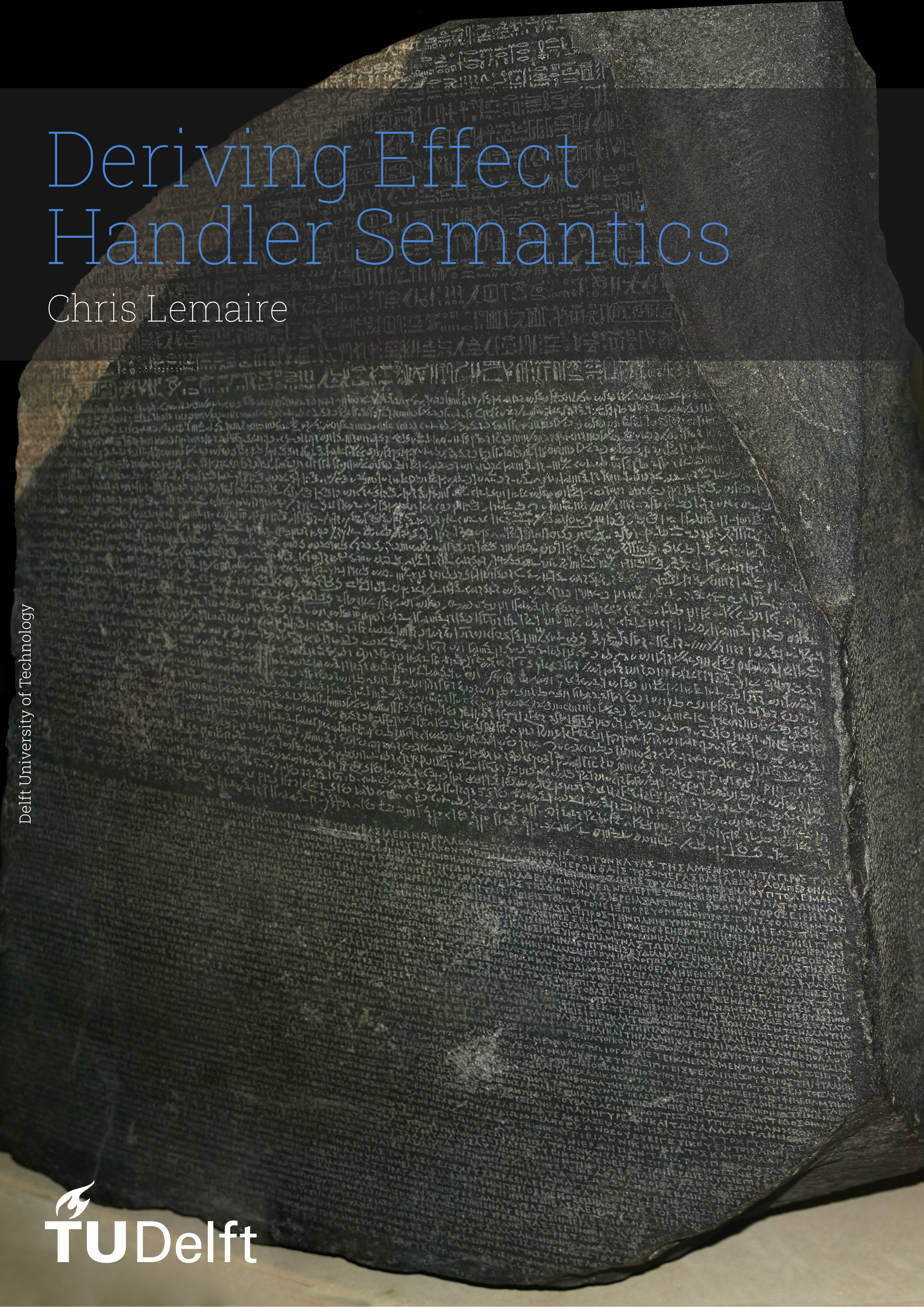
# Deriving Effect Handler Semantics

Chris Lemaire

Delft University of Technology



# Deriving Effect Handler Semantics

by

Chris Lemaire

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Wednesday March 29, 2023 at 9:00 AM.

| | | |
|---|---|---|
| Project duration: | January 9, 2022 – March 29, 2023 | |
| Thesis committee: | Prof. Dr. A. Zaidman, | TU Delft, thesis advisor |
| | Dr. C. B. Poulsen, | TU Delft, daily supervisor |
| | J. Reinders, | TU Delft, daily co-supervisor |

Cover: The Rosetta Stone in the British Museum by Hans Hillewaert under CC BY-SA 4.0 (Modified)
Style: TU Delft Report Style, with modifications by Daan Zwaneveld

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

# Preface

This represents the end of a three-and-a-half year long journey to earn the degree of MSc in Computer Science. A little longer due to various reasons, among which not least of all Covid-19, this degree has offered me insights into the numerous aspects of computer science. The thesis itself is, in my opinion, long overdue after a little over a year of reading, writing, programming, testing, and going back to the drawing board. Although I am at the time of writing mostly looking forward to the end, I am also certain I will look back at this period as one forming me for later years.

I want to thank Casper Bach Poulsen and Jaro Reinders for providing excellent steering on, among other things, which rabbit holes to go down and which to avoid. I also want to thank Andy Zaidman and Olivier Danvy for providing feedback on my thesis. Especially thanking Olivier for his short response times and incredibly helpful suggestions.

Finally, I want to thank everyone that I have met the past six-and-a-half years for the great interactions, deep conversations, and excessive venting we could each take part in. Seeing everyone grow into adult life so well has made me look forward to it and gives me confidence I too will embrace it with open arms.

*Chris Lemaire*
*Delft, March 2023*

# Contents

# Nomenclature

## Terms

| Term | Definition |
|---|---|
| Big-Step | Going directly from an initial state to a final state |
| Capture-avoiding Substitution | A technique for binding names to values without accidentally capturing names that were not in scope in the source program |
| Continuation-Passing Style | Interpreters are in this style when they use continuations to pass evaluation results |
| Denotational Semantics | A form of semantics describing a program as a number of mathematical entities interacting |
| Direct-Style | Without continuations |
| Effectful | Including impure computation |
| Expression | A common building block of a programming language that results in a value when executed |
| Expressivity | Usually regarding a programming language, the theoretical diversity of programs that can be expressed in the language |
| Operational Semantics | A form of semantics describing how a program should be interpreted |
| Program | Text describing instructions for a computer to follow |
| Programming Language | The description of a set of programs, their form (syntax) and meaning (semantics) |
| Pure | Refers to a program or language not allowing side-effects to occur |
| Semantics | A description of the meaning of a program |
| Side-Effect | An interaction outside the local environment of a program, not reflected in the primary result of that program |
| Small-Step | Going from one intermediate state to another intermediate state |
| Syntax | The rules describing the form/shape of a program, usually what text constitutes a program |

## Abbreviations

| Abbreviation | Definition |
|---|---|
| AE&H | Algebraic Effects and Handlers |
| CPS | Continuation-Passing Style |
| DTC | Data Types à la Carte |

# 1
# Introduction

In 1799, French officer Pierre-François Bouchard stumbled upon a carved rock near the Egyptian city of Rashid. Now known as the Rosetta Stone, after the Anglified name for Rashid (Rosetta), the stone is well known due to our efforts in understanding ancient text, as it was the first 'document' to be written in both Ancient Greek and Ancient Egyptian. This was key in translating one into the other and helped us relate texts written in the one to texts written in the other [4]. In programming language research, we often relate code to mathematical representations, but are required to prove their relation through our own equivalents to the Rosetta Stone[1]. In this work, we extend our knowledge of translating between code and mathematical representation by applying new and known transformations in a developing domain.

To explain what it is we achieve in this thesis, we must first discuss what this relation between 'code' and 'mathematical representation' is exactly. The domain we are interested in is that of interpreters and compilers. These are programs that receive some instructions as input and produce some result(s). Any particular list of input instructions is what we call a program. A programming language describes the rules required to write and understand programs. These are usually split into the exact combinations of text that may appear in a valid program, called syntax, and the meaning of a program in the language, or semantics. To transcribe the semantics of a programming language, we could describe the way an interpreter should execute a program, called operational semantics [71]. Instead we could also try to attach mathematical objects to terms of the language and describe the meaning of a program mathematically, called denotational semantics.[2] Where operational semantics is a useful tool for conveying to programmers what happens when executing a program, the mathematical representations of denotational semantics are a necessary abstraction for proving properties of programs [64]. These are what we elude to in the above paragraph as 'code' and 'mathematical representation', respectively.

In this thesis, we are interested in finding a correspondence between operational and denotational semantics for a specific group of programming languages. In our efforts, we build upon the work done by Olivier Danvy and his team. Danvy described steps to turn small-step interpreters - corresponding to operational semantics - into big-step interpreters - which are often closely related to denotational semantics [20]. We also use the inverse transformation, from big-step to small-step, as was described by Vesely and Fisher [78]. The group of languages we are interested in are those implementing what is now commonly known as effects and handlers, which came into existence as the solution to a long-standing and constantly evolving problem.

The problem was encountered by a committee designing the language called Haskell. The goal was for Haskell to be a language that could be used in production, while being, what is commonly referred to as, pure. A pure programming language does not allow any variables to be changed after declaration and, by doing so, prevents many defects. The problem that came up was modelling interactions with the real world, usually called Input/Output or I/O. How does one deal with the possibility of the outside world changing without the programmer's interactions? When writing pure code, the programmer should have

[1] Not referring to Rosetta Code. (https://rosettacode.org)

[2] A third way, axiomatic semantics, exists, but denotational semantics are usually preferred over them.

as little concern as possible for external factors. Indeed, the complexity of writing and understanding code is reduced when a written piece of code is predictable. A solution that satisfied all conditions was only found in 1996 with the introduction of monads. Monads are a formidable building block that require papers, tutorials, and hopelessly lost analogies to truly get into, but in the spirit of the thesis, all that we require to know now is that they allow us to 'fake' a mutable variable. Thus, I/O would be modelled as a monad, updating an invisible environment state in the background [36].

Of course, after solving this initial question, another quickly presented itself. The question regarded combinations of different monads. For instance, when handling I/O, one might want to also keep track of some local variable through monadic operations. There are many more examples of such interactions, including ones not using I/O at all. When used in combination as it was, however, Haskell code would end up a little mangled, nesting various actions in one another. The question was raised whether it might be possible to arbitrarily combine monads to prevent this nesting [46, 79]. Various solutions were proposed [42, 57, 59, 75], but none matched the requirements of arbitrary composition quite as perfectly as algebraic effect handlers, introduced in 2009 [70].

When combined with knowledge gained from the excellent functional pearl "Data Types à la Carte" by Swierstra [76], one could now write Haskell code that arbitrarily combines and composes user-defined effects [49, 44]. These effects represent a similar abstraction of interactions with an external environment as monads did, but they automatically compose. However, algebraic effects and handlers might still be limited by their algebraic requirement [27]. This is where we are now, solutions to this problem of expressivity of effects and handlers are being proposed and discussed. Examples of proposals to solve it are scoped effects [80], staged effects [72], latent effects [10], and hefty algebras [6]. In this relatively new field of research, these important works distinguish different semantics and argue for and against certain design decisions. Semantic decisions which are described in either operational or denotational style, but not often both.

This thesis concerns the semantics of simple languages with effects and handlers, but a softer introduction would be to start with a language without those. Consider a very simple language only offering integers and integer addition. Examples of terms in this language are `3 + 3`, `4 + (5 + 1)`, `(1 + 2) + 3 + (4 + 5)`, etc. The shape of these terms is what we call syntax, the meaning of programs in the language is called semantics. We could describe this as follows: when a +-term is encountered, the left-hand side argument is first reduced to an integer value, then the right-hand side, and finally the resulting values are added together. This process could be described with a form of operational semantics called small-step operational semantics:

$$\text{plus-left: } \frac{e_1 \rightsquigarrow e_1'}{e_1 + e_2 \rightsquigarrow e_1' + e_2} \qquad \text{plus-right: } \frac{e_2 \rightsquigarrow e_2'}{v_1 + e_2 \rightsquigarrow v_1 + e_2'}$$

$$\text{plus-apply: } \frac{}{v_1 + v_2 \rightsquigarrow v_3} (\text{where } v_3 = v_1 + v_2)$$

The plus-left rule shows that, when the left-hand side argument of an addition can still be reduced, it will be reduced. The plus-right rule shows that, when the left-hand side is irreducible - $v$ stands for integer values - and the right-hand side might still reduce, the right-hand side is reduced one step. Finally, the plus-apply rule shows that two values are added together with the classic mathematical meaning of additional.

Applying Danvy's guidelines [20], we can derive big-step semantics that correspond to the same language. These semantics are still operational in nature, as they describe how to interpret programs in the language, but they are also closer to a mathematical model than the earlier small-step semantics. Big-step semantics offer a different way of looking at the same behavioural rules to small-step semantics. Where small-step semantics take an initial configuration to produce a next configuration, big-step semantics produce a final configuration, which may not be further reduced by another semantic rule. In the following big-step semantics, the plus-big rule represents plus-left, plus-right, and plus-apply in one. It tells us to reduce the left-hand side and right-hand side arguments to a value each and add these values together to produce the result of an addition.

$$\text{plus-big: } \frac{e_1 \longrightarrow v_1 \quad e_2 \longrightarrow v_2}{e_1 + e_2 \longrightarrow v_3} (\text{where } v_3 = v_1 + v_2)$$

Finally, using the inverse of closure conversion, we can derive a denotational semantics from a big-step semantics [19].

For algebraic effects and handlers, we know both small-step and big-step operational semantics [7] and we are familiar with denotational semantics [8]. But, what is missing is a structured showing that one is equivalent to the other. On top of this, most efficient implementations of algebraic effects and handlers closely resemble their denotational semantics by encoding operations in what is called the free monad. In this work, we often refer to these implementations as freer monad-based embeddings of effects and handlers. These embeddings enable programmers to write effectful programs as though they are monadic programs. However, the derivation of such an embedding from an operational semantics remains thusfar unexplored. This work fills in the gap between a denotational interpreter derived by inverse closure conversion and the freer monad-based embedding of effects and handlers. We thus define and show program transformations that extend the steps needed to transform a denotational interpreter to a small-step interpreter and vice versa. All code for this thesis is written in Haskell and can be found on Github[3].

Additionally, we show that our added transformations can be reversed and combined with transformations for going from big-step to small-step semantics to obtain operational semantics from denotational semantics for shallow algebraic effects [32] and handlers and from scoped effects and handlers. Finally, to verify our transformations are correct, we provide a test suite for testing that every transformation produces an equivalent interpreter to the one before. In summary, we provide the following technical contributions:

1. We apply known transformations to derive a denotational interpreter from a small-step semantics for deep algebraic effects. We then describe and apply our own set of program transformations to derive a freer monad-based Haskell embedding from the denotational semantics (Chapter 3).
2. We describe the inverse program transformations to those we add in 1., to derive a denotational interpreter from a freer monad-based embedding of effects and handlers. We apply these added transformations and known transformations to derive a small-step semantics for shallow algebraic effects (Chapter 4).
3. We derive an operational semantics for scoped effects and handlers using the newly added transformations from 2. (Chapter 5).
4. We use state-of-the-art program synthesis techniques to generate test programs containing deep algebraic effects and handlers to verify that each of the interpreters we derived has the same behaviour as the interpreter it was derived from (Chapter 6).

[3] `https://github.com/chrislemaire/deriving-handler-semantics`

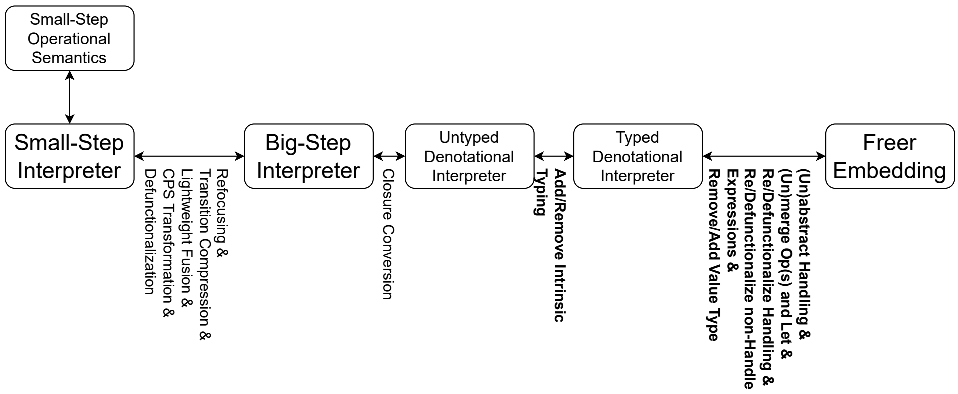



**Figure 1.1:** Overview of correspondences used and introduced in this thesis. We add the derivations in bold from typed to untyped denotational interpreters and from denotational interpreter to freer embedding.

## Overview

Previous work was able to outline step-by-step instructions to transform small-step interpreters into big-step interpreters [20] and back [78]. In this work, we apply these instructions to and extend them for language with effects and handlers to derive a small-step operational semantics from a freer monad embedding in Haskell and vice versa. More specifically, we apply derivations to get small-step semantics from a denotational semantics (passing through big-step semantics) and vice versa, and add our own derivations for obtaining a denotational interpreter from a freer monad embedding and vice versa.

This work builds a correspondence between the often-used embeddings of effects and handlers and a more traditional denotational semantics for languages with effects and handlers. Of note is that we show this correspondence in Haskell, a call-by-need language. This property of the *defining* language can determine parts of the semantics of the *defined* languages, as shown by Reynolds [74]. This correspondence builds on previous works showing syntactic and functional correspondences between various forms of semantics, including the aforementioned correspondence between big-step and small-step semantics [20]. Figure 1.1 shows an overview of the program transformations demonstrated in this thesis.

The overview shows that we derive a small-step interpreter from operational semantics, apply various transformation to derive a big-step interpreter, apply inverse closure conversion to derive an untyped denotational interpreter, add intrinsic typing to get a typed denotational interpreter, and apply a series of transformations of our own formulation to get the final freer monad embedding. This process is reversed by inversing each transformation and can be used to obtain other forms of semantics, such as small-step semantics for effects and handlers implemented with a freer monad embedding.

This thesis is divided into five main chapters. Chapter 2 demonstrates the existing correspondence between denotational and small-step operational semantics. Chapter 3 introduces the program transformations we use to derive a freer monad-based embeddings in Haskell of effects and handlers and immediately applies these derivations to derive the canonical freer monad embedding from a reduction semantics for a language with deep algebraic effects and handlers. Chapter 4 applies the inverse of the aforementioned transformations to derive an operational semantics for a language with shallow algebraic effects and handlers from a freer monad-based embedding of those effects and handlers. Chapter 5 applies the inverse program transformations once again, but to derive a novel operational semantics for deep scoped effects and handlers. Chapter 6 explains how we verified the interpreters we got from every program transformation.

### Small-step Operational to Denotational

The correspondence between small-step operational and denotational semantics is a known one [19, 22, 2]. In these works, Danvy, Biernacka, Sig Ager, Midtgaard, and Millikin show that, through simple program transformations, one can obtain interpreters corresponding to one type of semantics from interpreters corresponding to another type of semantics.

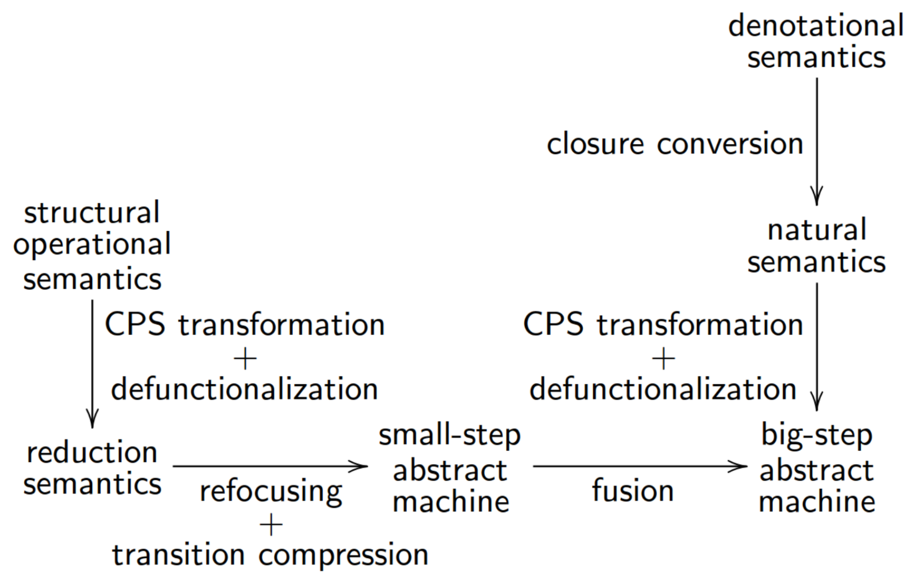



**Figure 1.2:** Functional and syntactical correspondences and the correspondence between small-step and big-step abstract machines. Image taken from [19].

Specifically, we apply the exact steps presented by Danvy in [20] to transform a reduction semantics to a big-step semantics (natural semantics in figure 1.2). The correspondence between denotational semantics and natural semantics is shown with closure conversion. We can selectively apply an inverse operation to closure conversion to derive an interpreter that represents the denotational semantics. This process is shown in Section 2.1. An overview of the various forms of semantics we pass through is shown in figure 1.2. In this, Danvy refers to the relation between reduction semantics and abstract machines as a syntactical correspondence and the relation between natural semantics and abstract machines as a functional one.

An inverse of these relations can be used to derive a small-step operational semantics from an interpreter corresponding to denotational semantics. Applying closure conversion to the corresponding interpreter gives us an interpreter corresponding to the big-step semantics. We apply the transformational steps presented by Vesely and Fisher [78] to further this big-step interpreter to an interpreter corresponding to the structural semantics.

## Reduction Semantics to Freer Monad-Based Embedding

In Chapter 3, we relate a denotational interpreter for a language with deep algebraic effects and handlers to the freer monad-based embedding of that same language. Starting with a reduction semantics, we apply a set of program transformations compiled by Danvy [20] to derive a big-step direct-style interpreter. We then replace closures with higher-order functions to get a denotational interpreter. Finally, we apply our own program transformations to derive the freer monad-based embedding of deep algebraic effects and handlers. These added steps embed pure computations and the handling construct as functions rather than data. They generalise the expression tree of the language until only pure and impure computation are represented. Such an expression tree naturally correspond to the free/freer monad.

The steps we add are as follows:

1. Lettify pure computations.
2. Intrinsically type the interpreter.
3. Generalise values.
4. Lettify the handling construct.
5. Merge *Let* with impure computations.

### Freer Monad-Based Embedding to Structural Semantics

To derive a denotational interpreter and subsequently the small step operational semantics for freer monad-based embeddings of effects and handlers, we show that the the inverse of our added steps can be applied. The impure computations present in the freer monad tree are split up into impure operations and let expressions. Our lettification steps may be inversed by adding expressions for smart constructors and inlining their evaluation instead. A value type is added rather than removed. Intrinsically typing the interpreter can be inverted by removing intrinsic typing instead.

Here are the steps we follow:

1. Split *Let* and impure computations.
2. Inline and lift the handling construct.
3. Inline and lift pure computations and specialise values.
4. Remove intrinsic typing from the interpreter.

Notice that we move around a few steps, but otherwise always apply the inverse of a transformation. The permutation of steps here is used for convenience, not out of necessity.

To obtain an operational semantics for a language with effects and handlers, we start by applying our transformations on the freer monad-based embedding of such a language. We then closure convert the resulting interpreter to derive a big-step direct-style interpreter. We use the order of program transformation steps described by Vesely and Fisher [78] to derive the small-step interpreter, and finally derive a structural operational semantics.

We first show the inverse transformations in Chapter 4, and apply them on a freer monad-based embedding of deep scoped effects and handlers in Chapter 5 to derive a novel operational semantics for deep scoped effects and handlers.

## Evaluation

Chapter 6 describes the evaluation of our derivations in Chapters 3 to 5. Here, we do not prove the desired properties for our added program transformations, instead we attempt to verify that the application of our transformations in the aforementioned chapter is without mistake. Ideally, one would prove that the derivation of each interpreter is perfectly behaviour preserving, meaning that for every possible input, the output of every interpreter is the same. However, writing a formal proof for every interpreter is tedious and, on top of that, incredibly challenging in just Haskell. Instead, we generate a test suite that checks this property to the best of our ability.

Creating tests is done in two steps: programs are generated (1), and converted to target interpreter expression trees (2). We implement program generation based on the type-derived program synthesis technique described by Pałka [67]. We extend this technique for the more complex construct of deep algebraic handlers. To also test shallow and scoped effect handlers, we convert these deep algebraic handlers to embeddings of those same handlers in the target effect handlers. We convert those generated untyped programs to the target expression trees. Typed expression trees present a problem here because we need to coerce the generated expression trees to a typed expression tree. Finally, if we run this test suite, we are ensured that, for all programs generated by the program generator, all interpreters behave the same.

# 2
# From Small-Step Operational to Denotational and Back

This thesis discusses the relation between the freer monad-based embedding of effects and handlers to a corresponding small-step operational semantics. We derive the such embeddings from operational semantics and vice versa. To do so, we take a number of readily available program transformations that have previously been shown to relate various semantic forms and we add our own. In this chapter, we discuss the program transformations that we use out-of-the-box, rather than the steps we add. We look at the relation between small-step and denotational interpreters and how one could derive one from the other using simple program transformations.

In this chapter, we show the transformations necessary to derive a denotational interpreter from a reduction semantics for a minimalistic language without effects and handlers in Section 2.1. We also show, for the same language, how one could reverse the steps previously done to obtain a small-step structural operational semantics in Section 2.2.

## 2.1. Small-Step to Big-Step

In this section, we follow the steps outlined by Danvy in his work "From Reduction Based to Reduction-Free Normalization" [20]. This work gives a step-by-step method for deriving a big-step direct-style interpreter from a set of reduction rules. That is, we follow a few steps to implement an interpreter from the small-step reduction rules and then use readily available program transformations to derive a big-step direct-style interpreter for the same language. In his work, Danvy illustrates these steps with multiple different small languages.

In this section, we repeat the lessons learned by Danvy with a small example language we call ***Ex***. We introduce syntax similar to the syntax used in later chapters and show some of the more standard parts of the transformations here. In later chapters, we only focus on the parts of the transformations that are interesting when effects and handlers are added to the language.

### 2.1.1. Syntax and Semantics of Ex

We start with a formal syntax and reduction rule semantics. ***Ex*** is a simple expression language merely implementing integer addition and multiplication, and lambda- and let-bindings. Its syntax is shown in Figure 2.1a. Values are either integer literals or function definitions and expressions come in a few forms. Lambda values, function application, and let-binding all contribute to binding variable names. Binary operations are written in infix notation.

To describe the meaning of programs within ***Ex***, we also need to provide some type of formal semantics. To use Danvy's method with as little extra steps as possible, we use small-step reduction rules to describe this. Reduction semantics consist of evaluation contexts and reduction rules. Evaluation contexts are used to search for the left-most inner-most part of the expression tree that may still be

| | | | |
|---|---|---|---|
| $v$ | ::= | *nat* | *Natural numbers* |
| | $\mid$ | **fun** $id_x \mapsto e$ | *Anonymous functions* |
| $bop$ | ::= | + $\mid$ $*$ | |
| $e$ | ::= | $v$ | *Value literals* |
| | $\mid$ | $id$ | *Variables* |
| | $\mid$ | $e_1\ e_2$ | *Function application* |
| | $\mid$ | **let** $id$ = $e$ **in** $e$ | *Let-binding* |
| | $\mid$ | $e_1\ bop\ e_2$ | *Unary and binary operations* |
| | $\mid$ | $(e)$ | *Parentheses* |

**(a)** Syntax for a ***Ex*** with lambdas, numeric operations, and ifs

**fun** $x \mapsto x$

$(a + (b * c))$

**let** $x = (2 + 5)$ **in** $(x * x)$

**fun** $x \mapsto (($**fun** $y \mapsto (x + y))\ (x + 1))$

**(b)** Example expressions in ***Ex***.

**Figure 2.1**

reduced. The reduction rules describe what forms of expression, when found in an evaluation context, can be reduced to a 'smaller' term.

Figure 2.2a shows the evaluation context $\mathcal{C}$ for ***Ex***. The first case is standard and describes an empty evaluation context. This empty context is called the context hole and it represents the inner-most left-most expression that is to be evaluated by a reduction rule. The second and third cases show the order in which to look at function applications: if a function application has a value as its first argument, we look into the second argument with $\mathcal{C}$, otherwise we look at the first argument first. In the same way, for binary operations, the left argument is evaluated to a value before the right argument. Finally, for let-expressions, only the binding is evaluated before the entire let-expression can be reduced.

| | | | |
|---|---|---|---|
| $\mathcal{C}$ | ::= | $[]$ | *Context hole* |
| | $\mid$ | $(\mathcal{C}\ e) \mid (v\ \mathcal{C})$ | *Application* |
| | $\mid$ | **let** $x$ = $\mathcal{C}$ **in** $e$ | *Let* |
| | $\mid$ | $(\mathcal{C}\ bop\ e) \mid (v\ bop\ \mathcal{C})$ | *Bin-ops* |

**(a)** Evaluation contexts for ***Ex***.

$\mathcal{C}[($**fun** $x \mapsto e)\ v] \longrightarrow \mathcal{C}[e\ [x/v]]$

$\mathcal{C}[$ **let** $x = v$ **in** $e] \longrightarrow \mathcal{C}[e\ [x/v]]$

$\mathcal{C}[v_1 + v_2] \longrightarrow \mathcal{C}[(v_3)]$ where $v_3 = v_1 + v_2$

$\mathcal{C}[v_1 * v_2] \longrightarrow \mathcal{C}[(v_3)]$ where $v_3 = v_1 * v_2$

**(b)** Reduction rules for ***Ex***. From top to bottom these are: bèta reduction, let-binding, integer addition, and integer multiplication.

**Figure 2.2**

The reductions that should be done on the inner-most left-most expression are described in the reduction rules in figure 2.2b. The left-hand side of a reduction rule describes what expressions are transformed by the rule. For instance, the left-hand side of the let-expression reduction rule matches **let** $x = v$ **in** $e$. The production of a reduced expression is shown on the right-hand side of the arrow in a reduction rule. This means that every **let** $x = v$ **in** $e$ in the left-most inner-most position of an expression is reduced to $e\ [x/v]$, meaning all occurrences of $x$ are replaced by value $v$ in expression $e$, and the resulting expression replaces **let** $x = v$ **in** $e$ in the expression tree. Similarly, beta-reductions are those where a function value is applied to some argument value. In these reductions, the entire application is replaced with the body of the function with $v$ subsituted in the place of every $x$.

### 2.1.2. An Interpreter

To understand how an interpreter of this language would work, we refer to one of the examples shown in figure 2.1b. Consider **let** $x = (2 + 3)$ **in** $(x * x)$. If we would evaluate this expression to a value, we would, intuitively, interpret this to mean $x = 2 + 3 = 5$ and the result of the entire expression would be $x * x = 5 * 5 = 25$. The reduction rules formalise this intuition by stepwise declaring how to interpret an expression. In the following, we show the manual reduction using evaluation contexts and reduction rules:

$\textbf{let } x = [\,] \textbf{ in } (x * x)$ where $[\,] = (2 + 3)$
*apply integer addition rule with* $v_1 = 2, v_2 = 3$ *to get* $[\,] = 5$
$[\,]$ where $[\,] = \textbf{let } x = 5 \textbf{ in } (x * x)$
*apply* **let** $-$ *binding rule with* $x = \texttt{"x"}, v = 5$ *to get* $[\,] = (5 * 5)$
$[\,]$ where $[\,] = (5 * 5)$
*apply integer multiplication rule with* $v_1 = v_2 = 5$ *to get* $[\,] = 25$
25 *with no further decompositions into an evaluation context*

We first find the inner-most left-most part of the expression that matches no evaluation context and apply a reduction rule on that part of the expression to reduce it. If no matching reduction rule can be found, the expression must be malformed. For instance 5 2 is not a well-formed expression because it cannot be further reduced while also not representing a value.

The idea for an interpreter based on reduction rules is to automate this process of searching and reducing. We start by representing the syntax of ***Ex*** as Haskell data types. In figure 2.3a we represent the two different types of values as data constructors. Lambda values store the name of the parameter as a *String* and the body of the function as an *Expr*. Integer values wrap a Haskell *Int*. In figure 2.3b we encode the different expressions as *Expr*. For instance, function applications are encoded as *App Expr Expr*, representing the left and right argument of an application as *Expr*s each. Integer addition and multiplication are grouped as *BinOp*s, using *BinOpOperator* to distinguish between the two. With these types, we can represent $\textbf{let } x = (2 + 3) \textbf{ in } (x * x)$ as:
*Let* `"x"` (*BinOp* (*Lit* (*IntV* 2)) *Add* (*Lit* (*IntV* 3))) (*BinOp* (*Var* `"x"`) *Mul* (*Var* `"x"`))

**data** *Value*
  = *LambdaV String Expr*
  | *IntV Int*

**data** *BinOpOperator* = *Add* | *Mul*

**(a)** Values and binary operations of ***Ex***.

**data** *Expr*
  = *Var String*
  | *App Expr Expr*
  | *Let String Expr Expr*
  | *BinOp Expr BinOpOperator Expr*
  | *Lit Value*

**(b)** Expressions of ***Ex***.

**Figure 2.3**

We also represent evaluation contexts (figure 2.4b) and the left-hand sides of reduction rules (figure 2.4a) as data types. For instance, *PRBeta* represents the left-hand side of the bèta reduction rule, storing the name $x$ as a *String*, value $v$ as a *Value*, and expression $e$ as an *Expr*.

We utilise these data structures with a few main functions and a few more helper types and functions. The main functions needed for evaluation are `decompose_context`, `decompose_expr`, `reduce`, and `iterate` and `normalise`. Besides those, we use helper functions `recompose`, and `subst`. The two decomposition functions are used to find the inner-most left-most expression matching the left-hand side of a reduction rule. The `contract` function is used to apply a single reduction rule step. `iterate` and `normalise` combine all the main steps to create an interpreting function in `normalise`. Furthermore, `recompose` is used to reconstruct a context into an expression and `subst` is used to substitute a certain variable for a value within some expression.

We start by defining the `subst` function and `contract` function in figure 2.5. `subst` replaces any *Var y* values with *Closed v* if $x \equiv y$ and recurses down every sub-expression otherwise. *Closed* expressions are closed under substitution, meaning that substitution does not continue down through these expressions to prevent name-capture. `contract` takes a left-hand side of a reduction and applies the reduction rule it represents if the captured expression is not otherwise malformed. For instance, $(2 + 3)$ could be captured and turned into the potential reduction *PRAdd* (*IntV* 2) (*IntV* 3). `contract` then reduces it to `Closed (IntV 5)` and returns this ‘`Contractum`’. A `PotentialRedex` such as *PRAdd* (*LambdaV* `"x"` (*IntV* 2)) (*IntV* 3) would, however, result in an `Error` as there is no way to reduce an expression such as this one. After all, adding a function and an integer has no meaning assigned to it in our language.

**data** *PotentialRedex*
  = *PRBeta String Expr Value*
    $(\mathcal{C}[(\textbf{fun}\ x \mapsto e)\ v])$
  | *PRLet String Expr Value*
    $(\mathcal{C}[\ \textbf{let}\ x = v\ \textbf{in}\ e])$
  | *PRAdd Value Value*
    $(\mathcal{C}[v_1 + v_2])$
  | *PRMul Value Value*
    $(\mathcal{C}[v_1 * v_2])$
  | *PRError String*

**(a)** Encoding of the left-hand side of each reduction rule (shown in grey) for ***Ex***.

**data** *Context*
  = *CEmpty*
    $[\,]$
  | *CAppL Context Expr*
    $(\mathcal{C}\ e)$
  | *CAppR Value Context*
    $(v\ \mathcal{C})$
  | *CLet String Context Expr*
    $\textbf{let}\ x = \mathcal{C}[\cdot]\ \textbf{in}\ e$
  | *CBinOpL Context BinOpOperator Expr*
    $\mathcal{C}\ bop\ e$
  | *CBinOpR Value BinOpOperator Context*
    $v\ bop\ \mathcal{C}$

**(b)** Encoding of the evaluation context cases (shown in grey) for ***Ex***.

**Figure 2.4**

The next step is to implement functionality to find and construct `PotentialRedex` instances. We do so with two functions: `decompose_expr` and `decompose_context`. Both of these try to find the inner-most left-most reducible expression and return a value or an error if no further decomposition exists. We show these functions in figure 2.6. Decomposition of expressions creates a `Context` that is zipped inside out, meaning the inner-most context represents the outer-most expression and, more usefully, the outer-most context represents the inner-most left-most expression. These functions dictate the order in which the expression is explored. For instance, when an expression such as *BinOp e1 Add e2* is decomposed, *e1* is first further looked into, before turning to *e2* in the case for `CBinOpL` in `decompose_context`, and finally the entire expression (`CBinOpR` in `decompose_context`). We see another helper data type is used for representing the result of decomposing: `ValueOrDecomposition`, meaning either a `Value` is directly found, or a reduction can be done, if no reduction can be done, we return a decomposition with an error instead.

Finally, we utilise these functions in the `iterate` and `normalise` functions. These use another helper function `recompose` to do their work. In figure 2.7 we show these functions. We encode results of evaluation as `Result`, so either a `Value` result or an error. Iteration is done by performing a decomposition, contracting, then iterating on the next decomposition of the recomposed expression after reduction. `normalise` does an initial call to `iterate` to start the evaluation process. If an error or value is encountered on the top-level expression, iteration is done and a `Result` is produced. `recompose` works by giving an `Expr` *t* to insert in the place of the empty context

```
recompose :: Context → Expr → Expr
recompose CEmpty t = t
recompose (CAppL c e2) t =
  recompose c $ App t e2
recompose (CAppR v1 c) t =
  recompose c $ App (Closed v1) t
recompose (CLet x c e) t =
  recompose c $ Let x t e
recompose (CBinOpL c bop e2) t =
  recompose c $ BinOp t bop e2
recompose (CBinOpR v1 bop c) t =
  recompose c $ BinOp (Closed v1) bop t
```

```
data Result
  = Result Value
  | Wrong String

decompose :: Expr → ValueOrDecomposition
decompose = decompose_expr CEmpty

iterate0 :: ValueOrDecomposition → Result
iterate0 (VODValue v) = Result v
iterate0 (VODDec pr c) = case contract pr of
  Contractum e → iterate0 (decompose (recompose c e))
  Error err → Wrong err

normalise0 :: Expr → Result
normalise0 e = iterate0 (decompose e)
```

**Figure 2.7:** Iteration over decompositions and contractions.

```
data Expr
  = ...
  | Closed Value
subst :: String → Value → Expr → Expr
subst x v (Var y)
  | x ≡ y = Closed v
  | otherwise = Var y
subst x v (Lit (LambdaV y e))
  | x ≡ y = Lit (LambdaV y e)
  | otherwise = Lit (LambdaV y (subst x v e))
subst x v (App e1 e2) =
  App (subst x v e1) (subst x v e2)
subst x v (Let y ev eb)
  | x ≡ y = Let y (subst x v ev) eb
  | otherwise = Let y (subst x v ev) (subst x v eb)
subst x v (BinOp e1 op e2) =
  BinOp (subst x v e1) op (subst x v e2)
subst _ _ e@(Lit _) = e
subst _ _ e@(Closed _) = e
```

```
data ContractumOrError
  = Contractum Expr
  | Error String
contract :: PotentialRedex → ContractumOrError
contract (PRBeta x e v) = Contractum (subst x v e)
  ⟶ C[e [x/v]]
contract (PRLet x e v) = Contractum (subst x v e)
  ⟶ C[e [x/v]]
contract (PRAdd (IntV n1) (IntV n2)) =
  Contractum (Closed (IntV (n1 + n2)))
  ⟶ C[(v3)]  where v3 = v1 + v2
contract (PRMul (IntV n1) (IntV n2)) =
  Contractum (Closed (IntV (n1 * n2)))
  ⟶ C[(v3)]  where v3 = v1 * v2
contract (PRError err) = Error err
contract pr = Error ("Cannot match types for: "
  <> show pr)
```

**Figure 2.5:** Substitution and contraction of inner-most left-most expressions matching reduction rules for ***Ex***.

This concludes the creation of an interpreter for ***Ex***. Running *normalise0 e* on some expression *e* now results in a `Value` or an error depending on whether the expression was well-formed. We continue transforming this interpreter from a small-step interpreter to a big-step interpreter in the following steps.

### 2.1.3. Step 1: Refocusing

From this point on, we suffix all functions that may be changed and adapted over different versions of the interpreter with a number indicating the step they belong to. For instance, `iterate` becomes `iterate1` in this step. This helps us separate different versions of interpreter functions and makes sure we call the right versions of functions.

For this first step we realise one simple fact: constantly decomposing and recomposing expressions is costly and this cost could be reduced. This reduction is done by removing recomposition entirely. As it turns out, when a contraction is done, we do not need to start over with our search for the left-most inner-most expression. Instead, we can restart the search on the left-most inner-most position, which is closer to finding a result that starting from the top. This saves us needing to recompose the entire expression every time. This process is called refocusing and the `refocus` function captures it. In the following snippet we see the changed lines of the `iterate` function highlighted:

```
refocus :: Context → Expr → ValueOrDecomposition
refocus = decompose_expr
iterate1 :: ValueOrDecomposition → Result
iterate1 (VODValue v) = Result v
iterate1 (VODDec pr c) = case contract pr of
  Contractum e → iterate1 ( refocus c e)
  Error err → Wrong err
```

### 2.1.4. Step 2: Inlining Contraction

This next step is to fuse the `contract` and `iterate` functions as `contract` is only called in the iteration process. We do so by unfolding the call to `contract` in `iterate` and rewriting the resulting function to perform pattern matches on `VODDec` in the top-level of the `iterate` function definition. The following

```
data ValueOrDecomposition
  = VODValue Value
  | VODDec PotentialRedex Context

decompose_expr :: Context
               → Expr
               → ValueOrDecomposition
decompose_expr c (Var s) =
  VODDec (PRError ("Free variable: " <> s)) c
decompose_expr c (App e1 e2) =
  decompose_expr (CAppL c e2) e1
decompose_expr c (Let x ev eb) =
  decompose_expr (CLet x c eb) ev
decompose_expr c (BinOp e1 bop e2) =
  decompose_expr (CBinOpL c bop e2) e1
decompose_expr c (Lit v) =
  decompose_context v c
decompose_expr c (Closed v) =
  decompose_context v c
```

```
decompose_context :: Value
                  → Context
                  → ValueOrDecomposition
decompose_context v CEmpty =
  VODValue v
decompose_context v (CAppL c e2) =
  decompose_expr (CAppR v c) e2
decompose_context v (CAppR (LambdaV x e) c) =
  VODDec (PRBeta x e v) c
decompose_context _ (CAppR v1 c) =
  VODDec (PRError (
    "Cannot apply non-function value: "
      <> show v1)) c
decompose_context v (CLet x c eb) =
  VODDec (PRLet x eb v) c
decompose_context v1 (CBinOpL c bop e2) =
  decompose_expr (CBinOpR v1 bop c) e2
decompose_context v2 (CBinOpR v1 Add c) =
  VODDec (PRAdd v1 v2) c
decompose_context v2 (CBinOpR v1 Mul c) =
  VODDec (PRMul v1 v2) c
```

**Figure 2.6:** Decomposition of an expression into a context and a potential reduction for ***Ex***.

snippet shows the new `iterate` function with some cases left out to make a shorter example:

```
iterate2 :: ValueOrDecomposition → Result
iterate2 (VODValue v) = Result v
iterate2 (VODDec (PRBeta x e v) c) =
  iterate2 $ refocus c (subst x v e)
iterate2 ... = ...
iterate2 (VODDec pr _) =
  Wrong ("Cannot match types for: " <> show pr)
```

### 2.1.5. Step 3: Lightweight Fusion

In this section we apply lightweight fusion [63]. We fuse `decompose_expr` and `decompose_context` with `iterate` through `refocus`. In practise, this means that occurrences of consecutive calls to `refocus` and then `iterate`, brought forth from the previous step, are replaced with calls to `refocus_expr`. `refocus_expr` and `refocus_context` are the results of the fusion. These are similar to `decompose_expr` and `decompose_context`, but their return types are changed to the return type of `iterate`, as instead of returning an intermediate decomposition, this decomposition is now directly passed to `iterate` and the resulting value is returned. We see this change most in the signatures of `refocus_expr` and `refocus_context`, which now return a `Result` value, like `iterate` did already.

```
refocus_expr3 :: Context → Expr → Result
refocus_expr3 c (Var s) =
  iterate3 $ VODDec (PRError ("Free variable: " <> s)) c
refocus_expr3 c (App e1 e2) =
  refocus_expr3 (CAppL c e2) e1
refocus_expr3 ... = ...
```

```
refocus_context3 :: Value → Context → Result
refocus_context3 v CEmpty =
  iterate3 $ VODValue v
refocus_context3 v (CAppL c e2) =
  refocus_expr3 (CAppR v c) e2
refocus_context3 v (CAppR (LambdaV x e) c) =
  iterate3 $ VODDec (PRBeta x e v) c
refocus_context3 ... = ...
```

```
iterate3 :: ValueOrDecomposition → Result
iterate3 (VODValue v) = Result v
iterate3 (VODDec (PRBeta x e v) c) = refocus_expr3 c (subst x v e)
iterate3 ... = ...
```

In the above snippets, we see how the three main functions of our interpreter are changed with this step. `refocus_expr` and `refocus_context` now return a `Result` and previous decomposition results are directly passed to the `iterate` function. Furthermore, `refocus DLR iterate` now is the same is `refocus_expr`, so occurrences of this combination are replaced with it. This step serves to make the three core functions of our interpreter mutually recursive and gives us the first big-step interpreter.

### 2.1.6. Step 4: Compress Corridor Transitions

In this step, we look at transitions between functions in the current interpreter and find 'corridor transitions'. This means we look for calls to functions that only have a single possible execution path, one which is also only reached through that specific call. In our example, we find that calls to `iterate` are all corridor transitions. We unfold these calls to get more involved `refocus_expr` and `refocus_context` functions:

```
refocus_context3 v (CAppR (LambdaV x e) c) = iterate3 $ VODDec (PRBeta x e v) c
...
iterate3 (VODDec (PRBeta x e v) c) = refocus_expr3 c (subst x v e)
  ⟶
refocus_context4 v (CAppR (LambdaV x e) c) = refocus_expr4 c (subst x v e)
```

The result of this transformation in our example language is that the `iterate` function now consists only of dead clauses and we have thus eliminated the need for the `ValueOrDecomposition` and `PotentialRedex` auxiliary data types. When applying this technique on other languages one might find that there are some parts of the `iterate` function that are still used in several places, so the two auxiliary data types might not be completely removed yet.

### 2.1.7. Step 5: Renaming and Flattening Configurations

In this step we rename the current functions to more commonly used names for the same functionality. For instance `refocus_expr` now represents an evaluation function, so we rename it to `eval`. `refocus_context` takes the part left to evaluate and find the next expression to evaluate, so we name it `continue`. Finally, if `iterate` would still contain any live clauses, we could split the function into the few remaining computations and rewritings done in that function to get rid of `ValueOrDecomposition` and `PotentialRedex` entirely. In our case, we rename `refocus_expr` to `eval`, `refocus_context` to `continue` and we remove `iterate` entirely:

| | | |
|---|---|---|
| *refocus_expr4* | $\longrightarrow$ | *eval5* |
| *refocus_context4* | $\longrightarrow$ | *continue5* |
| *iterate4* | $\longrightarrow$ | _ |

If one case of decomposition would be left in `iterate`, for instance that of `PRBeta`, the decomposition parameters become the parameters of the iteration function. This is what 'flattening configurations' refers to. For example, the iteration function for just this one decomposition would be as follows:

```
iterateBeta :: String → Expr → Value → Context → Result
iterateBeta x e v c = eval5 c (subst x v e)
```

### 2.1.8. Step 6: Refunctionalisation

In this step we merge `eval` and `continue` to a single evaluation function. The process through which we achieve this is called refunctionalisation [23, 74]. We do so by realising that evaluation `Context`s together with `continue` are the first-order counterpart of a higher-order function [20]. That is, there is a function that can be used to replace `continue` and `Context` in their entirety. This function is generally referred to as a continuation and it gets the type of `continue` without its `Context` parameter: `Value -> Result`.

To create the higher-order function representing full evaluation of an expression, we use `eval` as a basis and pass it the the continuation of type `Value -> Result` additionally. We then unfold every call to `continue` to a call to `eval` with a new continuation. For instance, the different clauses handling function applications are combined as like the following:

```
eval5 c (App e1 e2) = eval5 (CAppL c e2) e1
 ...
continue5 v (CAppL c e2) = eval5 (CAppR v c) e2
continue5 v (CAppR (LambdaV x e) c) = eval5 c (subst x v e)
continue5 _ (CAppR v1 _) = Wrong ("Cannot apply non-function value: " <> show v1)
   ⟶
eval6 (App e1 e2) k =
  eval6 e1 (λv1 →
    eval6 e2 (λv2 →
      case v1 of
        LambdaV x e → eval6 (subst x v2 e) k
        _ → Wrong ("Cannot apply non-function value: " <> show v1)))
```

The normalisation function is adjusted by passing a default continuation, which just wraps the resulting value with `Result`:

```
normalise6 :: Expr → Result
normalise6 e = eval6 e Result
```

### 2.1.9. Step 7: Back to Direct Style

The final step is to turn this interpreter function to direct-style. The current interpreter is in continuation-passing-like style, by passing the continuation function of type `Value -> Result` around. Turning the interpreting function into direct-style is as simple as pattern matching on the result type and using that result as the value that would have been passed to the continuation function. This eliminates the need for a continuation function. Our final interpreter for ***Ex*** is as follows:

```
eval7 :: Expr → Result
eval7 (Var s) = Wrong ("Free variable: " <> s)
eval7 (App e1 e2) =
  case eval7 e1 of
    Result v1 → case eval7 e2 of
      Result v2 → case v1 of
        LambdaV x e → eval7 (subst x v2 e)
        _ → Wrong ("Cannot apply non-function value: " <> show v1)
      err → err
    err → err
eval7 (Let x ev eb) =
  case eval7 ev of
    Result v → eval7 (subst x v eb)
    err → err
eval7 (BinOp e1 bop e2) =
  case eval7 e1 of
    Result v1 → case eval7 e2 of
      Result v2 → case (v1, bop, v2) of
        (IntV n1, Add, IntV n2) → eval7 (Closed (IntV (n1 + n2)))
        (IntV n1, Mul, IntV n2) → eval7 (Closed (IntV (n1 * n2)))
        _ → Wrong ("Cannot match types for binary operation: " <> show bop)
      err → err
    err → err
eval7 (Lit v) = Result v
eval7 (Closed v) = Result v
```

```
normalise7 :: Expr → Result
normalise7 = eval7
```

In the rest of this work, we try to avoid explicit error-handling like this to focus solely on translations of good-weather behaviour. So we will not have as many default cases for errors. Instead, we simply let errors delegate to top-level with `error`. This is fine because we build pure languages with no runtime exceptions in the following sections. Errors can thus only indicate a typing problem, meaning the program we input is malformed.

### 2.1.10. Step 8: From Big-Step to Denotational

At this point, we have a direct-style big-step interpreter of the language. This step turns the current abstract syntax tree represented by *Expr* into a higher-order abstract syntax [68] represented by *Expr8*. To do so, we replace all occurrences of name-binding with a Haskell function for binding instead.

In figure 2.8a, we show the changes done to values, expressions, and handlers. Here we see that in every place where a name was bound using a *String* and an *Expr*, names are now bound through a function *Value8* → *Expr8*.

```
data Value8
  = LambdaV8 (Value8 → Expr8)
  | ...
data Expr8
  = Let8 Expr8 (Value8 → Expr8)
  | ...
```

**(a)** Higher-order abstract syntax for ***Deep***.

```
eval8 :: Expr8 → Result8
eval8 (App8 ef ea) =
  check_result8 (eval8 ef)
    (λvf → check_result8 (eval8 ea)
      (λva → case vf of
        LambdaV8 body → eval8 (body va)
        _ → Wrong8 ("non-function value: "
          <> show vf))
eval8 (Let8 ev body) =
  check_result8 (eval8 ev)
    (λv → eval8 (body v))
eval8 ... = ...
```

**(b)** Example of evaluating handle-expressions with higher-order abstract syntax.

**Figure 2.8**

To demonstrate the transformation of the evaluation function, we show the cases for function application and let-binding in figure 2.8b. We adjust evaluation cases by replacing calls to *subst* with a call to the appropriate function.

## 2.2. ... and back

Turning a denotational interpreter back into a small-step interpreter is a process exactly inverse to the the process previously described. The work we base these transformations on is that of Minamide et al. [61] and Vesely and Fisher [78]. Minamide et al. describe typed closure conversion, which we use to obtain a big-step interpreter from a denotational interpreter. Vesely and Fisher describe 9 steps to transform a big-step direct-style interpreter to a small-step direct-style interpreter. In this work we go through each of these transformations by hand.

### 2.2.1. Step 0: From Denotational to Big-Step

Inversely to Section 2.1.10, we apply closure conversion to the interpreter we end off with in the previous section to get the first interpreter of this section. We apply this by replacing every instance of a higher-order function with a parameter name-body expression pair. Each of these pairs represents a closure, capturing the name of the previously free variable as its *String* argument. We end up with the same big-step direct-style interpreter as before, but without explicit error handling:

```
eval0 :: Expr → Value
eval0 (Var s) = error ("Free variable: " <> s)
eval0 (App e1 e2) = case eval0 e1 of
  v1 → case eval0 e2 of
    v2 → case v1 of
      LambdaV x e → eval0 (subst x v2 e)
      _ → error ("Cannot apply non-function value: " <> show v1)
eval0 (Let x ev eb) = case eval0 ev of
  v → eval0 (subst x v eb)
eval0 (BinOp e1 bop e2) = case eval0 e1 of
  v1 → case eval0 e2 of
    v2 → case (v1, bop, v2) of
      (IntV n1, Add, IntV n2) → eval0 (Closed (IntV (n1 + n2)))
      (IntV n1, Mul, IntV n2) → eval0 (Closed (IntV (n1 ∗ n2)))
      _ → error ("Cannot match types for binary operation: " <> show bop)
eval0 (Lit v) = v
eval0 (Closed v) = v
```

We take the liberty to remove runtime errors at this point as they have so far only contributed to the length of the work, rather than the depth. From this point, we add a suffix number again to indicate the step of evaluator function we use. Due to an editing issue, we start this at 0, rather than 1.

### 2.2.2. Step 1: CPS Conversion

In the first step we turn the direct-style interpreter into a CPS (continuation passing-style) interpreter. This is done by adding a `(Value -> Value)`-type argument to the evaluator, called the continuation. The continuation is called whenever a value would be resulted from the interpreter. Whenever a recursive call to `eval` is done, we need to construct a continuation that captures the parts of the evaluation function that depend on the result of that recursive call. For instance, the case for evaluating `Let`s defines a continuation `k1` that captures `eval1 (subst x v eb) k'`, as the part of the evaluation function dependent on the result of evaluating `ev`.

```
eval1 :: Expr → (Value → a) → a
eval1 (Let x ev eb) k =
  let k1 = λv →
    let k′ = k
      in eval1 (subst x v eb) k′
    in eval1 ev k1
```

The normalisation function for most of these transformations is rather uninteresting. This normalisation function simply passes in the last continuation, which wraps result values when necessary, or just returns the same value in our case:

```
normalise1 :: Expr → Value
normalise1 = flip eval1 id
```

### 2.2.3. Step 2: Generalisation

The difference between a small-step interpreter and a big-step interpreter is that big-steps fully evaluate an expression to a value, whereas small-step interpreters only step an expression to a 'more evaluated' form. Currently, continuations receive a `Value`-type parameter and result in a `Value`-type result. This step changes the continuation so that it may accept either `Value`s or `Expr`s. We achieve this in this work through the sum-type operator `:+:`. This sum-type operator allows us to define the type of the continuation as: `(Value :+: Expr) -> Value`, meaning the continuation takes either a `Value` or an `Expr` and needs to define how to deal with both cases. The sum-type comes with a convenience function to lift values to the sum-type (`inj0`) and two constructors `Inl0` and `Inr0` which are used to match for a value of the left- and right-type, respectively.

Every continuation in the evaluation function now matches on its parameter to find out whether the parameter is a finished computation (`Value`) or an expression. The `Value`-case is that of the big-step computation as we had it in the first step. The `Expr`-case of a continuation is added and simply passes its argument into the evaluation function with the current continuation as the `eval` continuation. Although this does nothing for the moment, as continuations are currently only called with result values, this step adds small stepping to the evaluation function.

The `Let` and `Lit` cases reflect the types of changes that are performed over this step:

*eval2* (*Let x ev eb*) *k* =
  **let** *k1* = λ**case**
    *Inl0 v* → *eval2* (*subst x v eb*) *k*
    *Inr0 ev′* → *eval2 ev′ k1*
    **in** *eval2 ev k1*
*eval2* (*Lit v*) *k* = *k* (*inj0 v*)

### 2.2.4. Step 3: Argument Lifting

We can categorise the various constituents of an expression used in a continuation during evaluation into two groups.

1. The expression that is currently under evaluation and is the main parameter of a continuation.
2. The expressions that are yet to be evaluated, values that have resulted from previous evaluation and various other constituents of an expression that will be used within the continuation but are not under evaluation in this continuation.

In this step we ensure that both of these types of parameters are passed to the continuation. This is done by adding all constituents of the second group to the parameter list of the continuation and partially applying all mentions of that continuation with the constituents.

For instance, for the `Let`-case the parameter name (`x`) and body expression (`eb`) constituents of `Let` are of the second category and are added to the parameter list of `k1`:

*eval3* (*Let x ev eb*) *k* =
  **let** *k1 x eb* = λ**case**
    *Inl0 v* → *eval3* (*subst x v eb*) *k*
    *Inr0 ev′* → *eval3 ev′* (*k1 x eb*)
    **in** *eval3 ev* (*k1 x eb*)

### 2.2.5. Step 4: Continuations Switch Control

In this step we simplify the evaluation function a little by noticing that the recursive calls to `eval` that switch control to a new continuation (in the `in` part of a `let` that defines a new continuation) are unnecessary. Instead of having this recursive call, we can call the continuation directly with the expression to be evaluated. This causes the `Inr0`-case to call the evaluation function recursively anyway. The `Let`-case is changed to the following:

```
eval4 (Let x ev eb) k =
  let k1 x eb = λcase
    Inl0 v → eval4 (inj0 $ subst x v eb) k
    Inr0 ev′ → eval4 ev′ (k1 x eb)
    in k1 x eb (inj0 ev)
```

### 2.2.6. Step 5: Defunctionalisation

Defunctionalisation is the process of eliminating higher-order functions in code at compile-time [24, 74]. In this step we use this process to extract the nested continuation declarations into a separate `apply` function. To do this, we perform the following changes:

1. Add a `Continuation`-type representing the various types of continuations with their parameters as last amended in step 3.
2. Add an `apply` function that takes a `Continuation` argument and the surrounding continuation. For every `Continuation` constructor, the matching continuation's body becomes the body of the `apply` function, adjusting `case` matching where necessary to refer to the last argument of the continuation.
3. Replace every inner definition and subsequent call of a continuation with a call to `apply` with its `Continuation` counterpart.

Important to note is that, to ensure the interpreter compiles in Haskell, we need to type the last argument to `Continuation`s the same as the general continuation parameter. In "One Step at a Time", Vesely and Fisher use a language with automatic sum-typing, which Haskell does not support out-of-the-box, so we need to use explicit sum-types.

```
data Continuation5
   = Cont5App1 Expr (Value :+: Expr)
   | Cont5App2 Value (Value :+: Expr)
   | Cont5Let1 String Expr (Value :+: Expr)
   | Cont5BinOp1 BinOpOperator Expr (Value :+: Expr)
   | Cont5BinOp2 Value BinOpOperator (Value :+: Expr)
apply5 :: Continuation5 → ((Value :+: Expr) → Value) → Value
apply5 (Cont5Let1 x eb ev) k = case ev of
  Inl0 v → eval5 (inj0 $ subst x v eb) k
  Inr0 ev′ → eval5 ev′ (λev″ → apply5 (Cont5Let1 x eb ev″) k)
eval5 (Let x ev eb) k = apply5 (Cont5Let1 x eb (inj0 ev)) k
```

### 2.2.7. Step 6: Remove Tail-Calls

Before showing that `Continuation`s can be turned into reconstructions of terms, we eliminate the recursive `apply` calls in `Inr0` cases. We do so by passing these `Continuation`s directly to the general continuation instead of calling `apply` for the `Continuation`:

```
apply6 (Cont6Let1 x eb ev) k = case ev of
  Inl0 v → eval6 (inj0 $ subst x v eb) k
  Inr0 ev′ → eval6 ev′ (λev″ → k (inj0 $ Cont6Let1 x eb ev″))
```

This eliminates the recursive calls to `apply` embedded in general continuations and leaves the general continuation in charge of control flow. We do not change the recursive calls to `apply` in `Inl0` cases. Finally, to accommodate this change, we need to change the continuation parameter type to include the `Continuation`-type in the sum. We also adapt the `eval` function to simply call `apply` in the case of a continuation passed to it:

$$\textit{eval6}\ (\textit{Inl0}\ (\textit{Let x ev eb}))\ k = \textit{apply6}\ (\textit{Cont6Let1 x eb}\ (\textit{inj0 ev}))\ k$$

### 2.2.8. Step 7: Convert Continuations into Terms

The previous step ensures that every `Inr0` case in `apply` passes `Continuation`s directly to the general continuation. This setup allows us to replace those `Continuation` values with `Expr` values that evaluate to the exact same call to `apply`, eliminating the need to passing `Continuation` values into general continuations and thus removing the case we added to `eval` in step 6.

To do so, we replace the general continuations passed to `eval` in the `Inr0` cases of `apply` to the body of that same `apply` case with its `Inr0` case substituted for the call to the general continuation instead. This may appear a little convoluted, but is required to ensure the result of `eval` is not a value. We then replace the `Continuation` in the innermost `Inr0` case with an `Expr` representing the leftover computation. For the case of `Let`, the result of these changes is as follows:

$$\begin{array}{l}
\textit{apply7}\ (\textit{Cont7Let1 x eb ev})\ k = \textbf{case}\ \textit{ev}\ \textbf{of} \\
\quad \textit{Inl0 v} \rightarrow \textit{eval7}\ (\textit{subst x v eb})\ k \\
\quad \textit{Inr0 ev}' \rightarrow \textit{eval7 ev}'\ \$\lambda\textbf{case} \\
\quad\quad \textit{Inl0 v} \rightarrow \textit{eval7}\ (\textit{subst x v eb})\ k \\
\quad\quad \textit{Inr0 ev}'' \rightarrow k\ (\textit{inj0}\ \$\ \textit{Let x ev}''\ \textit{eb})
\end{array}$$

### 2.2.9. Step 8: Inlining and Simplification

In this step we reconstruct a CPS interpreter from the `apply` and `eval` functions. We inline every call to `apply` and simplify the resulting `eval` function to get the following `Let`-case in `eval`:

$$\begin{array}{l}
\textit{eval8}\ (\textit{Let x ev eb})\ k = \textit{eval8 ev}\ \$\lambda\textbf{case} \\
\quad \textit{Inl0 v} \rightarrow \textit{eval8}\ (\textit{subst x v eb})\ k \\
\quad \textit{Inr0 ev}' \rightarrow k\ (\textit{inj0}\ (\textit{Let x ev}'\ \textit{eb}))
\end{array}$$

The body of this `eval` clause is the `Inr0` case of the corresponding `apply` clause.

### 2.2.10. Step 9: Back to Direct Style

Finally, to go back to a direct-style interpreter, we remove all mentions of the general continuation in `eval` and fix syntax where necessary. This is possible because the only usages of the general continuation is in the final computation of the evaluation function. We are left with a direct-style small-step interpreter:

```
eval9 :: Expr → (Value :+: Expr)
eval9 (Var s) = error ("Free variable: " <> s)
eval9 (App e1 e2) = case eval9 e1 of
  Inl0 v1 → case eval9 e2 of
    Inl0 v2 → case v1 of
      LambdaV x e → eval9 (subst x v2 e)
      _ → error ("Cannot apply non-function value: " <> show v1)
    Inr0 e2' → inj0 $ App (Lit v1) e2'
  Inr0 e1' → inj0 $ App e1' (inj0 e2)
eval9 (Let x ev eb) = case eval9 ev of
  Inl0 v → eval9 (subst x v eb)
  Inr0 ev' → inj0 $ Let x ev' eb
eval9 (BinOp e1 bop e2) = case eval9 e1 of
  Inl0 v1 → case eval9 e2 of
    Inl0 v2 →
      case (v1, bop, v2) of
        (IntV n1, Add, IntV n2) → eval9 (Closed (IntV (n1 + n2)))
        (IntV n1, Mul, IntV n2) → eval9 (Closed (IntV (n1 * n2)))
        _ → error ("Cannot match types for binary operation: " <> show bop)
    Inr0 e2' → inj0 $ BinOp (Lit v1) bop e2'
  Inr0 e1' → inj0 $ BinOp e1' bop e2
eval9 (Lit v) = inj0 v
eval9 (Closed v) = inj0 v
```

```
normalise9 :: Expr → Value
normalise9 e = case eval9 e of
  Inl0 v → v
  Inr0 e' → error ("STUCK: Irreducible expression: " <> show e')
```

### 2.2.11. Extracting Small-Step Operational Semantics

From the final `eval` function, we can extract a small-step structural operational semantics. We write these as transition rules where every single arrow denotes a small step. For example, we extract the small-step transitions for function application and let-binding and display them in figure 2.9. We could just as well derive an operational semantics for binary operations, but we choose to leave these out for space concerns.

$$\text{App-Left: } \frac{e_1 \to {e'}_1}{(e_1\ e_2) \to ({e'}_1\ e_2)} \qquad \text{App-Right: } \frac{e_2 \to {e'}_2}{(v_1\ e_2) \to (v_1\ {e'}_2)} \qquad \text{App-Beta: } \frac{}{((\lambda x \mapsto e_b)\ v_a) \to e_b\ [x/v_a]}$$

$$\text{Let-Bind: } \frac{e_a \to {e'}_a}{\textbf{let}\ x = e_a\ \textbf{in}\ e_b \to \textbf{let}\ x = {e'}_a\ \textbf{in}\ e_b} \qquad \text{Let-Apply: } \frac{}{\textbf{let}\ x = v_a\ \textbf{in}\ e_b \to e_b\ [x/v_a]}$$

**Figure 2.9:** Structural operational semantics for the simple language obtained through various program transformations, excluding binary operations to reduce clutter.

## Concluding

We have seen how to transform a small-step interpreter into a denotational interpreter for exactly the same language and vice versa in this chapter. In the following chapters, we extend this process with steps to transform a denotational interpreter into an embedding and back. In those chapters, we refer back to these transformations, but we do not explain every single one of these transformations in detail anymore. In Chapter 6, we describe ways to verify these steps, as well as those steps we introduce in the next chapters.

# 3

# Deriving a Freer Monad Embedding for Algebraic Effects and Handlers

In this chapter, we start with a description of a minimal language implementing algebraic effects and handlers, inspired by the language Pretnar used to introduce algebraic effects and handlers [73]. We do so by describing the syntax and small-step operational semantics (in the style of Felleisen and Hieb [26]) of such a language (Section 3.1). We call this language ***Deep*** to refer back to it within this chapter. From this description, we implement a small-step interpreter (Section 3.2). We take this interpreter through the steps to transform it to a denotational interpreter using the program transformations described in Section 2.1 (Section 3.3). We then perform the following added steps to transform the denotational interpreter into a freer monad:

1. Lettify pure computations. (Section 3.4)
2. Add intrinsic typing to values, expressions, handlers, binary operations, etc. (Section 3.5)
3. Generalise the Value type. (Section 3.6)
4. Lettify the handling abstraction. (Section 3.7)
5. Merge impure computations and let constructs into a single expression constructor. (Section 3.8)

## 3.1. The Model Language

The language we show our transformations on is inspired by that used to introduce effects and handlers by Pretnar [73]. By this we mean to say we adopt their syntax and semantics for effect handlers and handling of effects. This language consists of the following expressions: anonymous functions (lambdas), variables, function application, boolean constants, handlers, operation calls, sequencing through a **do**-expression, a handling expression to use handlers, and **if-then-else** expressions. We use this as a base for our language as it offers a convenient recognisable syntax and semantics for algebraic effects and handlers. We do, however, change a few things about the aforementioned syntax. We:

1. Allow effectful computation to occur anywhere, not just sequenced through **do**. Instead, we add **do**-expressions for easily sequencing operations.
2. Instead of passing a continuation to *op*-calls explicitely, we capture the continuation implicitly.
3. Add *nat*ural numbers, lists, pairs, and unit values, and unary and binary operations to be able to show and generate interesting programs.

### Syntax

To describe the language, we start by describing the syntax of the language. That is, the exact phrasing of expressions in ***Deep***. We give a BNF specification in figure 3.1. In short, this specification covers values, handlers, unary- and binary-operations, and expressions. The following concepts are covered:

**Common language constructs** including functions, function application, lists, pairs, integers, booleans, and unary and binary operations are mostly represented with Haskell-like syntax. Only unnamed functions (lambdas) can be constructed with **fun** $id_x \mapsto e$, where $id_x$ is the parameter name and $e$ is the body of the function. A function application such as ((**fun** $x \mapsto x$) 5) is a valid expression in ***Deep*** and we expect it to result in a value of 5 after evaluation.

**Handlers** define a **return**-implementation and zero or more operation implementations. The **return** $x \mapsto e$ implementation determines what to do when having to finish handling an expression under **handle** $h$ **with** $e_b$. When $e_b$ evaluates to a value $v$, the surrounding handler can be fully applied to the value by filling in $e$ with $v$ bound to the name $x$. Operation implementations are defined as $op_i\ (x, k) \mapsto e$, where $op_i$ is the name of the operation, $x$ is the name of the operation parameter, and $k$ is the name of the continuation bound in body $e$. $k$ represents all computation that uses the operation result and can be passed a result to execute those computations.

**Op-calls** are done with **op-call** $op\ e$, where $op$ is the name of the operation to call and $e$ is the argument to be passed to the operation. When handled, $e$ is evaluated and passed to the handler function for $op$. If that handler function calls a continuation with value $v$ as argument, in essence, the program evaluation is continued with $v$ in the place of **op-call** $op\ e$.

**Handling** is done through the **handle** $e$ **with** $h$ construction. Inside expression $e$, all occurrences of an **op-call** expression can be handled by handler $h$. When an **op-call** $op_i\ e$ is found, the nearest surrounding **handle** $e$ **with** $h$ where $h$ has an implementation of $op_i$ is used to handle it. When using an **op-call** expression, the programmer should be aware that every such expression needs to have a surrounding **handle** _ **with** $h$ block handling that specific operation. In some more complex languages, operations can be left unhandled as long as the type reflects what operations are left unhandled [56, 58].

**Do-sequencing** is added to more easily write examples with sequenced operations. In practice, these constructs act as sugar over multiple **let**-expressions. For example, one may write **let** $x$ = $e_1$ **in** (**let** $y$ = $e_2$ **in** $e$) as **do** $x \leftarrow e_1; y \leftarrow e_2; e$.

| | | | |
|---|---|---|---|
| $v$ | ::= | *nat* | *Natural numbers* |
| | \| | **fun** $id_x \mapsto e$ | *Anonymous functions* |
| | \| | **true** \| **false** | *Boolean constants* |
| | \| | $[\,]$ \| $v_h : v_t$ | *Lists* |
| | \| | $()$ \| $(v_1, v_2)$ | *Unit and pair* |
| $h$ | ::= | **handler** $\{$**return** $id_x \mapsto e_r, op_1(id_x, id_k) \mapsto e_1, ..., op_n(id_x, id_k) \mapsto e_n\}$ | *Handlers* |
| $uop$ | ::= | **fst** \| **snd** | |
| $bop$ | ::= | ++ \| + \| $*$ | |
| $e$ | ::= | $v$ | *Value literals* |
| | \| | $id$ | *Variables* |
| | \| | $(e_1\ e_2)$ | *Function application* |
| | \| | (**let** $id$ = $e$ **in** $e$) | *Let-binding* |
| | \| | $do\ (\{\{id \leftarrow\}\}\ e;)^*\ e$ | *Do-sequencing* |
| | \| | (**with** $h$ **handle** $e$) | *Handling* |
| | \| | (**op-call** $id\ e$) | *Effect operation call* |
| | \| | **if** $e_i$ **then** $e_t$ **else** $e_e$ | *If-then-else* |
| | \| | $e_h : e_t$ | *List construction* |
| | \| | $(e_1, e_2)$ | *Pair construction* |
| | \| | $(uop\ e)$ \| $(e_1\ bop\ e_2)$ | *Unary and binary operations* |
| | \| | $(e)$ | *Parentheses* |

**Figure 3.1:** Syntax of ***Deep***. {{...}} denotes optional syntax.

```
let helloWorld = do
  op-call print 43110
in ...
```

```
let getAndIncrement = do
  s ← op-call get ();
  op-call put (s + 1)
  s
in ...
```

```
let flipAndPut = do
  b ← op-call flip ();
  s ← op-call get ();
  if b
    then op-call put (s + 1)
    else op-call put (s + 2)
in ...
```

**Figure 3.2:** Example programs in ***Deep***. From left to right, the programs are a simple hello world print, getting and incrementing an integer state (corresponding to $s$ ++ in Java, C++, etc.), and flipping a coin to update a state.

```
helloWorld :: IO ()
helloWorld = do
  print 43110
```

```
getAndIncrement :: State Int Int
getAndIncrement = do
  s ← get
  put (s + 1)
  return s
```

**Figure 3.3:** Haskell programs corresponding to the two left-most ***Deep*** programs shown in figure 3.2.

### Examples

To illustrate what programs can be created and executed in a language such as ***Deep***, we show three programs in figure 3.2. These programs demonstrate the use of three different effects: I/O, state, and ambiguity. Each of these effects offer different operations.

1. I/O offers input/output actions such as reading text from a user and printing to screen. print simply takes a value and results in $()$.
2. State offers a put and get action for updating and retrieving an underlying state, respectively. put thus expects a state value and results in $()$. Conversely, get receives $()$ and results in a state value.
3. Ambiguity only offers the flip operation, representing the flip of a coin. The operation expects a $()$-value could either result in **true** or **false**.

The *helloWorld* and *getAndIncrement* program each only make use of a single effect. In these cases, we can write corresponding programs in Haskell to illustrate exactly what such a computation looks like. We do so in figure 3.3. The third program, however, is a little harder to transcribe with Haskell code as it makes use of both the ambiguity and state effects. A monad transformers solution would have a type such as *StateT Amb Int* $()$, or perhaps *AmbT* $($*State Int*$)$ $()$, where *Amb* is the ambiguity monad, and *AmbT* the corresponding monad transformer. However, either version makes more assumptions of the effect interactions within this program than the ***Deep*** program, as both types assume an ordering of state and ambiguity effects, so we do not show such a corresponding Haskell program.

As could be noticed from our descriptions, these programs do not offer any insight in the semantics of operations. Instead, the program is only concerned with the syntax of operations, i.e. which types of arguments need to be passed to operations and what types of values are returned. Instead, semantics of operations are given only by the handlers of those operations. For instance, we could handle the ambiguity effect in many different ways, such as those in figure 3.4.

Combining programs and handlers is done through the **handle** ... **with**... construct. Suppose the *flipAndPut* program and *hAmbBoth1* handler are available then **handle** *flipAndPut* **with** *hAmbBoth1* would fully take care of the ambiguity effect and thus all invocations of the flip operation within the *flipAndPut* program. Finally, let us consider a handler for the state effect called *hSt*, using this we would be able to fully handle all operations and retrieve a value result. We give an implementation for *hSt* below:

```
let hSt = handler {
  return x ↦ fun s ↦ (s, x),
```

**let** *hAmbConstant* =
  **fun** $b \mapsto$ **handler** {
    **return** $x \mapsto x$,
    flip $(\_, k) \mapsto k\ b$}
**in** ...

**let** *hAmbBoth1* = **handler** {
  **return** $x \mapsto x : [\,]$,
  flip $(\_, k) \mapsto k$ **true** ++ $k$ **false**}
**in** ...

**let** *hAmbBoth2* = **handler** {
  **return** $x \mapsto x : [\,]$,
  flip $(\_, k) \mapsto k$ **false** ++ $k$ **true**}
**in** ...

**Figure 3.4:** Three different handlers for the ambiguity effect, offering three different semantics for flip. From left-to-right these are always using either **true** or **false**, first trying **true** then trying **false** and concatenating lists of results, and finally first trying **false**, then trying **true** to do the same.

  put $(s, k) \mapsto$ **fun** $\_ \mapsto (k\ ())\ s$,
  get $(\_, k) \mapsto$ **fun** $s \mapsto (k\ s)\ s$}

The handler uses a function to pass along state, similar to how the state monad in Haskell keeps and modifies its state. In the return-case, a state-passing function simply wraps the result value and the resulting value is a pair of the state and result value (corresponding to the *pure*/**return** implementation of the state monad). The put operation is implemented by ignoring the function parameter of the state passing function and continue with the new state when applying the continuation result. The get operation continues with the same state, but passes that state to the continuation call, so that the state becomes the result of **op-call** get ().

Semantics

Next, we describe the semantics of algebraic effect handling precisely using reduction rules in the style of Felleisen and Hieb [26]. This requires two constructions: evaluation contexts to decide what needs to be captured for evaluation and reduction rules to describe how to reduce a captured context. We show the evaluation contexts (figure 3.6) and a subset of the reduction rules (figure 3.5) for ***Deep***. We base these rules on the reduction rules presented by Leijen et al [56]. The four rules shown are as follows:

1. Applying a function. We do this by filling in the replacing all occurrences of $x$ within the function body $e$ with the value $v$ the function is applied to.
2. The desugaring of **do**-syntax. In this language, we only use **do** as a syntactical sugar over many consecutive **let**-expressions. If no name for the binding is given, we simply use the name '_', instead.
3. Returning a value from a handler. Whenever the expression inside a **handle** is fully evaluated, the wrapping **handle**-expression is evaluated by applying the **return**-function defined by the **handler**.
4. Handling an operation within a **handle**-block. This makes use of the $X_{op}$ context to capture all expressions surrounding an **op-call**-expression wtihin a **handle**. The **op-call** operation is matched with an operation implementation in the **handler** and the operation argument and a continuation are supplied. The continuation is constructed by propagating the **handle**-expression, surrounding the $X_{op}$ context with the operation result in it.

We only show these four rules as they represent the essence of our model language. We leave out let-application - which is very similar to function application - and all other more common language features. We do this to leave the focus of this chapter on effects and handlers.

## 3.2. Step 1: A Model Interpreter

At this point, we have formal descriptions for the syntax and semantics given in figure 3.1, and figure 3.5 and figure 3.6, respectively. To start transforming interpreters, we must implement our first small-step interpreter based on these descriptions.

| | | | |
|---|---|---|---|
| $\mathcal{C}$ | ::= | [] | *Context hole* |
| | $\mid$ | $(\mathcal{C}\ e)\ \mid\ (v\ \mathcal{C})$ | *Application* |
| | $\mid$ | **let** $x$ = $\mathcal{C}$ **in** $e$ | *Let* |
| | $\mid$ | **if** $\mathcal{C}$ **then** $e$ **else** $e$ | *If-then-else* |
| | $\mid$ | ... | |
| | $\mid$ | **op-call** $op\ \mathcal{C}$ | *Op-calls* |
| | $\mid$ | (**with** $h$ **handle** $\mathcal{C}$) | *Handling* |
| $X_{op}$ | ::= | [] | *Context hole* |
| | $\mid$ | $(X_{op}\ e)\ \mid\ (v\ X_{op})$ | *Application* |
| | $\mid$ | **let** $x$ = $X_{op}$ **in** $e$ | *Let* |
| | $\mid$ | **if** $X_{op}$ **then** $e$ **else** $e$ | *If-then-else* |
| | $\mid$ | ... | |
| | $\mid$ | **op-call** $op\ X_{op}$ | *Op-calls* |
| | $\mid$ | (**with** $h$ **handle** $X_{op}$) if $op \notin X_{op}$ | *Handling* |

**Figure 3.5:** Evaluation contexts of ***Deep***

$$\mathcal{C}[((\textbf{fun}\ x \mapsto e)\ v)] \longrightarrow \mathcal{C}[(e\ [x/v])]$$

$$\mathcal{C}[(\textbf{do}\ x_1 \leftarrow e_1;\ ...;\ \ x_n \leftarrow e_n; e_r)] \longrightarrow \mathcal{C}[(\textbf{let}\ x_1 = e_1\ \textbf{in}\ (...\ \ \textbf{in}\ (\textbf{let}\ x_n = e_n\ \textbf{in}\ e_r)\ \ ...))]$$

$$\mathcal{C}[(\textbf{with}\ (\textbf{handler}\ \{\textbf{return}\ x_r \mapsto e_r,\ ...\})\ \textbf{handle}\ v)] \longrightarrow \mathcal{C}[(e_r\ [x_r/v])]$$

$$\mathcal{C}[(\textbf{with}\ h\ \textbf{handle}\ X_{op}[(\textbf{op-call}\ op_i\ e_v)])] \longrightarrow \mathcal{C}[(e_{op}\ [x/v, (k/\textbf{fun}\ y \mapsto (\textbf{with}\ h\ \textbf{handle}\ X_{op}[y]))])]$$

where

$$op_i\ (x, k) \mapsto e_{op}\ \in\ h$$

**Figure 3.6:** A subset of the reduction rules for ***Deep***.

| | |
|---|---|
| **data** *Value* | |
| = *LambdaV String Expr* | **fun** *id* $\mapsto$ *e* |
| \| *IntV Int* | *nat* |
| \| *BoolV Bool* | **true** $\vee$ **false** |
| \| *UnitV* | () |
| \| *PairV Value Value* | $(v, v)$ |
| \| *NilV* | [ ] |
| \| *ConsV Value Value* | $v : v$ |

*op* (*id*, *id*) $\mapsto$ *e*
**data** *OpI* = *OpI String String Expr*
**return** *id* $\mapsto$ *e*
**data** *ReturnI* = *ReturnI String Expr*
**handler** $\{$**return** $..,op_1 ..,..,op_n ..\}$
**data** *Handler* = *Handler* [*OpI*] *ReturnI*
**data** *UnOpOperator* = *Fst* | *Snd*
**data** *BinOpOperator* = *Add* | *Mul* | *Concat*

| | |
|---|---|
| **data** *Expr* | |
| = *Var String* | *id* |
| \| *App Expr Expr* | (*e e*) |
| \| *Let String Expr Expr* | **let** *id* = *e* **in** *e* |
| \| *OpCall String Expr* | **op-call** *op e* |
| \| *Handle Handler Expr* | **with** *h* **handle** *e* |
| \| *Lit Value* | *v* |
| \| *Closed Value* | |
| \| *Pair Expr Expr* | $(e, e)$ |
| \| *Cons Expr Expr* | $e : e$ |
| \| *UnOp UnOpOperator Expr* | *uop e* |
| \| *BinOp Expr BinOpOperator Expr* | *e bop e* |

**do** $(\{id \leftarrow\}\ e)^*\ \ e$
`do` :: [(*Maybe String*, *Expr*)] $\rightarrow$ *Expr* $\rightarrow$ *Expr*
`do` [ ] *res* = *res*
`do` ((*Nothing*, *eb*) : *t*) *res* = *Let* "_" *eb* (`do` *t res*)
`do` ((*Just nm*, *eb*) : *t*) *res* = *Let nm eb* (`do` *t res*)

**Figure 3.7:** ***Deep*** syntax modeled in Haskell.

Kicking off, we define data types representing values, unary and binary operations, and expressions in figure 3.7. These data types directly capture the syntax as described in figure 3.1. We add only one data constructor for interpreting purposes: *Closed*. This constructor is used in the same ways as *Lit* for value literals is used, except for being closed under substitution. When binding a name - through, for instance, function application -, we replace all unshadowed occurrences of that variable name with the bound value wrapped in *Closed*. This process does not enter *Closed* expressions, however, to make sure substitutions are capture-avoiding[1].

**data** *PotentialRedex*
  = *PRBeta*
    *String Expr Value*
  $\mathcal{C}[((\textbf{fun}\ x \mapsto e)\ v)]$
  | *PRHandleReturn*
    *String Expr Value*
  $(\mathcal{C}[(\textbf{with}\ (\textbf{handler}\ \{\textbf{return}\ x_r \mapsto e_r, ...\}))]$
    $\textbf{handle}\ v)$
  | *PRHandleOp*
    *Handler* (*Expr* → *Expr*)
    *String String Expr Value*
  $\mathcal{C}[(\textbf{with}\ h\ \textbf{handle}\ X_{op}[(\textbf{op-call}\ op_i\ e_v)])]$
  | *PRError String*
  | ...

**(a)** Potential Redexes representing the left-hand-side of a reduction rule.

*contract* :: *PotentialRedex* → *ContractumOrError*
*contract* (*PRBeta x e v*) =
  *Contractum* (*subst x v e*)
  $\longrightarrow\ \mathcal{C}[(e\ [x/v])]$
*contract* (*PRHandleReturn* $x_r$ $e_r$ *v*) =
  *Contractum* (*subst* $x_r$ *v* $e_r$)
  $\longrightarrow\ \mathcal{C}[(e_r\ [x_r\ /\ v])]$
*contract* (*PRHandleOp h* $X_{op}$ *x k* $e_{op}$ *v*) =
  *Contractum* (*subst x v*
      (*subst k* (*LambdaV* "y" (*Handle h* ($X_{op}$ (*Var* "y"))))
       $e_{op}$))
  $\longrightarrow\ \mathcal{C}[(e_{op}\ [x\ /\ v, k\ /\ (\textbf{fun}\ y \mapsto \textbf{with}\ h\ \textbf{handle}\ X_{op}[y])])]$
    where $op_i\ (x, k) \mapsto e_{op}\ \in\ h$
*contract* (*PRError err*) = *Error err*
*contract* ... = ...

**(b)** Contraction representing the function from left-hand to right-hand side of a reduction rule.

**Figure 3.8**

The next step is to implement the reduction rules from figure 3.6. First, we define a data type to represent the parts of expressions that are captured and used by a reduction rule, as seen on the left-hand side of a reduction rule. We call this data type *PotentialRedex*, as can be seen in figure 3.8a. Next, we apply the reduction in a *contract* function (figure 3.8b). This function takes the left-hand side of a reduction rule as a *PotentialRedex* and contracts it to an expression representing the right-hand side of a reduction rule. We list the relevant parts of the reduction rules beside each constructor or case in gray. Substitutions are done through the *subst* :: *String* → *Value* → *Expr* → *Expr* function and the definition of *ContractumOrError* is a simple data type either storing a contracted expression (*Contractum*) or an error. Finally, notice that we store the context $X_{op}[\cdot]$ as a function *Expr* → *Expr*. This is because we only use this captured context to reconstruct an expression with a value filled in in the context hole.

[1] Capture-avoiding substitutions prevent name-capture when a previously free variable would be bound. For instance ((**fun** *y* (**fun** *x y*)) *x*) might be substituted to get **fun** *x x*, rather than an error because the outer *x* is unbound.

**data** *Context*
  = *CEmpty*  $[\,]$
  | *CAppL Context Expr*  $(\mathcal{C}\ e)$
  | *CAppR Value Context*  $(v\ \mathcal{C})$
  | *COp String Context*  **op-call** *op* $\mathcal{C}$
  | *CHandle Handler Context*  **handle** *h* **with** $\mathcal{C}$
  | ...

**(a)** ***Deep*** evaluation contexts represented as a Haskell data-type.

*decompose_expr* :: *Context* $\rightarrow$ *Expr* $\rightarrow$ *ValueOrDecomposition*
*decompose_expr c* (*App e1 e2*) =
  *decompose_expr* (*CAppL c e2*) *e1*
  $(\mathcal{C}\ e)$
*decompose_expr c* (*Handle h eb*) =
  *decompose_expr* (*CHandle h c*) *eb*
  **with** *h* **handle** $\mathcal{C}$
*decompose_expr c* (*Lit v*) =
  *decompose_context v c*
*decompose_expr* ... = ...

**(b)** Decomposing expressions into contexts.

**Figure 3.9**

*decompose_context* :: *Value* $\rightarrow$ *Context* $\rightarrow$ *ValueOrDecomposition*
*decompose_context v CEmpty* = *VODValue v*
  $[\,]$
*decompose_context v* (*CAppL c e2*) = *decompose_expr* (*CAppR v c*) *e2*
  $(v\ \mathcal{C})$
*decompose_context v* (*CAppR* (*LambdaV x e*) *c*) = *VODDec* (*PRBeta x e v*) *c*
  $\mathcal{C}[((\textbf{fun}\ x \mapsto e)\ v)]$
*decompose_context* _ (*CAppR v1 c*) =
  *VODDec* (*PRError* (`"Cannot apply non-function value: "` <> *show v1*)) *c*
  $\mathcal{C}[(v\ v)]$
*decompose_context v* (*COp op c*) = *recompose_op op v id c*
  $\mathcal{C}[(\textbf{op-call}\ op\ v)]$
*decompose_context v* (*CHandle* (*Handler* _ (*ReturnI x e*)) *c*) = *VODDec* (*PRHandleReturn x e v*) *c*
  $\mathcal{C}[(\textbf{with}\ h\ \textbf{handle}\ v)]$

**Figure 3.10:** Further decomposing contexts after bottoming to a value.

Finally, we need a way to search through expressions to find the evaluation context hole. This means we search through an expression to see whether it matches a context and, if not, we recurse down the left-most part that requires further evaluation to continue the search. The functions we use to do this are *decompose_expr* (figure 3.9b) and *decompose_context* (figure 3.10), which both make use of the data-type encoding of evaluation contexts written as *Context* (figure 3.9a). The *decompose_expr* function tries to find the left-most hole in an expression whereas *decompose_context* takes a found value, fills it in in the place of the current context hole and continues the search in the next left-most hole. In either case, if the context matches a left-hand side of a reduction rule, the rule is applied by returning a decomposition with the *VODDec* constructor of *ValueOrDecomposition*. When an expression directly reduces to a value, it is returned with the *VODValue* constructor.

As an example, let us follow the decomposition of **with** *h* **handle** 5, with some arbitrary handler *h*. When applying *decompose_expr* to some prior context and this expression, we see that *decompose_expr* is first recursively called with context (**with** *h* **handle** $\mathcal{C}$) and expression 5. This results in a value 5, passed to *decompose_context*. Here we find that the *CHandle* case is reached, causing the function to correctly signal a decomposition is found in (**with** *h* **handle** *v*). Similarly, folloding the decomposition of **with** *h* **handle** (**op-call** *op* 4), where *h* handles *op*, we find that we reach the *decompose_context* case for *COp*. This case is a little special as we will need to retrace our steps to find $X_{op}$. To do this, we call the *recompose_op* function.

```
recompose_op :: String → Value → (Expr → Expr) → Context → ValueOrDecomposition
recompose_op op v _ CEmpty =
  VODDec (PRError ("Cannot handle free op: " <> op <> "(" <> show v <> ")")) CEmpty
recompose_op op v x_op c@(CHandle h@(Handler ops _) c′) =
  case find (λ(OpI op′ _ _ _) → op ≡ op′) ops of
    Just (OpI _ x k e) → VODDec (PRHandleOp h x_op x k e v) c′
    Nothing →
      let (_, f) = recompose_ss c
      in recompose_op op v (f.x_op) c′
recompose_op op v x_op c =
  let (c′, f) = recompose_ss c
  in recompose_op op v (f.x_op) c′
```

**Figure 3.11:** Recomposing the $X_{op}$ context up until the nearest handler handling an operation

An impression of the *recompose_op* function is given in figure 3.11. This function takes the name of the operation and argument value of an **op-call** *op v*, the built context $X_{op}$ as a function mapping expressions, and the surrounding context $\mathcal{C}[\cdot]$. This function looks at $\mathcal{C}[\cdot]$ to find the nearest **handle**-expression handling the **op-call** with an appropriate **handler**. If it does not find such a handler, it adds the current context to $X_{op}$ by recomposing it to an *Expr* → *Expr* function and composing it with the current $X_{op}$ function. It calls the *recompose_ss* function to do this, which recomposes a single step, meaning it shallowly turns a context into its containing context and an *Expr* → *Expr* function representing that shallow context wrapping.

```
recompose_ss :: Context → (Context, Expr → Expr)
recompose_ss (CAppL c e2) = (c, λt → App t e2)
recompose_ss (CAppR v1 c) = (c, λt → App (Closed v1) t)
recompose_ss (COp op c) = (c, λt → OpCall op t)
recompose_ss (CHandle h c) = (c, λt → Handle h t)
recompose_ss ... = ...

recompose :: Context → Expr → Expr
recompose CEmpty t = t
recompose c t = let (c′, f) = recompose_ss c in recompose c′ (f t)
```

**Figure 3.12:** Single-step and full recomposition of contexts into expressions.

In figure 3.12 we implement single-step and full recomposition of contexts. Every single step of recomposition takes a context and produces the inner context and a function as its result. The function fills in the context hole with its expression parameter, turning a single layer of the context back into an expression. Notice that this function has to be partial, as we cannot find an inner context for *CEmpty*. This partialness could be resolved with a *Maybe*, but we would rather keep *recompose* and *recompose_op* easier to read this way. *recompose* applies this single-step function and continues recursively until an empty context is encountered.

```
decompose :: Expr → ValueOrDecomposition
decompose = decompose_expr CEmpty

iterate0 :: ValueOrDecomposition → Result
iterate0 (VODValue v) = Result v
iterate0 (VODDec pr c) = case contract pr of
  Contractum e → iterate0 (decompose (recompose c e))
  Error err → Wrong err

normalise0 :: Expr → Result
normalise0 e = iterate0 (decompose e)
```

**Figure 3.13:** The first normalisation function for ***Deep***.

Finally, we connect all these pieces together in the *iterate0* and *normalise0* functions in figure 3.13 to do the following:

1. Take an expression and decompose it into either a value or a decomposition.
2. If we derive a value, we are done evaluating and it can be returned as a *Result*.
3. If we derive a decomposition, we contract its containing *PotentialRedex* and find the next decomposition of the recomposed expression.

## 3.3. Step 2: Apply Transformations to Derive Denotational Interpreter

We obtained a small-step interpreter in the style of Danvy's normalising interpreter [20]. We now transform this small-step interpreter, following the steps from [20], into a big-step interpreter of the same language. During every step, we re-number our functions with a new number. So, for instance, *iterate0* becomes *iterate1* in the first step. We thus also change references to functions with every step to a new postfix number. The steps to derive a big-step interpreter range steps 1 up to 7:

1. Refocusing decompositions
2. Inlining the contraction function
3. Lightweight fusion
4. Compressing corridor transitions
5. Renaming transition functions and flattening configurations
6. Refunctionalisation
7. Back to direct style

In the following subsections, we make notes on performing steps 4, 6, and 7 of this process. Besides these steps, we follow the transformations from [20] to the letter, exactly as also shown in Chapter 2. These steps demonstrate some maybe less obvious caveats required for effects and handlers.

Finally, we derive a denotational interpreter from the big-step interpreter by inversely applying closure conversion.

### 3.3.1. To Denotational Step 4: Compressing corridor transitions

In this step, we check find and compress 'corridor transitions'. This means we will look at any function application and check whether there are multiple computational paths for that application. If there is only one, we fill in this one path of computation in the place of the application and we might be able to remove some transitions from our interpreter entirely.

As per usual, we see that the *iterate4* function largely consists of dead clauses after this process, as all calls of the form *iterate4* (*VODDec*...) are considered corridor transitions. However, in addition to the usual transformations, we add one of our own in this step. We add it here to be able to demonstrate step 6 more clearly.

The transformation we add is to consider the only call to *refocus_op3* an opportunity for reducing the number of functions we need to deal with. We will merge *refocus_op3* into *refocus_context3*. We do so by introducing a data type that holds either the *Value* argument to *refocus_context3*, or the operation describing arguments to *refocus_op3*. We call this data structure *ValueOrOp*:

$$\begin{array}{l}\textbf{data}\ \mathit{ValueOrOp}\\ \quad = \mathit{NoOp}\ \mathit{Value}\\ \quad \mid \mathit{Op}\ \mathit{String}\ \mathit{Value}\ (\mathit{Expr} \to \mathit{Expr})\\ \mathit{refocus\_context4} :: \mathit{ValueOrOp} \to \mathit{Context} \to \mathit{Result}\end{array}$$

We change the type of *refocus_context3* to receive a *ValueOrOp*, rather than a *Value*. We adjust the existing cases of *refocus_context3* to match a *NoOp v* instead of just a *Value v*. We then merge the three cases of *refocus_op3* into this function and capture its arguments except for the context with the *Op* constructor. Finally, whenever a call to *refocus_op3* was made, we wrap its first three arguments in an *Op* and whenever a call to *refocus_context3* was made, we wrap its value argument in *NoOp*.

Consider the case for *COp*. Here, we change the call to *refocus_op3* to target *refocus_context4* instead and we wrap matches and values that are passed as arguments in *Op* or *NoOp*:

$$\begin{array}{l}\mathit{refocus\_context3}\ v\ (\mathit{COp}\ \mathit{op}\ c) = \mathit{refocus\_op3}\ \mathit{op}\ v\ \mathit{id}\ c\\ \quad \longrightarrow\\ \mathit{refocus\_context4}\ (\boxed{\mathit{NoOp}}\ v)\ (\mathit{COp}\ \mathit{op}\ c) = \boxed{\mathit{refocus\_context4}}\ (\boxed{\mathit{Op}}\ \mathit{op}\ v\ \mathit{id})\ c\end{array}$$

Finally, let us look at how the cases for *refocus_op3* are transformed. The generic case for recomposing single layers of the context is easily adjusted by retargeting the calls to *refocus_op3* and adding *Op* constructors. The case for handling is just as easily adjusted to fit in *refocus_context4*.

$$\begin{array}{l}\mathit{refocus\_op3}\ \mathit{op}\ v\ \mathit{x\_op}\ c =\\ \quad \textbf{let}\ (c', f) = \mathit{recompose\_ss}\ c\\ \quad \textbf{in}\ \mathit{refocus\_op3}\ \mathit{op}\ v\ (f.\mathit{x\_op})\ c'\\ \quad \longrightarrow\\ \boxed{\mathit{refocus\_context4}}\ (\boxed{\mathit{Op}}\ \mathit{op}\ v\ \mathit{x\_op})\ c =\\ \quad \textbf{let}\ (c', f) = \mathit{recompose\_ss}\ c\\ \quad \textbf{in}\ \boxed{\mathit{refocus\_context4}}\ (\boxed{\mathit{Op}}\ \mathit{op}\ v\ (f.\mathit{x\_op}))\ c'\end{array}$$

### 3.3.2. To Denotational Step 6: Refunctionalisation

After renaming *refocus_context* to *continue* and *refocus_expr* to *eval* and isolating unary and binary operation application into *applyUnOp5* and *applyBinOp5*, we are now tasked with merging *continue* into the *eval* function. The evaluation function gets the following signature:

$$\begin{array}{l}\mathit{eval6}\ ::\ \mathit{Expr}\\ \qquad \to (\mathit{Value} \to \mathit{Result})\\ \qquad \to (\mathit{String} \to \mathit{Value} \to (\mathit{Expr} \to \mathit{Expr}) \to \mathit{Result})\\ \qquad \to \mathit{Result}\end{array}$$

In this signature, we have turned *continue5* into two higher-order functions, each of their parameters determined by a case in *ValueOrOp*. To implement this function, we perform the usual refunctionalisation as done by Danvy, but we need to add two continuation functions per recursive call. This process involves filling in and simplifying computations done by *recompose_ss* for the second (op-finding) continuation. We show, for instance, what it looks like to evaluate an operation call in this refunctionalised interpreter:

```
eval5 c (OpCall op e) = eval5 (COp op c) e
 ...
continue5 (NoOp v) (COp op c) = continue5 (Op op v id) c
 ...
recompose_ss (COp op c) = (c, λt → OpCall op t)
   ⟶
eval6 (OpCall op e) kv ko =
  eval6 e (λv → ko op v id)
    (λop′ v x_op → ko op′ v ( (λt → OpCall op t) .x_op))
```

We see that two continuations are captured instead of a context: *kv* for value continuations and *ko* for operation continuations. The operation call first evaluated its argument. If a value is resulted from evaluation, the operation continuation is called with this operation as its arguments. If, instead, the argument requires another operation to be evaluated, the operation is passed along the operation continuation, wrapping the argment expression with the recomposition of the current operation. The highlighted part corresponds to the case for *recompose_ss* of an operation call context.

Taking a look at the case for handling operations, we see the same transformations. We combine all cases for handling within *eval5*, *continue5*, and *recompose_ss* to create the case for *eval6*. Note that we can link back every part of the resulting code to one of the four functions listed before the arrow. For instance, we highlight the part that came from unfolding *recompose_ss* again.

```
eval5 c (Handle h eb) = eval5 (CHandle h c) eb
 ...
continue5 (NoOp v) (CHandle (Handler _ (ReturnI x e)) c) = eval5 c (subst x v e)
 ...
continue5 (Op op v x_op) c@(CHandle h@(Handler ops _) c′) =
  case find (λ(OpI op′ _ _ _) → op ≡ op′) ops of
    Just (OpI _ x k e) →
      eval5 c′ (subst x v
               (subst k (LambdaV "y" (Handle h (x_op (Var "y"))))
                 e))
    Nothing →
      let (_, f) = recompose_ss c
      in continue5 (Op op v (f.x_op)) c′
 ...
recompose_ss (CHandle h c) = (c, λt → Handle h t)
   ⟶
eval6 (Handle h@(Handler ops (ReturnI xr er)) eb) kv ko =
  eval6 eb
    (λbv → eval6 (subst xr bv er) kv ko)
    (λop v x_op →
      case find (λ(OpI op′ _ _ _) → op ≡ op′) ops of
        Just (OpI _ x k e) →
          eval6 (subst x v
                  (subst k (LambdaV "y" (Handle h (x_op (Var "y"))))
                    e))
                kv ko
        Nothing → ko op v ( (λt → Handle h t) .x_op))
```

### 3.3.3. To Denotational Step 7: Back to direct style

Finally, to convert back to direct style, we use the same trick as used in step 4. We realise the result of evaluating an expression in ***Deep*** is either a value or an unhandled operation. This realisation does mean that the result of evaluating an expression can thus be incomplete if no handler is inserted to handle an operation. We do, however, assume that a type-checker would be in place to check that the property of having no unhandled operations is enforced. Indeed, we type our expressions later to enforce this property (see: Section 3.5.

To implement a direct style interpreter, we thus add a case to the *Result* data type. This result type can be seen in figure 3.14. The constructor *Op7 String Value* (*Expr* → *Expr*) represents unhandled operations with exactly the same types as the operation continuation from the last step.

```
data Result7
  = NoOp7 Value
  | Op7 String Value (Expr → Expr)
  | Wrong7 String
```

**Figure 3.14:** Result type for direct style evaluation.

When turning *eval6* into *eval7*, we translate every call to the value continuation to a *NoOp7* result and every call to an operation continuation to an *Op7* result. We show this transformation in the evaluation function for *OpCall* below. Specific to *OpCall* is that it has the only value-continuation not ending with a *NoOp7* result. Instead, it produces the unhandled operation it represents.

```
eval6 (OpCall op e) kv ko =
  eval6 e
    (λv → ko op v id)
    (λop′ v x_op → ko op′ v ((λt → OpCall op t).x_op))
  ⟶
eval7 (OpCall op e) =
  case eval7 e of
    NoOp7 v → Op7 op v id
    Op7 op v x_op → Op7 op v ((λt → OpCall op t).x_op)
```

Finally, we take a look at the evaluation of handling. The same transformations are applied on it to get the following evaluation case. However, the result of evaluating the body of a handle block is given somewhat special treatment. Instead of simply re-wrapping unhandled operations, they are checked and handled if possible:

```
eval7 (Handle h@(Handler ops (ReturnI xr er)) eb) =
  case eval7 eb of
    NoOp7 bv → eval7 (subst xr bv er)
    Op7 op v x_op →
      case find (λ(OpI op′ _ _ _) → op ≡ op′) ops of
        Just (OpI _ x k e) →
          eval7 (subst x v
            (subst k (LambdaV "y" (Handle h (x_op (Var "y")))) e))
        Nothing → Op7 op v ((λt → Handle h t).x_op)
    Wrong7 err → Wrong7 err
```

### 3.3.4. To Denotational Step 8: From Big-Step to Denotational

We use the inverse closure conversion or lifting of function arguments as described in Section 2.1.10. We need to not only apply this conversion to lambdas and lets, but also to the operation implementations and return implementations for effect handlers. The continuation argument for operation implementations is converted to a higher-order *Value8* → *Expr8* argument. The following code shows some of the changes made to the data types for our interpreter and the interpreting function itself:

```
data Value8
  = LambdaV8 (Value8 → Expr8)
  | ...
data OpI8 =
  OpI8 String
    (Value8 → (Value8 → Expr8) → Expr8)
data Handler8 =
  Handler8 [OpI8] (Value8 → Expr8)
data Expr8
  = Let8 Expr8 (Value8 → Expr8)
  | ...
data Result8
  = NoOp8 Value8
  | Op8 String Value8 (Expr8 → Expr8)
```

```
eval8 :: Expr8 → Result8
eval8  ...
eval8 (OpCall8 op e) =
  case eval8 e of
    NoOp8 v → Op8 op v id
    Op8 op v x_op → Op8 op v ((λt → OpCall8 op t).x_op)
eval8 (Handle8 h@(Handler8 ops ret) eb) =
  case eval8 eb of
    NoOp8 bv → eval8 (ret bv)
    Op8 op v x_op →
      case find (λ(OpI8 op′ _) → op ≡ op′) ops of
        Just (OpI8 _ body) →
          eval8 (body v (λy → (Handle8 h (x_op (Lit8 y)))))
        Nothing → Op8 op v ((λt → Handle8 h t).x_op)
eval8  ...
```

We see that, instead of substituting arguments into the bodies of lambdas, we now directly call a function, passing the argument into it instead.

## 3.4. Step 3: Lettify Pure Computations

In this step, we remove all pure computations except for *Let* and *Handle* from the expression tree. This means binary expressions, *App*-expressions, etc. are no longer a part of *Expr* at the end of this step. We do this by extracting the evaluation for every pure expression into its own smart-constructor like function. We start by pulling all cases for evaluation into a separate function for each:

```
app8 :: Expr8 → Expr8 → Result8
app8 ef ea =
  case eval8 ef of
    NoOp8 vf → case eval8 ea of
      NoOp8 va → case vf of
        LambdaV8 body → eval8 (body va)
        _ → error ("Cannot apply non-function value: " <> show vf)
      Op8 op v x_op → Op8 op v ((λt → App8 (Lit8 vf) t).x_op)
    Op8 op v x_op → Op8 op v ((λt → App8 t ea).x_op)
```

We take these functions, and 'lettify' each to find the desired form. For a lack of a better word, we use 'lettify' to say that we find a structure that only uses *Let*-expressions to perform boiler-plate constructions. The boiler-plate constructions here are the cases for reconstructing the surrounding context of an unhandled operation. This reconstruction is generalised by the evaluation case for *Let*-expressions[2]. In other words, evaluating this 'lettified' expression should always yield the same behaviour as the evaluating the original expression would have[3]. Such a 'lettified' function for function application is found below:

[2]This is also why the freer monad abstraction in the end works so well. That *Let* evaluation is a good generalisation for other pure computations foreshadows its relation to monadic bind.

[3]Unhandled operations would contain a slightly different context, but in behaviour, the expressions contained in this context are also equivalent.

```
app8′ :: Expr8 → Expr8 → Expr8
app8′ e1 e2 =
  Let8 e1 (λv1 →
    Let8 e2 (λv2 → case v1 of
      LambdaV8 body → body v2
      _ → error ("Cannot apply non-function value: " <> show v1)))
```

This re-expresses the application expression in terms of only *Let*-expressions. This step requires that *eval8* (*app8* $e_1$ $e_2$) = *eval8* (*App8* $e_1$ $e_2$) for arbitrary $e_1$ and $e_2$. Notice that with this abstraction, we fully remove the dependency on an expression data constructor.

In figure 3.16a, we show the new data structure to represent expressions and the new smart constructor-like function for constructing application expressions. The new expression tree no longer needs to contain constructors for pure computations such as function application, instead these computations are now embedded. In figure 3.16b, we show the full evaluation function at this point. We see that only *Let*-expression evaluation requires the *check_result9* function now as all other uses have been eliminated. Indeed all cases remain the same as last step, but we have removed the need for many cases of the evaluation function.

```
data Expr9
  = Let9 Expr9 (Value9 → Expr9)
  | OpCall9 String Value9
  | Handle9 Handler9 Expr9
  | Lit9 Value9
app9 :: Expr9 → Expr9 → Expr9
app9 e1 e2 =
  Let9 e1 (λv1 →
    Let9 e2 (λv2 → case v1 of
      LambdaV9 body → body v2
      _ → error ("[...]" <> show v1)))
```

**(a)** Full expression type after 'lettifying' all other expressions and smart constructors for op-calls and lists.

```
eval9 :: Expr9 → Result9
eval9 (Let9 ev body) =
  check_result9 (eval9 ev)
    (λv → eval9 (body v))
    (λt → Let9 t body)
eval9 (OpCall9 op v) = Op9 op v Lit9
eval9 (Handle9 h@(Handler9 ops ret) eb) =
  case eval9 eb of
    NoOp9 bv → eval9 (ret bv)
    Op9 op v x_op →
      case find (λ(OpI9 op′ _) → op ≡ op′) ops of
        Just (OpI9 _ body) →
          eval9 (body v (λy → (Handle9 h (x_op y))))
        Nothing → Op9 op v ((λt → Handle9 h t).x_op)
eval9 (Lit9 v) = NoOp9 v
```

**(b)** Full evaluation of expressions after 'lettifying'.

## 3.5. Step 4: Add Intrinsic Typing

To progress further, it is most illustrative and even necessary for the final typed interpreter to introduce the types we have so far implicitly enforced. To do this, we redefine value and expression types as GADTs, allowing us to intrinsically type values and expressions. We start with redefining the *Value*-type as *Value10 a*, where *a* represent the Haskell type that a *Value*-constructor represents. In the following, we see, for instance, lambda values types as a function from *Value10 s* to *Expr10 sig a*, where this expression type is an expression containing some unhandled operations described with *sig* and, under evaluation, might result in a value of type *a*. For example, we see that *NilV10* is a representation of the Haskell type $[x]$, and *BoolV10* is encoding a *Bool*.

```
data Value10 a where
  LambdaV10 :: (Value10 x → Expr10 sig a) → Value10 (Value10 x → Expr10 sig a)
  IntV10    :: Int → Value10 Int
  BoolV10   :: Bool → Value10 Bool
  UnitV10   :: Value10 ()
  PairV10   :: Value10 x → Value10 y → Value10 (x, y)
```

```
NilV10   :: Value10 [x]
ConsV10  :: Value10 x → Value10 [x] → Value10 [x]
```

To type effects and handlers, we use Data Types à la Carte [76]. This means that we encode all effects possibly left unhandled in an expression as a type-parameter *sig* :: $* \to *$. We show the encoding of handlers and expressions in figure 3.17. Here, we see that type parameters are added to every usage of a value or expression type, making the implicit typing rules we have enforced up until now explicit. For instance, *Let*-expressions take an arbitrary expression of type *x*, and a function taking a value of type *x* and producing a new expression of the same type as the final *Let*-expression. More complexly, handling an operation takes a handler that removes an effect *eff* from the signature of the expression and transforms its body result type from *a* to the answer type modification of *a*: *w*.

```
data Handler10 eff r a w where
  Handler10 :: (forall x.eff x → (Value10 x → Expr10 r w) → Expr10 r w)
               → (Value10 a → Expr10 r w)
               → Handler10 eff r a w
data Expr10 sig a where
  Let10     :: Expr10 sig x → (Value10 x → Expr10 sig a) → Expr10 sig a
  OpCall10  :: sig a → Expr10 sig a
  Handle10  :: Handler10 eff r a w → Expr10 (eff :++: r) a → Expr10 r w
  Lit10     :: Value10 a → Expr10 sig a
```

**Figure 3.17:** Intrinsically typed handlers and expressions.

To complete this transformation, we need to adjust the evaluation function and the result type. The *Result* type is now fully typed, making the type-based errors nearly impossible. We thus remove the previously required *Wrong* constructor to favour raising an exception in Haskell directly. Finally, the evaluation function is changed only for the case for *Handle*-expressions. Matching operations to operation implementations present in a handler used to be done with a *find*, but is now done using the *Inl* and *Inr* constructors. When an operation is part of the handler's operation set, the effect belonging to that operation is left-most in the effect signature. This means that any operation wrapped in *Inl* is part of the effect that is currently handled.

```
data Result10 sig a where
  NoOp10 :: Value10 a → Result10 sig a
  Op10 :: sig x → (Value10 x → Expr10 sig a) → Result10 sig a
eval10 :: Expr10 sig a → Result10 sig a
eval10 (Handle10 h@(Handler10 ops ret) eb) =
  case eval10 eb of
    NoOp10 bv → eval10 (ret bv)
    Op10 (Inl op) x_op → eval10 (ops op (λy → (Handle10 h (x_op y))))
    Op10 (Inr op) x_op → Op10 op ((λt → Handle10 h t).x_op)
...
```

## 3.6. Step 5: Generalise Values

In this step, we remove the *Value* type and generalise to allow, in theory, any type of value to inhabit the pure computation (*Lit*) branch of the expression tree. Instead of encoding Haskell values with this data type, we start using the Haskell values they represent directly. Generalising *Value* after adding intrinsic typing is very straight forward. We need to remove every occurrence of the *Value* type and we need to change smart constructors to directly use Haskells built-in value constructors for pairs and lists for instance. For example, the *Expr* tree is changed to no longer contain *Value*-types and the *pair* function now uses the Haskell constructor for pairs.

```
data Expr11 sig a where
  Let11 :: Expr11 sig x → (x → Expr11 sig a) → Expr11 sig a
  OpCall11 :: sig a → Expr11 sig a
  Handle11 :: Handler11 eff r a w → Expr11 (eff :++: r) a → Expr11 r w
  Lit11 :: a → Expr11 sig a
pair11 :: Expr11 sig x → Expr11 sig y → Expr11 sig (x, y)
pair11 e1 e2 =
  Let11 e1 (λv1 →
    Let11 e2 (λv2 → Lit11 (v1, v2)))
```

## 3.7. Step 6: Lettify Handling

There are only 4 constructors of the expression type left at this point. In this step and the next we eliminate *Handle* by moving its interpretation into a function, and we find that, after this move, we can merge two of the remaining constructors into one. This step involves writing a *handle*-function that correctly represents the interactions that a *Handle*-expression could have with all other expressions.

We again, make sure to abstract *handle* into a function that results in an expression by using *Let* to mimic the use of recursive calls to *eval*. In figure 3.18, the body of the handle is structurally matched to produce an expression that can directly be evaluated to yield a result. These cases are as follows:

- Case 1 corresponds to evaluating a *Handle* expression where the body is already a value and is thus immediately passed to the return function.
- Case 2 and 3 correspond to finding that the body evaluates to an unhandled operation and the op either being part of the operations handled by this handler (case 2) or not (case 3).
- Case 4 is an occurrence where a *Let*-expression harbours a literal in its argument, these can be immediately applied and the result can be further handled.
- Case 5 describes that, when *Let*-expressions are nested inside *Let*-expression arguments, the larger *Let* is normalised by moving the inner *Let* into the body of the outer *Let*.
- Case 6 ensures lone *OpCall* expressions are handled in the same way as a *Let*-expression with an *OpCall* in its argument and no meaningful body. This corresponds to *OpCall*s evaluating to an *Op* result with *Lit* as the initial recomposition function.

```
handle12 :: Handler12 eff r a w → Expr12 (eff :++: r) a → Expr12 r w
handle12 h@(Handler12 ops ret) eb = case eb of
  Lit12 bv → ret bv                                                          1
  Let12 (OpCall12 (Inl op)) body → ops op (λy → (handle12 h (body y)))      2
  Let12 (OpCall12 (Inr op)) body → Let12 (OpCall12 op) ((λt → handle12 h t).body)   3
  Let12 (Lit12 bv) body → handle12 h (body bv)                               4
  Let12 (Let12 e′ body′) body → handle12 h (Let12 e′ (λv′ → Let12 (body′ v′) body))   5
  OpCall12 op → handle12 h (Let12 (OpCall12 op) Lit12)                       6
```

**Figure 3.18:** The handle function representing the old *Handle* data constructor.

## 3.8. Step 7: Merge OpCall and Let

Finally, we see that the three constructors left have some extraneous parts to them. Specifically, expressions of the form *Let* (*Lit v*) *b* do little extra calculation. It would almost always make more sense to encode this as (*b v*) directly, without the need of an interpreter to do exactly that. Additionally, we take note that handling prefers to wrap occurrences of *OpCall* in a *Let*-expression to explicitly show what continuation should be passed to the operation implementation. Because of this, it makes sense to merge *OpCall*s and *Let*s into a single expression form, eliminating the possibility of having literals in the place of the *Let*-expression argument.

In figure 3.19 we show the new expression type and a replacement for constructing the old *Let*- and *OpCall*-expressions. The new *OpLet* constructor has the shape of *Let*, but only takes operations as its argument, thus representing *Let*-expressions that can only take (completed) *OpCall*s as their arguments. For the old *Let*-expressions, we introduce a **let** function. This function directly calls the let-body on literal values and propagates a let into the body of an *OpLet* otherwise, similar in function to cases 4 and 5 from figure 3.18. The *op* function receives a constructor for the operation and an argument expression. It produces the *OpLet* expression after first evaluating and embedding the operation argument in a similar fashion to case 6 from figure 3.18.

```
data Expr13 sig a where
  OpLet13 :: sig x → (x → Expr13 sig a) → Expr13 sig a
  Lit13 :: a → Expr13 sig a

let13 :: Expr13 sig x → (x → Expr13 sig a) → Expr13 sig a
let13 (Lit13 x) body = body x                                    4
let13 (OpLet13 op k) body = OpLet13 op (λx → let13 (k x) body)   5

op13 :: eff :«: sig ⇒ (x → eff a) → Expr13 sig x → Expr13 sig a
op13 op e = let13 e (λx → OpLet13 (inj $ op x) Lit13)            6
```

**Figure 3.19:** The expressions for ***Deep*** reduced to only *OpLet* and *Lit* after merging *OpCall* and *Let* and replacements for the old *Let*- and *OpCall*-expressions.

After merging these operators, we see that cases 4 to 6 of the *handle* function are indeed completely incorporated in the implementations of **let** and *op*. In figure 3.20, we show the changes to the *handle* function. We see matches and constructions of *OpCall*s nested in argument expression of a *Let* reduced to matching and constructing *OpLet* instead and we see cases 4 to 6 removed from the *handle* function. Indeed, what we have left is what we believe to be a fairly minimal version of the handling function, embodying algebraic effect handling in only three cases:

1. *Return* a value body by calling the return function for wrapping it in the answer type modification.
2. *Handle* an operation by calling its op-implementation with it and its continuation (captured by means of an *OpLet*).
3. *Delegate* any other operations to a handler further down the chain by skipping over it and handling further down the continuation of those operations.

```
handle13 :: Handler13 eff r a w → Expr13 (eff :++: r) a → Expr13 r w
handle13 h@(Handler13 ops ret) eb = case eb of
  Lit13 bv → ret bv                                                   1
  OpLet13 (Inl op) body → ops op (λy → (handle13 h (body y)))         2
  OpLet13 (Inr op) body → OpLet13 op ((λt → handle13 h t).body)       3
```

**Figure 3.20:** Handling function after merging *OpCall* and *Let* expressions into a single *OpLet*.

We are left with implementing the evaluation function. Its implementation is now rather short. The only cases left are those of *OpLet* and *Lit* where both directly correspond to a constructor in *Result*. Indeed, instead of using this *Result* data type to encode the same thing as *Expr* now embodies, we might as well only match those expressions that have no effects left, represented here by the empty effect *EPure*, such as is done with the *run* function.

```
eval13
  :: Expr13 sig a
  → Result13 sig a
eval13 (OpLet13 op body) = Op13 op body
eval13 (Lit13 v) = NoOp13 v

run13 :: Expr13 EPure a → a
run13 (Lit13 v) = v
```

## 3.9. Step 8: Freer Monad!

The final fact we take note of is the familiarity of the signature of *let13*. Its type is: *let13* :: *Expr13 sig x* → (*x* → *Expr13 sig a*) → *Expr13 sig a*. This should look familiar, as when we substitute *m* = *Expr13 sig*, we get a type of *m x* → (*x* → *m a*) → *m a*, which is exactly the signature of monadic bind. Indeed we have used *Let*-expressions as though they are the monadic bind operation since section 3.4 to abstract computations without operations. As it turns out, we can directly implement a *Monad* instance for *Expr13 sig*:

```
instance Monad (Expr13 sig) where
  return = Lit13
  (>>=) = let13
```

We can now implement, for instance, function application using do-notation, rather than having to manually write **let**-binds:

```
app13 :: Expr13 sig (x → Expr13 sig a) → Expr13 sig x → Expr13 sig a
app13 e1 e2 = do
  v1 ← e1
  v2 ← e2
  v1 v2
```

In fact, the signature of *Expr13* is exactly equivalent to the signature of the freer monad [48]. Either this or the free monad [44] is what is used most often to model algebraic effects and handlers. We can see now why this model is so useful: it separates pure actions from effectful ones by means of the *Lit* and *OpLet* constructors and it allows access to the large expressive power of monads. As a final showing of the equivalence of *Expr13* to the freer monad, we can expand the implementation of *let13* in the body of (>>=) to get the following monad instance:

```
instance Monad (Expr13 sig) where
  return = Lit13
  Lit13 x >>= body = body x
  OpLet13 op k >>= body = OpLet13 op (λx → k x >>= body)
```

This is the abstraction that is also used as a starting point for implementing other types of effects. In Chapter 4, we start with the exact same implementation for effect trees, but we change the handling abstraction. In Chapter 5, we use an adjusted tree that embeds scoped effects. Finally, in Chapter 6, we show a way of testing embedded interpreters such as this one and comparing it to any other interpreter.

# 4

# Deriving an Operational Semantics for Shallow Algebraic Effects

In this chapter we examine a variant of the semantics of algebraic effects called shallow algebraic effects. These semantics of handlers differ only in their titular component: effect handlers. We saw in Section 3.1 that handlers propagate themselves into the continuation argument of an operation implementation. Shallow handlers differ in that they instead leave that propagation in the hand of the programmer. This difference makes shallow effects somewhat more flexible to write. However, shallow effects and deep effects are otherwise equivalent in expressivity [32].

We introduce inverse program transformational steps that mirror our steps in Chapter 3. These steps turn the monadic implementation from Section 3.9 into shallowly handled effects (Section 4.3) and then to an untyped denotational interpreter. The steps we add are as follows:

1. Abstract a *handle* function that summarises the wanted behaviour.
2. Split the *Impure* constructor into *OpCall* and *Let* (inverse of Section 3.8).
3. Inline and lift the *handle* function as a *Handle* constructor in the expression tree (inverse of Section 3.7).
4. Inline and lift pure computations into the expression tree and add a *Value* type (inverse of Section 3.4 and Section 3.6).
5. Remove intrinsic typing from the language (inverse of Section 3.5).

The order of these inverse steps differs only in removing intrinsic typing. Indeed it should be possible to first remove intrinsic typing and later inline and lift pure language features into the expressiont tree, but we choose to have the interpreter safely intrinsically typed a little longer. After these custom steps, we apply closure conversion to get a big-step interpreter, after which we apply the list of steps shown by Vesely and Fisher in 'One step at a time' [78] to get the final small-step interpreter shown in Section 4.6. Finally, we derive a small-step operational semantics for shallow effects and handlers in Section 4.7.

## 4.1. Step 0: Specify Handle Function

The first step is to abstract handling from the general use of the freer monad. Specifically, we want to abstract a *handle* function that allows us to handle effects 'in a shallow manner'. We write a handling function for states using the freer monad below:

$$
\begin{aligned}
&\mathit{hSt0} :: \mathit{Freer}\ (\mathit{St}\ \mathit{st}\ {:}{+}{+}{:}\ r)\ a \to \mathit{Freer}\ r\ (\mathit{st} \to \mathit{Freer}\ r\ (\mathit{st}, a))\\
&\mathit{hSt0}\ (\mathit{Pure}\ a) && = \mathit{Pure}\ (\lambda \mathit{st} \to \mathit{return}\ (\mathit{st}, a))\\
&\mathit{hSt0}\ (\mathit{Impure}\ (\mathit{Inl}\ (\mathit{Get}\ ()))\ k) && = \mathit{Pure}\ (\lambda \mathit{st} \to \mathit{hSt0}\ (k\ \mathit{st}) \gg\!= (\$\ \mathit{st}))\\
&\mathit{hSt0}\ (\mathit{Impure}\ (\mathit{Inl}\ (\mathit{Put}\ \mathit{st}))\ k) && = \mathit{Pure}\ (\backslash\_ \to \mathit{hSt0}\ (k\ ())\gg\!= (\$\ \mathit{st}))\\
&\mathit{hSt0}\ (\mathit{Impure}\ (\mathit{Inr}\ \mathit{op})\ k) && = \mathit{Impure}\ \mathit{op}\ (\mathit{hSt0}.k)
\end{aligned}
$$

This function matches on the cases of *Freer* to define a *return*-like behaviour in the *Pure*-clause, operation handling behaviour in the clauses for *Impure* (*Inl* _), and deferring behaviour in the clause for *Impure* (*Inr* _). We could abstract these behaviours in at least two ways. The deep handling abstraction we have already seen abstracts all recursive calls to *hSt0*. The shallow abstraction leaves the place of applying the handler up to the programmer. The *sHandle* function and *SHandler* type together implement this abstraction:

```
data SHandler0 eff r a w where
  SHandler0 ::
    (forall x.eff x → (x → Freer (eff :++: r) a) → (Freer (eff :++: r) a → Freer r w) → Freer r w) →
    (a → Freer r w) →
    SHandler0 eff r a w
sHandle0 :: SHandler0 eff r a w → Freer (eff :++: r) a → Freer r w
sHandle0 hlr@(SHandler0 ops ret) (Pure a) = ret a
sHandle0 hlr@(SHandler0 ops ret) (Impure (Inl op) k) = ops op k (sHandle0 hlr)
sHandle0 hlr@(SHandler0 ops ret) (Impure (Inr op) k) = Impure op (sHandle0 hlr.k)
```

The only difference between this function and its deep counterpart is in the *Impure* (*Inl* _)-clause and its corresponding parameter in the *SHandler* constructor. Instead of always applying *sHandle0 hlr* to the continuation result, we leave this application to the programmer, leaving open the option of applying a different handler. In the usual configurations of such a language, recursive definitions are present in the language in one way or another. We do not make these available with some *letrec* or recursive function definition, so we pass the current handler to the operation implementation to use for convenience.

## 4.2. Step 1: Split Impure into *Let* and Impure Computation

This step inverses the merging of *OpCall* and *Let*. We do so to reach a state where all operations in our language are part of the expression data type. From this point on, we will name our expression data type *Expr*, instead of *Freer*.

Exactly inverse to how we merged *OpCall* and *Let* into *OpLet* and later *Impure* in Section 3.8, we now split *Impure* to get back *OpCall* and *Let*. Aside from the constructors we add to *Expr1*, we add a smart constructor to construct op-calls in the same way we did before.

```
data Expr1 sig a where
  Lit1 :: a → Expr1 sig a
  OpCall1 :: sig a → Expr1 sig a
  Let1 :: Expr1 sig x → (x → Expr1 sig a) → Expr1 sig a
opCall1 :: eff :«: sig ⇒ (x → eff a) → Expr1 sig x → Expr1 sig a
opCall1 eff xt = xt ≫= λx → OpCall1 (inj $ eff x)
```

Finally, we also need to adjust the running function. We rename *run* to *eval* and add a case for *Let*. Interpreting *Let* is done by applying the *body* function to the fully evaluated binding.

```
eval1 :: Expr1 EPure a → a
eval1 (Lit1 v) = v
eval1 (Let1 ev body) = eval1 (body (eval1 ev))
```

## 4.3. Step 2: Inline and Lift Handling

Embedding the *sHandle* function into the *Expr*-type and *eval*-function is a process inverse to the one described in Section 3.7. We take the signature of *sHandle* and use it as the type of the GADT constructor for *SHandle*.

```
data Expr2 sig a where
  Lit2 :: a → Expr2 sig a
  OpCall2 :: sig a → Expr2 sig a
  Let2 :: Expr2 sig x → (x → Expr2 sig a) → Expr2 sig a
  SHandle2 :: SHandler2 eff r a w → Expr2 (eff :++: r) a → Expr2 r w
```

Writing an evaluation function with type *Expr2 EPure a* → *a* hits a dead end when we try to write an implementation for the new *SHandle* case. In this case, an expression of type *eff* :++: *Expr* is introduced, which we do not know how to evaluate with this old evaluation function. Instead, we use a trick learned from Section 3.3.3: we add a *Result* type to say that unhandled operations are valid results of evaluation. We change the signature of *eval* to return this *Result* type, and add cases for *OpCall* and *SHandle*, as well as a case-match case for *Let*s.

```
data Result2 sig a where
  NoOp2 :: a → Result2 sig a
  Op2 :: sig x → (x → Expr2 sig a) → Result2 sig a
eval2 :: Expr2 sig a → Result2 sig a
eval2 (Lit2 v) = NoOp2 v
eval2 (OpCall2 op) = Op2 op Lit2
eval2 (Let2 ev body) = case eval2 ev of
  NoOp2 v → eval2 (body v)
  Op2 op k → Op2 op (λt → k t ≫= body)
eval2 (SHandle2 hlr@(SHandler2 ops ret) eb) = case eval2 eb of
  NoOp2 v → eval2 (ret v)
  Op2 (Inl op) k → eval2 (ops op k (SHandle2 hlr))
  Op2 (Inr op) k → Op2 op (SHandle2 hlr.k)
```

This *Result* type closely resembles the original *Freer* monad. The main difference, however, is that it does not refer to itself in the continuation argument of *Op*. The clauses for *Lit* and *OpCall* wrap their values in *NoOp* and *Op*, respectively. For *OpCall*s, we need to add a continuation, we initialise this with *Lit*, corresponding to *return*, i.e. merely wrapping values into the *Expr* monad.

The case-match for *Let* resembles the implementation of bind (≫=) for the freer monad quite closely. Indeed, if we try to emulate the behaviour of *Let* in the freer monad, we can use 'bind' directly. The implementation for the *SHandle* clause is obtained directly from the cases of *sHandle*, wrapped with recursive calls to *eval* unless an unhandled operation is intentionally returned.

## 4.4. Step 3: Inline and Lift Pure Computations and Specialise Values

The goal of these transformations is to lift Haskell embeddings into the expression tree. Although the *Expr* type already has constructors for the most crucial parts of a language centred around effects and handlers, a full-blown language requires other features and functionalities to be usable. This step unlifts these pure features from our embedding language into the expression tree. This makes it so that the characteristics of these pure features mimic the characteristics of Haskell [74]. We do two transformations to accomplish this feat.

1. We inline and implement functionalities that need to be part of the expression tree.
2. We specify and limit the types that values may assume.

### 4.4.1. Inline and Lift Pure Language Features

We select the same set of pure language features previously used by ***Deep*** and unlift them into the expression tree. We do this by first writing out the functionality we would like to unembed. We then inline the body of each functionality until a form is reached that only consists of *Let* and *Lit* constructors and the embedded functionality. For instance, for constructing pairs (or 2-tuples), we perform the following steps:

```
pair2 :: Expr2 sig x → Expr2 sig y → Expr2 sig (x, y)
pair2 = liftM2 (,)

pair2_1 :: Expr2 sig x → Expr2 sig y → Expr2 sig (x, y)
pair2_1 e1 e2 = do
  v1 ← e1
  v2 ← e2
  return (v1, v2)

pair2_2 :: Expr2 sig x → Expr2 sig y → Expr2 sig (x, y)
pair2_2 e1 e2 =
  e1 >>= λv1 →
    e2 >>= λv2 →
      return (v1, v2)

pair2_3 :: Expr2 sig x → Expr2 sig y → Expr2 sig (x, y)
pair2_3 e1 e2 =
  Let2 e1 (λv1 →
    Let2 e2 (λv2 →
      Lit2 (v1, v2)))
```

Notice that we use the definitions of the monad instance to convert (>>=) and *return* calls to expressions. To implement pair expressions in our new expression tree, we add a constructor for pair-expressions in the tree and we use the desugaring of the functionality into *Let*s and *Lit*s to write the *eval* implementation:

```
data Expr3 sig a where
  Pair3 :: Expr3 sig x → Expr3 sig y → Expr3 sig (x, y)
  ...

eval3 (Pair3 e1 e2) =
  eval3 (Let3 e1 (λv1 →
    Let3 e2 (λv2 →
      Lit3 (v1, v2))))
```

Finally, we can further reduce this expression by inlining *eval3* (*Let3*...), etc. We obtain the following equivalent implementation that we find to be convenient in its descriptivity while being close to the most performant equivalent:

```
eval3 (Pair3 e1 e2) =
  case eval3 e1 of
    NoOp3 v1 →
      case eval3 e2 of
        NoOp3 v2 → NoOp3 (v1, v2)
        Op3 op x_op → Op3 op ((λt → Pair3 (Lit3 v1) t).x_op)
    Op3 op x_op → Op3 op ((λt → Pair3 t e2).x_op)
```

### 4.4.2. Specialise Values

This step is exactly inverse to the transformation described in Section 3.6. We narrow the set of possible values that can result from a computation in our language. We do this by creating a *Value* type with a limited set of constructors. For this language, it is defined as follows:

```
data Value3 a where
  LambdaV3 :: (Value3 x → Expr3 sig a) → Value3 (x → Expr3 sig a)
  IntV3 :: Int → Value3 Int
  BoolV3 :: Bool → Value3 Bool
  UnitV3 :: Value3 ()
```

$PairV3 :: Value3\ x \rightarrow Value3\ y \rightarrow Value3\ (x, y)$
$NilV3 :: Value3\ [a]$
$ConsV3 :: Value3\ a \rightarrow Value3\ [a] \rightarrow Value3\ [a]$

We adapt the evaluation function to wrap returned values with the appropriate constructor and we add the *Value* type to the types of the arguments of embedded functions. This change makes it impossible to define a monad instance for the expression tree because there is no way to lift an arbitrary value into the expression tree.

## 4.5. Step 4: Remove Intrinsic Typing

Inverse to the step in Section 3.5, we remove intrinsic typing of the evaluation function in this step. We do so by removing all type parameters from the *SHandler*, *Expr*, and *Value* types. To differentiate operations and effects that were previously differentiated in *sig* type-parameters, we add an identifier of sorts to the *Op* and *OpCall* constructors. We also adjust the operation implementations to specify the operation each operates over and we change handlers to match on these names. Any types of unique identifier for effects and operations can be used, but for this example we use *String* names for both effects and operations within effects.

This causes the *Op* and *OpCall* constructors to have two *String*s and a *Value* added to each, representing the effect name, operation name, and operation argument, respectively. We change handlers to store the name of the effect they handle and we split the operation handling function into multiple functions that are paired with the name of the operation each handles. Finally, we adjust the evaluation function to accommodate for these changes in the *SHandle* clause:

*eval4* (*SHandle4 hlr*@(*SHandler4 eff ops ret*) *eb*) = **case** *eval4 eb* **of**
  *NoOp4 v* → *eval4* (*ret v*)
  *Op4 eff′ op va x_op* → **if** *eff* ≡ *eff′*
    **then** *eval4* (*ops op va x_op* (*SHandle4 hlr*))
    **else** *Op4 eff′ op va* (*SHandle4 hlr*.*x_op*)

We see that evaluating *SHandle* is quite similar to the last step, but we no longer match on *Inl* and *Inr*. Instead, we look at the effect names listed by the *Op* and *SHandler* constructors. Operations with an equal effect name are treated equivalently to the *Inl*-constructor, whereas unmatched operations are returned to be dealt with by a different handler, like with *Inr*-cases. This implementation handling performs exactly the same as before as long as duplicate effects in the signature are now referenced with unique effect names.

## 4.6. Step 5: Apply Transformations to Derive Small-Step Interpreter

Our evaluation function is now of a denotational style, but we wish to find a small-step semantics for our language. We find a small-step interpreter for the same language by first deriving a big-step direct-style interpreter before applying the program transformations described by Vesely and Fisher [78]. The process of deriving a big-step interpreter from the denotational interpreter is now rather systematic. We apply closure conversion to directly obtain the required interpreter.

Applying Vesely and Fisher's transformations can be done in a similarly systematic way, without much change needed for this specific language, so we refrain from giving a detailed re-explanation of the steps. Instead, we skip to the final interpreter and present the small-step direct-style evaluation function clause for *SHandle*:

$$
\begin{array}{l}
\textit{substHandleBody5} :: \textit{String} \to \textit{Value5} \to (\textit{Expr5} \to \textit{Expr5}) \to \textit{SHandler5} \to \textit{Expr5} \to \textit{Expr5} \\
\textit{substHandleBody5 param va x\_op hlr} = \\
\quad \textit{substHdl5}\ (\textit{SHandle5 hlr}) \circ \\
\quad\quad \textit{substAll5} \\
\quad\quad\quad [(\textit{param}, \textit{va}), \\
\quad\quad\quad\quad (\texttt{"resume"}, \textit{LambdaV5}\ \texttt{"\_\_\_y"}\ (\textit{x\_op}\ (\textit{Var5}\ \texttt{"\_\_\_y"}))) \\
\quad\quad\quad ] \\
\textit{eval5\_9}\ (\textit{SHandle5 hlr@}(\textit{SHandler5 eff ops}\ (\textit{RetI5 retArgNm retBody}))\ \textit{eb}) = \\
\quad \textbf{case}\ \textit{eval5\_9 eb}\ \textbf{of} \\
\quad\quad \textit{Inl0}\ (\textit{NoOp5 v}) \to \textit{eval5\_9}\ (\textit{subst5 retArgNm v retBody}) \\
\quad\quad \textit{Inl0}\ (\textit{Op5 op eff' va x\_op}) \to \textbf{if}\ \textit{eff} \equiv \textit{eff'} \\
\quad\quad\quad \textbf{then let}\ \textit{OpI5}\ \_\ \textit{param opI} = \textit{fromJust}\ \$\ \textit{find}\ (\lambda(\textit{OpI5 op'}\ \_\ \_) \to \textit{op} \equiv \textit{op'}) \\
\quad\quad\quad\quad\quad \textbf{in}\ \textit{eval5\_9}\ (\textit{substHandleBody5 param va x\_op hlr opI}) \\
\quad\quad\quad \textbf{else}\ \textit{inj0}\ \$\ \textit{Op5 op va}\ (\textit{SHandle5 hlr}.\textit{x\_op}) \\
\quad\quad \textit{Inr0 eb'} \to \textit{inj0}\ (\textit{SHandle5 hlr eb'})
\end{array}
$$

## 4.7. A Small-Step Operational Semantics

We now use the interpreter derived in the last step to find a structural operational semantics for shallow effects and handlers. This structural operational semantics consists of single steps that reduce an expression to a 'smaller' expression, a value, or an unhandled operation. To read these rules from the interpreter, we look at every case of interpretation. For instance, the final evaluation case for operation calls is as follows:

$$
\begin{array}{l}
\textit{eval5\_9}\ (\textit{OpCall5 eff op ea}) = \\
\quad \textbf{case}\ \textit{eval5\_9 ea}\ \textbf{of} \\
\quad\quad \textit{Inl0}\ (\textit{NoOp5 v}) \to \textit{inj0}\ \$\ \textit{Op5 eff op v id} \\
\quad\quad \textit{Inl0}\ (\textit{Op5 eff' op' va x\_op}) \to \textit{inj0}\ \$\ \textit{Op5 eff' op' va}\ ((\lambda t \to \textit{OpCall5 eff op t}).\textit{x\_op}) \\
\quad\quad \textit{Inr0 ea'} \to \textit{inj0}\ (\textit{OpCall5 eff op ea'})
\end{array}
$$

From this code, we can read 3 structural operational semantics rules. One for each clause covered in this implementation. The three clauses are, generally speaking:

1. The operand is a value: another operand can be interpreted or we can reduce the entire expression.
2. The operand is an unhandled operation, which is a special type of value: the current expression is made part of the continuation of the unhandled operation.
3. The operand can be further reduced: a stepping rule for the operand must exist and it is replaced by its reduction.

$$\text{OpCall-Value}\ \frac{}{\textbf{op-call}\ \textit{eff}_i\ \textit{op}_j\ v_a \to \textit{op}\ \textit{eff}_i\ \textit{op}_j\ v_a\ []}$$

$$\text{OpCall-Op}\ \frac{}{\textbf{op-call}\ \textit{eff}_i\ \textit{op}_j\ (\textit{op}\ \textit{eff}_k\ \textit{op}_m\ v_a\ X_{op}[]) \to \textit{op}\ \textit{eff}_k\ \textit{op}_m\ v_a\ (\textbf{op-call}\ \textit{eff}_i\ \textit{op}_j\ X_{op}[])}$$

$$\text{OpCall-Step}\ \frac{e_a \to {e'}_a}{\textbf{op-call}\ \textit{eff}_i\ \textit{op}_j\ e_a \to \textbf{op-call}\ \textit{eff}_i\ \textit{op}_j\ {e'}_a}$$

**Figure 4.1:** Structural operational semantics for operation call expressions with (shallow) algebraic effects.

In figure 4.1, we show these three rules for operation call expressions. Each corresponds with a case in the interpreter clause for operation calls. Evaluating an expression either causes it to be reduced a single step, or find that it is a value or unhandled operation. This is captured most explicitly by the OpCall-Step case, where the single step reduction is shown. In the other two steps, this is not explicitly shown because the operand expression is not further reduced in such a step. Notice that, in these descriptions, we use $X_{op}[]$ to denote the reconstruction function and $X_{op}[x]$ to denote the reconstruction function applied to term *x*. We perform the same process for handler semantics to find the operational semantics for our shallow handlers in figure 4.2.

$$\text{Handle-Value}\,\frac{}{\textbf{handle}\ \{..,\textbf{return}\ x \mapsto e,..\}\ v \to e\ [x/v]}$$

$$\text{Handle-Op}\,\frac{}{\begin{array}{c}\textbf{handle}\ h@\{\mathit{eff}_i,..,op_j\ x \mapsto e,..\}\ (op\ \mathit{eff}_i\ op_j\ v\ X_{op}[])) \to \\ e\ [x/v,(hdl\ e')/(\textbf{handle}\ h\ e'),\mathit{resume}/(x \mapsto X_{op}[x])]\end{array}}$$

$$\text{Handle-Op-Other}\,\frac{}{\begin{array}{c}\textbf{handle}\ h@\{\mathit{eff}_i,..\}\ (op\ \mathit{eff}_k\ op_j\ v\ X_{op}[]) \to op\ \mathit{eff}_k\ op_j\ (\textbf{handle}\ h\ X_{op}[]) \\ \text{if } \mathit{eff}_i \not\equiv \mathit{eff}_k\end{array}}$$

$$\text{Handle-Step}\,\frac{eb \to eb'}{\textbf{handle}\ h\ eb \to \textbf{handle}\ h\ eb'}$$

**Figure 4.2:** Structural operational semantics for shallow algebraic effect handling.

# 5

# Deriving an Operational Semantics for Deep Scoped Effects

In this chapter we apply the same transformations as in Chapter 4, but on scoped effects. At the time of writing, no published article lists an operational semantics for scoped effects. The operational semantics of scoped effects are an open topic of research [81]. With our method, we can derive one for scoped effects in a very similar fashion to shallow effects.

We introduce scoped syntax in Section 5.1 and Section 5.2. We list the peculiarities of applying the transformations on scoped effects in Section 5.3 until Section 5.6. We use Vesely and Fisher's transformations in Section 5.7 to derive a small-step operational semantics for scoped effects in Section 5.8.

## 5.1. The Monadic Implementation

For the monadic implementation of scoped effects, we start with Yang's implementation [81]. From that implementation, we derive a freer form, similar to how a freer monad is obtained in 'Freer Monads more Extensible Effects' [48].

```
data Freer sig gam a where
  Pure :: a → Freer sig gam a
  Call :: sig x → (x → Freer sig gam a) → Freer sig gam a
  Enter :: gam x → (x → Freer sig gam y) → (y → Freer sig gam a) → Freer sig gam a
instance Monad (Freer sig gam) where
  return = Pure
  Pure a >>= f = f a
  Call op k >>= f = Call op ((>>=f) <$> k)
  Enter scp k k′ >>= f = Enter scp k ((>>=f) <$> k′)
```

To convince the reader of their equivalence, we use the Yoneda lemma [13]. We can create conversion functions both ways using either the *Yo* type declaration or its *CoYo* counterpart. The existence of the *progToFreer* function and its counterpart shows the equivalence of *Freer* and *Prog* as data structures.

```
data Yo f a = Yo {unYo :: forall r.(a → r) → f r}

freerToProg :: Freer (Yo sig) (Yo gam) a → Prog sig gam a
freerToProg (Pure a) = Pure′ a
freerToProg (Call (Yo opF) k) = Call′ (opF (freerToProg.k))
freerToProg (Enter (Yo scpF) rec k) = Enter′ (scpF (freerToProg.fmap (freerToProg.k).rec))

progToFreer :: (Functor sig, Functor gam) ⇒ Prog sig gam a → Freer (Yo sig) (Yo gam) a
progToFreer (Pure′ a) = Pure a
progToFreer (Call′ op) = Call (Yo (<$> op)) progToFreer
progToFreer (Enter′ scp) = Enter (Yo (<$> scp)) progToFreer progToFreer
```

These show, in essense, that using the *Yo* wrapper, the same programs can be embedded in *Freer* as in *Prog*. For the rest of this work, we do not use *Yo*, as we only use it as a tool to show this equivalence. Instead, we would write operations algebraic operations that directly embed a value, rather than a continuation, as the continuation can be explicitly passed to *Call*. For *Enter*, we could construct scoped nodes in multiple ways with *Freer*. We could embed the scoped programs directly in the operation, without continuation (*catch*). Or we could embed a selector in the operation and select the appropriate scoped program in the first continuation (*catch′*).

```
catch :: SCatch :«: gam ⇒ Freer sig gam a → Freer sig gam a → Freer sig gam a
catch h r = Enter (inj $ SCatch h r) id return

data CatchArgs = CatchH | CatchR

catch′ :: SCatch :«: gam ⇒ Freer sig gam a → Freer sig gam a → Freer sig gam a
catch′ h r = Enter (inj $ SCatch CatchH CatchR) (λcase
  CatchH → h
  CatchR → r) return
```

Both can be written with exactly the same tree, but *catch* is an approach more faithful to the original scoped effects. *catch′*, although equivalent in behaviour to *catch*, demonstrates explicitly what the first continuation does: it takes a selection and presents the scoped program selected.

## 5.2. Step 0: Specify Handle Function

For specifying the handle function, we start with the *handleE* function from the paper. This function is already somewhat of a specification of the more generic *handle* function. This function takes, what we have commonly referred to as a *Handler* data type thusfar, an expression tree to be handled, and results in some wrapping of the result type of the expression tree. We need the result wrapping to still be within an expression tree, as we cannot write an interpreter for it otherwise. We also specify that the handling function should only try to handle a single layer of algebraic and scoped effects at the same time, so a *Freer* (*eff* :++: *sig*) (*scp* :++: *gam*) *a* should have its *eff* and *scp* effects handled at once by a handler. To put it in terms of the *handleE* function, the following type-signature signifies these specifications:

```
handleE′ :: EndoAlg sig gam (Compose (Freer sig′ gam′) f)
          → Freer sig gam a
          → (Compose (Freer sig′ gam′) f) a
handleE′ = handleE
```

We redefine this specific composition as a new handling function *handleE″*. In this redefinition we get rid of the obnoxious *Compose* type-constructors and by writing every part of the *EndoAlg* data type and replacing *Compose f1 f2 a* with *f1* (*f2 a*):

```
handleE″ :: (forall x.x → Freer sig′ gam′ (f x))
    → (forall x w.sig x
      → (x → Freer sig′ gam′ (f w))
      → Freer sig′ gam′ (f w))
    → (forall x y w.gam x
      → (x → Freer sig′ gam′ (f y))
      → (y → Freer sig′ gam′ (f w))
      → Freer sig′ gam′ (f w))
    → Freer sig gam a
    → Freer sig′ gam′ (f a)
handleE″ hReturn hOps hScps (Pure x) = hReturn x
handleE″ hReturn hOps hScps (Call op k) = hOps op (handleE″ hReturn hOps hScps.k)
handleE″ hReturn hOps hScps (Enter scp rec k) =
  hScps scp (handleE″ hReturn hOps hScps.rec) (handleE″ hReturn hOps hScps.k)
```

*handleE″* is still equivalent to *handleE* in meaning, but its signature is slightly easier to write programs and transformations for. However, in practice, it is very hard to write modular handlers with this signature alone. For those, we would need a 'weaving function' [80, 72]. We introduce a similar concept to be able to weave together the results of handling with subsequent continuations. We call the added function the 'mending function' and its purpose is to implement 'weaving' in a way that is compatible with our transformations. It has the type $\mathit{forall}\ x\ w.f\ x \rightarrow (x \rightarrow \mathit{Freer}\ \mathit{sig}'\ \mathit{gam}'\ (f\ w)) \rightarrow \mathit{Freer}\ \mathit{sig}'\ \mathit{gam}'\ (f\ w)$ and thus defines the way an answer type modification can affect a following computation. Using the mending function, we can now modularise our handlers, meaning we can pop off one level of scoped and algebraic effects with a single handler. Without this function, we would be unable to thread the handler in cases where unhandled operations and scopes must be treated.

```
data Handler0 eff r scp rg f where
  Handler0 ::
    {hReturn :: forall x.x → Freer r rg (f x),
      hOps :: forall x a.eff x → (x → Freer r rg (f a)) → Freer r rg (f a),
      hScp :: forall x y a.scp x → (x → Freer r rg (f y)) → (y → Freer r rg (f a)) → Freer r rg (f a),
      hMend :: forall x a.f x → (x → Freer r rg (f a)) → Freer r rg (f a)
    } →
    Handler0 eff r scp rg f
handle0 :: Handler0 eff r scp rg f → Freer (eff :++: r) (scp :++: rg) x → Freer r rg (f x)
handle0 (Handler0 ret _ _ _) (Pure a) = ret a
handle0 h@(Handler0 _ ops _ _) (Call (Inl op) k) =
  ops op (λx → handle0 h (k x))
handle0 h (Call (Inr op) k) =
  Call op (handle0 h.k)
handle0 h@(Handler0 _ _ scps _) (Enter (Inl scp) rec k) =
  scps scp (handle0 h.rec) (handle0 h.k)
handle0 h@(Handler0 _ _ _ mend) (Enter (Inr scp) rec k) =
  Enter scp (handle0 h.rec) (λfx → mend fx (handle0 h.k))
```

Note that the mending function is used only to connect the handled result of scoped programs with the continuation. The rest of the computation is standard. The mending function might also be used by the implementation of handlers to mend scoped program results and the continuation, but handlers might also define mending per operation, not using a single mending function. For this reason, we do not apply the mending function by default for handled scoped operations.

## 5.3. Step 1: Split Impure into *Let* and Impure Computation

We split the impure operations into *OpCall*, *ScopeCall* and *Let*. In the case of scoped effects, we have two: *Call* and *Enter*. Both of these constructors currently carry a generic continuation. The *Let* constructor will take these generic continuations, as *Call* and *Enter* have their continuations removed:

```
data Expr1 sig gam a where
  Lit1 :: a → Expr1 sig gam a
  OpCall1 :: sig a → Expr1 sig gam a
  ScopeCall1 :: gam x → (x → Expr1 sig gam a) → Expr1 sig gam a
  Let1 :: Expr1 sig gam x → (x → Expr1 sig gam a) → Expr1 sig gam a
```

Handling is changed in the same way as for shallow handlers, and we adjust evaluation to return a *Result*, possibly containing unhandled algebraic and scoped operations:

```
data Result1 sig gam a where
  NoOp1 :: a → Result1 sig gam a
  Op1 :: sig x → (x → Expr1 sig gam a) → Result1 sig gam a
  Scope1 :: gam x → (x → Expr1 sig gam y) → (y → Expr1 sig gam a) → Result1 sig gam a
eval1 :: Expr1 sig gam a → Result1 sig gam a
eval1 (Lit1 a) = NoOp1 a
eval1 (OpCall1 op) = Op1 op return
eval1 (ScopeCall1 scp rec) = Scope1 scp rec return
```

```
eval1 (Let1 x k) = case eval1 x of
  NoOp1 xv → eval1 $ k xv
  Op1 op x_op → Op1 op (x_op >=> k)
  Scope1 scp rec x_op → Scope1 scp rec (x_op >=> k)
```

## 5.4. Step 2: Inline and Lift Handling

We lift the *handle* function signature into the expression tree. The expression tree is extended with a *Handle* constructor as follows:

```
data Expr2 sig gam a where
  ...
  Handle2 :: Handler2 eff r scp rg f → Expr2 (eff :++: r) (scp :++: rg) x → Expr2 r rg (f x)
```

The evaluation function is also adjusted. An initial version can be implemented as *eval2* (*Handle2 h eb*) = *eval2* (*handle2 h eb*), where *handle2* is *handle1*, but in *Expr2*. However, to get an informative small-step semantics, we cannot work with these structure desugarings. Instead, we simplify, inline, and evaluate the case-matches from *handle* to find the following implementation:

```
eval2 :: Expr2 sig gam a → Result2 sig gam a
eval2 ...
eval2 (Handle2 h@(Handler2 ret ops scps mend) eb) = case eval2 eb of
  NoOp2 a → eval2 $ ret a
  (Op2 (Inl op) k) → eval2 $ ops op (λx → Handle2 h (k x))
  (Op2 (Inr op) k) → Op2 op (Handle2 h.k)
  (Scope2 (Inl scp) rec k) → eval2 $ scps scp (Handle2 h.rec) (Handle2 h.k)
  (Scope2 (Inr scp) rec k) → Scope2 scp (Handle2 h.rec) (λfx → mend fx (Handle2 h.k))
```

## 5.5. Step 3: Inline and Lift Pure Computations and Specialise Values

This step is the same as for shallow handlers. The main difference is that we end up with a slightly different evaluation function because of the third alternative result type. We must not only accumulate surrounding context/continuations for unhandled algebraic operations, but also for unhandled scoped operations. This means that, for instance, the evaluation case for function applications, gets extra cases for dealing with unhandled scoped operations:

```
eval3 (App3 ef ea) =
  case eval3 ef of
    NoOp3 vf@(LambdaV3 f) →
      case eval3 ea of
        NoOp3 va → eval3 (f va)
        Op3 op x_op → Op3 op ((λt → App3 (Lit3 vf) t).x_op)
        Scope3 scp k x_op → Scope3 scp k ((λt → App3 (Lit3 vf) t).x_op)
    Op3 op x_op → Op3 op ((λt → App3 t ea).x_op)
    Scope3 scp k x_op → Scope3 scp k ((λt → App3 t ea).x_op)
```

## 5.6. Step 4: Remove Intrinsic Typing

Just like with algebraic operations, scoped operations need to be qualified by something other than their constructors. Same as before, we add effect names to handlers, op calls, and scoped calls. We also add a name for every algebraic operation and every scoped operation. The evaluation function is changed for the *Handle* case by replacing the match on *Inl* and *Inr* constructors with an if-then-else checking whether the handler effect and operation effect are the same. We could add a second effect name to separate algebraic effects and scoped effects, but we think that, because we already handle both a layer of algebraic effects and scoped effects at once, we might as well use the same name for both. The change equates to an if-expression rather than a case-match in:

```
eval4 (Handle4 h@(Handler4 eff ret ops scps mend) eb) = case eval4 eb of
  NoOp4 a → eval4 $ ret a
  Op4 eff′ op vo k →
    if eff ≡ eff′
      then eval4 $ ops op vo (λx → Handle4 h (k x))
      else Op4 eff′ op vo (Handle4 h.k)
  Scope4 eff′ scp vs rec k →
    if eff ≡ eff′
      then eval4 $ scps scp vs (Handle4 h.rec) (Handle4 h.k)
      else Scope4 eff′ scp vs (Handle4 h.rec) (λfx → mend fx (Handle4 h.k))
```

## 5.7. Step 5: Apply Transformations to Derive Small-Step Interpreter

We apply the transformations composed by Vesely to get a direct-style small-step interpreter. We show the *Handle*-case for the final evaluation function:

```
eval5_9 (Handle5 h@(Handler5 eff (retP, retB) ops scps (mendP, mendB)) eb) = case eval5_9 eb of
  Inl0 (NoOp5 a) → eval5_9 (inj0 $ subst5 retP a retB)
  Inl0 (Op5 eff′ op vo x_op) →
    if eff ≡ eff′
      then eval5_9 (inj0 $ substAll5 [(opParamP, vo), ("resume", resumption5 h x_op)] opBody)
      else inj0 $ Op5 eff′ op vo (Handle5 h.x_op)
    where
      (_, opParamP, opBody) = fromJust $ find (λ(op′, _, _) → op ≡ op′) ops
  Inl0 (Scope5 eff′ scp vs recP recB x_op) →
    if eff ≡ eff′
      then eval5_9 (inj0 $ substAll5 [
        (scpParamP, vs),
        (scpRecP, LambdaV5 recP (Handle5 h recB)),
        ("resume", resumption5 h x_op)] scpB)
      else inj0 $ Scope5 eff′ scp vs recP (Handle5 h recB)
        (λfx → Let5 fx mendP (subst5 "resume" (resumption5 h x_op) mendB))
    where
      (_, scpParamP, scpRecP, scpB) = fromJust $ find (λ(scp′, _, _, _) → scp ≡ scp′) scps
  Inr0 eb′ → inj0 $ Handle5 h eb′
```

## 5.8. A Small-Step Operational Semantics

Finally, we take the final implementation of the interpreter to find a structural operational semantics for scoped effects. We only extract the semantics for handling to save space, all other semantic descriptions, such as those for function application, seem to be the same as for deep algebraic effects and handlers.

$$\text{Handle-Value}\ \frac{}{\textbf{handle}\ \{\textbf{return}\ x \mapsto e, ..\}\ v \to e\ [x/v]}$$

$$\text{Handle-Op}\ \frac{}{\textbf{handle}\ h@\{\mathit{eff}, .., op_i\ x \mapsto e, ..\}\ (op\ \mathit{eff}\ op_i\ v\ X_{op}[\,])) \to e\ [x/v, \mathit{resume}/(x \mapsto \textbf{handle}\ h\ X_{op}[x])]}$$

$$\text{Handle-Op-Other}\ \frac{}{\textbf{handle}\ h@\{\mathit{eff}, ..\}\ (op\ \mathit{eff}'\ op_i\ v\ X_{op}[\,]) \to op\ \mathit{eff}'\ op_i\ (\textbf{handle}\ h\ X_{op}[\,])}\quad \text{if } \mathit{eff} \not\equiv \mathit{eff}'$$

$$\text{Handle-Scope}\ \frac{}{\begin{array}{c}\textbf{handle}\ h@\{\mathit{eff}, .., scp_i \mapsto e, ..\}\ (scope\ \mathit{eff}\ op_i\ v\ x_{rec}\ e_{rec}\ X_{op}[\,])) \to \\ e\ [x/v, rec/(x_{rec} \mapsto \textbf{handle}\ h\ e_{rec}), \mathit{resume}/(x \mapsto \textbf{handle}\ h\ X_{op}[x])]\end{array}}$$

$$\text{Handle-Scope-Other}\ \frac{}{\begin{array}{c}\textbf{handle}\ h@\{\mathit{eff}, .., mend\ x_m \mapsto e_m\}\ (scope\ \mathit{eff}'\ scp_i\ v\ x_{rec}\ e_{rec}\ X_{op}[\,]) \to \\ scope\ \mathit{eff}'\ scp_i\ v\ x_{rec}\ (\textbf{handle}\ h\ e_{rec})\ (\textbf{let}\ x_m = [\,]\ \textbf{in}\ e_m\ [\mathit{resume}/(y \mapsto \textbf{handle}\ h\ X_{op}[y])])\end{array}}\quad \text{if } \mathit{eff} \not\equiv \mathit{eff}'$$

$$\text{Handle-Step}\ \frac{eb \to eb'}{\textbf{handle}\ h\ eb \to \textbf{handle}\ h\ eb'}$$

**Figure 5.1:** Small-step structural operational semantics for handling of scoped effects.

# 6 Evaluation

After showing our additional program transformations to derive a freer monad embedding for effects and handlers from an operational semantics and vice versa, we discuss the quality of our steps and applications in this chapter. We split this evaluation into two parts:

1. We review our program transformations and argue that new formal proofs are not required to prove that each transformation maintains the behaviour of the interpreter that is transformed.
2. We evaluate the applications of our program transformations from Chapters 3 to 5. For this, we generate a test suite that attempts to verify that each of the interpreters from the aforementioned chapters displays the same behaviour.

In the following sections, we first discuss the validity of the program transformations to relate a freer monad-based embedding of effects and handlers to an untyped denotational interpreter. We argue that the program transformations we use are well-known derivations that have been proven to preserve behaviour of the program under transformation, and thus require no additional proving in this work. The second part to evaluating our work comes in the form of a generated test suite. This test suite uses state-of-the-art program synthesis techniques to automatically synthesise programs with effects and handlers that can be executed by our interpreters. Finally, we discuss the conclusions we can take away from these methods of evaluation and what may be considered to be done differently in future work.

## 6.1. The Added Program Transformations

In this section, we discuss the program transformations we add to derive a freer monad-based embedding of effects and handlers from an untyped denotational interpreter of the same language. We look at every step individually to see where the difficulties of application lie and to relate each step to the works that most closely describe a similar or even exactly the same transformation. We indicate when we involve some creativity in a transformation and when we think the process can be completely automated. We claim that every step can be inverted without loss, albeit with a few caveats for specifically the step that introduces intrinsic typing. Additionally, we claim that, again with a few caveats for intrinsically typing, our transformations relate so closely to existing program transformations that no additional proof is necessary to convince the reader of their correctness.

We discuss the added steps in the order of introduction in Chapter 3. The inverse of every step is discussed in the same section. We also discuss the step we only introduce when deriving an operational semantics from a freer monad-based embedding, such as is done for shallow algebraic and deep scoped effects and handlers. This step we discuss additionally is the step of specifying a handle function. The steps, with section of introduction within parentheses, are as follows:

1. Lettify/Inline and lift pure computations. (Section 3.4)
2. Add/Remove intrinsic typing to values, expressions, handlers, binary operations, etc. (Section 3.5)
3. Generalise/Specialise the Value type. (Section 3.6)
4. Lettify/Inline and lift the handling abstraction. (Section 3.7)

5. Merge/Split impure computations and let constructs into a single expression constructor. (Section 3.8)
6. Specify Handle Function (Section 4.1)

### 6.1.1. Lettify/Inline Pure Computations

At first glance, this transformation reminded us of refunctionalisation/defunctionalisation [24, 23]. This is because we replace a data constructor with an equivalent higher order program, built with *Let*-expressions. However, at this moment, we cannot fully confirm that this step uses refunctionalisation and defunctionalisation exactly because of a few reasons. Firstly, because the conversion is partial: it only removes part of the data involved and leaves part of that same data untouched. Secondly, because no higher-order functions appear to be involved in the transformation at first glance. And finally, because *Let*-expressions are apparently introduced where none were present before.

For lettification, we introduce expressions using mostly *Let* for every pure computation that, when evaluated, are behaviourally equivalent to the original expressions. The equivalence is most apparent if we perform the inlining direction of the transformation. For example, we can inline the smart constructor for function application to find the following:

```
eval8 (app8 ef ea)
 →
case eval8 ef of
  NoOp8 vf → case eval8 ea of
    NoOp8 va → case vf of
      LambdaV8 body → eval8 (body va)
      _ → error ("Cannot apply non-function value: " <> show vf)
    Op8 op v x_op → Op8 op v ((λt →
        Let8 t
          (λva → case vf of
            LambdaV8 body → body va
            _ → error ("Cannot apply non-function value: " <> show vf))).x_op)
  Op8 op v x_op → Op8 op v ((λt →
      Let8 ef
        (λvf → Let8 ea
          (λva → case vf of
            LambdaV8 body → body va
            _ → error ("Cannot apply non-function value: " <> show vf)))).x_op)
```

We then notice that *eval8* (*Let8* (*Lit8* *v*) *f*) = *eval8* (*f* *v*), apply it to the inner operation context reconstruction function. Finally, we notice that the resulting operation continuation functions ($\lambda t \rightarrow$ ...) are the right-hand side of *app8*. Thus rewriting gives:

```
case eval8 ef of
  NoOp8 vf → case eval8 ea of
    NoOp8 va → case vf of
      LambdaV8 body → eval8 (body va)
      _ → error ("Cannot apply non-function value: " <> show vf)
    Op8 op v x_op → Op8 op v ((λt → app8 (Lit8 vf) t).x_op)
  Op8 op v x_op → Op8 op v ((λt → app8 t ea).x_op)
```

Which is the evaluation case of *App8*, with the exception of using the smart constructor to reconstruct function application expressions. By induction, this shows that *App8* and *app8* are behaviourally equivalent under evaluation with *eval8*.

A proof like the above should be possible to implement in Coq or Agda just the same for every other pure expression. On an alternate, but related note, the correctness of this transformation appears to be closely related to the manner in which *Let* bindings are later abstracted: as monadic binding. Monadic bind threads evaluation of expressions in the final monadic embedding. This step prepares this threading by abstracting away threading through *Let* expressions. The final result of lettifying appears to be nested *Let*-expressions for every subsequently evaluated sub-expression, with a final pure computation when all expressions are evaluated.

### 6.1.2. Add/Remove Intrinsic Typing

This step makes all typing rules assumed or otherwise explicit or, inversely, removes all explicit mention of typing in the types of expressions. Intrinsically typed terms are have their type rules built into their definition [9]. We do this using GADTs in Haskell by explicitely adding each type rule onto the expressions of our language.

In our work, we assume type rules for each expression we transform in this way. However, explicitly defining type rules that match ones semantics should be a more preferred approach. We know of no automatic way as of yet, but we suspect it is possible to derive a set of type rules from the untyped denotational interpreter that fit its expressions. In fact, this is the approach we took: we looked at every expression, looking at what types of values were matched and constructing type rules within the intrinsically typed expression tree as we went.

The main difference between the untyped denotational interpreter and typed denotational interpreter lies in the transformation for impure operations. These impure computations are represented as a product of effect name, operation name, and operation argument when left untyped and as a value of a constructor when typed. The effect name (untyped) corresponds to the data type (typed), the operation name (untyped) corresponds to the data constructor (typed), and the operation argument (untyped) corresponds to the constructor arguments (typed):

```
data Eff a { Op :: Value () → Op () }
Op UnitV
  -- corresponds to
OpCall "Eff" "Op" UnitV
```

These constructions encode almost exactly the same information, with the exception of the typed variant holding more information on the type of the operation.

In summary, adding type information to an expression tree is a well-known operation. Impure computations in a typed setting hold the same information as in the untyped setting. However, one might not have enough information from just the untyped denotational interpreter to introduce intrinsic typing. Instead, one might need to resort to typing rules accompanying the formal semantics of the language to be able to fully derive a typed denotational interpreter.

### 6.1.3. Generalise/Specialise Values

This transformation either removes or adds a *Value* type that specialises the types of values that can inhabit the language.

When generalising, the restriction on values already exists and is simply removed with little consequence. *Value* turns out to only be a wrapper of the underlying Haskell-native values, such as native pairs, *Int*s, etc. Whatever matching we did on wrapped values may just as well be done on the underlying value and operations we applied on underlying values can now be directly applied.

When specialising, restrictions on the types of values are added. We base these restrictions on the types of computations that should be possible within the language we are trying to form an operational semantics for. Exactly opposite to generalising, we now impose the need to wrap and unwrap values, but computation is otherwise left unchanged.

This appears to us as a very standard operation. One which any programmer can relate to as being a fairly common activity.

### 6.1.4. Lettify/Inline Handling

Here we perform a similar transformation for handling as we have done for pure computations before. However, the result is different, as shown in Section 3.7. The resulting function structurally matches the body of the handling expression and splits it up in cases. Each of these cases corresponds to an result of evaluation. We think this lettification can be shown to correspond to the original evaluation case for handling. However, we did not have the time to formulate the proof.

### 6.1.5. Merge/Split OpCall and Let

This step merges or splits impure computations and *Let*-binding into a single expression.

We convince the reader of the behaviour preserving nature of this transformation through an observation: there are only very few expressions left in the expression tree, namely *Let*-binding, impure computations, and pure computations. Combining these expressions can only yield a very limited number of interactions. Namely, *Let* can have a pure computation or impure computation in its argument. For pure computations, however, a value is readily available to be passed to the continuation of *Let*. Thus leaving it a rather unnecessary wrapping of function application at this point. Seeing as the only interactions left are those between impure computations and *Let*-binding, we can merge the two. When we go the other way and split impure computation and *Let*-expressions, we merely re-introduce the possibility to use *Let* to unnecessarily represent function application.

When splitting, the resulting evaluation function is obtained by simple inlining and simplification. When merging, we introduce the split between a pure result and an impure result and evaluate each accordingly. The only other function affected is the handling function. Its cases are simplified when we merge, as many combinations of expressions can no longer occur and computations are inherently coupled with their continuation. When splitting, the cases we add to handling are those relating to the possible nestings of *Let*-expressions and pure computations and are inlined parts of the evaluation function.

### 6.1.6. Specify Handle Function

Specifying the handle function is only done when we start with a freer monad-based embedding of effects and handlers. This is done to make sure that the handling function we use during the transformations is modular and encodes the semantics we would like to end up with. However, this step is exempt from needing to be behaviour preserving, as we have no defined behaviour for handling before this point. The freer monad in itself is not enough to restrict the way handling is implemented. In fact, deep and shallow handling differ only in the handling abstraction, not in the freer monad that abstracts computations.

However, because there is no defined handling behaviour before, this step is absolutely crucial in determining the semantics one ends up with. If the semantics of the handling function one invents at this point do not equal the desired semantics, the resulting operational semantics would of course not offer any insights into those desired semantics.

## 6.2. The Applications of Program Transformations

We have seen that the program transformations we add to derive a freer monad-based embedding of effects and handlers from an untyped denotational interpreter closely relate to existing and well-known transformations. In Chapters 3 to 5 we have applied these and other already described steps to derive the freer monad-based embedding from a small-step operational semantics and vice versa. These applications of the program transformations are what we evaluate in this section. Knowing what we know from Section 6.1, if we applied all transformations to the letter, there is little doubt that the final embedding implements the same language as the original operational semantics describes. However, humans are imperfect, so while applying these steps we may well have made mistakes along the way. To gain confidence in our application of the program transformations and thus in the resulting embedding and operational semantics, we test the interpreters we present in Chapters 3 to 5.

The first observation we have when we think of how to test these interpreters is that every interpreter should interpret the same language, with the same semantics. Secondly, the semantics of the language we interpret is introduced by the first interpreter we implement, a definitional interpreter if you will [74]. The property we want to test is that, indeed, each interpreter interprets the language the same way. For a program transformation, we call the transformation 'behaviour preserving' if the initial program and the final program have the same domain and codomain. This definition comes from the definition of equivalence of mathematical functions and can be applied to our programs as long as the program we test is pure[1]. When transforming interpreters, the behaviour preserving property exactly describes what we wish to check, that the syntax (domain) and semantics (related to the codomain) of the language an interpreter describes is the same as before. To effectively test this property, one could construct

[1]Our interpreters are not pure exactly, as they may throw errors with *error*. We mention and avoid this issue later in this section.

proofs that rigorously equate the domain and codomain for the interpreter before transformation and after transformation. However, we chose to test this using dynamic tests, as Haskell lends itself to proofs of this type less easily than a dependently typed language with proof assistant such as Agda or Coq.

To come close to the result of a rigorous proof, we generate programs (the domain) and run each program with every interpreter (codomains) to check that indeed, the domains and codomains of transformed interpreters are the same. Pałka [67] has previously described a basic technique for generating programs based on the type of expression expected. We use this technique and extend it with generation rules for deep algebraic handler semantics. We convert deep algebraic handlers to an embedding of those same handlers in shallow and scoped handler syntax. This is to say, we were not able to generate shallow and scoped handlers directly, but we describe what is necessary to add generation rules for these in future work.

This section is meant to explain how we wrote the tests for testing interpreters. The following components require explanation:

1. The architecture of the test suite in Section 6.2.1 and why the codebase for the test suite is split into a few components.
2. The program generation component that is responsible for synthesising programs in an untyped expression tree only used in the test suite in Section 6.2.1.
3. The conversion component that provides conversion functions to convert programs of the generic untyped expression tree to other, possibly intrinsically typed, expression trees in Section 6.2.1.

In the final part of this chapter, we reflect on the testing process by showing the types of errors it caught as well as offer a look forward to see what could be done better in the future (Section 6.3).

### 6.2.1. Infrastructure

In figure 6.1, we show an overview of the functions necessary to test our interpreters. We write programs, either by generating them or by manually writing them, as values of the *PG.Expr* expression tree. We can use a few 'dialects' within this expression tree to represent the three different languages with effects and handlers (with Deep, Shallow, and Scoped effects, respectively). Every test program uses the same expression tree to promote re-use of utility functions across tests.

When we test a specific interpreter with a program, we convert the program using the appropriate conversion function. These conversions should represent the program in such a way that the meaning of the program is conserved exactly. Verifying that these program conversions are behaviour-conserving amounts to a problem similar in nature to verifying the transformations themselves, although perhaps slightly less involved. For this work, we assume program conversions are done carefully enough so as not to lose meaning. We make this easier by making sure that the test expression tree does not change or add any language features with respect to the deep, shallow, and scoped languages.

This section is meant to explain how we wrote the tests for testing interpreters. The following components require explanation:

1. The program generation component that is responsible for synthesising programs in an untyped expression tree only used in the test suite.
2. The conversion component that provides conversion functions to convert programs of the generic untyped expression tree to other, possibly intrinsically typed, expression trees in.

#### Program Generation

We wished to generate programs that type-check, to check the behaviours of interpreters rather than the exceptional cases. Although this does mean we cannot detect differences in exceptional behaviours, the goal of this work is to connect semantics that require well-typed programs to make sense anyway. We use the generation technique presented by Klein et al. [50] adjusted for use in Haskell with the QuickCheck testing framework [16] as a basis.

The idea is to generate a program recursively using a function with at least two arguments: an environment of bound variables, and the type to be generated. The function simulates a theorem-proving method, where the theorem is that a program for the given type can be generated, and the

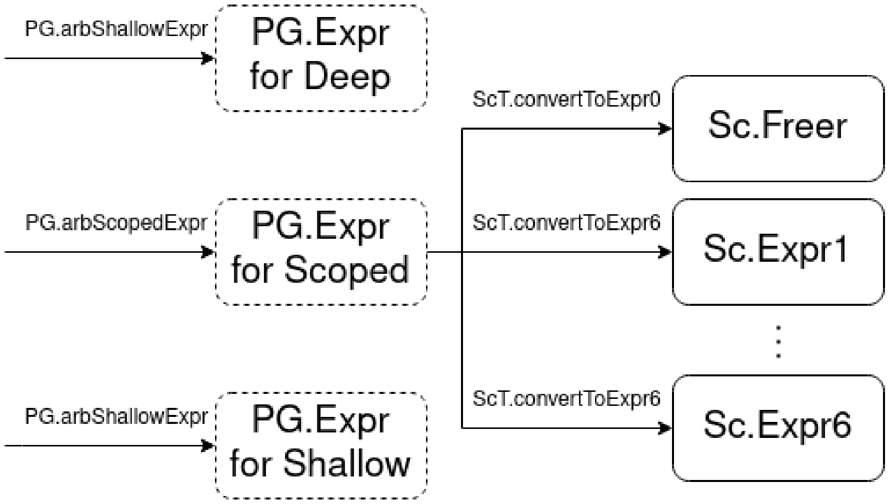



**Figure 6.1:** Diagram of the process of generating and converting programs for various interpreters. Nodes stand for the type of values generated and arrows stand for a function generating or converting programs. *PG.*, *Sc* and *ScT* stand for the program generation, scoped effect interpreters, and tests for scoped effect interpreters modules, respectively.

proof is such a program [60]. Every call to this function may cause recursive calls to generate sub-expressions that might have a different type-goal. We use recommendations from Pałka et al. to limit the size of a generated expression tree and add weights to interesting structures [67]. In our cases, the interesting structures are lambdas and applications (same as for Pałka et al.) and of course handling constructs and operation calls. These get a higher weight than other constructs such as binary operators.

Our languages are more complex than the simple lambda terms used by Pałka et al., but most language constructs can be generated without much trouble. To write generation rules for these constructs, we need to find a typing judgment for that construct first. We use the intrinsic typing of the language in either the first or final step of the process to derive a typing rule and generation rule. For all non-handler constructs, this process has, to some degree, been done before. For handlers and operation calls, however, we discuss our generation rules for every language, as these offer the biggest challenge in program generation.

**The Basic Generator**

The basis of the expression generator is to generate expression trees recursively based on the type of the required expression. It takes, as arguments, the aforementioned environment and expected type. The environment may sometimes be represented as several arguments for several namespaces. In this work, we pass an environment of effect signatures and an environment of variable name-bindings.

The expression tree itself represents all language features present in the three target languages as closely as possible. It annotates some expressions with its constituent types whenever necessary for conversion. For instance, the type of effect that is handled by a handler is described in the expression tree node for the handler. All variables and types that are introduced and need to be referencable later in generation are referred to with De Bruijn indices [25].

**Non-Handler Language Features**

Constructs like if-then-else expressions can be generated based on the intrinsic typing rules for each of those constructs. For instance, we can take the type for the **if** smart-constructor from Section 3.5 for the Deep algebraic language. From this type, we can derive a typing rule directly, using row-type notation for adding the effect signature [55].

Finally, a generation rule can be derived. Within a generation step, the type that is required is known. In the case of *If*, the output may be anything as long as the 'then' and 'else' expressions are that same type and the 'if' expression is of type *Bool*. We thus generate 3 expressions according to these rules, such that the first is of the boolean type and the latter 2 are of the type required. The resulting expressions are used to create an *If*-expression. We show the signature, typing judgement,

$$\begin{aligned} \textit{if10} ::\ & \textit{Expr10 sig Bool} \\ & \rightarrow \textit{Expr10 sig a} \\ & \rightarrow \textit{Expr10 sig a} \\ & \rightarrow \textit{Expr10 sig a} \end{aligned}$$

$$\frac{e_c : \langle\sigma\rangle \textit{Bool}, e_t : \langle\sigma\rangle a, e_e : \langle\sigma\rangle a}{\textbf{if } e_c \textbf{ then } e_t \textbf{ else } e_e : \langle\sigma\rangle a}$$

$$\begin{aligned} & \textbf{do} \\ & \quad e_c \leftarrow \textit{generateExpr effs nv BoolT} \\ & \quad e_t \leftarrow \textit{generateExpr effs nv a} \\ & \quad e_e \leftarrow \textit{generateExpr effs nv a} \\ & \quad \textbf{return}\ \$\ \textit{If}\ e_c\ e_t\ e_e \end{aligned}$$

**Figure 6.2:** The type of an if-expression and its derived typing judgement (left), and a generation rule generated from it (right). For the program generation, we leave out size limitation and general plumbing arguments from 'generateExpr' calls to reduce cognitive load for the reader.

$$\begin{aligned} \textit{Handler10} ::\ & (\textit{forall x.eff x} \\ & \quad \rightarrow (\textit{Value10 x} \rightarrow \textit{Expr10 r w}) \\ & \quad \rightarrow \textit{Expr10 r w}) \\ & \rightarrow (\textit{Value10 a} \rightarrow \textit{Expr10 r w}) \\ & \rightarrow \textit{Handler10 eff r a w} \end{aligned}$$

$$\begin{aligned} \textit{Handle10} ::\ & \textit{Handler10 eff r a w} \\ & \rightarrow \textit{Expr10 (eff :++: r) a} \\ & \rightarrow \textit{Expr10 r w} \end{aligned}$$

$$\frac{\textit{ops} :: \forall\ \textit{x.eff x} \rightarrow (x \rightarrow \langle r\rangle w) \rightarrow \langle r\rangle w, \quad \textit{ret} :: a \rightarrow \langle r\rangle w}{\textit{handler ops ret} : \textit{handler eff r a w}}$$

$$\frac{h :: \textit{handler eff r a w}, \qquad e : \langle \textit{eff}|\sigma\rangle a}{\textit{with } h \textit{ handle } e : \langle\sigma\rangle w}$$

**Figure 6.3:** The types for handlers and handle expressions and matching typing judgements. In the typing judgements, we use a single colon to denote that the expression on the left evaluates to a value of the type on the right. Double colons denote that the term on the left is a value of the type on the right already, as handlers are only used directly in our language.

and eventual generation rule all in figure 6.2.

**Deep Algebraic Handlers**

Deep algebraic handlers are generated with a slightly more guided generation process than the rest of the non-handler constructs. Figure 6.3 shows the types and typing judgements derived for handlers and handle expressions. In our languages, we simplify handling by only allowing handlers to be directly embedded values in a handle expression. We thus type handlers with double colons to separate typing of handlers from typing of expressions.

During generation, we give the generator function access to an environment of variable name bindings as well as an environment of effect types that may be used. For generating the body of a handle-expression, we simply add the handled effect type to the latter environment, similar to how we generate function bodies à la Klein [50]. Handlers are a little more challenging to generate, however. For generating handlers, we need to generate expressions that represent the bodies of the operation and return implementations. Both can be generated by attempting to generate a type $\langle r\rangle w$ recursively. The environment is populated with an argument of type *a* for return implementations, and *x* and $x \rightarrow \langle r\rangle w$ arguments for operation implementations. An outline of the program generation steps needed to generate handle expressions is shown in figure 6.4. To avoid generation failures as much as possible, we guide operation implementation generation in the following way:

1. A number of expressions of type *a* are generated and passed directly to the continuation function argument to get a number of expressions of the type $\langle r\rangle w$.
2. A number of expressions of type $\langle r\rangle w$ is generated without the use of the continuation function in such a way that no value of type *w* is required, if possible.
3. All these expressions are made available in the environment by generating nested *Let* expressions that introduce a name for each.
4. The final body expression is generated with all generated final values in the environment.

```
data Handler = Handler Int [OpI] RetI
data OpI = OpI
  { opISig :: OpType,
    opIBody :: Expr}
data RetI = RetI
  { retISig :: RetType,
    retIBody :: Expr}
generateHandle effs nv a = do
  eff ← generateEffect
  hlr ← generateHandler eff
  eb ← generateExpr (effs <> [eff]) nv a
```

```
generateHandler effs nv eff@(EffType {..}) a = do
  ops ← traverse (generateOp effs nv) effOpTypes
  ret ← generateRet effs nv eff a
  return (Handler ops ret)
generateOp effs nv opT@(OpType {..}) = do
  [contN, emptyN] ← sequence
    [chooseInt (1, 3), chooseInt (1, 3)]
  conts ← replicateM contN (fmap Resume $
    generateExpr effs nv opArgT)
  empties ← replicateM emptyN (generateEmptyOp opResT)
  eb ← generateExpr effs
    (nv <> replicate (length conts + length empties) opResT)
    opResT
  return (OpI opT (foldr Let eb (conts <> empties)))
```

**Figure 6.4:** A simplified outline of the process of generating deep handlers. Empty op generation can, for instance, yield an empty list if the op result type is $w = [a]$. The implementation for 'generateEffect' and 'generateRet' are missing, because we do not attempt to give an exact implementation of the generation function in this chapter.

**Shallow Algebraic Handlers**

Shallow algebraic handlers can currently not be generated to their full potential by our program generator. We currently generate deep handlers and convert them to shallow handlers during the generation process. This is done by consistently calling the recursive handling function on every result of a call to the continuation. This embeds deep handlers in shallow handler syntax, in the same way as described by Hillerström and Lindley [32].

To generate handlers that utilise shallow handler semantics, we would need to make multiple handlers available for every effect (currently we only generate 1). Ideally, we might make mutually recursive shallow handlers possible through the use of a recursive let expression. This might well, however, generate programs that introduce infinite recursion, which is currently prevented by rigorously avoiding recursion. A generator might thus generate programs that do not terminate and would need to be time-boxed when tested.

**Scoped Handlers**

Scoped handlers are also not currently generated to their full potential. Similar to shallow handlers, we convert deep algebraic handlers into scoped handlers during the program generation process. This is done by leaving the list of scoped operations empty and generating a bogus mending function, to be removed during conversion. Additionally, the expression tree dialect for scoped handlers includes a representation of the answer type modification to later be used during conversion.

It is possible to extend the program generation to introduce fully fledged scoped effects, although we did not have the time to do it in this work. An extension would be based on the deep algebraic effect generation. It would further need to introduce well-typed bodies for scoped operation bodies and mending functions.

Scoped operation implementations are of type $\mathit{scp}\ x \to (x \to \mathit{Freer\ sig\ gam}\ y) \to (y \to \mathit{Freer\ sig\ gam}\ a) \to \mathit{Freer\ sig\ gam}\ a$. The first function passed to this implementation is defined within a scoped operation call: $\mathit{ScopeCall} :: \mathit{gam}\ x \to (x \to \mathit{Freer\ sig\ gam}\ a) \to \mathit{Freer\ sig\ gam}\ a$. We would need to generate scoped operation types, similar to how we generate algebraic operation types. We would also need to generate values for $x$, such that $x$ can be used to select a scoped program using this function argument to scoped operation calls. $x$ would be used to select a scoped program within the function using **if**-expressions or further introduced branching functionality.

The main challenge for an implementation of program generation lies elsewhere, however. The mending function has a signature of $f\ x \to (x \to \mathit{Freer\ sig\ gam}\ (f\ a)) \to \mathit{Freer\ sig\ gam}\ (f\ a)$. In words, the mending function 'unpacks' a value of the answer type modification of $x$, namely $f\ x$. Generating interesting implementations of this function is challenging. This is because a value of *Freer sig gam a*

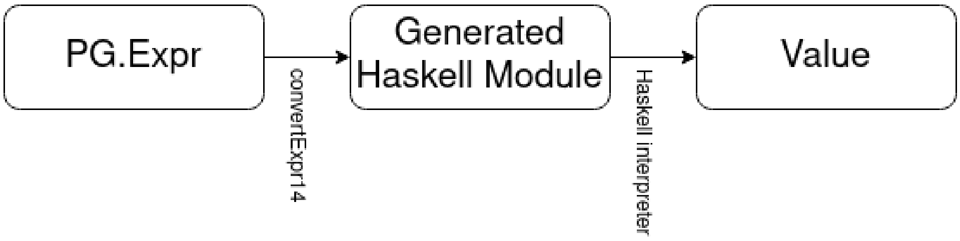



**Figure 6.5:** The process of converting a test expression into an interpreted value through means of a generated Haskell module.

can almost solely[2] be obtained through the function, and a value of *x* can only be obtained through this 'unpacking'. In general, we have not needed to do this sort of generation before in the generator, so this would form the biggest challenge. We imagine one would steer program generation towards values of that type by looking ahead through the type and finding expressions as paths from *f x* to *x*. Without further looking into such a solution, it's hard to say whether this approach would be able to fully enumerate all programs as well.

### Program Conversion

The generated expression tree programs are generated into various target expression tree syntaxes. This is done in one of two ways. Either the expression tree is directly converted to a new expression tree by inspecting every sub-tree and gradually converting that node to an equivalent sub-tree in the new expression tree (1), or every sub-tree is instead converted to Haskell code that represents that sub-tree (2). In both cases, the conversion function makes sure that all De Bruijn indices [25] present in the generated program tree are converted to names and replaced wherever referenced.

**Untyped Expression Trees**

These are the target for the first type of conversion. An untyped tree is created to represent the same program as our generated untyped expression tree represents. This is often done by simply converting an expression to its corresponding expression in the target tree. For instance, *PG.If* corresponds directly to *D.If*, where *PG* is the program generation module and *D* is the deep language module. Recursively converting an expression to its correponding target expression or to a call to its corresponding smart constructor is thus enough to convert most expressions. Even handlers can be converted rather easily.

**Typed Expression Trees**

Typed expression trees are a lot more difficult to convert to. The main difficulty in writing a conversion function is that it's impossible to type such a conversion function without arbitrarily introducing type information. To illustrate, the signature of such a conversion function for *Freer* trees is $PG.Expr \to Freer\ sig\ a$. The *sig* and *a* type parameters need to be introduced, but can be anything, as *PG.Expr* may expect any effects and return any type of value. We only know what these types should be at run-time. We circumvent this typing restriction by generating Haskell code to represent the target tree instead of converting to the tree directly. In the test, we make sure to run the generated Haskell code, thus delaying type-checking to this step. Figure 6.5 shows the process as a simple graph.

We use the template-haskell[3] library for its complete AST representation of Haskell with pretty-printing capabilities. The conversion function for converting expressions turns a generated expression into a Haskell AST expression representing the same (partial) program. For instance, we can convert a program like $BinOp\ (Lit\ (IntV\ 5))\ Add\ (Lit\ (IntV\ 1))$ to a program in the *Freer* tree:

$$binOp14\ (Pure\ 5)\ Add\ (Pure\ 1)$$

[2]It is also possible to obtain 'empty' values of type *f a* in some cases, but always mending to an empty value is quite boring, so one would want to use the function. For example, if *f a* = $[a]$, an returning empty list will satisfy the type.

[3]`https://hackage.haskell.org/package/template-haskell`

Where *binOp14* is a smart constructor to construct binary operation expressions in the *Freer* tree. Our conversion function would, however, return a representation of this Haskell code, to circumvent the need to type-check at compile-time. The binary operations-case for a conversion function from generated programs to a program in the *Freer* tree looks like this:

```
convertExpr0To14 (BinOp e1 bop e2) =
  appsE [
    varE $ mkName "binOp14",
    convertExpr0To14 e1,
    convertBinOp0To14 bop,
    convertExpr0To14 e2]
```

Every program generated through this way will need to be interpreted with a Haskell interpreter. We use hint[4] for this. After writing out a module of a number of programs, hint is run to compile the module (and thus type-check our generated programs) and run every individual program afterwards. This has as an added benefit that, although our program generator produces untyped expressions, tests in this step can still be used to check that our program generator produces well-typed expressions indirectly.

**Handlers**
Handlers offer another challenge for this type of program generation. With embedded effects and handlers, effects are represented as GADTs. Every constructor for such a GADT represents an operation that requires the effect. In the constructor, its parameter types represent the operation parameters and the instantiation of the last type parameter of the GADT represents the return type of the operation. With this in mind, we can convert every effect present in the expression tree into one of these GADTs.

The operation implementations of handlers differ a lot during the various transformations. In their most denotational form, operation implementations are functions that take the effect to handle and perform a case-match on that effect to find the operation. In the most operational form, however, multiple operation implementations each handle a single operation. We make sure that the conversion function for a specific target outputs the correct form of operation implementation.

**Scoped handlers**
Scoped handlers introduce a slight caveat on this generation process. Scoped effect handlers are parameterised with the answer type modification *f*, rather than the polymorphic *w*[5] seen in algebraic handlers. We thus need to offer a sensible type for *f* to allow our generated programs to type-check. We generate a **newtype**[6] declaration for the answer type modification *f* for every handler. The generated answer type modification is used in every place where a type *f* would be needed. To make sure the program type-checks, we then need to wrap the operation implementation and return results in this **newtype** and unwrap results of continuations and handle expressions. This makes sure we do not need to take care of the wrapping in our program generator.

### Testing Philosophy
For every program we perform the same tests that have as goal to confirm that interpreters maintain the same semantics. A program, generated or otherwise, is targeted with a specific language in mind already, but we still need to convert every program into values of the target expression trees. A test case consists of a few steps:

1. Convert the program into the representations for 2 interpreters, one is the reference interpreter and the other is the interpreter under test.
2. Run each program with its target interpreter.

[4] `https://hackage.haskell.org/package/hint`
[5] *w* is for most intents and purposes actually equivalent to *f a*, but allows no wrapping of *a*, unlike *f a*.
[6] A **type** declaration would be more desirable, but because Haskell does not allow partial type-alias application, we need to make due with **newtype**.

3. Convert both interpreting results to a single result representation (often that of the reference interpreter).
4. The test passes if the converted results are equal.

The reference interpreter is the same for all tests for the same language. Although this is not needed for verification reasons, using the same reference interpreter limits the number of conversions necessary. We thus test every step individually, but we also know that any two interpreters other than the reference interpreter yield the same results by transitivity of equality. This puts our intention of testing that the domain and codomain of two functions are the same into action.

The programs we test are 10000 generated programs per target interpreter. We expect each of these 10000 generated programs to be interpreted without erroring or returning an unhandled operation by every interpreter. We expect the target interpreter to give the same result as the reference interpreter, to verify that interpreters have the same semantics.

## 6.3. Conclusions and Considerations

We provide evidence that the program transformations we use to derive a freer monad-based embedding of effects and handlers from a corresponding denotational interpreter are well-known, standard transformations. However, we were unable to provide a full correspondence for every step. To truly know that each step preserves the semantics of the language an interpreter interprets, we need to complete the picture. We think it is possible to connect 'lettification', as we call it, to defunctionalisation and refunctionalisation, because the parts transformed show the typical signs of defunctionalisation, but we have yet to fully show their correspondence. Aside from this, we claim that our introduction of intrinsic typing in the expression tree assumes typing rules that should be preferred to be specified explicitly in future work.

As for the application of our program transformations, we describe the test suite we use to evaluate the correctness of our implementations. These tests check that interpreters have the same codomain for a generated domain. In other words, we test that interpreters have the same behaviour by generating a set of programs as input to these interpreters and check that each interpreter outputs the same values.

During the application of the program transformations, many things could go wrong. A couple instances of such problems are:

- Copying *interpreter1*, we would search 1 and replace it with 2 to update the names of functions and types. However, variable names would also sometimes contain 1, introducing name conflicts if another variable with the same name was already present.
- Another simple type of copy paste issue could happen when working on binary operations. We could sometimes introduce a case with the wrong semantics for one of the binary operations.
- At times, we missed cases when we had to implement a new functionality such as substitution.
- Handling shallow effects experiences a subtle change when we just as subtly removed a dependency on the laziness of the *defining* language (Haskell).

All these types of mistakes in applying and implementing the program transformations were caught by our generated tests.

However, the tests we ran could have been executed a little better if we took the time to do any of the following improvements:

- We could have not only ran the tests just to see that they passed, but we could have looked into the coverage that these tests had. For instance, we might have ran simple line or branch coverage. However, because of how we convert untyped expression trees into typed expression trees, a coverage measuring tool should take into account in-execution calls to GHC. We may well have also looked into mutation testing to find exactly what type of errors were left uncaught by the tests.
- The programs we generate currently are restricted to be deterministic and always well-typed. A future iteration of such a generator might also consider not well-typed and non-deterministic programs. However, the assumption of determinism is interwoven into the program generation and testing method rather tightly at this moment. A solution that should support both these, might need to rethink the approach to program generation.

- On top of this, program generation of effect handlers is currently limited to handlers that display deep algebraic-like handling semantics. These handlers are then converted and embedded in shallow and deep scoped handlers, simulating only deep algebraic semantics. We have explained a way to extend the program generation rules to include shallow and scoped handlers, but we have yet to implement these rules.

More structurally, dynamic tests could only verify that a set of interpreters all implement the same semantics to some degree of confidence. To know for sure that this is the case, a proof would be required. Although it is indeed possible to write such a proof for every pair of interpreters in Coq or Agda, we have not tried this, as our work was written in Haskell and would take a considerable time to convert and prove.

# 7 Related Work

We relate the small-step operational semantics for languages with effects and handlers to their corresponding free monad embeddings in Haskell, closely modelling their denotational semantics. Previous work has related the denotational semantics for effects and handlers to their free monad model [28], as well as a model for delimited control. In fact, previous work relates effects and handlers to delimited control more often [18, 69]. These works show that both in typed and untyped settings, deep handlers are related to the *shift*$_0$ operator, and, likewise, shallow handlers are related to the *control*$_0$ operator. These works provide mathematical approaches to proving a relation between delimited control and effects and handlers. More practically, previous works have shown both partial [56] and full [33, 32] CPS translations for deep and shallow algebraic effect handlers.

We relate different semantics through series of small, discernible program transformations, like demonstrated by Danvy [20] and Vesely and Fisher [78]. Specifically, work involving exceptions demonstrates stack unwinding relates closely to this work: Danvy demonstrates the relation between [19], and Hutton and Wright show how to derive a compiler from a corresponding interpreter [39]. This is in the spirit of Hoare's seminal work *Unifying Theories of Programming* [34], wherein Hoare states "What is needed is a deep understanding of the relationships between the different models and theories, and a sound judgment of the most appropriate area of application of each of them". Danvy relates small-step to big-step operational semantics, Vesely and Fisher relate big-step back to small-step operational semantics [78]. Work relating operational to denotational semantics exists as well, relating the worst-case execution paths mathematically [77], relating semantics for web services [82], and relating semantics for Verilog [37, 38]. All of the aforementioned works relating operational and denotational semantics do so through a single conversion, unlike how Danvy takes us through a semantic park [21] when he shows the various types of semantics in between small-step operational and big-step operational semantics.

The work we consider to be closest to this work in its goals and results is [1], in which Sig Ager, Danvy, and Midtgaard show the relation between various monadic evaluators and an abstract machine for languages with computational effects. Another work very closely related to this work is the PhD thesis of Biernacki, in which he relates the various semantics of programs with delimited continuations [12]. Other work similar in nature are ones relating the abstract machines and denotational semantics of the gradually-typed lambda calculus [30] and functional languages [11].

We test our program transformations with programs generated using type-directed program synthesis techniques. We make use of random testing using QuickCheck [16], where inputs are generated to check that a user-defined property holds. Approaches such as Adaptive Random Testing [15, 35] focus on the uniformity of the generated inputs. Extensions of which exist for generating non-numeric inputs [43]. An extension on this method of testing is that of symbolic execution [47, 17]. This system of testing would describe the properties of the interpreter under test and test properties hold symbolically.

The program generation approach is based on Pałka's work [67], which of itself is an implementation of constraint-based program synthesis [65]. A work by Juhošová basing itself on Pałka's work like ours focuses on generating both well-typed and ill-typed expressions [43]. For Haskell specifically, tools such as TYGAR [29], Scythe [65], Djinn [1], and Wingman [2] synthesize Haskell expressions to help auto-

[1] `https://hackage.haskell.org/package/djinn`

[2] `https://hackage.haskell.org/package/hls-tactics-plugin`

completions. Aside from randomly searching, many works instead base their approach on example programs [66, 51]. Alternatively, one might look toward symbolic execution for generating tests [14]. Recently, this hybrid approach was used to generate test suites for interpreters written in Scala [5].

As for evaluating our program generation procedure, we might have looked towards various forms of coverage, such as the well-known line coverage, branch coverage, but also pairwise combinatorial coverage [31], multi-way combinatorial coverage [52], mutation adequacy score [40], or even higher-order mutation adequacy [41]. Tools such as QuickCover [31] and MuCheck [54] make these evaluations available in Haskell. Using such coverage measures, we could have improved our generated test inputs incrementally [53, 62, 31, 45]. Work by Allwood et al. supports fully black box test suite generation in Haskell [3].

# 8
# Conclusion

After presenting the generally known transformations to derive a denotational interpreter from a small-step operational semantics and vice versa in Chapter 2, we showed our own transformations to take an operational semantics all the way to a freer monad-based embedding for deep effects and handlers in Chapter 3, and back for shallow effects in Chapter 4 and scoped effects in Chapter 5. We evaluated the transformations we add to derive the embedding from a denotational interpreter and back, and our applications of these same transformations in Chapter 6.

In Chapter 4, we derive a novel set of operational semantics for deep scoped effects and handlers. These operational semantics show that scoped effects defined with the helping construct of a 'mending function' use the mending function only to thread handling in the case of an unhandled scoped operation (figure 8.1). Other than this case, scoped operations appear to be very similar to algebraic operations, threading handling into two continuations rather than the single continuation provided by algebraic operations.

$$\frac{}{\begin{gathered}\textbf{handle}\ h@\{\mathit{eff}, .., \mathit{mend}\ x_m \mapsto e_m\}\ (\mathit{scope}\ \mathit{eff}'\ \mathit{scp}_i\ v\ x_{rec}\ e_{rec}\ X_{op}[]) \to \\ \mathit{scope}\ \mathit{eff}'\ \mathit{scp}_i\ v\ x_{rec}\ (\textbf{handle}\ h\ e_{rec})\ (\textbf{let}\ x_m = []\ \textbf{in}\ e_m\ [\mathit{resume}/(y \mapsto \textbf{handle}\ h\ X_{op}[y])]) \\ \text{if}\ \mathit{eff} \not\equiv \mathit{eff}'\end{gathered}}$$

**Figure 8.1:** Part of the scoped effects structural operational semantics

The program transformations we add are those that transform an untyped denotational interpreter to a freer monad-based embedding in Haskell and vice versa. We claim that these program transformations are standard and can be shown to originate in known transformations such as specialisation and generalisation, and defunctionalisation and refunctionalisation. The testing we have done has yet to indicate otherwise.

We evaluated the applications of program transformations in Chapters 3 to 5 with generated and manual programs rather than a formal proof. The process described in Chapter 6 generates untyped expression trees and converts those trees into untyped and typed expression trees to be interpreted with various interpreter functions. This process is not perfect. Not all features of the languages we transform can be generated. We only generate deep handler semantics and convert those to the semantics required for each language. Therefore, we do not generate programs that test scoped operations or utilise the possibility of mutually recursive shallow effect handlers, for instance. Future work may remedy this by extending the program generation functions we use here, but we believe this might require rethinking our assumptions. This might require generation of non-deterministic programs, which our program generation function explicitly avoids.

We show that it is possible to test the applications of our program transformations on embeddings of effects and handlers, but we cannot show a complete proof that the transformations work on every language. We claim that it can be proven that our interpreters implement the same semantics, however. By breaking up the transformation process into small steps, we do not only facilitate understanding of each transformation, but we also facilitate writing a proof. Without a proof, our testing method is good, but could be better. We could have measured the coverage of our tests through mutation testing and it is still possible to improve that coverage by implementing generation rules for shallow and scoped effects and handlers. Additionally, one might look into generating recursive programs as well as erroring programs to also investigate bad-weather behaviours of the interpreters under test.

## Future Work

**Generalisation of our transformations**
The transformational steps we introduced may not generalise to be applicable to every single language that implements effects and handlers. We might see that some languages cannot be represented with a freer-monad-like tree and might indeed need some different abstraction. This might hinder our ability to generically apply these steps to get the best abstraction. We cannot currently guarantee that these steps are generally applicable and future work would thus be needed to try to convert different implementations of effects and handlers. Future work may start by converting the model for denotational semantics of latent effects [10] to an operational semantics, as a first check. If the steps cannot generically apply to these semantics, perhaps adjustments can be made.

**A complete proof**
As it stands, we justified our strong suspicion that the program transformations we add are ‘standard’ transformations, known to preserve the behaviour of the transformed programs. However, a full connection and proof of that connection is yet to be produced. A future work could look into these connections and formally connect lettification to refunctionalisation, for instance. Formalisations of such kinds would also give some insight into the way these transformations generalise to other types of effects and handlers.

**Incomplete program generation**
We currently generate programs in a target language with deep algebraic handler semantics turned into the target handler semantics. To properly cover the search space, we believe the program generator would need to be extended to support generating scoped syntax. A solution would need to be created to generate mending functions of type $f\ x \to (x \to \mathit{Freer\ sig\ gam\ a}) \to \mathit{Freer\ sig\ gam\ a}$, as discussed in Section 6.2.1. The difficulty with this function is that a search for a value of type $x$ from the argument of type $f\ x$ is hard to imagine to generally enumerate all possible expressions to fill the gap.

**Different program generation techniques**
Our untyped approach is useful for being relatively easy to implement and describe, but does not offer type-level guarantees about the programs generated. A program generator written with dependent types might be able to generate typed expression trees that can be directly converted to other typed expression trees. This would remove the need for special conversion functions that generate Haskell code. However, this would come with its own challenges, such as typing the environment of effects and scoped effects of which operations can be called.

**Evaluating quality of generated programs**
Currently, we have no way of properly evaluating the quality of the programs generated. For this, we would look towards a metric such as combinatorial coverage. Combinatorial coverage can give an indication of how many 2-way combinations of interactions are tested with a test-set and can reduce the test-set to the set of tests that is most relevant to getting a high coverage. A measurement can be made using the framework introduced by Goldstein et al. [31], to find how much of the search space is likely covered by our generated programs.

In combination with this, we might be able to perform some additional tests with less effort to find out how much our generated programs may cover. We may define a set of features that we wish to be covered by generated programs and see whether each of those features are used by a program and, importantly, tested by the interpreter. The second part is important because not every part of a program is eventually executed. This would give an indication of what features might need to be prioritised in generation to optimally generate interesting programs.

# References


[1] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. “A functional correspondence between monadic evaluators and abstract machines for languages with computational effects”. In: *Theoretical Computer Science* 342.1 (2005). Applied Semantics: Selected Topics, pp. 149–172. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2005.06.008. URL: https://www.sciencedirect.com/science/article/pii/S0304397505003439.

[2] Mads Sig Ager et al. “A Functional Correspondence between Evaluators and Abstract Machines”. In: *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming*. PPDP ’03. Uppsala, Sweden: Association for Computing Machinery, 2003, pp. 8–19. ISBN: 1581137052. DOI: 10.1145/888251.888254. URL: https://doi.org/10.1145/888251.888254.

[3] Tristan Allwood, Cristian Cadar, and Susan Eisenbach. “High Coverage Testing of Haskell Programs”. In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ISSTA ’11. Toronto, Ontario, Canada: Association for Computing Machinery, 2011, pp. 375–385. ISBN: 9781450305624. DOI: 10.1145/2001420.2001465. URL: https://doi.org/10.1145/2001420.2001465.

[4] Carol AR Andrews. *The Rosetta Stone*. British Museum Publications London, 1981.

[5] Wesley Baartman. “Towards Automatic Test Suite Generation for Functional Programming Assignments using Budgeted Compositional Symbolic”. In: (2022).

[6] Casper Bach Poulsen and Cas van der Rest. “Hefty Algebras: Modular Elaboration of Higher-Order Algebraic Effects”. In: *Proc. ACM Program. Lang.* 7.POPL (Jan. 2023). DOI: 10.1145/3571255. URL: https://doi.org/10.1145/3571255.

[7] Andrej Bauer and Matija Pretnar. “An Effect System for Algebraic Effects and Handlers”. In: *Algebra and Coalgebra in Computer Science*. Ed. by Reiko Heckel and Stefan Milius. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–16. ISBN: 978-3-642-40206-7.

[8] Andrej Bauer and Matija Pretnar. “Programming with algebraic effects and handlers”. In: *Journal of Logical and Algebraic Methods in Programming* 84.1 (2015). Special Issue: The 23rd Nordic Workshop on Programming Theory (NWPT 2011) Special Issue: Domains X, International workshop on Domain Theory and applications, Swansea, 5-7 September, 2011, pp. 108–123. ISSN: 2352-2208. DOI: https://doi.org/10.1016/j.jlamp.2014.02.001. URL: https://www.sciencedirect.com/science/article/pii/S2352220814000194.

[9] Nick Benton et al. “Strongly Typed Term Representations in Coq”. In: *Journal of Automated Reasoning - JAR* 49 (Aug. 2012), pp. 1–19. DOI: 10.1007/s10817-011-9219-0.

[10] Birthe van den Berg et al. “Latent Effects for Reusable Language Components”. In: *Programming Languages and Systems*. Ed. by Hakjoo Oh. Cham: Springer International Publishing, 2021, pp. 182–201. ISBN: 978-3-030-89051-3.

[11] Ma lgorzata Biernacka. “A derivational approach to the operational semantics of functional languages”. PhD thesis. Citeseer, 2006.

[12] Dariusz Biernacki. “The theory and practice of programming languages with delimited continuations”. PhD thesis. Citeseer, 2005.

[13] Guillaume Boisseau and Jeremy Gibbons. “What You Needa Know about Yoneda: Profunctor Optics and the Yoneda Lemma (Functional Pearl)”. In: *Proc. ACM Program. Lang.* 2.ICFP (July 2018). DOI: 10.1145/3236779. URL: https://doi.org/10.1145/3236779.

[14] Cristian Cadar and Koushik Sen. “Symbolic execution for software testing: three decades later”. In: *Communications of the ACM* 56.2 (2013), pp. 82–90.

[15] T. Y. Chen, H. Leung, and I. K. Mak. “Adaptive Random Testing”. In: *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making*. Ed. by Michael J. Maher. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 320–329. ISBN: 978-3-540-30502-6.

[16] Koen Claessen and John Hughes. “QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs”. In: *SIGPLAN Not.* 35.9 (Sept. 2000), pp. 268–279. ISSN: 0362-1340. DOI: 10.1145/357766.351266. URL: https://doi.org/10.1145/357766.351266.

[17] Lori A. Clarke. “A Program Testing System”. In: *Proceedings of the 1976 Annual Conference*. ACM ’76. Houston, Texas, USA: Association for Computing Machinery, 1976, pp. 488–491. ISBN: 9781450374897. DOI: 10.1145/800191.805647. URL: https://doi.org/10.1145/800191.805647.

[18] Youyou Cong and Kenichi Asai. “Understanding Algebraic Effect Handlers via Delimited Control Operators”. In: *Trends in Functional Programming*. Ed. by Wouter Swierstra and Nicolas Wu. Cham: Springer International Publishing, 2022, pp. 59–79. ISBN: 978-3-031-21314-4.

[19] Olivier Danvy. “Defunctionalized Interpreters for Programming Languages”. In: *SIGPLAN Not.* 43.9 (Sept. 2008), pp. 131–142. ISSN: 0362-1340. DOI: 10.1145/1411203.1411206. URL: https://doi.org/10.1145/1411203.1411206.

[20] Olivier Danvy. “From Reduction-Based to Reduction-Free Normalization”. In: *Electronic Notes in Theoretical Computer Science* 124 (Apr. 2005), pp. 79–100. DOI: 10.1016/j.entcs.2005.01.007.

[21] Olivier Danvy, Jacob Johannsen, and Ian Zerny. “A Walk in the Semantic Park”. In: *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. PEPM ’11. Austin, Texas, USA: Association for Computing Machinery, 2011, pp. 1–12. ISBN: 9781450304856. DOI: 10.1145/1929501.1929503. URL: https://doi.org/10.1145/1929501.1929503.

[22] Olivier Danvy and Kevin Millikin. “On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion”. In: *Information Processing Letters* 106.3 (2008), pp. 100–109. ISSN: 0020-0190. DOI: https://doi.org/10.1016/j.ipl.2007.10.010. URL: https://www.sciencedirect.com/science/article/pii/S0020019007003018.

[23] Olivier Danvy and Kevin Millikin. “Refunctionalization at work”. In: *Science of Computer Programming* 74.8 (2009). Special Issue on Mathematics of Program Construction (MPC 2006), pp. 534–549. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2007.10.007. URL: https://www.sciencedirect.com/science/article/pii/S0167642309000227.

[24] Olivier Danvy and Lasse R. Nielsen. “Defunctionalization at Work”. In: *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. PPDP ’01. Florence, Italy: Association for Computing Machinery, 2001, pp. 162–174. ISBN: 158113388X. DOI: 10.1145/773184.773202. URL: https://doi.org/10.1145/773184.773202.

[25] N.G de Bruijn. “Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem”. In: *Indagationes Mathematicae (Proceedings)* 75.5 (1972), pp. 381–392. ISSN: 1385-7258. DOI: https://doi.org/10.1016/1385-7258(72)90034-0. URL: https://www.sciencedirect.com/science/article/pii/1385725872900340.

[26] Matthias Felleisen and Robert Hieb. “The revised report on the syntactic theories of sequential control and state”. In: *Theoretical Computer Science* 103.2 (1992), pp. 235–271. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(92)90014-7. URL: https://www.sciencedirect.com/science/article/pii/0304397592900147.

[27] Yannick Forster et al. “On the Expressive Power of User-Defined Effects: Effect Handlers, Monadic Reflection, Delimited Control”. In: *Proc. ACM Program. Lang.* 1.ICFP (Aug. 2017). DOI: 10.1145/3110257. URL: https://doi.org/10.1145/3110257.

[28] Yannick Forster et al. “On the Expressive Power of User-Defined Effects: Effect Handlers, Monadic Reflection, Delimited Control”. In: *Proc. ACM Program. Lang.* 1.ICFP (Aug. 2017). DOI: 10.1145/3110257. URL: https://doi.org/10.1145/3110257.

[29] Didier Galmiche. “Constructive system for automatic program synthesis”. In: *Theoretical Computer Science* 71.2 (1990), pp. 227–239. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(90)90199-R. URL: https://www.sciencedirect.com/science/article/pii/030439759090199R.

[30] Álvaro García-Pérez, Pablo Nogueira, and Ilya Sergey. “Deriving Interpretations of the Gradually-Typed Lambda Calculus”. In: *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*. PEPM ’14. San Diego, California, USA: Association for Computing Machinery, 2014, pp. 157–168. ISBN: 9781450326193. DOI: 10.1145/2543728.2543742. URL: https://doi.org/10.1145/2543728.2543742.

[31] Harrison Goldstein et al. “Do Judge a Test by its Cover”. In: *European Symposium on Programming*. Springer, Cham. 2021, pp. 264–291.

[32] Daniel Hillerström and Sam Lindley. “Shallow Effect Handlers”. In: *Programming Languages and Systems*. Ed. by Sukyoung Ryu. Cham: Springer International Publishing, 2018, pp. 415–435. ISBN: 978-3-030-02768-1.

[33] Daniel Hillerström et al. “Continuation passing style for effect handlers”. In: *2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2017)*. Ed. by Dale Miller. Leibniz International Proceedings in Informatics (LIPIcs). Germany: Dagstuhl Publishing, Sept. 2017, 18:1–18:19. DOI: 10.4230/LIPIcs.FSCD.2017.18. URL: https://doi.org/10.4230/LIPIcs.FSCD.2017.18.

[34] Charles Antony Richard Hoare and He Jifeng. *Unifying theories of programming*. Vol. 14. Prentice Hall Englewood Cliffs, 1998.

[35] Rubing Huang et al. “A Survey on Adaptive Random Testing”. In: *IEEE Transactions on Software Engineering* 47.10 (2021), pp. 2052–2083. DOI: 10.1109/TSE.2019.2942921.

[36] Paul Hudak et al. “A History of Haskell: Being Lazy with Class”. In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. San Diego, California: Association for Computing Machinery, 2007, pp. 12–1–12–55. ISBN: 9781595937667. DOI: 10.1145/1238844.1238856. URL: https://doi.org/10.1145/1238844.1238856.

[37] Z. Huibiao, J.P. Bowen, and He Jifeng. “Deriving operational semantics from denotational semantics for Verilog”. In: *Proceedings Eighth Asia-Pacific Software Engineering Conference*. 2001, pp. 177–184. DOI: 10.1109/APSEC.2001.991475.

[38] Zhu Huibiao, Jonathan P. Bowen, and He Jifeng. “From Operational Semantics to Denotational Semantics for Verilog”. In: *Correct Hardware Design and Verification Methods*. Ed. by Tiziana Margaria and Tom Melham. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 449–464. ISBN: 978-3-540-44798-6.

[39] Graham Hutton and Joel J Wright. “Calculating an exceptional machine.” In: *Trends in Functional Programming* 5 (2004), pp. 49–64.

[40] Yue Jia and Mark Harman. “An Analysis and Survey of the Development of Mutation Testing”. In: *IEEE Transactions on Software Engineering* 37.5 (2011), pp. 649–678. DOI: 10.1109/TSE.2010.62.

[41] Yue Jia and Mark Harman. “Higher Order Mutation Testing”. In: *Information and Software Technology* 51.10 (2009). Source Code Analysis and Manipulation, SCAM 2008, pp. 1379–1393. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2009.04.016. URL: https://www.sciencedirect.com/science/article/pii/S0950584909000688.

[42] Mark P Jones and Luc Duponcheel. *Composing monads*. Tech. rep. Technical Report YALEU/DCS/RR-1004, Department of Computer Science. Yale …, 1993.

[43] Sára Juhošová. “Validating Type Checkers Using Property-Based Testing”. In: (2021).

[44] Ohad Kammar, Sam Lindley, and Nicolas Oury. “Handlers in Action”. In: *SIGPLAN Not.* 48.9 (Sept. 2013), pp. 145–158. ISSN: 0362-1340. DOI: 10.1145/2544174.2500590. URL: https://doi.org/10.1145/2544174.2500590.

[45] Manju Khari et al. “Performance analysis of six meta-heuristic algorithms over automated test suite generation for path coverage-based optimization”. In: *Soft Computing* 24.12 (2020), pp. 9143–9160.

[46] David J. King and P Wadler. "Combining Monads". English. In: *Glasgow Workshop on Functional Programming*. Workshops in Computing. United Kingdom: Springer London, 1992, pp. 134–143. ISBN: 978-3-540-19820-8. DOI: 10.1007/978-1-4471-3215-8_12.

[47] James C. King. "Symbolic Execution and Program Testing". In: *Commun. ACM* 19.7 (July 1976), pp. 385–394. ISSN: 0001-0782. DOI: 10.1145/360248.360252. URL: https://doi.org/10.1145/360248.360252.

[48] Oleg Kiselyov and Hiromi Ishii. "Freer Monads, More Extensible Effects". In: *SIGPLAN Not.* 50.12 (Aug. 2015), pp. 94–105. ISSN: 0362-1340. DOI: 10.1145/2887747.2804319. URL: https://doi.org/10.1145/2887747.2804319.

[49] Oleg Kiselyov, Amr Sabry, and Cameron Swords. "Extensible Effects: An Alternative to Monad Transformers". In: *SIGPLAN Not.* 48.12 (Sept. 2013), pp. 59–70. ISSN: 0362-1340. DOI: 10.1145/2578854.2503791. URL: https://doi.org/10.1145/2578854.2503791.

[50] Casey Klein, Matthew Flatt, and Robert Bruce Findler. "Random Testing for Higher-Order, Stateful Programs". In: *SIGPLAN Not.* 45.10 (Oct. 2010), pp. 555–566. ISSN: 0362-1340. DOI: 10.1145/1932682.1869505. URL: https://doi.org/10.1145/1932682.1869505.

[51] Tristan Knoth et al. "Resource-Guided Program Synthesis". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 253–268. ISBN: 9781450367127. DOI: 10.1145/3314221.3314602. URL: https://doi.org/10.1145/3314221.3314602.

[52] Rick Kuhn, Yu Lei, and Raghu Kacker. "Practical Combinatorial Testing: Beyond Pairwise". In: *IT Professional* 10.3 (2008), pp. 19–23. DOI: 10.1109/MITP.2008.54.

[53] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. "Coverage Guided, Property Based Testing". In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019). DOI: 10.1145/3360607. URL: https://doi.org/10.1145/3360607.

[54] Duc Le et al. "MuCheck: An Extensible Tool for Mutation Testing of Haskell Programs". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISSTA 2014. San Jose, CA, USA: Association for Computing Machinery, 2014, pp. 429–432. ISBN: 9781450326452. DOI: 10.1145/2610384.2628052. URL: https://doi.org/10.1145/2610384.2628052.

[55] Daan Leijen. "Koka: Programming with Row Polymorphic Effect Types". In: *Electronic Proceedings in Theoretical Computer Science* 153 (June 2014), pp. 100–126. DOI: 10.4204/eptcs.153.8. URL: https://doi.org/10.4204%2Feptcs.153.8.

[56] Daan Leijen. "Type Directed Compilation of Row-Typed Algebraic Effects". In: *SIGPLAN Not.* 52.1 (Jan. 2017), pp. 486–499. ISSN: 0362-1340. DOI: 10.1145/3093333.3009872. URL: https://doi.org/10.1145/3093333.3009872.

[57] Sheng Liang, Paul Hudak, and Mark Jones. "Monad Transformers and Modular Interpreters". In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '95. San Francisco, California, USA: Association for Computing Machinery, 1995, pp. 333–343. ISBN: 0897916921. DOI: 10.1145/199448.199528. URL: https://doi.org/10.1145/199448.199528.

[58] Sam Lindley, Conor McBride, and Craig McLaughlin. "Do be do be do". In: *CoRR* abs/1611.09259 (2016). arXiv: 1611.09259. URL: http://arxiv.org/abs/1611.09259.

[59] Christoph Lüth and Neil Ghani. "Composing Monads Using Coproducts". In: *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*. ICFP '02. Pittsburgh, PA, USA: Association for Computing Machinery, 2002, pp. 133–144. ISBN: 1581134878. DOI: 10.1145/581478.581492. URL: https://doi.org/10.1145/581478.581492.

[60] Zohar Manna and Richard Waldinger. "A Deductive Approach to Program Synthesis". In: *ACM Trans. Program. Lang. Syst.* 2.1 (Jan. 1980), pp. 90–121. ISSN: 0164-0925. DOI: 10.1145/357084.357090. URL: https://doi.org/10.1145/357084.357090.

[61] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. "Typed Closure Conversion". In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '96. St. Petersburg Beach, Florida, USA: Association for Computing Machinery, 1996, pp. 271–283. ISBN: 0897917693. DOI: `10.1145/237721.237791`. URL: `https://doi.org/10.1145/237721.237791`.

[62] Saahil Ognawala et al. "Improving Function Coverage with Munch: A Hybrid Fuzzing and Directed Symbolic Execution Approach". In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. SAC '18. Pau, France: Association for Computing Machinery, 2018, pp. 1475–1482. ISBN: 9781450351911. DOI: `10.1145/3167132.3167289`. URL: `https://doi.org/10.1145/3167132.3167289`.

[63] Atsushi Ohori and Isao Sasano. "Lightweight fusion by fixed point promotion". In: *ACM SIGPLAN Notices* 42.1 (2007), pp. 143–154.

[64] C-H Luke Ong. "Correspondence between operational and denotational semantics: the full abstraction problem for PCF". In: *Handbook of logic in computer science* 4 (1995), pp. 269–356.

[65] Peter-Michael Osera. "Constraint-Based Type-Directed Program Synthesis". In: *Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development*. TyDe 2019. Berlin, Germany: Association for Computing Machinery, 2019, pp. 64–76. ISBN: 9781450368155. DOI: `10.1145/3331554.3342608`. URL: `https://doi.org/10.1145/3331554.3342608`.

[66] Peter-Michael Osera and Steve Zdancewic. "Type-and-Example-Directed Program Synthesis". In: *SIGPLAN Not.* 50.6 (June 2015), pp. 619–630. ISSN: 0362-1340. DOI: `10.1145/2813885.2738007`. URL: `https://doi.org/10.1145/2813885.2738007`.

[67] Michał H. Pałka et al. "Testing an Optimising Compiler by Generating Random Lambda Terms". In: *Proceedings of the 6th International Workshop on Automation of Software Test*. AST '11. Waikiki, Honolulu, HI, USA: Association for Computing Machinery, 2011, pp. 91–97. ISBN: 9781450305921. DOI: `10.1145/1982595.1982615`. URL: `https://doi.org/10.1145/1982595.1982615`.

[68] F. Pfenning and C. Elliott. "Higher-Order Abstract Syntax". In: *SIGPLAN Not.* 23.7 (June 1988), pp. 199–208. ISSN: 0362-1340. DOI: `10.1145/960116.54010`. URL: `https://doi.org/10.1145/960116.54010`.

[69] Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. "Typed Equivalence of Effect Handlers and Delimited Control". In: *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. Ed. by Herman Geuvers. Vol. 131. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 30:1–30:16. ISBN: 978-3-95977-107-8. DOI: `10.4230/LIPIcs.FSCD.2019.30`. URL: `http://drops.dagstuhl.de/opus/volltexte/2019/10537`.

[70] Gordon Plotkin and Matija Pretnar. "Handlers of Algebraic Effects". In: *Programming Languages and Systems*. Ed. by Giuseppe Castagna. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 80–94. ISBN: 978-3-642-00590-9.

[71] Gordon D Plotkin. *A structural approach to operational semantics*. Aarhus university, 1981.

[72] Casper Bach Poulsen, Cas van der Rest, and Tom Schrijvers. "Staged effects and handlers for modular languages with abstraction". In: *Workshop on Partial Evaluation and Program Manipulation (PEPM)*. 2021.

[73] Matija Pretnar. "An Introduction to Algebraic Effects and Handlers. Invited tutorial paper". In: *Electronic Notes in Theoretical Computer Science* 319 (2015). The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI)., pp. 19–35. ISSN: 1571-0661. DOI: `https://doi.org/10.1016/j.entcs.2015.12.003`. URL: `https://www.sciencedirect.com/science/article/pii/S1571066115000705`.

[74] John C Reynolds. "Definitional interpreters for higher-order programming languages". In: *Proceedings of the ACM annual conference-Volume 2*. 1972, pp. 717–740.

[75] Guy L. Steele. "Building Interpreters by Composing Monads". In: *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '94. Portland, Oregon, USA: Association for Computing Machinery, 1994, pp. 472–492. ISBN: 0897916360. DOI: `10.1145/174675.178068`. URL: `https://doi.org/10.1145/174675.178068`.

[76] Wouter Swierstra. "Data types à la carte". In: *Journal of Functional Programming* 18.4 (2008), pp. 423–436. DOI: 10.1017/S0956796808006758.

[77] Fairouz Tchier. "Demonic Operational and Denotational Semantics". In: *Applied Mathematical Sciences* 2.18 (2008), pp. 861–881.

[78] Ferdinand Vesely and Kathleen Fisher. "One Step at a Time". In: *European Symposium on Programming*. Springer, Cham. 2019, pp. 205–231.

[79] Philip Wadler. "Monads for functional programming". In: *Advanced Functional Programming*. Ed. by Johan Jeuring and Erik Meijer. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 24–52. ISBN: 978-3-540-49270-2.

[80] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. "Effect Handlers in Scope". In: *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*. Haskell '14. Gothenburg, Sweden: Association for Computing Machinery, 2014, pp. 1–12. ISBN: 9781450330411. DOI: 10.1145/2633357.2633358. URL: https://doi.org/10.1145/2633357.2633358.

[81] Zhixuan Yang et al. "Structured Handling of Scoped Effects". In: Jan. 2022, pp. 462–491. ISBN: 978-3-030-99335-1. DOI: 10.1007/978-3-030-99336-8_17.

[82] Huibiao Zhu et al. "Linking denotational semantics with operational semantics for web services". In: *Innovations in Systems and Software Engineering* 6 (2010), pp. 283–298.