Towards Modular Compilation using Higher-order Effects

Anonymous author
Anonymous affiliation

Abstract

Compilers transform a human readable source language into machine readable target language. Nanopass compilers simplify this approach by breaking up this transformation into small steps that are more understandable, maintainable, and extensible. We propose a semantics-driven variant of the nanopass compiler architecture exploring the use a effects and handlers to model the intermediate languages and the transformation passes, respectively. Our approach is fully typed and ensures that all cases in the compiler are covered. Additionally, by using an effect system we abstracts over the control flow of the intermediate language making the compiler even more flexible. We apply this approach to a minimal compiler from a language with arithmetic and let-bound variables to a minimal subset of X86 machine code with variables. In the future, we hope to extend this work to compile a larger and more complicated language and we envision a formal verification framework from compilers written in this style.

2012 ACM Subject Classification Theory of computation → Program semantics; Software and its engineering → Compilers

Keywords and phrases algebraic effects and handlers, higher-order effects, monadic semantics, modularity, compilation, nanopass

Digital Object Identifier 10.4230/OASIcs.CVIT.2016.23

1 Introduction

The essence of a compiler is a function from a source language, suitable for humans, to a machine language, suitable for computers. As our computers have become more powerful we have seen increasingly complex compilers providing extensive safety guarantees and powerful optimizations. To manage this complexity, modern compilers are designed as a composition of multiple passes. However, the total number of passes has traditionally been kept low for the sake of performance, because each pass adds extra overhead. Thus, compiler passes are more complicated than necessary and therefore harder to *understand*, *maintain*, and *extend*.

To address this problem, Sarkar, Waddell, and Dybvig [7] introduce the nanopass compiler architecture. In the nanopass architecture, each pass is designed to be as simple as possible. It is not a problem to use many more passes than are used in traditional compilers. To address concerns about the performance of this architecture, Keep and Dybvig [4] show that it is possible to write a competitive commercial compiler using this architecture.

While the development of the nanopass architecture and the development of a commercial compiler is a great engineering achievement, we believe the theoretical foundation is underexplored. In this paper, we present our ongoing work on developing a semantics-driven nanopass compiler architecture. Our approach has the practical advantage of preventing type errors in the compiler and ensuring all cases are covered. Additionally, our approach abstracts over the control flow giving many of the benefits of continuation-passing-style while retaining a simple monadic interface. During the development of our approach, we are anticipating the possibility of verifying the correctness of compilers written in this style. Concretely, we make the following contributions:

- We introduce a novel approach to designing and implementing practical compilers while staying close to formal denotational semantics specifications (Section 2).

- We demonstrate our approach on a simple language with arithmetic and let-bound variables (Section 2). We compile this language to a subset of X86 with variables. We are still working a method to allocate these variables on the stack.

2 Compiling with Higher-order Effects

In this section, we present our approach by applying it to a very simple language with arithmetic and let-bound variables. The target language of our compiler is X86 machine code. We explain the required concepts as we develop our compiler for this language. The specifications and compiler passes are presented in a simplified notation, but we have implemented all the work we present here in the Agda programming language [2].

We start off by assuming our parser and possibly type checker has finished and produced an abstract syntax tree which follows the grammar described in Figure 1. Our language has integers, addition, subtraction, negation, a read operation to read an input integer, and a let to bind variables and a var to refer to bound variables. The abstract syntax is unusual because we reuse the variable binding facilities from our host language in the form of the $\lambda x.$ binding in the let constructor. This style of abstract syntax is called parametric higher-order abstract syntax (PHOAS) [3]. It allows us to avoid the complexities of variable binding and thus simplify our presentation.

$$\begin{aligned}
\mathit{expr} ::=\ & \mathrm{int}(n) \\
\mid\ & \mathrm{add}(\mathit{expr}, \mathit{expr}) \\
\mid\ & \mathrm{sub}(\mathit{expr}, \mathit{expr}) \\
\mid\ & \mathrm{neg}(\mathit{expr}) \\
\mid\ & \mathrm{read} \\
\mid\ & \mathrm{let}(\mathit{expr}, \lambda x.\, \mathit{expr}) \\
\mid\ & \mathrm{var}(x)
\end{aligned}$$

Figure 1 Abstract syntax of our simple language with arithmetic and let-bound variables.

The first step of our compilation pipeline will be to denote these syntactic constructs onto a set of semantic algebraic operations in the sense of algebraic effects [5]. As is customary when using algebraic effects in functional programming languages, we group these operations under units we call effects. We could group every operation under a single effect, however with the benefit of hindsight we decide to distribute the operations over four effects: Int, Arith, Read, and Let.

Figure 2 shows the operations that correspond directly to our source language. However, there are some particularities we address individually:

- To keep our example simple, we have chosen to use this single type for all values, but we keep the type abstract and simply call it 'val'. It is crucial to keep this type abstract for reasons we explain at the end of this section. Also note that we now need a special *int* operation to inject integers into this abstract value type.
- We write the surrounding monadic context as m, which provides the standard $\gg\!=$, $\gg$, and *return* operations. The monadic context of each operation always includes at least the effect that the operation belongs to, but it can accomodate other effects too. For

$$\begin{array}{ll} \mathit{int} : \mathbb{Z} \to m\ \mathit{val} & \\ & \mathit{read} : m\ \mathit{val} \\ \mathit{add} : \mathit{val} \to \mathit{val} \to m\ \mathit{val} & \\ \mathit{sub} : \mathit{val} \to \mathit{val} \to m\ \mathit{val} & \mathit{let} : m\ \mathit{val} \to (\mathit{val} \to m\ \mathit{val}) \to m\ \mathit{val} \\ \mathit{neg} : \mathit{val} \qquad\quad \to m\ \mathit{val} & \end{array}$$

Figure 2 The operations of our source language with their signatures. Divided into the following effects from left to right and then top to bottom: Int, Read, Arith, and Let.

example, the monadic context of *int* 1 $\gg\!= \lambda x.\, \mathit{add}\ x\ x$ contains at least the Int and Arith effects.

- Readers knowledgeable about effect systems might notice that the *let* operation has arguments that are themselves monadic computations. In standard algebraic effects and handlers this is not allowed, however our approach uses a novel higher-order effect formalism that does support such effectful subcomputations [1].

The first pass of our compiler pipeline is to map our abstract syntax from Figure 1 onto the operations we have defined for our source language from Figure 2. This mapping, called a denotation and written using the $[\![\cdot]\!]$ notation, is a recursive traversal of the abstract syntax tree shown in Figure 3. The result of this mapping is a monadic computation involving the Int, Arith, Read, and Let effects.

$$\begin{aligned} [\![\mathrm{int}(n)]\!] &= \mathit{int}\ n \\ [\![\mathrm{add}(e_1, e_2)]\!] &= [\![e_1]\!] \gg\!= \lambda x.\, [\![e_2]\!] \gg\!= \lambda y.\, \mathit{add}\ x\ y \\ [\![\mathrm{sub}(e_1, e_2)]\!] &= [\![e_1]\!] \gg\!= \lambda x.\, [\![e_2]\!] \gg\!= \lambda y.\, \mathit{sub}\ x\ y \\ [\![\mathrm{neg}(e)]\!] &= [\![e]\!] \gg\!= \lambda x.\, \mathit{neg}\ x \\ [\![\mathrm{read}]\!] &= \mathit{read} \\ [\![\mathrm{let}(e, f)]\!] &= \mathit{let}\ [\![e]\!]\ (\lambda x.\, [\![f\ \ x]\!]) \\ [\![\mathrm{var}(x)]\!] &= \mathit{return}\ x \end{aligned}$$

Figure 3 Denotational mapping from our abstract syntax onto our initial set of effectful operations.

Now that we have denoted our syntactic elements into our semantic domain as operations, we can start refining these operations to get closer to the desired target language which is X86 in our case. In the practice of algebraic effects, this refinement is facillitated by handlers as introduced by Plotkin and Pretnar [6]. These handlers give us access to the operations that occur in the program and the continuation of the program which we will simply call k. We only have to provide the rules that map the operation and continuation onto our semantic domain consisting of primitives, existing operations, or newly introduced operations.

The effect that we choose to handle first is the Let effect, which only has the *let* operation. We handle this operation by running the right hand side of the binding, passing the resulting value to the body, and finally passing the result of that to the continuation. In code that looks as follows:

$$\text{handle } (\mathit{let}\ e\ f)\ k = e \ggg\!= \lambda x.\, f\ x \ggg\!= \lambda z.\, k\ z$$

Note that this defines a strict semantics for our let bindings. By using a different handler we could give different semantics to our language. This is an example of the flexiblility of algebraic effects and handlers.

$$\begin{aligned}
\mathit{addq} &: \mathit{val} \to \mathit{val} \to m\ () \\
\mathit{subq} &: \mathit{val} \to \mathit{val} \to m\ () \\
\mathit{negq} &: \mathit{val} \qquad\quad\ \to m\ () \\
\mathit{movq} &: \mathit{val} \to \mathit{val} \to m\ () \\
\mathit{callq} &: \mathit{lab} \qquad\quad\ \to m\ () \\
\mathit{reg} &: \text{Register} \quad \to m\ \mathit{val}
\end{aligned}
\qquad\qquad
\mathit{x86var} : m\ \mathit{val}$$

Figure 4 X86 related effects. On the left X86. On the right X86Var.

At this point, since our language is so simple we can already begin translating into our target language. In Figure 4 we show the a minimal subset of X86 that we need to compile our Arith and Read effects. This subset contains in-place arithmetic instructions, the ubiquitous move instruction, the call instruction, and an operation to inject concrete registers into our abstract value type. Additionally, we add an operation to generate fresh variables and inject them into our abstract value type.

We can trasnlate our Arith effect operations into X86 operations by creating a fresh X86 variable, populating it, and then applying the in-place arithmetic operation to the variable. So, we write handler as follows:

$$\begin{aligned}
&\text{handle } (\mathit{add}\ x\ y)\ k = \mathit{x86var} \ggg\!= \lambda z.\, \mathit{movq}\ x\ z \gg \mathit{addq}\ y\ z \gg k\ z \\
&\text{handle } (\mathit{sub}\ x\ y)\ \ k = \mathit{x86var} \ggg\!= \lambda z.\, \mathit{movq}\ x\ z \gg \mathit{subq}\ y\ z \gg k\ z \\
&\text{handle } (\mathit{add}\ x) \quad\ k = \mathit{x86var} \ggg\!= \lambda z.\, \mathit{movq}\ x\ z \gg \mathit{negq}\ z \gg k\ z
\end{aligned}$$

The *read* operation requires us to call a function that we will assume is defined in a standard library called read_int. This function places its output in the %rax register, so we have to move it to avoid it being overwritten by other parts of our program. The full definition of our handler for the Read effect is as follows:

$$\text{handle } \mathit{read}\ k = \mathit{x86var} \ggg\!= \lambda z.\, \mathit{callq}\ \text{read\_int} \gg \mathit{reg}\ \%\text{rax} \ggg\!= \lambda x.\, \mathit{movq}\ x\ z \gg k\ z$$

The final challenge to complete this minimal compiler is to allocate the X86 variables on the stack. Conceptually, this requires us to give each *x86var* operation and give each its own location of the stack. Keeping track of such information in our handler, however, is something we have not yet needed to do for the passes up to this point. The main problem is that we need to pass a valid value as input to the continuation of each operation to get access to the operations in the contination. When the type of the argument of the contination is a unit type, such as for the X86 arithmetic, move, and call instructions, then we can simply construct the unit value and pass that to the contination. However, if we allow arbitrary concrete types as results of our operations then there is no guarantee that we can construct a value of that type and it is not always possible to reconstruct the higher-order representation if we do manage to construct such a value.

Luckily, we have chosen to keep our value type abstract. So, we still have a choice to instantiate it to a type that suits our purpose. Instantiating it to the unit type would make it possible for us to construct a value to pass to the continuation. However, we would no longer be able to distinguish values passed to different continuations, so it would not be possible to reconstruct a higher order representation for further manipulation. Instead, our solution is to instatiate it to the type of natural numbers and to pass a unique number to each continuation. Whenever we encounter such a natural number in the rest of the program, we know which continuation it originated from. Hence, we are able to reconstruct a higher order operation.

This process sounds complicated, but we expect it is possible to expose the ability to keep track of the required information through an easy to use API.

3 Related Work

The origins of this work can be traced back to Eelco Visser's work on the Spoofax Language Workbench [9]. As part of Spoofax, Eelco is one of the designers of the Stratego [8] program transformation language. While Stratego can be used for developing compilers, Eelco was still looking for a way of specifying compilers that also abstracts over the control flow. We hope that this work can be the start of an answer to that research direction.

As mentioned in the introduction, our approach embraces the nanopass architecture [7, 4]. We improve upon this work by putting it on more formal foundations, making it fully typed to prevent common errors and even check that all cases are covered, and abstracting over the control flow in the compiler.

Our semantics-driven approach using an effect system is inspired by the work on symbolic execution by Wei et al. [10].

For our vision on verification of compilers we have taken inspiration from Interaction Trees [11]. Interaction Trees use algebraic laws of effects to prove the correctness of compilers. We hope to learn from that technique to prove the correctness of compilers written using our approach.

4 Conclusions and Future Work

We have presented a new semantics-driven approach to writing compilers by using effect operations as an intermediate representation. We use effect handlers to iteratively refine operations in terms of increasingly lower level operations to finally reach a target machine language.

We have shown a concrete example of this approach applied to a very simple language with arithmetic and let-bound variables. We show the implementation of a denotation function and handlers which compile this language is compiled in several passes to X86 machine language with variables. Currently, we are working on a stack allocation pass to complete this minimal compiler.

In the future, we would like to complete the minimal compiler and extend it with more complicated language constructs such as conditionals and anonymous functions. Furthermore, we would like to explore the verification of our compilers using algebraic laws for our effect operations.

References

1 Casper Bach Poulsen and Cas van der Rest. Hefty algebras – modular elaboration for higher-order algebraic operations. Under submission, 2022.

2 Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda – a functional language with dependent types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 73–78, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

3 Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. *SIGPLAN Not.*, 43(9):143–156, sep 2008. URL: `https://doi-org.tudelft.idm.oclc.org/10.1145/1411203.1411226`, `doi:10.1145/1411203.1411226`.

4 Andrew W. Keep and R. Kent Dybvig. A nanopass framework for commercial compiler development. *SIGPLAN Not.*, 48(9):343–350, sep 2013. URL: `https://doi-org.tudelft.idm.oclc.org/10.1145/2544174.2500618`, `doi:10.1145/2544174.2500618`.

5 Gordon Plotkin and John Power. Adequacy for algebraic effects. In Furio Honsell and Marino Miculan, editors, *Foundations of Software Science and Computation Structures*, pages 1–24, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

6 Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In Giuseppe Castagna, editor, *Programming Languages and Systems*, pages 80–94, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

7 Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. A nanopass infrastructure for compiler education. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, ICFP '04, page 201–212, New York, NY, USA, 2004. Association for Computing Machinery. URL: `https://doi-org.tudelft.idm.oclc.org/10.1145/1016850.1016878`, `doi:10.1145/1016850.1016878`.

8 Eelco Visser. Stratego: A language for program transformation based on rewriting strategies system description of stratego 0.5. In Aart Middeldorp, editor, *Rewriting Techniques and Applications*, pages 357–361, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

9 Guido H. Wachsmuth, Gabriël D.P. Konat, and Eelco Visser. Language design with the spoofax language workbench. *IEEE Software*, 31(5):35–43, 2014. `doi:10.1109/MS.2014.100`.

10 Guannan Wei, Oliver Bračevac, Shangyin Tan, and Tiark Rompf. Compiling symbolic execution with staging and algebraic effects. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020. URL: `https://doi-org.tudelft.idm.oclc.org/10.1145/3428232`, `doi:10.1145/3428232`.

11 Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: Representing recursive and impure programs in coq. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019. URL: `https://doi-org.tudelft.idm.oclc.org/10.1145/3371119`, `doi:10.1145/3371119`.