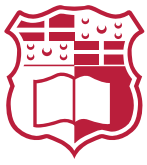
Expressing Monitors using Algebraic Effects in OCaml

Marietta Galea

Supervisor: Prof. Adrian Francalanza

May 2024

Submitted in partial fulfilment of the requirements
for the degree of BSc. (Hons.) in Mathematics and Computer Science.



L-Università ta' Malta
Faculty of Information &
Communication Technology

Abstract

This study explores the application of algebraic effects and handlers in expressing runtime verification monitors within the OCaml programming language. Runtime verification is a lightweight technique for ensuring that a system's behavior conforms to specified properties during execution. Traditional approaches to implementing monitors can be cumbersome and intrusive, often intertwining monitoring logic with the program's core functionality. By leveraging the theory of algebraic effects, we propose a more modular and systematic approach. Algebraic effects allow the separation of effectful operations from their handling logic, providing a clear and flexible framework for defining and managing runtime monitors. Utilizing OCaml's recently introduced Effects library, we transform theoretical models of monitors into practical, executable programs. This approach not only simplifies the development and maintenance of monitors but also enhances their correctness and reliability. We demonstrate the viability of this method by implementing several classes of monitoring systems, both stand-alone and layered, and evaluate their performance and effectiveness. Our findings suggest that algebraic effects provide a powerful and expressive tool for runtime verification, offering significant advantages in terms of modularity, composability, and clarity over traditional methods. This work opens new avenues for research and development in the domain of runtime verification, particularly in functional programming environments like OCaml.

Acknowledgements

The work presented here would not be possible without the following people. I am deeply grateful to Professor Adrian Francalanza for his patience and excellent mentorship over the past year. I deeply appreciate our productive meetings and amiable conversations, but above all, I am thankful for his ongoing support and encouragement.

I am also deeply grateful to my partner, Philip, and my circle of friends for their unwavering support, compassionate listening, and uplifting conversations during moments of stress. Last but not least, to my dear parents and brother, for their unconditional love, unwavering belief in my abilities, and steadfast support throughout every step of this challenging yet rewarding journey. Their encouragement has been the cornerstone of my success, and I am profoundly grateful for their presence in my life.

Contents

Abstract i

Acknowledgements ii

Contents iv

List of Figures v

List of Listings vi

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Aim . . . 2
- 1.3 Objectives . . . 2
- 1.4 Overview . . . 2

2 Algebraic Theories 3
- 2.1 An Introduction to Algebraic Effects . . . 3
- 2.2 Computational Effects as Algebraic Effects . . . 8
- 2.3 Algebraic Effect Handlers . . . 10

3 Stand-alone Monitors 12
- 3.1 The General Program . . . 12
- 3.2 Instrumentation . . . 14
- 3.3 A Stateless Monitor . . . 15
- 3.4 Stateful Monitors . . . 18
  - 3.4.1 Aggregate Value Monitor . . . 18
  - 3.4.2 Alternating Parity Monitor . . . 21
- 3.5 A Monitor for the Aliasing Problem . . . 22
- 3.6 Joining Two Monitors Together . . . 23

4 Layered Monitoring 25

5 Conclusion 28

5.1 Limitations 28
5.2 Related and Future Work 28
5.2.1 Algebraic Effects versus Monads 29

A Further Background on Algebraic Effects **33**

List of Figures

Figure 2.1 Free Model 7

Figure 4.1 Composing Monitors 26

Listings

3.1 Defining the effects put and get in OCaml 14
3.2 Defining the effects put and get in OCaml 14
3.3 Instrumenting the program 15
3.4 Handler Signature 16
3.5 Stateless Monitor 16
3.6 Continuation Signature 18
3.7 Aggregate Value Monitor 19
3.8 Enhanced Aggregate Value Monitor 20
3.9 Alternating Parity Monitor 21
3.10 Sample Program 22
3.11 Aliasing Monitor 23
3.12 Two Joined Monitors 24

1 Introduction

1.1 Motivation

Functional programming languages, such as Haskell, allow developers to write code in a declarative style, making it easier to transform a mathematical specification of a program into code [1]. Moreover, they are statically-typed and have powerful type inference techniques, ensuring correctness of their programs [2]. Functional programs aim to be free from side-effects, that is, they do not perform any operations that change the state of the program outside its local environment [3]. Examples of side-effects include I/O, exceptions and modifying a variable passed by reference. The more pragmatic functional languages allow some of these side-effects, for example, Haskell implements I/O operations using monads. In general, functional languages have little support on how to handle side-effects.

A widely used family of functional programming languages is ML, which gave birth to several other languages such as Standard ML [4], OCaml [5] and F# [6]. An updated version of OCaml was recently released with a new library called $\mathrm{Effects}$ that implements user-defined effects and effect handlers [7]. Effect handlers were initially introduced by Plotkin and Power [8–10] in terms of algebra theory and then were applied to computations. A user-defined effect is any operation which produces a side-effect. On the other hand, an effect handler is an entity designed to respond to these effects by defining the subsequent steps of execution once they are detected. Whenever a user-defined effect is encountered, the program is suspended while it searches for a corresponding handler. If one is found, the commands found in the handler are executed and control is reinstated to the main program. This concept is similar to exceptions and exception handlers, with the main difference being that once the effect handler's code has been fully executed, the program can resume computation from the point it was last suspended [11]. Therefore, effect handlers are a generalisation of exception handlers and provide a dynamic control-flow mechanism. Moreover, effect handlers can be nested within each other. That is, when an effect is raised, the program searches for a matching handler starting from the innermost and progressively moving outward. If the current handler does not describe the behaviour of our current effect, then, that handler simply passes on the effect to the next handler.

Effectful programs can be difficult to test or analyse statically since they depend on external factors, such as user input, the state of a file or network conditions, whose behaviour is determined at runtime. For this purpose, we focus on one type of verification technique referred to as runtime verification. It is a system consisting of software entities called monitors that run concurrently with a program and then report whether

the program violates a given property or not [12]. These monitors continuously inspect the runtime behavior of a system and trigger actions or alerts when violations of specified properties are detected. Monitors are frequently viewed as integral components of the Trusted Computing Base (TCB), a set of software and hardware components which are responsible for enforcing the system's overall security. Hence, it is imperative that monitors themselves are designed in the most systematic way possible to ensure their correctness and reliability.

1.2 Aim

The aim of this project is to express monitors using algebraic handlers to produce a more systematic way of creating them and to ensure their correctness. With the introduction of OCaml's Effects library, providing strong backing for working with effect handlers, our goal is to convert these abstract ideas into functional programs, effectively connecting theory with real-world application.

1.3 Objectives

In this project we set out to:

(**O1**) Describe effectful programs using algebraic effects and to describe monitors as algebraic handlers.

(**O2**) Translate the theory of algebraic effects and handlers into an OCaml program using the Effects library.

(**O3**) Create multiple examples of stand-alone monitors and of composed monitors.

1.4 Overview

The remaining chapters in this project are structured as follows. In Chapter 2, we introduce the mathematical structure behind algebraic theories, algebraic effects and handlers. This chapter can be skipped if the reader is familiar with the theory of algebra and algebraic effects. Then, we provide an application of algebraic effects to runtime verification by going through some stand-alone monitors in Chapter 3. These examples highlight both the theoretical and practical aspects of expressing monitors algebraically. In Chapter 4, we discuss how monitors may be layered for one particular program to test a program for several properties. Finally, Chapter 5 provides an analysis of this research, critique and limitations, along with possibilities for future work.

2 Algebraic Theories

In this chapter, we cover mathematical structures that are equipped with operations satisfying equational laws, called algebraic theories. We start by covering the fundamental mathematical concepts that describe these structures, then proceed by seeing how these are used in computer science to describe computational effects. Algebraic effects go hand in hand with their counterpart, algebraic handlers, used to modify the behaviour of a program's side effects. We remark that the definitions and results provided in this chapter are adapted from a note written by A. Bauer [13] describing the relationship between algebra theory and algebraic effects.

2.1 An Introduction to Algebraic Effects

Definition 2.1. *A **signature** $\Sigma = \{(op_i, ar_i)\}_i$ is a collection of operation symbols, denoted by op_i, and the number of arguments the operation expects as input, referred to as arities and are denoted by ar_i.*

The operation symbols can be anything as they are simply syntactic entities and do not possess any meaning. Since arities represent the number of arguments their corresponding operation takes, they must be non-negative integers. An operation symbol whose arity is zero is called **constant**, whilst those with arity one, two and three are referred to as **unary, binary** and **ternary** respectively. For example, if we consider integers, one such symbol could be Plus, whose arity is two and another symbol would be Zero with arity zero. Then, the signature would be given by $\Sigma_{\mathsf{Nat}} = \{(\mathrm{Plus}, 2), (\mathrm{Zero}, 0)\}$.

Operations can be combined together to form expressions. To do this, we need to come up with a way to represent unknown or changing values. A placeholder is a symbol that represents a particular value which can be referred to repeatedly in an expression.

Definition 2.2. *A **context** is a list of distinct variables $x_1, x_2, \ldots, x_k$.*

Using variables, we can build expressions that can only refer to the variables in the given context. Some variables may be left unused and there may be no variables in a context at all. When a context is empty, then the Σ-term is referred to as a **closed Σ-term**. We can inductively build expressions using these variables and operations from Σ as follows.

Definition 2.3. *A **Σ-term** t in context $x_1, \ldots, x_k$, denoted by $x_1, \ldots, x_k \mid t$, is built inductively using the rules below:*

1. *each variable x_i is a Σ-term in context $x_1, \ldots, x_k$,*

2. *if $t_1, \ldots, t_{ar_i}$ are Σ-terms in context $x_1, \ldots, x_k$ then $op_i(t_1, \ldots, t_{ar_i})$ is a Σ-term in context $x_1, \ldots, x_k$.*

Consider the set of integers again, with the binary operation Plus and constant operation Zero. Some examples of Σ-terms follow below.

$$\emptyset \mid \mathsf{Zero}, x \mid \mathsf{Plus}(x, \mathsf{Zero}), x, y \mid \mathsf{Plus}(x, y), x, y \mid \mathsf{Plus}(\mathsf{Plus}(x, y), \mathsf{Zero}), x \mid \mathsf{Plus}(\mathsf{Zero}, x)$$

A **closed term** is a Σ-term with an empty context, such as the first term in the Σ-term above.

We can also create equations for the constructed terms.

Definition 2.4. *A Σ-**equation** is a pair of Σ-terms ℓ and r in a context $x_1, \ldots, x_k$ and is denoted by $x_1, \ldots, x_k \mid \ell = r$. For brevity, we may write an equation as simply $\ell = r$.*

Some examples of expressions using Σ_{Nat} are $x \mid \mathsf{Plus}(\mathsf{Zero}, x) = x$, meaning that any term added to zero remains unchanged and $x, y \mid \mathsf{Plus}(x, y) = x, y \mid \mathsf{Plus}(y, x)$, describing commutativity of the plus operation.

We can collect signatures and equations into what we call an algebraic theory.

Definition 2.5. *For a signature Σ_T and a collection $\mathcal{E}_T$ of Σ_T-equations, an **algebraic theory** T is a pair (Σ_T, $\mathcal{E}_T$).*

Example 2.6. *In algebraic theory terms, the theory of Groups can be defined as follows. Firstly, the signature Σ_G is given by $\Sigma_G = \{$(u,0), (i, 1), (b, 2)$\}$. The operation u represent the unit value in a group, the operation i represents the inverse of an element and b is the binary operation on a group. The equations of the group, denoted by $\mathcal{E}_G$, are:*

$$\begin{aligned} g, h, k \mid b(b(g, h),\ k) &= b(g,\ b(h, k)), \\ g \mid b(u(),\ g) &= g, \\ g \mid b(g,\ u()) &= g, \\ g \mid b(i(g),\ g) &= u(), \\ g \mid b(g,\ i(g)) &= u(). \end{aligned} \tag{2.1}$$

In the example above, only three operation symbols are required since any group only makes use of the identity, the inverse and the binary operation. The notion of variables (or placeholders) allows us to construct terms and equations that refer to arbitrary terms from a group. In this example, the only closed term is u() as it does not make use of any variables. Hence its context can be left empty.

The equations given in Equation (2.1) are valid Σ_G-terms. For instance, the left-hand side of the first equation in (2.1) is a valid Σ_G-term because $g, h,$ and k are Σ_G-terms in the context g, h, k. Using part 2 of Definition 2.3, b(g, h) is another Σ_G-term since b is a

binary operation in Σ_{G}, therefore, the terms $\mathsf{b}(\mathsf{b}(g,h),\ k)$ and $\mathsf{b}(g,\ \mathsf{b}(h,k))$ are Σ_{G}-terms, making the first equation valid.

So far, we have only worked with symbols and variables, that are just syntactic entities. Once a signature and a set of equations have been defined for some theory T, a meaning is given to this theory by defining an interpretation for signatures.

Definition 2.7. *For a given signature Σ, an **interpretation I** of Σ is given by:*

1. *a set $\overline{I}$, called the **carrier set** and*
2. *for each operation symbol op_i, there is a map $[\![op_i]\!]_I$: $\overline{I} \times \cdots \times \overline{I} \to \overline{I}$, which is called an **operation**.*

We refer to the double bracket $[\![\ ]\!]_I$ as the **semantic bracket**, which typically maps syntactic entities to their corresponding mathematical meaning. For brevity, we denote the Cartesian product $\overline{I} \times \cdots \times \overline{I}$ by $\overline{I}^n$. Note that when $n = 0$, the only element contained in $\overline{I}^0$ is the empty tuple $()$ and we write $\overline{I}^0$ as 1.

Example 2.8. *Recall the theory of Group given in Example 2.6. An interpretation for Group is given by a carrier set $\overline{G}$ and the three maps:*

$$[\![u]\!]_G : 1 \to \overline{G},\ [\![m]\!]_G : \overline{G} \times \overline{G} \to \overline{G},\ [\![i]\!]_G : \overline{G} \to \overline{G}.$$

The maps given above interpret the operation symbols in Σ_G and $(\overline{G}, [\![u]\!]_G, [\![m]\!]_G, [\![i]\!]_G)$ forms a group, with the difference that the identity is seen as a map rather than an element of $\overline{G}$. For instance, if the group under consideration is $\mathbb{Z}$ whose binary operation is addition, it has carrier set $\overline{G} = \mathbb{Z}$ along with the following maps:

$$[\![u]\!]_G() = 0,\ [\![m]\!]_G(a,b) = (a+b),\ [\![i]\!]_G(a) = -a.$$

Since a theory is also constructed using terms and equations, we extend an interpretation I to Σ-terms. Recall that terms are constructed inductively, therefore, their interpretation exploits their structure by recursively defining the following semantic maps.

Definition 2.9. *For a given Σ-term t in context $x_1, \ldots, x_k \mid t$, its interpretation is a map $[\![x_1, \ldots, x_k \mid t]\!]_I : \overline{I}^k \to \overline{I}$, defined by the following rules:*

1. *the variable x_i is interpreted as the i^{th} projection, that is, $[\![x_1, \ldots, x_k \mid x_i]\!]_I = \pi_i : \overline{I}^k \to \overline{I}$,*
2. *an operation $op_i(t_1, \ldots, t_{ar_i})$ is interpreted as $[\![op_i]\!]_I([\![t_1]\!]_I, \ldots, [\![t_{ar_i}]\!]_I)$.*

Less formally, the first point means that given a k-tuple, the i^{th} projection returns the i^{th} component of the tuple. Moreover, if a term is an operation, then by Definition 2.3, its arguments are terms themselves, all in the same context as the operation. Therefore, we get the interpretation of each argument and form a tuple of size ar_i. These are then passed to the map $[\![\mathsf{op}_i]\!]$ to return a value in the carrier set $\overline{I}$.

Finally, we put together the syntax and semantics of a theory so that we can reason about its operations and achieve a description of their behaviour.

Definition 2.10. *A **model M** of an algebraic theory T, referred to as a T-**model**, is an interpretation of the signature Σ_T which satisfies all the equations $\mathcal{E}_T$.*

For the interpretation of the signature to satisfy every equation in $\mathcal{E}_\mathsf{T}$ having the form $x_1, \ldots, x_k \mid \ell = r$, the maps $[\![x_1, \ldots, x_k \mid \ell]\!]_M : \overline{M}^k \to \overline{M}$ and $[\![x_1, \ldots, x_k \mid r]\!]_M : \overline{M}^k \to \overline{M}$ are equal. That is, a k-tuple in M^k is mapped to the values v and v' in $\overline{M}$ by the first map and second map, respectively, and $v = v'$.

It is easy to check that the equations given in Example 2.6 hold using the interpretation given in Example 2.8. For instance, $g \mid \mathsf{m}(\mathsf{u}(), g)$ is the map $[\![g \mid \mathsf{m}(\mathsf{u}(), g)]\!] : \overline{G} \times \overline{G} \to \overline{G}$ such that $(a, b) \mapsto (0 + g) = g$. Indeed, $g \mid g$ is the map $[\![g \mid g]\!] : G \to G$ mapping g to itself, as required. Similarly, $[\![g \mid \mathsf{m}(g, \mathsf{u}())]\!]$ returns the value g, satisfying equation three in Equation (2.1).

We can also define mappings between two models of a theory T using homomorphisms, mappings that commute with the operations between the two models.

Definition 2.11. *Let L and M be models of a theory T. A **T-homomorphism** $\phi : \overline{L} \to \overline{M}$, from the carrier set of L into the carrier set of M, commutes with the operations op_i. That is, for every operation symbol op_i of T we have*

$$\phi \circ [\![op_i]\!]_L = [\![op_i]\!]_M \circ (\phi, \ldots, \phi),$$

where the size of the tuple $(\phi, \ldots, \phi)$ is ar_i.

Using Example 2.6, a homomorphism between two groups G and H is a map $\phi : \overline{G} \to \overline{H}$ such that, for all $a, b \in \overline{G}$,

1. $\phi([\![\mathsf{u}()]\!]_G) = [\![\mathsf{u}()]\!]_H$,
2. $\phi([\![\mathsf{m}]\!]_G(a, b)) = [\![\mathsf{m}]\!]_H(\phi(a), \phi(b))$,
3. $\phi([\![\mathsf{i}]\!]_G(a)) = [\![\mathsf{i}]\!]_H(\phi(a))$

This means that the identity element of G is mapped to the identity element of H, the inverse of an element a in G is mapped the inverse of the image of a in H, and the binary operation on two elements in G is again mapped to the binary operation in H applied to the image of these two elements.

When formulating the list of equations for a theory, we must proceed with caution as too many equations or too little may result in contradictions. For one theory T, there can be many models that satisfy $\mathcal{E}_{\mathsf{T}}$, but one that makes use of the least number of elements and equations, called the free model, is preferred over the other models.

Definition 2.12. *For a theory T and a set X, the **free T-model** generated by X, denoted by $\mathit{Free}_T(X)$ is a model M and a map $\eta : X \to \overline{M}$ such that, for every T-model L and every map $f : X \to \overline{L}$, there is a unique T-homomorphism $g : M \to L$ for which Figure 2.1 commutes.*

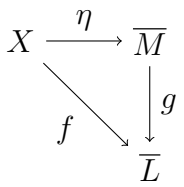


Figure 2.1 Free Model

For a diagram to commute, all the directed paths in the diagram with the same start and end points yield the same end result. For instance, in Figure 2.1 the edges symbolise homomorphisms between the sets in the diagram. Starting at X and ending at $\overline{L}$ by following the two directed paths on the diagram should yield the same mapping, that is, each element of X is mapped to the same element by f and by $g \circ \eta$ in $\overline{L}$.

Every algebraic theory has a free model generated by any set and there is a construction that yields such a free model, given by Bauer in [13]. Here we provide the main idea behind the construction. Whilst a Σ-term can be defined recursively using Definition 2.3, we can look at terms as trees. An operation $\mathsf{op}_i(t_1, \ldots, t_{\mathsf{ar}_i})$ is represented as a tree whose root is labelled op_i and has ar_i many sub-trees corresponding to its arguments $t_1, \ldots, t_{\mathsf{ar}_i}$. The leaves of the tree would be the variables in the context $x_1, x_2, \ldots, x_k$, labelled by return x_i, since these cannot be reduced further. These trees are grouped together into a set denoted by $\mathsf{Tree}_\Sigma(X)$, where $X = \{x_1, x_2, \ldots, x_k\}$. It is important to note that our trees are **well-founded**, that is, they do not contain infinite paths. Consequently, the free model for a theory T may be expressed in terms of $\mathsf{Tree}_\Sigma(X)$ as follows. The carrier set of the free model is the set of all the normal forms that the trees may have. These forms are obtained by applying the equations to the trees until no more equations are applicable. Note that the leaves cannot be reduced further, hence, they are always part of the free model.

The current algebra definitions describe some structures, but many other algebraic theories cannot be captured by our current framework. The reason is that either some operations need to consider an arbitrary number of arguments or there are some operations that require an extra element which is not an operation nor an existing variable.

Recall that earlier we shortened the Cartesian product $\overline{I} \times \cdots \times \overline{I}$ to $\overline{I}^n$. In fact, $\overline{I}^n$ is isomorphic to $\overline{I}^{[n]}$, where $[n] = \{0, 1, \ldots, n-1\}$. To generalise the number of arguments

an operation acts on, we replace $[n]$ with an arbitrary set A. Hence, for an operation to have A many arguments, we first represent these arguments as a map $\kappa : A \to \overline{I}$ then apply it to our operation, $\mathsf{op}_i(\kappa)$.

In addition, there will be situations where some operations take in the function κ and a parameter from a set different to $\overline{I}$. In light of this, we shall expand the previous definition for a signature to allow these parameters and general arities.

Definition 2.13. *A **signature** Σ is a collection of operation symbols op_i, together with parameter sets P_i and arities A_i. This is denoted by $\Sigma = \{\mathsf{op}_i : P_i \rightsquigarrow A_i\}$.*

As before, the op_i are symbols whilst P_i and A_i are arbitrary sets. The notation $P_i \rightsquigarrow A_i$ is not a mapping from the set of parameters to arity set, it is simply part of the notation. We denote an operation with parameter p and A many arguments by $\mathsf{op}_i(p, \kappa)$, where $\kappa : A \to \overline{I}$.

We mainly use the definition of a signature given in Definition 2.13 as it models more closely computational effects. The rest of the definitions remain mostly unchanged, with the main difference being interchanging $\overline{I} \times \cdots \times \overline{I}$ to a function κ as described above. Refer to Appendix A for a detailed explanation on how the definitions are altered.

2.2 Computational Effects as Algebraic Effects

A **pure** computation is one which does not throw any effects whilst an **effectful** computation throws one or more effects. The effects thrown in a computation are referred to as **computational effects**. These are operations that interact with a program's environment or state, for instance, I/O, state mutation and exceptions are all types of computational effects. Plotkin and Power in [10, 14] suggested that several computational effects can be described using algebra theory. That is, the operations of a program that interact with their environment correspond to operations in an algebraic theory.

A pure computation is one that does not perform any computational effects, that is it returns a value v. In algebra theory, we represent this computation by $\mathsf{return}\ v$. Such a value could be an integer or a boolean, an entity that does not require further computation. Effectful computations are represented as a pair $\mathsf{op}(p, \kappa)$ where p is a parameter passed to the computational effect op and κ captures the rest of the computation. The computation is suspended when an effect is thrown, however, once the effect is carried out, the rest of the computation is resumed through κ. For instance, the operation could be read, with parameter p being a memory location to be read and the continuation being resumed once the contents of the memory location are returned.

When applying the theory of algebraic effects to describe computational effects, the arity sets for the operations correspond to the set of values V the program employs. The map $\kappa : A_i \to \overline{\mathsf{Free}_{\mathsf{T}}(X)}$ now corresponds to the suspended computation κ, which can be

invoked if it is given an argument from the arity set A_i. After establishing our signature and equations, the computations that can be performed arise from the elements in the free model $\mathsf{Free}_{\mathsf{T}}(V)$.

The first example we shall see here models I/O in a program. The signature for this theory is given by $\Sigma_{\mathsf{IO}} = \{\mathsf{print} : \mathsf{String} \rightsquigarrow 1, \mathsf{read} : 1 \rightsquigarrow \mathsf{String}\}$. This theory has no equations, hence, $\mathcal{E}_{\mathsf{IO}} = \emptyset$. The operation print takes a string as a parameter, however, its continuation acts on the unit value only. On the other hand, the operation read acts only on the unit value and returns the string read. An example of a computation is $\mathsf{print}(\text{“Hello World!”}, \lambda\_.\mathsf{return}())$. Here, we supply print with a string, after the operation is performed it returns the unit value, and the continuation is simply returning the unit value. The free model for this theory is the set $\mathsf{Tree}_{\Sigma_{\mathsf{IO}}}(V)$, where $V = \mathsf{String} \cup \{()\}$ since $\mathcal{E}_{\mathsf{IO}} = \emptyset$. Some examples of trees are

$$\mathsf{return}\ v \tag{2.2}$$

$$\mathsf{print}(s, \lambda\_.\mathsf{return}()) \tag{2.3}$$

$$\mathsf{read}((), \lambda s.\mathsf{return}\ s) \tag{2.4}$$

$$\mathsf{read}((), \lambda s.\mathsf{print}(s, \lambda\_.\mathsf{return}())) \tag{2.5}$$

In Equation (2.2), the tree $\mathsf{return}\ v$ is a description of a program which produces no side effects, therefore, it is a pure computation which returns some value v. On the other hand, Equation (2.3) is an abstract description of an effectful program which prints a string s and then terminates.

Another well-known event is exceptions. The theory of exceptions is given by the signature $\Sigma_{\mathsf{Exn}} = \{\mathsf{abort} : 1 \rightsquigarrow \emptyset\}$ and it has no equations. The operation abort is an operation which does not have a continuation since its arity set is $\emptyset$.

Let's consider another example by describing the state operations in algebraic terms. Here we assume that we have a single unnamed memory location. The signature is given by $\Sigma_{\mathsf{State}} = \{\mathsf{put} : S \rightsquigarrow 1, \mathsf{get} : 1 \rightsquigarrow S\}$, where S is the set of states. For $n, m \in S$ and continuations κ, the equations $\mathcal{E}_{\mathsf{State}}$ are given by,

$$\mathsf{put}(n, \lambda\_.\mathsf{put}(m, \kappa)) = \mathsf{put}(m, \kappa) \tag{2.6}$$

$$\mathsf{put}(n, \lambda\_.\mathsf{get}((), \kappa)) = \mathsf{put}(n, \lambda\_.\kappa\ n) \tag{2.7}$$

$$\mathsf{get}((), \lambda n.\mathsf{put}(n, \kappa)) = \kappa() \tag{2.8}$$

$$\mathsf{get}((), \lambda n.\mathsf{get}((), \lambda m.\kappa\ n\ m) = \mathsf{get}((), \lambda n.\kappa\ n\ n) \tag{2.9}$$

An example of a computation we can write using Σ_{State} is $\mathsf{get}((), \lambda n.\mathsf{put}(n+1, \lambda\_.\mathsf{return}\ n))$. This describes a computation that first reads the current state, then puts back the state incremented by one and return the original value. This is precisely the behaviour of the

operation $i++$ found in many programming languages.

Some elements in the free model include the tree return v, for any $v \in V$. Some other examples of trees in the free model are put$(n, \lambda_.\text{return}())$ and get$((), \lambda s.\text{return } s)$.

A tree such as put$(n, \lambda_.\text{put}(m, \lambda_.\text{return}()))$ describes a computation which stores the value n followed by m in the unnamed memory location. Using Equation (2.2), this can be reduced to the term put$(m, \lambda_.\text{return}())$. They are both valid computations, however, the final state of these two computations is the same. In fact, since $\mathcal{E}_{\text{State}} \neq \emptyset$, several trees may be reduced to simpler forms using the equations established in Equation (2.6).

2.3 Algebraic Effect Handlers

Effect handlers are a programming construct used to manage computational effects in a structured and modular way. Effect handlers capture the effects that occur during the execution of a program and provide specific mechanisms to handle these effects, making the effects more manageable. Expressing effect handlers using algebraic theory provides a formal framework for describing and reasoning about effects and their handlers. This rigor ensures that the definitions and properties of effects are precise, reducing ambiguities and potential errors in their implementation. When an effect is raised, the program is suspended and the handler acting on this effect defines the next steps of the computation. From a programming point of view, handlers act on return statements or operations of a theory T and re-define their behaviour in terms of operations in the theory T'. The general idea is that handlers can entirely change the behaviour of an operation and may even transform an effectful computation into a pure one and vice versa.

To express handlers in algebraic theory, the first step is to define a mapping from one free model to the other since computations form part of the free model. We could go further and claim that this mapping is a homomorphism, but we also choose which theory's free model is preserved. For this purpose, we define a homomorphism on the free model for T as defined below.

Definition 2.14. *A **handler** $H : \overline{\text{Free}_T(X)} \to \overline{\text{Free}_{T'}(X')}$ is a T-homomorphism from computations $\text{Free}_T(X)$ to computations $\text{Free}_{T'}(X')$ is given by the following:*

1. *a map $f : X \to \overline{\text{Free}_{T'}(X')}$,*
2. *for every operation $\text{op}_i : P_i \rightsquigarrow A_i$ in Σ_T, there is a map*

$$h_i : P_i \times \overline{\text{Free}_{T'}(X')}^{A_i} \to \overline{\text{Free}_{T'}(X')}$$

   such that

3. *the maps h_i form a T-model on $\overline{\text{Free}_{T'}(X')}$ by validating the equations $\mathcal{E}_T$.*

The first map relates values from the set X for T to trees in the free model for T'. Then, for each operation in T, we define its behaviour in the free model for T' such that the equations $\mathcal{E}_{\mathsf{T}}$ still hold. That is, the behaviour exhibited by the operations in T remains unchanged in T'. A handler $H : \overline{\mathsf{Free}_{\mathsf{T}}(X)} \to \overline{\mathsf{Free}_{\mathsf{T}'}(X')}$ may be denoted by $\mathsf{Free}_{\mathsf{T}}(X) \Rightarrow \mathsf{Free}_{\mathsf{T}'}(X')$. Alternatively, a handler can be written as

$$\mathsf{handler}\{\mathsf{return}\ v \mapsto f(v); \mathsf{op}_i(p, \kappa) \mapsto h_i(p, \kappa)_{\mathsf{op}_i \in \Sigma_{\mathsf{T}}}\}.$$

If a computation is described as a return statement, then we map the returned value to some other expression in the free model for T'. If the computation encountered an operation, then, we execute the map h_i which describes the next computation after op_i.

A simple example of a handler considers the theory of exceptions. Let $H : \overline{\mathsf{Free}_{\mathsf{Exn}}(X)} \to \overline{\mathsf{Free}_{\mathsf{T}'}(X)}$ be defined by

$$\mathsf{handler}\{\mathsf{return}\ v \mapsto f(v); \mathsf{abort}((), \kappa) \mapsto c\}$$

where $f : X \to \overline{\mathsf{Free}_{\mathsf{T}'}(X')}$ and $c \in \overline{\mathsf{Free}_{\mathsf{T}'}(X')}$. Note that c does not depend on the unit value or on $\kappa : \emptyset \to \overline{\mathsf{Free}_{\mathsf{T}'}(X')}$ since they do not provide any useful information. The handler is immediately well-defined since the theory of exceptions has no equations, so point (3) in Definition 2.14 automatically holds. The handler maps a return statement to some tree in the free model for T', for instance, to the tree $\mathsf{return}\ v$ which means that return statements are unchanged. Then, an abort is mapped to a computation in the free model for T'.

More examples of handlers will be discussed in the following chapter.

3 Stand-alone Monitors

This chapter investigates how monitors can be expressed using effect handlers. We use the theory developed in the previous chapter to formally define effectful programs and their respective monitors to ensure correctness of our monitors, addressing objective (**O1**). The concepts developed in the previous chapter are employed to compare and contrast the theoretical side of monitors with their implementation counterpart to achieve objective (**O2**).

3.1 The General Program

Common side-effects found in any program relate to accessing or modifying values stored in memory. To formalise this problem, we define two operations, put and get, that store a value in a particular variable and retrieve the value stored in a particular variable, respectively. In algebraic theory terms, we shall refer to the theory of our program as State, whose signature is given by Equation (3.1).

$$\Sigma_{\mathsf{State}} = \{\mathsf{put} : \mathsf{String} \times \mathsf{Int} \rightsquigarrow 1, \mathsf{get} : \mathsf{String} \rightsquigarrow \mathsf{Int}\}. \tag{3.1}$$

This means that the operation put acts on a pair (s, n), where s is a string that identifies the memory location and n is an integer that the location is updated with, and returns the unit value. Moreover, the operation get acts on a string and returns an integer held at the respective location. The theory of State is also equipped with a few equations $\mathcal{E}_{\mathsf{State}}$, which are given in Equation (3.2). They describe the relationship between the operations of our theory and the behaviour of composing two operations with each other. First we look at what happens on successive put and get operations.

For all $s \in \mathsf{String}$, $n, m \in \mathsf{Int}$ and all continuations κ, we have that:

$$\mathsf{put}((s, n), \lambda\_.\mathsf{put}((s, m), \kappa)) = \mathsf{put}((s, m), \kappa) \tag{3.2}$$

$$\mathsf{put}((s, n), \lambda\_.\mathsf{get}(s, \kappa)) = \mathsf{put}((s, n), \lambda\_.\kappa(s, n)) \tag{3.3}$$

$$\mathsf{get}(s, \lambda n.\mathsf{put}((s, n), \kappa)) = \kappa() \tag{3.4}$$

$$\mathsf{get}(s, \lambda n.\mathsf{get}(s, \lambda m.\kappa\ n\ m)) = \mathsf{get}(s, \lambda n.\kappa\ n\ n) \tag{3.5}$$

Equation (3.2) describes the case when the program has two successive puts. The first put stores the integer n is memory location identified by s, it returns the unit, then puts another integer m in memory location identified by s with κ representing the rest of the program. This is equivalent to storing the final value m in the memory location for s and then continuing the rest of the program through κ. On the other hand, Equation

(3.4) describes how a get followed by a put cancels out. If we read the contents of a memory location labelled s, then we store the read state again in s, the state has not have changed, therefore, it is equivalent to resuming the rest of the program.

Another set of equations describes the behaviour of our operations for distinct locations, referenced through different variables names which are unailiased, as follows below.

For $s, t \in \mathsf{String}$ such that $s \neq t$:

$$\mathsf{put}((s,n), \lambda\_.\mathsf{put}((t,m),\kappa)) = \mathsf{put}((t,m), \lambda\_.\mathsf{put}((s,n),\kappa)) \tag{3.6}$$

$$\mathsf{put}((s,n), \lambda\_.\mathsf{get}(t,\kappa)) = \mathsf{get}(t, \lambda m.\mathsf{put}((s,n), \lambda\_.\kappa\ n)) \tag{3.7}$$

$$\mathsf{get}(s, \lambda n.\mathsf{get}(t, \lambda m.\kappa\ n\ m)) = \mathsf{get}(t, \lambda m.\kappa\ m\ n) \tag{3.8}$$

Another theory we consider is the theory of Alert which makes use of two operations, the print and exception. Their signature is given by Equation (3.9).

$$\begin{aligned}\Sigma_{\mathsf{Alert}} = \{&\mathsf{print} : \mathsf{String} \rightsquigarrow 1, \mathsf{raise} : 1 \rightsquigarrow \emptyset, \mathsf{find} : \mathsf{List\ of\ (String} \times \mathsf{Int)} \times \mathsf{String} \rightsquigarrow \mathsf{Int},\\ &\mathsf{update} : \mathsf{List\ of\ (String} \times \mathsf{Int)} \times \mathsf{String} \times \mathsf{Int} \rightsquigarrow \mathsf{List\ of\ (String} \times \mathsf{Int)}\}.\end{aligned} \tag{3.9}$$

The operation print acts on a string and returns the unit whilst the operation raise acts on the unit parameter and does not return any values. The operations find and update operate on lists of pairs (s,n), where s is a string and n is an integer. The operation $\mathsf{find}(\ell, s)$ searches for the entry in the list ℓ whose first element is s and returns the second element in that pair whilst $\mathsf{update}(\ell, s, n)$ changes the value of s in ℓ to n. There are some equations for this theory regarding consecutive finds and updates for a list using find and update. These equations in $\mathcal{E}_{\mathsf{Alert}}$ are similar to those found in Equation (3.2) and Equation (3.6) with an added parameter of type List in all of the equations.

We can combine the operations found in Σ_{State} and Σ_{Alert} to create another theory Mon, where $\Sigma_{\mathsf{Mon}} = \Sigma_{\mathsf{State}} \cup \Sigma_{\mathsf{Alert}}$, with the addition of four new ones found below.

$$\mathsf{put}((s,n), \lambda\_.\mathsf{raise}(())) = \mathsf{raise}(()) \tag{3.10}$$

$$\mathsf{get}(s, \lambda n.\mathsf{raise}(())) = \mathsf{raise}(()) \tag{3.11}$$

$$\mathsf{find}((\ell, s), \lambda m.\mathsf{raise}(())) = \mathsf{raise}(()) \tag{3.12}$$

$$\mathsf{update}((\ell, s, n), \lambda\_.\mathsf{raise}(())) = \mathsf{raise}(()) \tag{3.13}$$

The main idea here is that the behaviour of an operation a put or a get is followed by an exception is the same as when an exception is raised immediately since the program is terminated immediately in both cases.

Below is an OCaml implementation of the operations put and get with three memory

locations mem1, mem2, mem3 and a sample program comp.

```
let mem1 = ref 0
let mem2 = ref 0
let mem3 = ref 0

let put var n = update_entry names_hash var (ref n)

let get var = let x = get_entry (names_hash) (var) in !x

let x = (ref 0) in
let comp () = put("mem1", 5); x:= get ("mem1"); put("mem2", 6);
   put("mem3", -1); put("mem2", 100) in comp ();;
```

Listing 3.1 Defining the effects put and get in OCaml

A Hashtable names_hash has been added to store pairs (s, n) where s is a string and n is an integer reference. A put takes a string var and an integer n and updates names_hash accordingly. Similarly, a get retrieves the record in names_hash and returns the value in the second element of the pair. The sample computation comp is updating the three memory locations, with the final values for mem1, mem2 and mem3 being 5, 100 and -1 respectively.

3.2 Instrumentation

The main goal of runtime verification is for the monitor to test whether the program under consideration satisfies some properties. However, the monitor needs to recognise which side-effects to analyse and which can be ignored. To bridge the gap between the program and the monitor, the **instrumentation** phase generates a series of events from the program so that the monitor is aware of the events it needs to analyse.

From the programming perspective, the process of instrumentation requires two major additions provided to us by OCaml's Effects library. First we need to define the side-effects we wish the monitor the act on, which in our case are put and get.

```
type _ Effect.t += Put : (string * int) -> unit Effect.t
type _ Effect.t += Get : string -> int Effect.t
```

Listing 3.2 Defining the effects put and get in OCaml

The Effects library contains a pre-defined extensible variant type, called Effect.t as seen in Listing 3.2. This means that we can add more constructors, not necessarily all of the same type, to others already present in Effect.t. Here we have added a constructor Put that takes a pair (s, n), where s is a string and n is an integer and when the effect is performed, it returns the unit value. We also define a Get constructor which takes a value of type string and returns an integer. The type of any user-defined effect is the

return type of the constructor added to Effect.t. For instance, Put is said to be an effect of type unit Effect.t and Get is said to be an effect of type int Effect.t. For our purposes, these effects are defined in an OCaml file Utils.ml so that it can be shared amongst other files as needed.

Once the effects for a program have been established, we take the original program and instrument it using these effects as follows. Consider the program established in Listing 3.1 whose effectful operations are put and get. To each of these operations, we append another operation perform as seen in Listing 3.3. This keyword takes a user-defined effect a Effect.t which has already been defined in Effect.t. When the operation put or get is called within main and the perform keyword is reached, the program is suspended while it searches for handler that defines the behaviour for these effects. If one is found, the computations found within that handler are executed and the program is resumed from the point it was last suspended. If no suitable handler is found, an Unhandled exception is thrown.

```
1 let put var n = update_entry names_hash var (ref n); perform(Put (var, n))
2
3 let get var = let x = get_entry names_hash var in !x;
4                perform(Get var)
```

Listing 3.3 Instrumenting the program

The perform keyword has two purposes. Firstly that it performs the side-effect and invokes a search for a corresponding handler. Since our monitors are defined as handlers, perform raises an event that the monitor can act on. Conveniently, we define our monitors as effect handlers so that with each effectful operation, the necessary checks are carried out to verify whether the program has violated our properties or not.

In terms of algebraic effects and handlers, there is no need for an instrumentation process. When a computation c is handled by a handler H, the behaviour of the operations in c is determined by H without the need for special keywords or operations. Thus, the instrumentation process is only necessary from the programming perspective of the monitor.

3.3 A Stateless Monitor

The first monitor we consider is an instance of the class of **stateless** monitors. It asserts that a -1 is never stored in memory. First we describe the monitor using an algebraic handler, then proceed to show its implementation in OCaml.

For our purposes, all our handlers are defined on the trees from the free model $\mathsf{Free}_{\mathsf{State}}(Int)$. The handler is given in Equation (4.1) below. The handler only analyses the input for put since it is the only operation that modifies the value in memory.

Whilst get and return are also in the free model, the monitor does not need to analyse them further.

$$
\begin{aligned}
H = \mathsf{handler}\{&\mathsf{return}\ v \mapsto f(v),\\
&\mathsf{put}((loc, n), \kappa) \mapsto\\
&\ \lambda\_.\ \mathsf{if}\ n = -1,\ \mathsf{then}\\
&\ \mathsf{print}(\text{“Put with value -1 encountered.”}, \lambda\_.\mathsf{raise}(())), \mathsf{else}\ \kappa()\ ),\\
&\mathsf{get}(loc, \kappa) \mapsto \lambda n.\kappa n\}.
\end{aligned}
\tag{3.14}
$$

where $f : Int \to \overline{\mathsf{Free}_{\mathsf{Mon}}(Int)}$ is defined such that $f(v) = \mathsf{return}\ v$, that is, the first map in Equation (4.1) is the identity map. Similarly, the map for the operation get is also the identity map. The only operation that alters the value of a location is put, hence, upon a put, we still store that value in memory and then check whether that value is a -1. If that is the case, then we employ the operations from Alert to print a message and then terminate the program. Otherwise, the rest of the computation is executed through $\kappa()$. The unit is passed as an argument to the continuation since the continuation for put expects an argument from the set $1 = \{()\}$ to resume.

In OCaml, the algebraic effect handler is translated seamlessly into code as the Effects library is equipped with a handler type type (a,b) handler that acts on values of type a and returns values of type b, as seen in Listing 3.4. Similarly to handler H, we have the case retc for when the computation returns a value and a case effc for when the computation is one of the operations in our signature. For each user-defined effect, there is a case inside the handler that defines its behaviour.

```
type (a,b) handler = { retc: a -> b;
                       effc: c.c eff -> (c,b) continuation -> b; }
```

Listing 3.4 Handler Signature

When an effect is performed, case effc expects two arguments; an effect of a general type c eff and a continuation of type (c,b) continuation. The latter is a *delimited continuation*, that is, a function that takes a value of type c and returns a value of some type b from which the rest of the program may be resumed. The continuation may only be resumed if it is given a value of the same type as the effect which has been performed.

In OCaml, the handler H given in Equation (4.1) is programmed as the handler found in Listing 3.5.

```
let monitor1 () = match_with (main) ()
{
  effc = (fun (type b) (eff: b Effect.t) ->
    match eff with
    | Put (loc,s) ->
      Some (fun (k: (b,_) continuation) ->
```

```
7       if s = (-1) then (printf "Put with value -1 encountered.\n";
          discontinue k (Invalid_value s))
8       else continue k ())
9     | Get x ->
10      Some (fun (k: (b,_) continuation) -> let x = get_entry names_hash x
                                             in continue k (!x))
11    | _ -> None
12  );
13  exnc = raise;
14  retc = fun x-> x
15 }
```

Listing 3.5 Stateless Monitor

Recall that in algebra theory, for a computation c to be transformed into another computation via a handler H, we write with H handle c. The equivalent of this in OCaml is $\mathrm{match\_with}\ (\mathrm{main}\ ())\ h$, where the computation $\mathrm{main}\ ()$ is executed under an effect handler h. The construct $\mathrm{match\_with}$ on line 1 has type $(\mathrm{unit}\ \text{->}\ \mathrm{a})\ \text{->}\ (\mathrm{a,b})\ \mathrm{handler}\ \text{->}\ \mathrm{b}$ whose first argument is the computation to be handled by the second argument, the handler. It is important to note that the computation must be a function that takes in the unit due to the type of $\mathrm{match\_with}$. When the program is run against the handler, the entire computation should return a value of type b. The computation can result in three cases, either it returns a value, it raises an exception or performs an effect, which is accounted for by the handler through the retc, exnc and effc fields respectively. In this program, the outcomes of each case are as follows.

1. Case retc on line 14: if the computation returns with a value x, then the handler returns x.

2. Case exnc on line 13: if the computation raises an exception, the handler raises the same exception.

3. Case effc on line 3: here we return a function that defines a locally abstract type, $\mathrm{type\ b}$. This function takes as a parameter the variable eff of type $\mathrm{b\ Effect.t}$. Then if eff is the effect:

   (a) Put with the input argument $(\mathrm{loc,s})$: we return another function whose argument is k of type $(\mathrm{b,\ \_})\ \mathrm{continuation}$ representing the continuation of the program. Within this function, we check whether s is -1 or not. If so, then an error message is printed on the console and on line 7 the continuation is resumed by raising an exception $\mathrm{Invalid\_value}$. Otherwise, on line 8 we resume the continuation by giving it the unit value since Put has type $\mathrm{Unit\ Effect.t}$ (refer to Listing 3.2).

(b) Get with the input argument x and with continuation k: we resume the continuation by giving it the integer value of x since Get has type int Effect.t (refer to Listing 3.2).

There are some points worth mentioning that arise from the cases above. Firstly, type b Effect.t is an abstract type that allows the function eff to be polymorphic over any effect of type b. In our case, all the operations have been instrumented even though the only effect of interest is Put, hence, the Get effect also needs to be handled. The dynamic semantics for this library, found in [11], forces the handler to always have the case retc, whilst the cases for excn and effc are optional. Note that on line 11 the effc case has a catch-all case _ -> None, which forwards an unmatched effect to the outer handler. Since this is the only handler running with our program, if some other effect is raised, it is left unhandled and an Unhandled exception is thrown [11].

The continue and discontinue keywords, whose signature is given in Listing 3.6, resume the program from the point it was last suspended. The main difference between the two is that continue resumes the program given an argument with the correct type whilst discontinue resumes the program with an exception. This is useful when we want the program to terminate upon a particular exception. In our case, if a -1 is stored in memory, we raise an exception to immediately stop the program since the property the monitor was checking has failed.

```
1 val continue: (b,a) continuation -> b -> a
2 val discontinue: (b,a) continuation -> exn -> a
```

Listing 3.6 Continuation Signature

3.4 Stateful Monitors

In the previous section we have seen a stateless monitor, one that analyses each event in isolation. That is, the monitor does not keep an internal state that depends on previous events. The monitors in this section are examples of the **stateful** class of monitors that only analyse one type of event.

3.4.1 Aggregate Value Monitor

The purpose of the aggregate value monitor is to verify that the total sum of the integers stored across all variables is less than some arbitrary integer, for example, 500. For this purpose, the monitor makes use of a variable, sum, that stores the running total of each integer passed to Put.

The procedure to define this handler is similar to the handler define in the previous section, except that here we introduce a parameter-passing handler to keep a record of

the total sum.

$$
\begin{aligned}
H = \mathsf{handler}\{&\mathsf{return}\ v \mapsto \lambda\_.\mathsf{return}\ v,\\
&\mathsf{put}((loc, n), \kappa) \mapsto \lambda sum.\ \mathsf{if}\ sum < 500,\\
&\quad \mathsf{then}\ (k())(sum + n)\\
&\quad \mathsf{else}\ \mathsf{print}(\text{“Total sum of puts exceeded 500.”}, \lambda\_.\mathsf{raise}(())),\\
&\mathsf{get}(loc, \kappa) \mapsto \lambda sum.(\lambda s.\kappa\ s)(sum)\}.
\end{aligned}
\tag{3.15}
$$

Since the monitor has to keep a running total of the integers stored in memory, we use a technique called parameter-passing, where the handled computation is transformed into a function that passes around a parameter. In our case, this parameter is the aggregate sum. When a put is encountered, the handler returns a function with a parameter sum which then checks whether it has exceeded 500. If not, the continuation is resumed by passing it the unit value. However, the continuation is again handled by the handler, therefore, we apply $\kappa\ ()$ to the new value of sum. Otherwise, we print an error message and raise an exception to terminate the program. Meanwhile, if a get is found, we return a function with the parameter sum and apply the continuation $\kappa\ s$ to sum since this operation does not alter states. For the case when we reach a return statement, we still return a function.

Below is part of the implementation of the aggregate value monitor. The effect handler describing this monitor only analyses the events performed by Put, hence, the effc fields not concerning Put, the retc and exnc fields remain the same as seen in Listing 3.5.

```
1 let sum : int ref = ref 0
2
3 | Put (loc,s) -> Some (fun (k: (b,_) continuation) ->
4                     if !(sum) < 500 then (sum := !(sum) + s; continue k ())
5                     else printf "Total sum of puts exceeded 500.\n";
6                     discontinue k (Exceeded_value 500))
```

Listing 3.7 Aggregate Value Monitor

Here we note that sum in line 1 is an integer reference since references are mutable, thus, their value can be updated. The monitor updates its running total with each Put, given that the current value is less than 500. Otherwise, an error message is printed to the screen and the continuation is resumed with the exception Exceeded_value. As a first attempt, the sum variable is defined globally, however, this can be implemented in a continuation passing style passed as similarly to the theoretical handler in Equation (3.15). Due to time constraints, the latter could not be implemented. We slightly enhance this monitor by keeping an aggregate sum of each memory location. The handler for this monitor is given in Equation (3.16).

$$
\begin{aligned}
H = \mathsf{handler}\{&\mathsf{return}\ v \mapsto \lambda\_.\mathsf{return}\ v,\\
&\mathsf{put}((loc, n), \kappa) \mapsto \lambda sum\_lst.\mathsf{find}((sum\_lst, loc),\\
&\quad \lambda m.\ \mathsf{if}\ m < 500,\ \mathsf{then}\ (k())(\mathsf{update}(sum\_lst, loc, m + n))\\
&\quad \mathsf{else}\ \mathsf{print}(\text{“Total sum of puts exceeded 500.”}, \lambda\_.\mathsf{raise}(())))),\\
&\mathsf{get}(loc, \kappa) \mapsto \lambda sum\_lst.(\lambda s.\kappa\ s)(sum\_lst)\}.
\end{aligned}
\tag{3.16}
$$

Similarly to Equation (3.15), we pass around a list sum_lst of pairs (ℓ, m) where ℓ is a memory location and m stores the aggregate sum of all the puts made to ℓ. When a get is found, we handle it by returning a function whose parameter is sum_lst and then we apply the continuation to sum_lst. If a put is found, we return a function with sum_lst as a parameter then get the current aggregate sum for loc using an operation find. This operation takes a list and a memory location as input and returns the aggregate sum of the memory location. If the value returned is less than 500, then we resume the computation and apply it to the new list of pairs with the value for loc being updated to the newer value. This is done through another operation update which takes three parameters, a list of pairs sum_lst, a memory location loc and an integer. It finds the entry for loc in sum_lst, updates the second entry in the pair with the integer argument and returns the updated list.

In OCaml, this handler is given by the code snippet in Listing 3.8.

```
let sums_hash : (string , int ref) Hashtbl.t = Hashtbl.create 10

| Put (loc,s) -> Some (fun (k: (b,_) continuation) ->
                    let x = get_entry sums_hash loc in
                    let y = (!x) + s in
                    update_entry sums_hash loc (ref y);
                    if y < 500 then continue k ()
                    else
                    printf "Total sum of puts in one memory location
   exceeded 500.\n"; discontinue k (Exceeded_value 500))
```

Listing 3.8 Enhanced Aggregate Value Monitor

First we add a Hashtable sums_hash that relates the names of the variables to their value in memory. When a Put is found, the monitor retrieves the current total for the variable loc, adds this value to the newly stored integer and updates the value in sums_hash. Then, the monitor compares the newly stored value with 500, and similarly to Listing 3.7 an exception is thrown when the value is greater than 500. Similarly to the previous aggregate value monitor, a list of pairs relating the names of memory locations with their sums can be passed as a parameter to the continuation to adhere to functional language principles, however, this was not possible due to time limitations.

3.4.2 Alternating Parity Monitor

Another stateful monitor is one that checks whether the integers stored in the variables are alternating between odd and even. We can express the handler for this monitor as follows.

$$
\begin{aligned}
H = \text{handler}\{&\text{return } v \mapsto \lambda\_.\text{return } v,\\
&\text{put}((loc, n), \kappa) \mapsto \lambda flag.\\
&\quad \text{if } (n \bmod 2 == 0 \wedge !flag) \vee (n \bmod 2 == 1 \wedge flag),\\
&\quad \text{then } \text{print}(\text{“Integers did not alternate in parity.”}, \lambda\_.\text{raise}(()))\\
&\quad \text{else } (k())(!flag),\\
&\text{get}(loc, \kappa) \mapsto \lambda flag.(\lambda s.\kappa\ s)(flag)\}.
\end{aligned}
\tag{3.17}
$$

Similarly to previous handlers, we pass a boolean $flag$ which determines whether the previous integer was odd, in which case the value would be 1 or even, with value 0. When a put is found, we check whether n is odd or even by seeing the remainder when divided by 2. If, for instance, the current integer is even and the flag's value is false, then we have found two consecutive even numbers. We print an error message on the screen and abort the program. The case when two consecutive odds are found is analogous. Otherwise, we resume the computation and apply it to the flag's new value. If previously the flag was false then now it should be true and vice versa, hence, we reverse the value of $flag$. If a get is found, we return a function with the $flag$ parameter, resume the continuation and apply it to the current value for $flag$.

The handler is expressed in OCaml code as seen in Listing 3.9.

```
let flag : int ref = ref 2
| Put (loc,s) ->
    Some (fun (k: (b,_) continuation) ->
    if !flag == 2 then
        (if s mod 2 == 0 then (flag := 0; continue k ())
        else (flag := 1; continue k ()))
    else if (s mod 2 == 0) then
        (if !flag == 0 then
        (printf "Two consecutive evens found.\n";
        discontinue k Not_alternating)
        else flag := 0; continue k ())
    else (if !flag == 1 then
      (printf "Two consecutive odds found.\n";
      discontinue k Not_alternating)
        else flag := 1; continue k ()) )
```

Listing 3.9 Alternating Parity Monitor

The monitor has an integer reference flag to keep track whether the last integer stored was odd or even. The flag's uninitialised value is 2 so that upon the first put, the flag's value does not indicate that some odd or even integer was stored previously. When an even integer is passed, the value of flag is updated to zero so that if the next integer is also even, an error message is printed to the console and the program is terminated by an exception. The case when two odds are encountered is analogous. Otherwise, if the integers are alternating, the flag is updated accordingly and the program is resumed.

3.5 A Monitor for the Aliasing Problem

One of the most pressing problems in programs is to keep track of the variables which have been aliased. Aliasing is when two or more different variables are associated with the same location in memory. This can lead to unexpected side effects because changes made to one variable can inadvertently affect other parts of the program that use the same data. This can make it difficult to predict the behavior of the program and identify the source of bugs. While aliasing offers benefits such as memory efficiency and code reuse, we must diligently manage its complexities to avoid bugs and ensure program reliability.

If two variables $\mathrm{mem1}$ and $\mathrm{mem2}$ are pointing to the same block of memory, then, whenever one variable is updated, so is the other. Therefore, retrieving the values from both of these variables would return the same result. In a program such as Listing 3.10, if the two variables are not associated with the same location in memory, then $\mathrm{get}(\text{"mem1"})$ will return 5 and $\mathrm{get}(\text{"mem2"})$ will return 41, however, if $\mathrm{mem1}$ has been assigned to $\mathrm{mem2}$, the value for $\mathrm{mem1}$ will be 41.

```
1 put("mem1", 5); put("mem2", 41); get("mem1")
```

Listing 3.10 Sample Program

We design and implement a monitor to check and compare the values stored in memory with the values read from memory to ensure consistency and accuracy. If there is any discrepancy or inconsistency, the monitor will alert the system and abort the program. This monitor is an instance of the stateful class of monitors that analyse two types of events thrown by our program.

The handler for such a monitor is defined as follows.

$$
\begin{aligned}
H = \mathsf{handler}\{&\mathsf{return}\ v \mapsto \lambda\_.\mathsf{return}\ v,\\
&\mathsf{put}((loc, n), \kappa) \mapsto \lambda prev\_lst.(k())(\mathsf{update}(prev\_lst, loc, n)),\\
&\mathsf{get}(loc, \kappa) \mapsto \lambda prev\_lst.\lambda s.\ \mathsf{if}\ s \neq (\mathsf{find}(prev\_lst, loc))\ \mathsf{then}\\
&\ \mathsf{print}(\text{"Alias Detected"}, \lambda\_.\mathsf{raise}(()))\\
&\ \mathsf{else}\ (ks)(prev\_lst)\}.
\end{aligned}
\tag{3.18}
$$

We pass around a list $prev\_lst$ of pairs (s, n) where s is a string identifying a memory location and n is the last integer that was stored in the memory location for s. The put effect is handled by storing the passed integer in $prev\_lst$. On the other hand, get is handled by checking whether the current state read s is equal to the last stored value of loc from $prev\_lst$ through find. If they do not match, an error message is printed on the screen and an exception is raised. Otherwise, the continuation is resumed and applied to $prev\_lst$ unchanged.

The monitor is translated into OCaml and works in a similar way.

```
let prev_hash : (string , int ref) Hashtbl.t = Hashtbl.create 10
| Put (loc,s) ->
    Some (fun (k: (b,_) continuation) ->
        update_entry prev_hash loc (ref s); continue k ())
| Get y ->
    Some (fun (k: (b,_) continuation) ->
        let current = get_entry names_hash y in
        let prev = get_entry prev_hash y in
        if (!current) != (!prev)
        then (printf "The previous [put] stored value %d but the current
             [get] retrieved value %d.\n" (!prev) (!current);
            discontinue k Inconsistent)
        else continue k (!current)))
```

Listing 3.11 Aliasing Monitor

Similarly to previous monitors, we have introduced another Hashtable $\mathrm{prev\_hash}$ that relates the variables names to the last stored integer in their variable counterpart. In this handler, the Put effect is handled by simply updating the Hashtable with the integer passed. The effect Get is handled by first retrieving the last value stored in the variable loc from $\mathrm{prev\_hash}$ and by obtaining the value curerntly stored in the variable from $\mathrm{names\_hash}$. These two values are compared against each other and if they disagree, an error message is printed to the console and the program resumed by the exception $\mathrm{Inconsistent}$, otherwise, the program is resumed by passing the current value stored in the variable to the continuation. This approach enables the monitor to verify that the first get retrieves the value of the last put. If the values do not coincide, it signifies that a change has been made outside the program itself. As in previous cases, a list can be passed to the continuation similarly to the theoretical handler instead of globally defining a hashtable, however, time limitations hindered the execution of this change.

3.6 Joining Two Monitors Together

So far we have seen examples of stand-alone monitors with only one property to verify. Two effect handlers may be merged into each other by simply putting together the

fields from the stand-alone monitors into one handler. For instance, if we want to check whether our current program has no aliasing and that each variable's running total does not exceed 500, we would create a new handler from Equation (3.15) and Equation (3.18) as seen in Equation (3.19).

$$
\begin{aligned}
H = \mathsf{handler}\{&\mathsf{return}\ v \mapsto \lambda\_, \_.\mathsf{return}\ v,\\
&\mathsf{put}((loc, n), \kappa) \mapsto \lambda sum, prev\_lst.\ \mathsf{if}\ sum < 500,\ \mathsf{then}\\
&\ (k())(sum + n, \mathsf{update}(prev\_lst, loc, n))\\
&\ \mathsf{else}\ \mathsf{print}(\text{“Total sum of puts exceeded 500.”}, \lambda\_.\mathsf{raise}(()))\\
&\mathsf{get}(loc, \kappa) \mapsto \lambda sum, prev\_lst.\lambda s.\ \mathsf{if}\ s \neq (\mathsf{find}(prev\_lst, loc))\ \mathsf{then}\\
&\ \mathsf{print}(\text{”Alias Detected”}, \lambda\_.\mathsf{raise}(()))\ \mathsf{else}\ (ks)(sum, prev\_lst)\}.
\end{aligned}
\tag{3.19}
$$

In this case, we pass around two parameters, the aggregate sum and the list of pairs keeping track of previous entries. The only major change from Equation (3.15) and Equation (3.18) is that the continuation is applied to a pair instead of one argument since the monitor is verifying more than one property. In OCaml, a similar approach is taken to join two monitors together, as seen in the code listing below.

```
| Put (loc,s) -> Some (fun (k: (b,_) continuation) ->
                    update_entry prev_hash loc (ref s);
                     if !(sum) < 500 then
                     (sum := !(sum) + s; continue k ())
                     else printf "Total sum of puts exceeded 500.\n";
                     discontinue k (Exceeded_value 500))
| Get y -> Some (fun (k: (b,_) continuation) ->
                     let current = get_entry names_hash y in
                     let prev = get_entry prev_hash y in
                     if (!current) != (!prev)
                     then (printf "The previous [put] stored value %d but the
   current [get] retrieved value %d.\n" (!prev) (!current);
                     discontinue k Inconsistent)
                     else continue k (!current)))
```

Listing 3.12 Two Joined Monitors

Although we can combine two or more monitors together, this is not a composition of monitors. Here we modified the handler internally to be able to verify more than one property of our program, whereas a way to use the existing monitors as components in a larger monitoring system would be akin to compositionality. This approach would provide modularity, reusability, and easier maintenance of the monitoring system. We discuss this further in the next chapter.

4 Layered Monitoring

In Chapter 3, we have seen several instances of monitors ranging from stateless monitors to stateful monitors observing two events and how these can be described using effect handlers. We would like to find a way how to combine two or more monitors together so that a program may be tested for more than one property. The aim is to use the existing monitors and chain them together so that each monitor's check is carried out on our program in a compositional way. The last example found in Listing 3.12 illustrates how two monitors can be combined into one, however, it is not a composition of two monitors. Here, we modified the code of the aggregate value monitor from Listing 3.8 to add the necessary checks for the aliasing monitor from Listing 3.11, however, this increases the complexity of the handler and adds redundancy by duplicating code from existing handlers. This chapter discusses whether a compositional treatment can be applied to monitors described by algebraic handlers.

For efficient verification with minimal impact on computational complexity and correctness, it is essential to keep monitors lightweight. Hence, there is a need to explore reusing existing handlers rather than creating overly complex ones. A possible approach would be to sequence a number of monitors $\mathrm{mon1}, \mathrm{mon2}, \ldots, \mathrm{monN}$ with each other as $\mathrm{mon1}; \mathrm{mon2}; \cdots; \mathrm{monN}$. In this way, the program is tested for all the properties described by the monitors and when one monitor's check fails, an exception is raised and the entire program is terminated. This allows us to leverage existing stand-alone monitors and handlers to test the plain-vanilla program for multiple properties without adding complexity to the handlers or monitors. However, the plain-vanilla program is executed as many times as there are monitors, making the entire process unnecessarily computationally expensive.

We show that there is a way how the monitors from Chapter 3 can be reformulated to useful components for a larger monitoring system. The two approaches provided above are far from ideal, partly due to the fact that we have neglected a powerful property of algebraic handlers; compositionality. Algebraic handlers are mappings themselves, therefore, they can be composed with each other exactly like functions. A computation c which is to be handled by k handlers may be written as with H_1 handle (with H_2 handle $(\ldots$ with H_k handle $c))$, where an effect E thrown by c is handled by the first handler that defines the alternative behaviour for E. Once the effect is handled by some handler H_i, the effect is no longer able to be acted upon by the rest of the handlers, unless the effect is raised again by H_i. From the monitors shown in Chapter 3, we can see that a monitored program can produce other side-effects of its own which include raising exceptions and modifying the contents of memory locations. We can monitor these side-effects to verify whether the expected changes are being made or not by attaching another monitor

to the monitored program. We highlight the fact that a monitored program is another program itself, therefore, we can add another monitor to it to verify other properties. Figure 4.1 illustrates how the handlers, or monitors, can be layered onto each other. The program at the core raises an effect E_1 which is then handled by monitor 1. In turn, this monitor can raise some other effect E_2 to be handled by one of the handlers in the outer layers and so on. The catch all monitor would be the final monitor in our system of layered monitors that successfully terminates the entire verification process given that the other monitors did not report a problem. Therefore, we can design several self-contained handlers which are then composed together to create a larger, more complex handled computation. Breaking down complex computations into smaller parts enhances code clarity and organization. This modular design also simplifies debugging and maintenance since each handler can be reasoned about on its own and modified independently without affecting the overall functionality of the program. Furthermore, this would allow us to reuse already defined handlers, significantly reducing code redundancy.

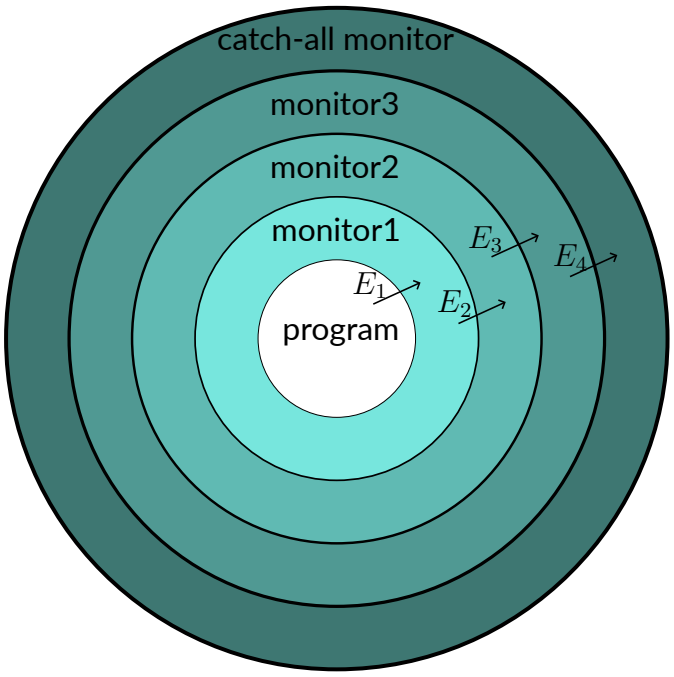


Figure 4.1 Composing Monitors

In OCaml, handlers can also be nested within each other, noting that each handler has to be wrapped inside a function that consumes the unit value $\mathrm{fun}\ ()\ \text{->}$ due to the semantics of the $\mathrm{match\_with}$ operation, found in [11]. Since a handled effect is no longer visible for the other handlers afterwards, composing handlers is a useful technique when the monitors in each layer act on different events. This nested composition of handlers models closely the existing layered monitoring technique, however, it still fails to precisely describe the behaviour of a layered system. To more accurately model layered monitoring using algebraic handlers, we need to ensure that any handled effect is prop-

agated again, allowing subsequent handlers to intercept and respond accordingly.

The compositionality of handlers together with re-performing effects could be the best possible model to express a layered monitoring system using algebraic effect handlers. In this way, we can build a complex monitoring system by composing smaller, simple monitors and establishing a coherent, modular system. Theoretically, handlers can be composed with each other naturally. To raise an effect E which has just been handled, we simply raise this operation before continuing the rest of the program in the handler. For instance, consider the stateless monitor in Equation (4.1). If it is to be attached to another monitor that analyses some other property of put, the handler would be altered as seen in the equation below.

$$
\begin{aligned}
H = \mathsf{handler}\{&\mathsf{return}\ v \mapsto f(v),\\
&\mathsf{put}((loc, n), \kappa) \mapsto\\
&\quad \lambda\_.\ \mathsf{if}\ n = -1,\ \mathsf{then}\\
&\quad \mathsf{print}(\text{“Put with value -1 encountered.”}, \lambda\_.\mathsf{raise}(())), \mathsf{else}\ \mathsf{put}((loc, n)); \kappa()\ ),\\
&\mathsf{get}(loc, \kappa) \mapsto \lambda n.\kappa n\}.
\end{aligned}
\tag{4.1}
$$

In OCaml, this would translate to attaching a $\mathrm{perform(E)}$ right before resuming the rest of the program. For our current implementation we believe this is sufficient to achieve what we want, however, if it is to be altered to remove globally defined v variables, one must proceed with caution when structuring the sequence of operations.

5 Conclusion

From the three objectives **(O1)** - **(O3)** listed in the introduction, we have successfully attained objectives **(O1)** and **(O2)** in Chapter 2 and Chapter 3. We first define effects and handlers using algebra theory then we apply them to describe computational effects. In Chapter 3 we describe monitors in terms of algebraic handlers, then, we transform the handler into an executable OCaml program. The third objective **(O3)** was partially reached in Chapter 3 since several instances of stand-alone monitors are found from Section 3.3 till Section 3.5. In Chapter 4 we discussed a way to approach layered monitoring, however, the attempt at its implementation has room for improvement.

5.1 Limitations

One limitation of the implementation in OCaml is that we assume that our variables are all defined in the same scope. Having different scopes in a program, through a function definition or let construct, allows variables to be defined only within a particular part of the code. In this case, the labelling of the variables needs to be improved to include the names of the scopes, which is not present in our implementation.

Another limitation is that the plain-vanilla program we start off with needs to be modified substantially for it to be instrumented, as we have seen in Section 3.2. This is not a problem with a small number of effectful operations, however, as the program is expanded, the instrumentation process may become cumbersome. In a fully fledged solution, this process would be automated through standard tools. However, this was beyond the scope of our study. See [15] for an outline on how to do this.

5.2 Related and Future Work

The current implementation of the monitors can be significantly improved in the following ways. Firstly, we we modify the handlers so that the handled effects return a function that passes around some parameters required by the monitor. In this way, we avoid globally-defined variables and the implementation of the monitor aligns with the associated algebraic handler. Then, as discussed in Chapter 4, we can implement a layered monitoring system using the compositionality of handlers and raising the effects again for other monitors to pick up again. This generalisation is a natural extension of stand-alone monitors and would showcase a powerful property of algebraic handlers.

From a theoretical point of view, Pretnar in [16] showcases an alternative interpretation of effects and handlers by providing operational semantics, typing and effect systems for effects and handlers. This approach would make effects and handlers easier

to understand and reason about for those familiar with programming language design. In the future, we can further explore this interpretation to achieve a more computer science-oriented understanding.

OCaml is not the only functional programming language that offers support for effectful operations. Haskell implements the monad structure to handle several well-known computational effects, include I/O and State. Algebraic effects are closely related to monads, as we discuss in the following subsection.

5.2.1 Algebraic Effects versus Monads

Monads originated from category theory and were later introduced to computer scientists by Moggi [17] to aid in establishing semantics for computations. Now they are mainly known for their ability to handle side-effects in Haskell, such as error handling, state and asychronous operations. More importantly, monads are a powerful tool to sequence effectful computations in Haskell in a controlled and well-behaved manner.

A monad is a triple $(\mathrm{M},\ \mathrm{unit},\ \mathrm{bind})$, where M is a type constructor and the other two are polymorphic operations defined on it. unit takes a value of type a and returns $(\mathrm{M\ a})$ whilst bind, typically denoted by $>>=$, takes a container $(\mathrm{M\ a})$ and a function a -> $\mathrm{M\ b}$, and returns a container of type $\mathrm{M\ b}$.

Monads also have three laws that must be satisfied, namely left identity, right identity and associativity. Together, these allow monads to be composed with each other in a similar way to function compositionality, which ensures that monad compositionality is well-behaved. Haskell is also equipped with the 'do' notation, some syntactic sugar that allows programmers to sequence monadic computations, where a computation is executed only if the previous one succeeds.

How do monads relate to algebraic effects and handlers? Both of these structures stem from category theory and provide a way to describe and manage computational effects. The main practical difference lies in where the effectful behavior is specified. In monads, it is within the definition of the operators bind and unit, which then statically set the behavior of the monadic code found inside the 'do' block. On the other hand, code making use of algebraic effects can invoke effectful operations that inherently possess no specific behavior on their own. Instead, their behaviour is dynamically determined through the closest enclosing handler. This provides increased flexibility and modularity in composing effectful code.

Nevertheless, Forster et al. in [18] show that monads can be somewhat expressed in terms of algebraic effects and vice versa. A program which is written using algebraic effects can be translated into a monadic one by doing some local structural changes, thus, preserving the overall structure of the program. Such a property is referred to as macro-expressivity. However, one important limitation is that there cannot be macro-

expressivity of effect handlers into monads (or vice versa) while maintaining typeability.

From a computational perspective, an algebraic handler is similar in behaviour to the 'goto' operation, where the program jumps to some other place in the code once the keyword is reached. Programs written using algebraic handlers may be converted to simulate monadic programs by still using the 'goto' notation, however, we jump to the next step of the code as we would do in a monadic program. Meanwhile, monads are similar to the sequencing of computations, where we link one computation with another. Moreover, algebraic handlers offer this separation of concerns between raising effects and handling them, whereas monads encapsulate raising an effect and handling it in one entity.

In conclusion, our exploration of monads and algebraic effect handlers has shed light on two powerful constructs for managing side effects and sequencing computations in functional programming. While monads offer a well-established and widely-used approach, algebraic effect handlers provide a more flexible and modular framework for handling effects. These two constructs differ from each other on where the effectful behaviour is defined, however, there are strong relationships between them with some trade-offs. As part of future work, we can study this relationship more closely.

References

[1] I. Leandersson. "Six Surprising Reasons the OCaml Programming Language is Good for Business." (), [Online]. Available: https://tarides.com/blog/2022-11-22-six-surprising-reasons-the-ocaml-programming-language-is-good-for-business/.

[2] D. Scott, R. Sharp, T. Gazagnaire, and A. Madhavapeddy, "Using Functional Programming within an Industrial Product Group: Perspectives and Perceptions," vol. 45, Dec. 2010, pp. 87–92. DOI: 10.1145/1863543.1863557.

[3] D. A. Spuler and A. S. M. Sajeev, "Compiler Detection of Function Call Side Effects," *Informatica (Slovenia)*, vol. 18, 1994. [Online]. Available: https://api.semanticscholar.org/CorpusID:27166185.

[4] D. MacQueen, M. Tofte, R. Harper, and R. Milner, *The Definition of StandardML*. MIT Press, 1990.

[5] A. Madhavapeddy, J. Hickey, and Y. Minsky, *Real World OCaml*. Cambridge University Press, 2013.

[6] D. Fancher, *The Book of F#:Breaking Free with Managed Functional Programming*. No Starch Press, 2014.

[7] O. Tutorials. "Ocaml Multicore Branch." (), [Online]. Available: https://github.com/ocaml-multicore/ocaml-multicore.

[8] G. D. Plotkin and J. Power, "Adequacy for Algebraic Effects," *FoSSaCS. Springer-Verlag*, 2001.

[9] G. D. Plotkin and J. Power, "Notions of Computation Determine Monads," *FoSSaCS. Springer-Verlag*, 2002.

[10] G. D. Plotkin and J. Power, "Algebraic Operations and Generic Effects," *Appl. Categ. Structures*, pp. 69–94, 2003.

[11] K. Sivaramakrishnan, S. Dolan, L. White, T. Kelly, S. Jaffer, and A. Madhavapeddy, "Retrofitting effect handlers onto OCaml," ser. PLDI 2021, Virtual, Canada: Association for Computing Machinery, 2021, pp. 206–221, ISBN: 9781450383912. DOI: 10.1145/3453483.3454039. [Online]. Available: https://doi.org/10.1145/3453483.3454039.

[12] E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger, "Introduction to Runtime Verification," *Lectures on Runtime Verification. Introductory and Advanced Topics*, pp. 1–33, 2018.

[13] A. Bauer, "What is algebraic about algebraic effects and handlers?" *CoRR*, 2018. [Online]. Available: http://arxiv.org/abs/1807.05923.

[14] G. Plotkin and J. Power, "Semantics for algebraic operations," *Electronic Notes in Theoretical Computer Science*, vol. 45, pp. 332–345, 2001, MFPS 2001,Seventeenth Conference on the Mathematical Foundations of Programming Semantics, ISSN: 1571-0661. DOI: https://doi.org/10.1016/S1571-0661(04)80970-8. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1571066104809708.

[15] I. Cassar, A. Francalanza, L. Aceto, and A. Ingólfsdóttir, "A survey of runtime monitoring instrumentation techniques," *Electronic Proceedings in Theoretical Computer Science*, vol. 254, pp. 15–28, Sep. 2017. DOI: 10.4204/EPTCS.254.2.

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

[17] E. Moggi, "Notions of computation and monads," *Information and Computation*, vol. 93, no. 1, pp. 55–92, 1991, Selections from 1989 IEEE Symposium on Logic in Computer Science, ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(91)90052-4. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0890540191900524.

[18] Y. Forster, O. Kammar, S. Lindley, and M. Pretnar, "On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control," *Journal of Functional Programming*, vol. 29, e15, 2019. DOI: 10.1017/S0956796819000121.

Appendix A Further Background on Algebraic Effects

Using Definition 2.13, the context must also be generalised to allow an arbitrary number of variables. Therefore, the list of distinct variables is replaced by a set X of variables. This generalisation is also reflected in how we define Σ-terms. Here we define terms immediately in terms of well-founded trees, where terms in a context X have the operation symbols op_i as the roots, and each op_i has A_i-many sub-trees. These trees together form a set $\mathsf{Tree}_\Sigma(X)$ and are built inductively using the rules in the definition below.

Definition A.1. *A **well-founded tree** is built inductively as follows:*

1. *for every x in X, there is a tree return x,*
2. *for a parameter p and $\kappa : A_i \to \mathsf{Tree}_\Sigma(X)$, $\mathsf{op}_i(p, \kappa)$ is a tree whose root is labelled op_i and has A_i-many sub-trees produced by κ.*

Therefore, equations may be formed in terms of trees as follows.

Definition A.2. *A Σ **equation** is a set X and a pair of Σ-terms $\ell, r \in \mathsf{Tree}_\Sigma(X)$ written as $X \mid \ell = r$.*

Usually, the context X of an equation is omitted. The definition of an algebraic theory remains unchanged from that given in Definition 2.5.

The interpretation of a signature can be extended to operation with parameters and general arities as follows.

Definition A.3. *An **interpretation** I of a signature Σ is given by:*

1. *a carrier set $\overline{I}$,*
2. *for each operation symbol op_i with parameter set P_i and arity set A_i, there is a map $[\![\mathsf{op}_i]\!]_I : P_i \times \overline{I}^{A_i} \to \overline{I}$.*

However, a theory is also constructed using terms and equations, thus, we extend an interpretation I to Σ-terms.

Definition A.4. *A tree $t \in \mathsf{Tree}_\Sigma(X)$ is a map $[\![t]\!]_I : \overline{I}^X \to \overline{I}$ such that:*

1. *the tree return x is interpreted as the x^{th} projection,*
   $[\![\mathsf{return}\ x]\!] : f \mapsto f(x)$
2. *the tree $\mathsf{op}_i(p, \kappa)$ is the map $[\![\mathsf{op}_i(p, \kappa)]\!]_I : \overline{I}^X \to \overline{I}$ such that*
   $[\![\mathsf{op}_i(p, \kappa)]\!]_I : f \mapsto [\![\mathsf{op}_i]\!]_I(p, \lambda a.[\![\kappa(a)]\!]_I(f))$.

If the tree is labelled return x, then, its interpretation returns the value of x in $\overline{I}$ determined by f. Otherwise, each argument of an operation, given by κ, is a tree itself, therefore, we interpret each sub-tree and pass it as the second argument for $[\![\mathsf{op}_i]\!]_I$.

The definition of a model remains unchanged, however, we note that definitions 2.13, A.3 and A.4 are used instead.

For the interpretation of the signature to satisfy every equation in $\mathcal{E}_\mathsf{T}$ having the form $X \mid \ell = r$, the maps $[\![\ell]\!]_M : \overline{M}^{A_i} \to \overline{M}$ and $[\![r]\!]_M : \overline{M}^{A_i} \to \overline{M}$ are equal. The definition of a free model also remains unchanged.