Towards Modular Compilation using Higher-order Effects

Jaro S. Reinders
Delft University of Technology, Netherlands

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 abstract 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 string of pretty printed X86 instructions. 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.EVCS.2023.19

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 et al. introduce the nanopass compiler architecture [11]. In the nanopass architecture, each pass is designed to be as small as possible and has only a single purpose. To make development of nanopass compilers easier Sarkar et al. proposed a methodology where each pass only has to specify transformation rules for those language elements which they actually modify. Additionally, intermediate representations of nanopass compilers can be specified by listing language elements which are removed or added to an existing intermediate representation. To address concerns about the performance of this architecture, Keep and Dybvig have developed a competitive commercial compiler using this architecture [6].

In this paper, we present our ongoing work on developing an improved nanopass architecture. As our foundation we use a higher-order effect system [1] which is a state of the art technique for modeling the semantics of programming languages with side effects. Our approach has the practical advantage of preventing type errors in the compiler and ensuring all cases are covered, which means that the passes defined using our architecture are guaranteed to be syntactically correct by construction. Additionally, our approach abstracts

over the control flow giving many of the benefits of continuation-passing-style while retaining a simple monadic interface.
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 3).
- We demonstrate our approach on a simple language with arithmetic and let-bound variables by compiling it to a subset of X86 (Section 3).

2 Monads and Effects

The compiler architecture we propose is based on the concept of monads, algebraic effects and handlers, and higher-order effect trees and their algebras. In this section, we briefly introduce this required background. This section contains no novel work except for the explanations themselves.

2.1 Monads

Monads are a concept from category theory that have been applied to the study of programming language semantics by Moggi [7] and introduced to functional programming by Wadler [17]. A monad as used in this paper consists of four parts:

- a type m
- a binary operator $\gg\!= : m\ a \to (a \to m\ b) \to m\ b$ (known as "bind")
- a binary operator $\gg : m\ a \to m\ b \to m\ b$
- a function $return : a \to m\ a$

The operator $x \gg y$ is always defined to be equal to $x \gg\!= \lambda z.\, y$ where z is free in y. Furthermore, monads must satisfy three laws:

- left identity: $return\ x \gg\!= \lambda y.\, k\ y = k\ x$
- right identity: $x \gg\!= \lambda y.\, return\ y = x$
- associativity: $x \gg\!= \lambda y.\, (k\ y \gg\!= \lambda z.\, h\ z) = (x \gg\!= \lambda y.\, k\ y) \gg\!= \lambda z.\, h\ z$

We can use monads is to define computations with side effects. An example is the *Maybe* monad which defines partial computations. The type consists of two constructors: $Just : a \to Maybe\ a$ and $Nothing : Maybe\ a$. The bind operator sequences partial computations such that if one subcomputation fails then the whole computation fails. That behavior can be defined using the following two equations: $Nothing \gg\!= \lambda x.\, k\ x = Nothing$ and $Just\ x \gg\!= \lambda y.\, k\ y = k\ x$. The return function is the *Just* constructor. From these definitions it is straightforward to prove that the laws hold, so we will not show the proofs in this paper.

Using the maybe monad we can define a checked division function assuming we have access to an unchecked division function:

$$checkedDiv\ x\ y = \begin{cases} Just\ (div\ x\ y) & \text{if } y \neq 0 \\ Nothing & \text{if } y = 0 \end{cases}$$

Using this checked division function we can define another function that divides a number c by all the numbers contained in a list and return the result in a list:

$$\begin{aligned} divAll\ c\ Nil &= return\ Nil \\ divAll\ c\ (x :: xs) &= checkedDiv\ c\ x \gg\!= \lambda x'.\, divAll\ c\ xs \gg\!= \lambda xs'.\, return\ (x' :: xs') \end{aligned}$$

In this case, the monad has allowed us to implicitly get the control flow behavior that the whole computation will abort if there is at least one zero in the list.

2.2 Effects

One problem with using monads for specifying the semantics of side effects of programming languages is that you need to define a whole new monad for every programming language, even if many programming languages have side effects in common.

Plotkin and Power addressed this problem by introducing algebraic effects [9], focusing on the effectful operations themselves rather than the concrete monad type. Any set of effectful operations gives rise to a monad for free. In this way, the same operations can be reused in two different modular monads.

In this paper we group effectful operations into units which we call 'effects'. For example we can define the interface of an effect called 'Abort' as follows:

effect Abort **where**
  $abort : m\ a$

This definition shows that the Abort effect has one operation called 'abort' which takes no arguments. The type of abort is polymorphic in both the effectful context m and the return type a. In this paper, we assume the obvious implicit constraint that the defining effect, in this case Abort, must be part of the effectful context m.

Any combination of effects gives rise to a monad. So, we can start using the Abort effect to define the *checkedDiv* function instead of using the concrete *Maybe* monad as follows:

$$checkedDiv\ x\ y = \begin{cases} return\ (div\ x\ y) & \text{if } y \neq 0 \\ abort & \text{if } y = 0 \end{cases}$$

The implementation of the *divAll* function does not need to be changed.

Now, let us consider the meaning of the Abort effect. The fact that the *abort* operation claims to produce any polymorphic a as result might seem strange. A normal interface or type class in most programming languages would not be able to implement such a function, because it is impossible to produce a value if the type of that value is not fixed. The reason that we are able to define our effectful operation like this is because one possible side effect is to stop the rest of the computation. That way we do not actually have to produce such a value at all.

In general, to define the meaning of effects we use handlers as introduced by Plotkin and Pretnar [10]. Effect handlers can be thought of as exception handlers with the ability to resume the computation at the location where an effect operation which is handled was used. For the Abort effect, the usual implementation is defined by the following handler:

handle
  $abort\ k \rightarrow Nothing$
  $return\ x \rightarrow Just\ x$

This example shows that the handler of an effect lists each operation and how it should be handled. In addition to the operations themselves, the handler has access to the continuation k from the point where the operation was used. Furthermore, the Abort handler has a *return* case for when the computation contains no usage of the *abort* operation. For most handlers in this paper, the *return* case is just $return\ x \rightarrow return\ x$ in which case we simply leave it out.

Finally, in this paper we use a higher-order effect system introduced by Bach Poulsen and Van der Rest [1]. Higher-order effect operations are those operations that take effectful

computations as arguments. That can be useful for scoping operations, such as exception catching, but also thunks in lazy programming languages. Bach Poulsen and Van der Rest show that it is possible to use their effect system to define interpreters for such higher-order effects. So, it is more than expressive enough for the minimal compiler we present in this paper and even we expect it gives us room to expand our compiler in the future.

3 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]. Our code can be found on GitHub[1].

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 $v \to expr$ function 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. We believe other name binding approaches, such as De Bruijn indices [4], could be used instead. However, changing the name binding mechanism would also require changing our representation of effectful computations.

$$
\begin{aligned}
expr ::= \ & \mathrm{int}(n) \\
| \ & \mathrm{add}(expr, expr) \\
| \ & \mathrm{sub}(expr, expr) \\
| \ & \mathrm{neg}(expr) \\
| \ & \mathrm{read} \\
| \ & \mathrm{let}(expr, v \to expr) \\
| \ & \mathrm{var}(v)
\end{aligned}
$$

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 an effectful computation. We have chosen to divide our source language into four effects: Int, Arith, Read, and Let.

Figure 2 shows the operations that correspond directly to our source language.

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'. Also note that we now need a special *int* operation to inject integers into this abstract value type.

[1] `https://github.com/heft-lang/hefty-compilation`

effect Int **where**
$int : \mathbb{Z} \to m\ val$

effect Read **where**
$read : m\ val$

effect Arith **where**
$add : val \to val \to m\ val$
$sub : val \to val \to m\ val$
$neg : val \to m\ val$

effect Let **where**
$let : m\ val \to (val \to m\ val) \to m\ val$

Figure 2 The effects of our source language and their operations with type signatures.

The reason for keeping the value type abstract is twofold. Firstly, this prevents us from accidentally attempting to use information from these run-time values at compile-time. Secondly, at the end of the compilation, these values stand for register or memory locations. Keeping the values abstract gives us the freedom to choose the concrete representation later on in the compilation pipeline. We use this freedom in the last handler of this section where we convert the program to a string. There we choose the values to be strings.

The first pass of our compiler pipeline maps 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.

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

Figure 3 A 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. 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:

handle $(let\ e\ f)\ k \to e \gg\!= \lambda x.\, f\ x \gg\!= \lambda z.\, k\ z$

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 flexibility of algebraic effects and handlers.

At this point, since our language is so simple we can already begin translating into our target language. In Figure 4 we show a minimal subset of X86 that we need to compile our

effect X86 **where**
$addq : val \to val \to m\ ()$
$subq : val \to val \to m\ ()$
$negq : val \to m\ ()$
$movq : val \to val \to m\ ()$
$callq : lab \to m\ ()$
$reg : \text{Register} \to m\ val$
$deref : \text{Register} \to \mathbb{Z} \to m\ val$

effect X86Var **where**
$x86var : m\ val$

Figure 4 The effects related to X86 and their operations with type signatures.

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 translate 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:

handle
$(add\ x\ y)\ k \to x86var \gg\!= \lambda z.\ movq\ x\ z \gg addq\ y\ z \gg k\ z$
$(sub\ x\ y)\ k \to x86var \gg\!= \lambda z.\ movq\ x\ z \gg subq\ y\ z \gg k\ z$
$(neg\ x)\ \ k \to x86var \gg\!= \lambda z.\ movq\ x\ z \gg negq\ z \gg k\ z$

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:

handle $read\ k \to x86var \gg\!= \lambda z.\ callq$ read_int $\gg reg$ %rax $\gg\!= \lambda x.\ movq\ x\ z \gg k\ z$

The final challenge to complete this minimal compiler pipeline 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. Until now, we have handled each effect by translating into other effects directly. Instead, we can parameterize our handlers which means we pass along an extra parameter while handling our operations. Parameterized handlers take one extra parameter and need to pass one extra argument to the continuation[2]. Now we can write the parameterized handler for the X86Var effect which assigns each variable to its own stack location as follows:

handle $x86var\ k\ n \to deref$ %rbp $(-8 \cdot n) \gg\!= \lambda z.\ k\ z\ (n+1)$

Note that we assume sufficient space is allocated on the stack. Additionally, when applying this handler we need to provide the starting value of the parameter n, which we will choose to be 1.

[2] We ignore effectful subcomputations, because they were already removed in an earlier pass.

At this point, we have a full compiler pipeline from our source language to a subset of X86, but is still in the form of an effectful computation. To get a concrete representation, we implement two handlers for the remaining Int and X86 effects to produce an output string. As part of choosing this concrete representation, we also choose the concrete type for the variables *val* and *lab* to be the string type. We define the handler that turns our effectful computation of Int and X86 effects into a concrete string representation as follows:

handle

$(int\ n)$ $k \to k\ (\text{showInt}\ n)$

$(addq\ x\ y)$ $k \to \texttt{"addq "} +\!\!+ x +\!\!+ \texttt{", "} +\!\!+ y +\!\!+ \texttt{"\textbackslash n"} +\!\!+ k\ ()$

$(subq\ x\ y)$ $k \to \texttt{"subq "} +\!\!+ x +\!\!+ \texttt{", "} +\!\!+ y +\!\!+ \texttt{"\textbackslash n"} +\!\!+ k\ ()$

$(negq\ x)$ $k \to \texttt{"negq "} +\!\!+ x +\!\!+ \texttt{"\textbackslash n"} +\!\!+ k\ ()$

$(movq\ x\ y)$ $k \to \texttt{"movq "} +\!\!+ x +\!\!+ \texttt{", "} +\!\!+ y +\!\!+ \texttt{"\textbackslash n"} +\!\!+ k\ ()$

$(callq\ l)$ $k \to \texttt{"callq "} +\!\!+ l +\!\!+ \texttt{"\textbackslash n"} +\!\!+ k\ ()$

$(reg\ r)$ $k \to k\ (\text{showReg}\ r)$

$(deref\ r\ n)$ $k \to k\ (\text{showInt}\ n +\!\!+ \texttt{"("} +\!\!+ \text{showReg}\ r +\!\!+ \texttt{")"})$

$return\ x$ $\to x$

4 Related Work

Our work is influenced by Eelco Visser's work on the Spoofax Language Workbench [5]. Eelco Visser was the original designer of the Stratego program transformation language [16] which is part of Spoofax. Stratego can be used to implement a whole compiler back end, however Spoofax was still lacking a way to specifying programming language semantics at a higher level of abstraction.

One step in that direction by Vlad Vergu, Pierre Neron, and Eelco Visser was the DynSem DSL for dynamic semantics specification [14]. Later, Vlad Vergu and Eelco Visser developed a way to improve the performance of programs written in languages specified in DynSem by using just-in-time compilation [15]. However, they were not able to fully eliminate the interpretation overhead that DynSem imposes. Furthermore, DynSem uses big-step operational reduction rules, which require explicit managing of the control flow. For languages which include complicated control flow constructs, such as exceptions, this requires non-trivial glue code in every reduction rule.

In the meantime, Eelco Visser started working on adding type systems to Spoofax. Firstly, together with Van Antwerpen et al., he developed Statix [13], which is a language for defining the static semantics of languages defined in Spoofax. Later, Eelco Visser and Jeff Smits added a gradual type system to the meta-language Stratego [12].

Recently, Thijs Molendijk, who was supervised by Eelco Visser, has developed the Dynamix [8] dynamic semantics specification language for Spoofax. Dynamix is more amenable to compilation and it allows for specifying the semantics of complicated control flow constructs independently from other language constructs. The monadic style of Dynamix makes it similar to our work, but the main difference is that Dynamix has a fixed intermediate representation where the only way to extend it is to add new primitives.

As mentioned in the introduction, our approach embraces the nanopass architecture [11, 6]. The main idea of nanopass compilers is that they consist of many small single purpose passes, which aids understanding. We improve upon this work by making it fully typed to prevent

common errors and even check that all cases are covered, and by abstracting over the control flow in the compiler.

5 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. This example application consists of an implementation of a denotation function and handlers which compile this language is compiled in several passes to X86 machine language.

In the future, we would like to extend this minimal compiler with more complicated language constructs such as conditionals, exceptions, and anonymous functions.

Additionally, we would like to implement more complicated analyses on this effectful representation, such as register allocation. We expect these analyses to consist of two stages: first derive concrete structures such as control-flow graphs and interference graphs from our effectful representation, and then perform a pass over the effectful computation that uses the results of the analysis over these structures to transform the program. The first stage would be similar to our handler that turns the effectful computation into a concrete string and the second stage would employ a parameterized handler similar to our stack allocation handler.

Furthermore, we would like to explore the verification of our compilers using algebraic laws for our effect operations, inspired by Interaction Trees [18]. To be specific, we can define a set of laws that describe the behavior of each of the effects in our compiler pipeline. If these laws are sound and complete, with respect to for example definitional interpreters for the effects, then we can prove compiler correctness by proving that these laws are preserved by each of our handlers.

References

1 Casper Bach Poulsen and Cas van der Rest. Hefty algebras: Modular elaboration of higher-order algebraic effects. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023. URL: `https://doi-org.tudelft.idm.oclc.org/10.1145/3571255`, `doi:10.1145/3571255`.

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 N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972. URL: `https://www.sciencedirect.com/science/article/pii/1385725872900340`, `doi:https://doi.org/10.1016/1385-7258(72)90034-0`.

5 Lennart C.L. Kats and Eelco Visser. The spoofax language workbench: Rules for declarative specification of languages and ides. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 444–463, New York, NY, USA, 2010. Association for Computing Machinery. URL: `https://doi-org.tudelft.idm.oclc.org/10.1145/1869459.1869497`, `doi:10.1145/1869459.1869497`.

6 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.

7 E. Moggi. Computational lambda-calculus and monads. In *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, 1989. doi:10.1109/LICS.1989.39155.

8 Thijs Molendijk. Dynamix: A domain-specific language for dynamic semantics. Master's thesis, Delft University of Technology, 2022. URL: http://resolver.tudelft.nl/uuid:8653ab24-a782-41f0-aefc-6b1c8d9a37d5.

9 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.

10 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.

11 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, pages 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.

12 Jeff Smits and Eelco Visser. Gradually typing strategies. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2020, pages 1–15, New York, NY, USA, 2020. Association for Computing Machinery. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/3426425.3426928, doi:10.1145/3426425.3426928.

13 Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. Scopes as types. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/3276484, doi:10.1145/3276484.

14 Vlad Vergu, Pierre Neron, and Eelco Visser. DynSem: A DSL for Dynamic Semantics Specification. In Maribel Fernández, editor, *26th International Conference on Rewriting Techniques and Applications (RTA 2015)*, volume 36 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 365–378, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: http://drops.dagstuhl.de/opus/volltexte/2015/5208, doi:10.4230/LIPIcs.RTA.2015.365.

15 Vlad Vergu and Eelco Visser. Specializing a meta-interpreter: Jit compilation of dynsem specifications on the graal vm. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, ManLang '18, New York, NY, USA, 2018. Association for Computing Machinery. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/3237009.3237018, doi:10.1145/3237009.3237018.

16 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.

17 Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 61–78, New York, NY, USA, 1990. Association for Computing Machinery. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/91556.91592, doi:10.1145/91556.91592.

18 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.