymorphism in the *Hand* rule.

4 Elaboration

The elaboration show how the source language can be transformed into the core language. Most rules are straightforward except for the *Do* rule.

(pure) type A, B ::=	$A \to \underline{C}$	function type
|	$\underline{C} \Rightarrow \underline{D}$	handler type
|	α	**type variable**
|	$\forall\alpha.\underline{C}$	**polytype**
dirty type $\underline{C}, \underline{D}$ ::=	$A \,!\, \Delta$	
dirt Δ ::=	{R}	
R ::=	$\mathsf{Op}\,;R$	row
|	δ	row variable
|	.	end of row

Figure 6. Types of the explicitly type core language

typing contexts $\Gamma ::= \epsilon \mid \Gamma, x : A$

Expressions

$$\textsc{Val}\ \frac{}{\Gamma \vdash v : A} \qquad \textsc{Var}\ \frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{Const}\ \frac{(\mathsf{k} : A) \in \Sigma}{\Gamma \vdash \mathsf{k} : A}$$

$$\textsc{Fun}\ \frac{\Gamma, x : A \vdash c : \underline{C}}{\Gamma \vdash \lambda(x : A).c : A \to \underline{C}} \qquad \textsc{Type Abstraction}\ \frac{\Gamma, \alpha \vdash c : \underline{C}}{\Gamma \vdash \Lambda\alpha.c : \forall\alpha.\underline{C}}$$

$$\textsc{Hand}\ \frac{\begin{array}{c} \underline{C} = A \,!\, \{Op_i \,;\, R\} \\ \underline{D} = B \,!\, \{Op_i \,;\, R\} \qquad (\mathsf{Op}_i : A_{\mathsf{Op}} \to B_{\mathsf{Op}}) \in \Sigma \\ h = \{\mathtt{return}\ x \mapsto c_r, [\mathsf{Op}\, y\, k \mapsto c_{\mathsf{Op}}]_{\mathsf{Op} \in O}\} \\ \Gamma, x : A_{\mathsf{Op}} \vdash c_r : \underline{D} \\ \Gamma, y : A_{\mathsf{Op}}, k : B_{\mathsf{Op}} \to \underline{D} \vdash c_{op} : \underline{D} \end{array}}{\Gamma \vdash h : \underline{C} \Rightarrow \underline{D}}$$

Computations

$$\textsc{Comp}\ \frac{}{\Gamma \vdash c : \underline{C}} \qquad \textsc{App}\ \frac{\Gamma \vdash v_1 : A \to \underline{C} \qquad \Gamma \vdash v_2 : A}{\Gamma \vdash v_1\, v_2 : \underline{C}}$$

$$\textsc{Type App}\ \frac{\Gamma \vdash v : \forall\alpha.\underline{C}}{\Gamma \vdash v\, A : \underline{C}[A/\alpha]}$$

$$\textsc{LetRec}\ \frac{\Gamma, f : A \to \underline{C}, x : A \vdash c_1 : \underline{C} \qquad \Gamma, f : A \to \underline{C} \vdash c_2 : \underline{D}}{\Gamma \vdash \mathtt{let\ rec}\ f\, x = c_1\ \mathtt{in}\ c_2 : \underline{D}}$$

$$\textsc{Ret}\ \frac{\Gamma \vdash v : A}{\Gamma \vdash \mathtt{return}\ v : A \,!\, \emptyset}$$

$$\textsc{Op}\ \frac{(\mathsf{Op} : A \to B) \in \Sigma \qquad \Gamma \vdash v : A \qquad \underline{C} : B \,!\, \{\mathsf{Op} \,;\, R\}}{\Gamma \vdash \mathsf{Op}\, v : \underline{C}}$$

$$\textsc{Do}\ \frac{\Gamma \vdash c_1 : A \,!\, \Delta \qquad \Gamma, x : A \vdash c_2 : B \,!\, \Delta}{\Gamma \vdash \mathsf{do}\ x \leftarrow c_1 \,;\, c_2 : B \,!\, \Delta}$$

$$\textsc{With}\ \frac{\Gamma \vdash v : \underline{C} \Rightarrow \underline{D} \qquad \Gamma \vdash c : \underline{C}}{\Gamma \vdash \mathtt{handle}\ c\ \mathtt{with}\ v : \underline{D}}$$

Figure 7. Typing of the explicitly typed language

typing contexts $\Gamma ::= \epsilon \mid \Gamma, x : A$

Expressions

$$\textsc{Val}\ \frac{}{\Gamma \vdash v : A \rightsquigarrow v'} \qquad \textsc{Var}\ \frac{(x : S) \in \Gamma \qquad S = \forall\bar{\alpha}.A}{\Gamma \vdash x : A[\bar{S}/\bar{\alpha}] \rightsquigarrow x\, \bar{S}'}$$

$$\textsc{Const}\ \frac{(\mathsf{k} : A) \in \Sigma}{\Gamma \vdash \mathsf{k} : A \rightsquigarrow \mathsf{k}'}$$

$$\textsc{Fun}\ \frac{\Gamma, x : A \vdash c : \underline{C} \rightsquigarrow c'}{\Gamma \vdash \mathsf{fun}\ x \mapsto c : A \to \underline{C} \rightsquigarrow \lambda(x : A).c' : A \to \underline{C}}$$

$$\textsc{Hand}\ \frac{\begin{array}{c} \underline{C} = A \,!\, \{Op_i \,;\, R\} \\ \underline{D} = B \,!\, \{Op_i \,;\, R\} \qquad (\mathsf{Op}_i : A_{\mathsf{Op}} \to B_{\mathsf{Op}}) \in \Sigma \\ h = \{\mathtt{return}\ x \mapsto c_r, [\mathsf{Op}\, y\, k \mapsto c_{\mathsf{Op}}]_{\mathsf{Op} \in O}\} \\ \rightsquigarrow h' = \{\mathtt{return}\ x \mapsto c'_r, [\mathsf{Op}\, y\, k \mapsto c'_{\mathsf{Op}}]_{\mathsf{Op} \in O}\} \\ \Gamma, x : A_{\mathsf{Op}} \vdash c_r : \underline{D} \rightsquigarrow c'_r : \underline{D} \\ \Gamma, y : A_{\mathsf{Op}}, k : B_{\mathsf{Op}} \to \underline{D} \vdash c_{op} : \underline{D} \rightsquigarrow c'_{op} : \underline{D} \end{array}}{\Gamma \vdash h : \underline{C} \Rightarrow \underline{D} \rightsquigarrow h' : \underline{C} \Rightarrow \underline{D}}$$

Figure 8. Elaboration of source to core language: expressions

5 Conclusion

Algebraic effects and handlers are a very active area of research. An important aspect is the development of an optimising compiler. Without a type-&-effect system with explicit typing, it is easy for type checking bugs to be introduced during the construction of optimised compilation. A core language with row-based effects was introduced. The core language is explicitly typed in order to reduce bugs in the optimised compilation.

Acknowledgments

I would like to thank Amr Hany Saleh for his continuous guidance and help. I would also like to thank Matija Pretnar for his support during my research.

References

[1] 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

[2] Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.* 84, 1 (2015), 108–123. https://doi.org/10.1016/j.jlamp.2014.02.001

[3] Stephen Dolan and Alan Mycroft. 2017. Polymorphism, Subtyping, and Type Inference in MLsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 60–72. https://doi.org/10.1145/3009837.3009882

$$\text{typing contexts } \Gamma \ ::= \ \epsilon \mid \Gamma, x : A$$

Computations

Comp

$$\frac{}{\Gamma \vdash c : \underline{C}' \rightsquigarrow c'}$$

App

$$\frac{\Gamma \vdash v_1 : A \to \underline{C} \rightsquigarrow v_1' \qquad \Gamma \vdash v_2 : A \rightsquigarrow v_2'}{\Gamma \vdash v_1\, v_2 : \underline{C} \rightsquigarrow v_1'\, v_2' : \underline{C}}$$

LetRec

$$\frac{\begin{array}{c}\Gamma, f : A \to \underline{C}, x : A \vdash c_1 : \underline{C} \rightsquigarrow c_1' s \\ \Gamma, f : A \to \underline{C} \vdash c_2 : \underline{D} \rightsquigarrow c_2'\end{array}}{\begin{array}{c}\Gamma \vdash \texttt{let rec } f\, x = c_1 \texttt{ in } c_2 : \underline{D} \\ \rightsquigarrow \texttt{let rec } f\, x = c_1' \texttt{ in } c_2' : \underline{D}\end{array}}$$

Ret

$$\frac{\Gamma \vdash v : A \rightsquigarrow v'}{\Gamma \vdash \texttt{return } v : A \,!\, \emptyset \rightsquigarrow \texttt{return } v' : A \,!\, \emptyset}$$

Op

$$\frac{\begin{array}{c}(\texttt{Op} : A \to B) \in \Sigma \\ \underline{C} = B \,!\, \{\texttt{Op} \,;\, R\} \qquad \Gamma \vdash v : A \rightsquigarrow v'\end{array}}{\Gamma \vdash \texttt{Op}\, v : \underline{C} \rightsquigarrow \texttt{Op}\, v' : \underline{C}}$$

Do

$$\frac{\begin{array}{c}\Gamma \vdash c_1 : \underline{C} \rightsquigarrow c_1' \qquad S = \forall \bar{\alpha}.A \\ \bar{\alpha} = FTV(A) - TV(\Gamma) \qquad \Gamma, x : S \vdash \underline{D} \rightsquigarrow c_2'\end{array}}{\Gamma \vdash \texttt{do } x \leftarrow c_1 \,;\, c_2 : \underline{D} \rightsquigarrow (\lambda(x : A).c_2')(\Lambda\bar{\alpha}.c_1')}$$

With

$$\frac{\Gamma \vdash v : \underline{C} \Rightarrow \underline{D} \rightsquigarrow v' \qquad \Gamma \vdash c : \underline{C} \rightsquigarrow c'}{\Gamma \vdash \texttt{handle } c \texttt{ with } v : \underline{D} \rightsquigarrow \texttt{handle } c' \texttt{ with } v' : \underline{D}}$$

Figure 9. Elaboration of source to core language: computations

[4] Daniel Hillerström and Sam Lindley. 2016. Liberating Effects with Rows and Handlers. In *Proceedings of the 1st International Workshop on Type-Driven Development (TyDe 2016)*. ACM, New York, NY, USA, 15–27. https://doi.org/10.1145/2976022.2976033

[5] Daan Leijen. 2014. Koka: Programming with row polymorphic effect types. *arXiv preprint arXiv:1406.2061* (2014).

[6] 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)*. ACM, New York, NY, USA, 486–499. https://doi.org/10.1145/3009837.3009872

[7] 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)*. ACM, New York, NY, USA, 500–514. https://doi.org/10.1145/3009837.3009897

[8] Matija Pretnar. 2015. An introduction to algebraic effects and handlers. invited tutorial paper. *Electronic Notes in Theoretical Computer Science* 319 (2015), 19–35.