Towards 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.

2 Background

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

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. The type $\underline{C} \Rightarrow \underline{D}$ 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.

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 [10]. The terms of the core language are seen in Figure 3, the types are seen in the Figure 4.

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

Figure 1. Terms of Eff as described in previous work

3 Results and Contributions

Preliminary results show that optimised compilation of Eff reaches the same performance as an implementation without algebraic effect handlers in OCaml (Figure 5). Unfortunately, the development of several optimisations proved to be very error-prone, illustrating the need for an explicitly-typed core language.

The proposed core language with row-based effects makes it easier to develop an optimised compiler due to the explicit typing. Since Eff focusses on ease-of-use and usability, the programmer will not be burdened with providing more type information than minimally required [8, 9]. The combination of these languages gives the best of both worlds.

In this work, we presented an idea of a core language for optimised compilation. Planned in future work is the implementation. We will integrate the presented core language in the optimising compiler for Eff and benchmark its impact. The metatheory of the core language is still under development. We will test the explicitly typed core language in other typing systems. As mentioned in this work, another interesting research direction is the development of an unification based algorithm for the subtyping based type-&-effect system which we will also explore in future work.

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

Figure 2. Types of Eff as described in previous work

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

Figure 3. Terms of the explicitly typed core language

(pure) type A, B	$::=$	$A \to \underline{C}$	function type
	$\mid$	$\underline{C} \Rightarrow \underline{D}$	handler type
	$\mid$	α	**type variable**
	$\mid$	$\forall\alpha.\underline{C}$	**polytype**
dirty type $\underline{C}, \underline{D}$	$::=$	$A \,!\, \Delta$	
dirt Δ	$::=$	$\{\mathsf{Op}_1, \ldots, \mathsf{Op}_n\}$	

Figure 4. Types of the explicitly type core language

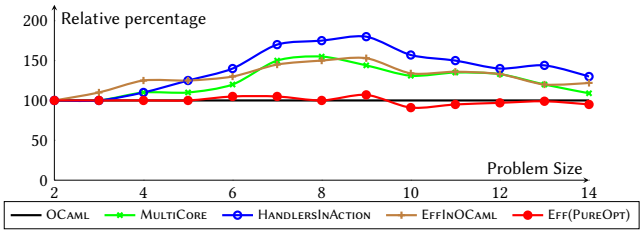

Figure 5. Results of running N-Queens for all solutions on multiple systems

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

[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] 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

[9] Matija Pretnar. 2014. Inferring Algebraic Effects. *Logical Methods in Computer Science* 10, 3 (2014). https://doi.org/10.2168/LMCS-10(3:21)2014

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