NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

TIAGO LOPES SOARES

Bachelor in Computer Science and Engineering

HANDLE WITH CARE AND CONFIDENCE – EXTENDING CAMELEER WITH ALGEBRAIC EFFECTS AND EFFECT HANDLERS

AN ANALYSIS OF ALGEBRAIC EFFECTS AND TECHNIQUES TO DEDUCTIVELY VERIFY THEM

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
December, 2022

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

HANDLE WITH CARE AND CONFIDENCE – EXTENDING CAMELEER WITH ALGEBRAIC EFFECTS AND EFFECT HANDLERS

AN ANALYSIS OF ALGEBRAIC EFFECTS AND TECHNIQUES TO DEDUCTIVELY VERIFY THEM

TIAGO LOPES SOARES

Bachelor in Computer Science and Engineering

Adviser: Mário José Parreira Pereira
Assistant Professor, NOVA University Lisbon

Co-adviser: António Maria Lobo César Alarcão Ravara
Associate Professor, NOVA University Lisbon

Examination Committee

Chair: Maria Armanda Simenta Rodrigues Grueau
Associate Professor, School of Science and Technology
NOVA University Lisbon

Rapporteur: François Pottier
Principal Investigator, Inria Paris

Adviser: Mário José Parreira Pereira
Assistant Professor, School of Science and Technology
NOVA University Lisbon

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
December, 2022

Handle with Care and Confidence – Extending Cameleer with Algebraic Effects and Effect Handlers
An analysis of algebraic effects and techniques to deductively verify them

Copyright © Tiago Lopes Soares, NOVA School of Science and Technology, NOVA University Lisbon.
The NOVA School of Science and Technology and the NOVA University Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.10.16) [29].

I dedicate this thesis to all the haters and losers.

Acknowledgements

I would like to first thank my thesis supervisor Mário Pereira for all the support he has given me over the past three years. I feel immensely grateful for having you by my side for this extremely difficult journey. When I compare my experience with some of my colleagues', I know I made the right choice in going into program verification under your wing. It's hard to explain how much help and kindness you've given me in just one short paragraph, so I'll just say thank you for always giving me what I needed and always making me feel like I had someone watching my back.

I would like to thank professor António Ravara for his contributions to this project. Additionally, I would also like to thank professor François Pottier for being the rapporteur for this thesis. Finally, I would also like to extend a thanks to professor João Lourenço for providing this incredible LaTex template [29] (if you're reading this for whatever reason, let me tell you, you're a saint for doing this for free. Also you should be paid more).

I would like to thank my five best ~~friends~~ colleagues, Carolina, Diogo and Xavier. Even though none of you helped with or really even understood my thesis, I would never have gotten this far without all of you. The only thing that kept me from going insane from all the work these 5 years have entailed is being able to share this crazy ride with people as unhinged as me.

Também queria agradecer à minha mãe, ao meu pai e à minha irmã pelo apoio que me deram nestes difíceis 5 anos. Sempre puseram o meu bem estar acima de tudo o resto, e só por isso estarei sempre grato. Eu sempre senti-me amado incondicionalmente: mesmo se não acabasse a tese, sabia que teria sempre a minha família ao meu lado. Sem este apoio crítico, eu nunca teria conseguido fazer tudo o que fiz nestes 5 anos. Esta tese é tanto minha quão é vossa. Eu já vos disse isto tantas vezes, mas em caso de dúvida, amo-vos tanto.

"There can be no freedom without the abolition of the state.
When there is no state there will be freedom. "

(Lenin)

Abstract

The new major release of the OCaml compiler is set to be an important landmark in the history and ecosystem of the language. The 5.0 version introduces Multicore OCaml, a multi-threaded implementation of the OCaml runtime. Two new important paradigms shall arise in the language: parallelism via domains and direct-style concurrency via algebraic effects and handlers. In this work, we focus precisely on the latter and try to answer the following research question: "what tools and principles must be developed in order to apply automated deductive proofs to OCaml programs featuring effects and handlers?".

Algebraic effects and handlers are a powerful abstraction to build non-local control-flow mechanisms such as resumable exceptions, lightweight threads, co-routines, generators, and asynchronous I/O. All of such features have very evolved semantics, hence they pose very interesting challenges to deductive verification techniques. In fact, there are very few proposed techniques to deductively verify programs featuring these constructs, even fewer when it comes to automated proofs. In this report, we outline some of the currently available techniques for the verification of programs with algebraic effects. We then build off them to create a mostly automated verification framework by extending Cameleer, a tool which verifies OCaml code using GOSPEL and Why3. This framework embeds the behavior of effects and handlers using exceptions and defunctionalized functions.

Keywords: Deductive Verification, Algebraic Effects, Effect Handlers, Multicore OCaml, GOSPEL, Why3, Cameleer

Resumo

A próxima iteração do compilador OCaml será histórica no que diz respeito ao ecosistema da linguagem. A versão 5.0 introduzirá Multicore OCaml, uma implementação *multi-threaded* do *runtime* OCaml. Nesta versão, dois paradigmas serão adicionados: paralelismo utilizando *domains* e concorrência em estilo direto na forma de efeitos algébricos e *handlers*. Neste relatório, focar-nos-emos no segundo ponto, tentado responder à seguinte questão: "que ferramentas e princípios deveremos desenvolver de modo a applicar provas dedutivas automáticas a programas com efeitos e handlers?".

Efeitos algébricos e *handlers* são uma abstrações poderosas que nos permite construir mecanismos para controlar o curso de um programa como, por exemplo, exceções que nos permitem recomeçar a computação, *threads lightwheight*, corotinas, geradores e I/O asíncrono. Todos estes paradigmas são um grande desafio no contexto de verificação dedutiva pois têm semanticas bastante complexas. Neste relatório iremos abordar algumas das técnicas existentes para provar programas com efeitos algébricos. Ademais, propomos uma estratégia de verificação para provar automáticamente programas com *handlers*. Para este efeito, extendemos a ferramenta Cameleer, um verificador de código OCaml que utiliza a linguagem de especificação GOSPEL e o prover Why3. Esta extensão visa aproximar o comportamento de *handlers* utilizando exceções e funções desfuncionalizadas.

Palavras-chave: Verificação dedutiva, Efeitos Algébricos, *Effect Handlers*, Multicore OCaml, GOSPEL, Why3, Cameleer

Contents

List of Figures **xi**

1 Introduction **1**

1.1 Motivation 1

1.2 Problem Definition 2

1.3 Goals and Contributions 2

1.4 Thesis Structure 3

2 Background **4**

2.1 Functional Languages 4

2.2 Hoare Logic 5

2.2.1 Separation Logic 6

2.3 GOSPEL and Cameleer 7

2.4 Effect Handlers 9

2.4.1 Deep and Shallow handlers 11

2.4.2 Joining Streams 11

2.5 Peeking Under the Curtain : Handler Implementation 16

2.5.1 Implementing Handlers With Exceptions 17

2.5.2 Okay, But Why Does This Matter? 18

2.6 Exceptions: What Are They Good For? 18

3 State of the Art **20**

3.1 Typing Programs With Algebraic Effects 20

3.2 Modelling Algebraic Effects Using Equational Theory 21

3.3 Verification of Algebraic Effects with Separation Logic 21

3.3.1 Language Syntax 22

3.3.2 Protocols 23

3.3.3 Definition of Protocols 24

3.3.4 Formal Interpretation of Protocols 25

4 Defunctionalization 27
4.1 Defunctionalization in a Nutshell . . . 27
4.2 Defunctionalization Coupled with Verification . . . 28
4.3 Defunctionalization with State . . . 31

5 Reasoning about Effects and Handlers in Why3 33
5.1 GOSPEL Extension . . . 33
5.1.1 GOSPEL Protocols . . . 33
5.1.2 Effect Handlers . . . 34
5.1.3 Performs Clause . . . 35
5.1.4 The Remaining Grammar . . . 36
5.2 WhyML Formalization . . . 36
5.3 Proving an OCaml program with protocols . . . 38
5.3.1 Translating effects into WhyML . . . 41
5.3.2 Representing continuations in WhyML . . . 41
5.3.3 Specifying continuations . . . 42
5.3.4 Verification of the Program . . . 44
5.4 General Translation Scheme . . . 46
5.5 Limitations . . . 53

6 Case Studies 56
6.1 Cameleer Implementation . . . 56
6.2 Division Interpreter . . . 57
6.2.1 Interpreter Specification . . . 57
6.2.2 Translating The Interpreter . . . 58
6.2.3 Verifying the Interpreter . . . 60
6.2.4 Translation of the Handler . . . 60
6.2.5 Verifying the handler . . . 60
6.3 Mutable Reference . . . 61
6.3.1 Client implementation and Specification . . . 61
6.3.2 Handler Implementation and Specification . . . 62
6.3.3 Verification of the Handler . . . 65
6.4 Generators . . . 66
6.4.1 Client Specification . . . 66
6.4.2 Handler Implementation . . . 68
6.4.3 Handler Specification . . . 69
6.4.4 Verification . . . 70
6.5 Shallow Handlers . . . 72
6.5.1 Extending The Grammar . . . 72
6.5.2 Shallow to Deep . . . 73
6.5.3 Translation . . . 74

6.5.4 Verification 77
6.6 Overview 79

7 Conclusions **80**
7.1 Contributions 80
7.2 Future Work 82

Bibliography **83**

List of Figures

2.1 Stream join. 13
2.2 An OCaml fiber [38]. 17

4.1 Fully defunctionalized length. 29
4.2 Fully annotated `length`. 30
4.3 Defunctionalized proof of `length_cps`. 31

5.1 Extended GOSPEL Syntax. 37
5.2 WhyML Syntax. 38
5.3 Full OCaml program and GOSPEL specification. 44
5.4 WhyML proof. 45
5.5 State Definition. 47
5.6 Top Level Declarations. 48
5.7 Effect Translation Rule. 49
5.9 Defunctionalization Rule. 51
5.10 Translation rule for effect handlers. 54
5.11 Inductive Translation Rules. 55

6.1 Handler for the `xchg` function implemented in OCaml. 57
6.2 Division Interpreter. 57
6.3 Specified Division Interpreter with Handler. 59
6.4 Complete `sum_fun` handler. 64
6.5 Client Specification. 67
6.6 Axiomatized Definition of `iter_inv` 70
6.7 Handler for the `iter` function. 71
6.8 Partial translation of a shallow handler. 76
6.9 Complete translation of the `deep` function. 77
6.10 Fully Translated Program. 78

1

Introduction

1.1 Motivation

Our physical lives are becoming evermore intertwined with the digital realm, seeing as entertainment, communications and global finances have all migrated to virtual platforms, essentially flattening many limitations posed by the physical world. With such a great load to bear, it is not a bold statement to say that all these mechanisms must not only be efficient and easy to use but must also provide some sort of guarantee that they will always present the expected result and will never suffer any catastrophic failure. This is without a doubt one of the biggest (arguably impossible) challenges that computer scientists have struggled with for as long as computer science has been a respectable field. Although overcoming this challenge involves many different areas of study, in this thesis we will focus on one: the correctness of code.

The most widespread way of checking if a program produces the expected result is testing. However, as Djikstra put it, "program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence" [14]. In order to prove such a rigid condition, we need formal proofs of a program's correctness. Seeing as writing these proofs by hand would be far too cumbersome, especially for the complex systems that warrant such a methodology, we will use automated verifiers that will dispatch proof obligations and feed these into SMT (Satisfiability Modulo Theory) solvers [25], programs that can determine if a mathematical formula is valid or not; this is referred to as *deductive verification* [16].

This style of verification is best applied to software written in functional languages, mainly because these tend to avoid effectful constructs, making it easier to apply a logical model on them. Another reason being that functional conventions such as pattern matching and recursion go hand in hand with a mathematically minded approach. For this project, we will consider OCaml as our target language.

More specifically, the class of OCaml programs we will consider are those that feature *effects* and *effect handlers* [33]. In short, effect handlers behave similarly to *try-catch* blocks: when an effect is performed, execution is halted, and control is given to the handler. The

main difference lies in the fact that an effect handler exposes a *continuation*: a function that, when called, will resume computation at the point the effect was performed.

Effect handlers can be employed for a myriad of applications: they can be used in creating lightweight threads, developing highly modular code and exposing control of higher-order iteration. Handlers have been gaining popularity as a way of encoding the interactions a program has with its environment. As such, they have been, over the course of the past decade, implemented in a series of research programming languages [6, 4, 12, 26, 19]. More recently, the latest alpha release of Multicore OCaml features effect handlers with *delimited one-shot continuations* [38].

1.2 Problem Definition

Although handlers enable us to write functional programs in interesting new ways, they are not trivial to write proofs for, given the fact that the computation might be suspended and restarted, breaking the normal control flow we would expect from a sequential program. Luckily, some research has been done [13] that presents reasoning rules for this kind of program, albeit none of them explore automated proofs. Therefore, the research question of this work is

> *what tools and principles must be developed in order to apply automated deductive proofs to OCaml programs featuring effects and handlers?*

1.3 Goals and Contributions

This thesis proposes a framework to automatically verify programs that employ algebraic effects. To achieve this, we support to prove these programs to Cameleer [31], a deductive verification tool for OCaml whose specifications are written in GOSPEL [10] (Generic OCaml SPEcification Language) and then translated to Why3 [5], an automatic theorem prover. Cameleer is also the only automated deductive verification platform (we are aware of) that targets OCaml programs, making it an ideal choice for this thesis.

To achieve this goal we must be able to represent the continuation function exposed by the handler in our proofs, which we will do using a translation of algebraic effects into Why3 that employs defunctionalization [37], a transformation that turns higher order programs into first-order ones. Although some research has been done using this technique in the context of program verification [40], we will refine it further in order to employ it to algebraic effects. Our goals are twofold:

1. to develop an embedding of OCaml handlers and effects into WhyML, employing defunctionalization; and

2. extending Cameleer to automatically translate GOSPEL annotated programs into this embedding.

1.4 Thesis Structure

- Chapter 2 covers the necessary grounds to understand the problem at hand, such as existing techniques to verify and specify programs, as well as a clearer definition of what algebraic effects are.
- Chapter 3 explains the existing methodologies with which to prove programs with algebraic effects. We will give special emphasis to protocols.
- Chapter 4, we show how defunctionalization can help with the verification of higher-order programs with side effects such as mutable state.
- Chapter 5 contains the formalization of two ML languages similar to GOSPEL annotated OCaml and WhyML. We will also showcase the translation rules that will allow us to encode the meaning of the specifications of our OCaml-like language into our WhyML-like language.
- Chapter 6 shows four concrete examples of OCaml annotated GOSPEL programs proved using our encoding.
- Chapter 7 will summarize our contributions, limitations and future prospects.

2 Background

2.1 Functional Languages

A common critique of projects of this type is: why OCaml? Or why functional languages in general? This sort of reasoning usually springs from the fact that OCaml and its peers never seem to reach the same level of notoriety as their imperative counterparts [1], mostly because the learning curve for functional languages is quite steep, especially if one is already ingrained in imperative patterns. However, sidelining these languages simply due to their popularity would be a grave error, seeing as many small and major corporations rely on OCaml code to implement a wide variety of systems [11]. One of the biggest sharks that use OCaml is Jane Street, an extremely large Wall Street firm, who have been writing OCaml code for over 15 years and developed Core, one of the most commonly used open source OCaml libraries [18].

OCaml applies a combination of features that, although quite pervasive in many other languages, are rarely wedded in a single tool. For example, we do not need to declare a variable's type, like in Python; nonetheless, we have static type checking, like in Java, since the compiler can infer, by context alone, if each variable is used in a safe manner. Like in Javascript and Python, we have first class functions that can be used as ordinary values. Additionally, we also have access to many other features that, although quite useful, are rarely seen in more popular languages such as algebraic data types, pattern-matching and parameterized modules [18].

More generally, the main difference between the imperative and functional style of programming is that functional languages take a more declarative approach. Value and function definitions are commonly trees of expressions, unlike in imperative languages where they are a list of instructions that may modify or access the global state of the program. Functional programs also make extensive use of recursion coupled with pattern matching as opposed to loops, as well as a greater focus on purity than stateful effects.

These qualities make functional languages more amenable to deductive verification, seeing as these programs are already quite similar to mathematical formulas, both in style and substance. Nonetheless, OCaml still gives us access to a rich ecosystem that

encompasses many imperative features, most notably, mutable references and exceptions, making it more versatile than other functional languages such as Haskell.

2.2 Hoare Logic

In deductive verification, we generally aim to express some relation between the arguments a function takes and its result. For this purpose we can use Hoare Logic [21], a formal system built on top of Hoare Triples, which consist of a precondition P, a postcondition Q and a piece of code s. The Hoare triple $\{P\}s\{Q\}$ is said to be valid under Hoare logic if an execution of s starting in any state in which P holds implies that, after s halts, Q will be true.

Although Hoare Logic is very useful, it can fall short when verifying effectful programs, given that it can lead to very complex specifications. To demonstrate this problem, we will present a method which copies the contents of an input array to an output array, written in Dafny [27], a program verifier built on top of Hoare Logic and implicit Dynamic Frames [39]. Although the most obvious postcondition we want to prove is that the contents are copied, we could also try to ensure that the contents of the input array are unmodified.

```
method copyArray(output : array<int>,
    input : array<int>, offset : int)
ensures ∀ i • 0 ≤ i < input.Length ⟹
   old(input[i]) = input[i] ∧ input[i] = output[i + offset]
modifies output
//rest of the specification omitted
{
    var i := 0;
    while(i < input.Length) {
        output[i + offset] := input[i];
        i := i + 1;
    }
}
```

The **ensures** clause allows us to define our postcondition, in this case, that the input array is unmodified and that the contents of the input array have been copied to the output array at the correct offset. We also have a **modifies** clause which states which variables this program will modify. The immutability of the input array appears trivial to prove, seeing as it is not accessed and we omitted it from the **modifies** clause. However, Dafny is unable to prove it, given that the input and output array may be aliased, meaning that modifying the output array would imply changing the input array. The only way to fix this problem is to add the clunky precondition : $input != output$. This specification is a

clear example of Hoare Logic interpreting specifications with state variables in ways that do not make intuitive sense.

The fundamental issue is that, with Hoare Logic, we must have a clear and complete definition of the state of memory in all our proofs. Since these issues can crop up even in relatively small programs, using Hoare Logic with state variables is an invitation for a great deal of complications.

2.2.1 Separation Logic

Naturally, an extension of Hoare Logic was created which aimed to patch these problems: Separation Logic [8]. The general idea is similar to Hoare Logic: we have a triple made up of a precondition, a postcondition and a piece of code. Only this time, our logical assertions reason over a fragment of memory instead of having to model it completely. We do this using heap predicates, of type $Heap \rightarrow Prop$, which describe the state of the memory fragment that this program operates over. Another important element in Separation Logic is the star operator: given two heap predicates $H1$ and $H2$, $H1 * H2$ means not only that $H1$ and $H2$ are valid, but that they refer to disjoint fragments of memory, one that validates the predicate $H1$, the other validates $H2$. But if we are only describing a fragment of memory, what is to be said of the rest we are effectively ignoring? Separation Logic includes a very powerful rule known as the *frame rule*. Essentially, it states that any piece of memory not explicitly referred to in our precondition will not be accessed nor modified. More formally, given two disjoint heap predicates H and H', if the triple $\{H\}t\{Q\}$ is valid then $\{H * H'\}t\{Q * H'\}$ is also valid for any arbitrary heap predicate. This is described by the following inference rule:

$$\frac{\{\text{H}\}\ \text{t}\ \{\text{Q}\}}{\{\text{H * H'}\}\ \text{t}\ \{\text{Q * H'}\}}\ \text{Frame rule}$$

Additionally, if a piece of memory is referred to in our precondition, but not in our postcondition, then that memory fragment is now inaccessible.

Another important operator is $x \rightsquigarrow v$, which states that the value the memory reference x holds can be defined by the logical value v. To assert a pure predicate P (i.e not a heap dependent predicate) holds we write P. Finally, if we do not wish to reason over any memory fragment we use , essentially asserting an empty state. In order to show these operators in action and their expressive power, let's examine a very simple specification for an allocation of a memory cell (this example is taken from [8]):

$$\{\ \}\ ref\ v\ \{\ \lambda r.\exists p.r = p * p \rightsquigarrow v\ \} \tag{2.1}$$

Assuming the r argument is the result of evaluating expression $ref\ v$, the postcondition states that the program allocates a memory cell p that holds the value v. Interestingly, it is left implicit that this pointer is different from every other pointer in the program, due to the frame rule: seeing as we started with the empty state, we can infer that any existing

memory references are disjoint from the pointer p, seeing as none were accessed, without any extra guidance from the logician.

To make our proofs slightly easier, we will use an extension of separation logic which adds read only permissions [9]. Essentially, predicates marked as read-only can only be read, and therefore they need only to appear in the precondition and not the postcondition. More formally:

$$\frac{\{H * RO(H')\}\ t\ \{Q\}}{\{H * H'\}\ t\ \{Q * H'\}}\ \text{Read only frame rule}$$

We should note here that this variant of Separation Logic has not been proven to be sound in a language with effect handlers.

Another operator that will be relevant in Chapter 3 is the separating implication, most commonly referred to as the *magic wand* H1 $-\!\!*$ H2. In short, this is very similar to a normal implication where instead of propositions we have heap predicates: if H1 is true of a heap fragment, H1 * (H1 $-\!\!*$ H2) is equivalent to H2. Important to note that H1 and H2 don't necessarily need to describe two disjoint pieces of memory. The magic wand principle is captured by the following rule:

$$\frac{H1 * (H1 -\!\!* H2)}{H2}\ \text{Magic Wand}$$

2.3 GOSPEL and Cameleer

Although Separation Logic is a much more convenient way of handling mutable state, the proofs can still be rather verbose, which is why we will use GOSPEL [10] (Generic OCaml SPEcification Language), a higher level specification language built on top of Separation Logic with read-only permissions. GOSPEL's syntax is very different from what we have seen thus far in regards to Separation Logic. Nonetheless it is much simpler, seeing as it makes certain assumptions about state variables that simplify the process of writing our specifications. Generally speaking, we assume that only variables that are read-only can be aliased. By making this assumption we are technically reducing the program space we can verify.

To demonstrate the expressive power of GOSPEL specifications, we will specify the `copyArray` function from section 2.2 as follows:

GOSPEL + OCaml

```
val copyArray (input : 'a array) (output : 'a array) (offset : int)
(*@ requires length input >= offset + length output
    ensures forall i. 0 <= i < length input ==>
        input[i] == output[i + offset] && old(input[i]) == input[i]
    modifies output *)
```

Note that GOSPEL specifications are added as comments beginning with @ at the end of the function definition. These conditions are quite similar to the Dafny specification, however, since we state that the `output` array is in the **modifies** clause, we can infer it is not aliased with input. This is because GOSPEL is built on Separation Logic, unlike Dafny, which is built on Dynamic Frames[39], an extension of Hoare Logic that remedies some of the problems in verifying programs with mutable data, but is still more verbose then Separation Logic.

GOSPEL, nonetheless, is only a specification language; it is not geared towards any specific tool [10]. Indeed, if one so desires, GOSPEL specifications can act only as a sort of formal documentation. If we wish to prove these, we must translate them in such a way that they can be used by some program verifier. One of the most obvious tools we could use is CFML [7], the only deductive verification tool we are aware of that is built specifically for OCaml code. Another option would be Iris [23], an extremely powerful and expressive logic which can handle concurrency, higher-order effectful functions and many other constructs. Nevertheless, these two fall outside the scope of our project, since we focus on automated theorem provers and both of these are based on interactive proof assistants. The main difference between these two categories is that the latter requires some assistance from the user to build proofs, whereas the former builds them automatically.

Given this, we turn to Cameleer, which translates GOSPEL annotated OCaml into Why3, an automated theorem prover which dispatches proofs written in WhyML, a language belonging to the ML dialect, making it very similar to OCaml. Like Dafny, Why3 is also built on top of Hoare Logic, but disallows untracked aliasing. For example:

WhyML

```
let x = ref 0
let y = x
```

Is allowed seeing as Why3 can keep track that x and y are the same reference. However, if we have a function where we pass two references, these cannot be aliased, unless they are read-only. Although these limitations line up very neatly with GOSPEL's infrastructure, problems arise when we introduce recursive data structures with mutable fields, such as:

WhyML

```
type link_list 'a = Nil | Cons 'a (ref (link_list 'a))
```

Since Why3 cannot keep track if this type forms a closed loop, seeing as the `Cons` constructor's second argument is a reference to a `link_list`, it will not allow constructs of this nature to be used in WhyML, specifically, recursive types with mutable fields. If we wanted to prove an OCaml program that uses this list, we would eventually have to use some other verifier, such as CFML, or alternatively, use a memory model implemented in Why3 [32].

2.4 Effect Handlers

Effect handlers have been around for over a decade [33], but have recently picked up momentum as a unique approach to model effectful behavior in a functional setting. To illustrate their usage, let's first examine the following function written in standard OCaml.

```
type exp = Int of int | Div of exp * exp

exception Div_by_zero

let rec eval (e : exp) : int = match e with
|Int x -> x
|Div(l, r) ->
  let eval_l = eval l in
    let eval_r = eval r in
      if eval_r = 0
        then raise Div_by_zero
        else eval_l / eval_r

let main e =
  try Printf.printf "%d\n" (eval e) with
  |Div_by_zero -> print_endline "Division by zero. Exiting"
```

In this example we have a simple program which evaluates an expression that can be either a constant or a division. However, if the right hand size of the division evaluates to 0, the interpreter raises an exception, since divisions by zero are undefined. As we mentioned in section 2.1, programs written in functional languages are stylistically very similar to mathematical definitions, and this interpreter is no exception. Indeed, if we were to write an inductive definition for this function's behavior, it would be nearly identical to `eval`'s implementation, barring the exception, since in standard logic functions are assumed to terminate with a result. Naturally, if we wish to specify this function, we will need additional rules and syntax to model exceptions.

Since in most programming languages it is quite common to control the flow of a program using exceptions, virtually every prover (Why3 included) has support for specifying and proving exceptional behavior. In this case, we would like to prove that an exception is thrown whenever there is a sub-expression whose right-hand evaluates to zero. To do so, we could use the following WhyML program.

WhyML

```
function eval_ind (e : exp) : int = match e with
|Int n -> n
|Div e1 e2 -> eval_ind e1 / eval_ind e2
end
```

```
predicate has_zero (e : exp) = match e with
|Int n -> false
|Div e1 e2 -> eval_ind e2 = 0 || has_zero e1 || has_zero e2
end

let eval (exp : exp) : int
raises{Div_by_zero -> has_zero exp} =  (*implementation omitted*)
```

We first define the evaluation of expressions using the logical function `eval_ind`. This function also uses the division operator. In WhyML logical functions, when we attempt to divide by 0, no exceptions are thrown. Instead the result is undefined, meaning it cannot be used constructively in our specifications. We then define a predicate `has_zero` which checks if there are any invalid sub expressions. In the `raises` clause attached to the `eval` function, we state that when a `Div_by_zero` exception is thrown, there was a sub-expression whose right hand side evaluated to zero. Since our focus for the time being is in capturing exceptional behavior we will not add any additional specification clauses.

As we stated in chapter 1, however, our goal is to prove programs with effects, meaning they may restart at the point the effect was performed. This adds a much higher burden of proof: not only must we prove the conditions in which the effect is performed, we must also specify the conditions in which control is returned to the function. Before discussing strategies for proving these, let us examine the following implementation of the division interpreter using effects. Important to note that OCaml does not have a dedicated syntax for effect handlers as of yet, they are simply exposed with functions from the novel `Effect` module. Nevertheless, in order to increase clarity, we have created our own syntax which we will formalize in section 5.1.2. This syntax is inspired by what was proposed in an experimental branch of Multicore OCaml [2].

OCaml

```
effect Div_by_zero : int

let rec eval (e : exp) : int = match e with
|Int x -> x
|Div(l, r) ->
  let eval_l = eval l
    let eval_r = eval r in
      if eval_r = 0
        then perform Div_by_zero
        else eval_l / eval_r

let main e =
  try Printf.printf "%d\n" (eval e) with
  |effect Div_by_zero k -> continue k max_int
```

The only difference between this and our previous approach is that `Div_by_zero` is now an effect, not an exception, and it has type `int`. This means when this effect is performed it will generate a suspended continuation, which receives as argument an integer: the value the caller wants the undefined division to hold.

To expose this continuation, we use a `try-with` expression, thereby installing a handler that will catch any effects performed by the call to `eval`. Whenever an effect is performed, this handler will have access to the suspended continuation by means of the variable `k`. It will then call `k` and pass it as argument `max_int`, an OCaml constant that defines the largest `int` the runtime environment can represent. This call will resume the call to `eval` at the point the effect was performed. We cannot call `k` directly, we must use the `continue` function, which takes a continuation and its argument, due to certain implementation decisions that we will explain in section 2.5. When this continuation is called, the function returns to where the effect was performed and resumes execution, replacing the `perform Div_by_zero` expression with `max_int`. We could also call the `discontinue` function which receives as argument the continuation. This function simply discards the continuation and raises a special exception.

2.4.1 Deep and Shallow handlers

If we call a continuation, what happens if it performs another effect? Does the handler keep catching them or will we need a new one? This will depend if the handler is *shallow* or *deep*. Shallow handlers only catch one effect: the continuations they generate will perform more effects. Deep handlers however, will remain installed and continue handling any other effects that are performed.

Deep handlers are more commonly used then their shallow counterparts. They are also easier to reason over since, if a deep handler catches every effect a function produces, its continuations will not produce effects. This is important since our encoding for continuations makes it difficult to represent effectful behavior. We will go over this in more detail in Chapter 6. Because of this, they will be the main focus of this thesis. Therefore, any instance when we show implementations using handlers, the reader can assume the handler is deep, unless explicitly stated otherwise.

2.4.2 Joining Streams

Although the example in section 2.4 is illustrative of how effects work, it confines them to the niche of resumable exceptions. In truth, effects are extremely versatile: in a general sense, we can think of effects as abstract operations and effect handlers as their concrete implementation [24]. One of the most powerful ways one can use handlers is by storing the continuation for later use instead of consuming it immediately, which can lead to new possibilities for asynchronous programming. For example, let's say we are working with a finite, ascending stream of integers, exposed by a function of the following type:

```
type int_stream = (int -> unit) -> unit
```

This stream takes a single argument: a lambda with one argument of type `int`, which represents the current element in the stream, and returns `unit`, an OCaml type that is inhabited by the element `()` which represents a void value. The stream also returns `unit`. When values of type `int_stream` are called, they will produce an `int` value, apply the lambda by passing it as an argument, and repeat until the stream has been exhausted. Since this function effectively has no return value, any non-trivial application of a stream would use a lambda that writes to some mutable data structure, such as an array.

Our goal is to implement a function which takes a list of `int_stream` and combine all their values into a single sorted array. This is possible without effects: we simply call each stream, add each element into an array

```
let res_arr : int array = ...
let join (l : int_stream list) =
    let i = ref 0 in
    List.iter (fun () -> l (fun x -> res_arr.(!i) <- x; i:=!i+1)) l;
    Array.sort res_arr compare
```

However, there is a more efficient alternative: since we know that streams are already sorted, we could suspend a stream using an effect when we know there is another stream whose next value is smaller than the one the current stream has just produced. An implementation of this function is presented in figure 2.1 (this example is original)

In lines 1-2, we define the `int_stream` type, which is the same as our previous definition and the `Yeild` effect of type `int -> unit`, meaning that in order to perform it, we must give it an argument of type `int`. Additionally, the continuation exposed by the handler will receive as argument a value of type `unit`

OCaml
```
type int_stream = (int -> unit) -> unit
effect Yield : int -> unit
```

We then declare the join function which takes a list of streams and returns an array of integers. On lines 5-8 we define four state variables: `res_arr` `stop_at`, `i` and `n_end`.

OCaml
```
  let res_arr : int array = Array.init 0 10000
  let stop_at = ref min_int in
  let i = ref 0 in
  let n_end = ref 0 in
```

The `res_arr` variable is the array we will return at the end of the function with all the values from each stream. This array is initialized with an initial capacity of 10000. To keep the code relatively simple, we do not resize the array when it has reached full capacity.

The next variable is used by the stream which is currently iterating: if the value saved in `stop_at` is smaller then the value we are currently iterating, then that means there is a

OCaml

```
1  type int_stream = (int -> unit) -> unit
2  effect Yield : int -> unit
3
4  let join (l : int_stream list) : int array =
5    let res_arr : int array = Array.init 0 10000
6    let stop_at = ref min_int in
7    let i = ref 0 in
8    let n_end = ref 0 in
9
10   let add (elt : int) =
11     if elt > !stop_at then perform (Yield elt) else ();
12     res_arr.(!i) <- elt;
13     i := !i + 1 in
14
15   let block_stream stream =
16     let rec curr = ref (fun () ->
17       try stream add;
18         n_end := !n_end + 1;
19         max_int
20       with effect (Yield x) k -> (curr := continue k; x)) in
21     fun () ->  !curr () in
22
23   let map_fun = fun stream ->
24     let bs = block_stream stream in bs, bs () in
25   let it_list = List.map map_fun l in
26
27   let it_arr = Array.of_list it_list in
28   let sort = (fun (_, n1) (_, n2) -> compare n1 n2) in
29
30   while !n_end <> Array.length it_arr do
31     Array.sort sort it_arr;
32     let next_it, _ = it_arr.(0) in
33     stop_at := (let _, n = it_arr.(1) in n);
34     let block_value = next_it () in
35     it_arr.(0) <- next_it, block_value;
36   done;
37   res_arr
```

Figure 2.1: Stream join.

stream whose next value will be smaller then the one we are currently iterating, meaning we should yield and surrender control to that stream. This reference is initialized with `min_int`, an OCaml constant that represents the smallest possible integer.

The variable `i` will hold the current index in the array and the `n_end` variable stores how many streams have finished processing.

We then define the ancillary `add` function, defined within the scope of `join`. This will be the function our stream will execute at each element: it first checks if it should yield, sets the current index of the array to the current element and increments `i`.

OCaml
```
let add (elt : int) =
  if elt > !stop_at
    then perform (Yield elt) (*yields control to another stream*)
    else ();
  res_arr.(!i) <- elt;
  i := !i + 1 in
```

We then define the `block_stream` function, which will transform a normal `int_stream` into a stream that will block when necessary and returns the value it stopped at. We do this using `curr`, a reference to a function which will hold the current step in the iteration, defined within the scope of `block_stream`. This reference holds a function that calls the stream by passing it the add function defined previously. Once completed, the `n_end` counter is incremented and we return `max_int`. If at any point a value is yielded, we set `curr` to be the suspended continuation and return the yielded element. The implication being that `curr` will be a function that updates itself depending on the current state of the iteration. The return of `block_stream` will simply be a lambda which calls `curr`.

OCaml
```
let block_stream stream =
  let rec curr = ref (fun () ->
    try
      stream add; (* start stream iteration *)
      n_end := !n_end + 1;
      max_int with
    |effect (Yield x) k -> (curr := continue k; x) ) in
                          (* update curr, return yeilded value *)

  fun () ->  !curr () in
```

In lines 23-28 we create an array where each element will be a tuple comprised of a blocking stream and the first element that blocking stream will handle. We do this by applying a map over the list of streams. To get the first element in the stream, we simply call it; since `stop_at` was set to `min_int`, the stream will surrender control immediately and return the current value.

OCaml
```
  let map_fun = fun stream ->
    let bs = block_stream stream in bs, bs () in
  let it_list = List.map map_fun l in
```

We then turn the resulting list into an array and define a `sort` function to order the array by the next element in each stream. This function takes two arguments, both tuples, the first element of both is ignored and we simply compare the second (the `compare` function used is part of the OCaml standard library).

OCaml
```
let it_arr = Array.of_list it_list in
let sort = (fun (_, n1) (_, n2) -> compare n1 n2) in
```

Finally, we have the loop in which we call fill the `res_arr` by the appropriate blocked stream.

OCaml
```
while !n_end <> Array.length it_arr do
  Array.sort sort it_arr;
  let next_it, _ = it_arr.(0) in
  stop_at := (let _, n = it_arr.(1) in n);
  let block_value = next_it () in
  it_arr.(0) <- next_it, block_value;
done;
```

At each step of the loop we do the following:

- check if all streams have completed
- sort the array using the `sort` function
- get the stream whose current element is the smallest (the one at position 0)
- set the `stop_at` variable to the current element of the stream at position 1
- call the stream and change the element at position 0 with the value the stream blocked at, or `max_int` if the stream is complete; this way, after the array is sorted, completed streams remain at the back of the array. (for simplicity's sake, we assume that streams never contain values equal to `max_int`)

Once this algorithm terminates, our result array is returned having every element from each stream as well as begin sorted. One might wonder however, if the overhead necessary to stop and re execute these streams might cancel out any performance gains we achieve by not having to sort the array after adding all the elements. To test which is faster, we created three streams with one million elements, each constructed in a way that we are forced to yield at every step of all iterations. After testing the initial solution solution and the effectful solution with these three streams, we concluded that the effectful

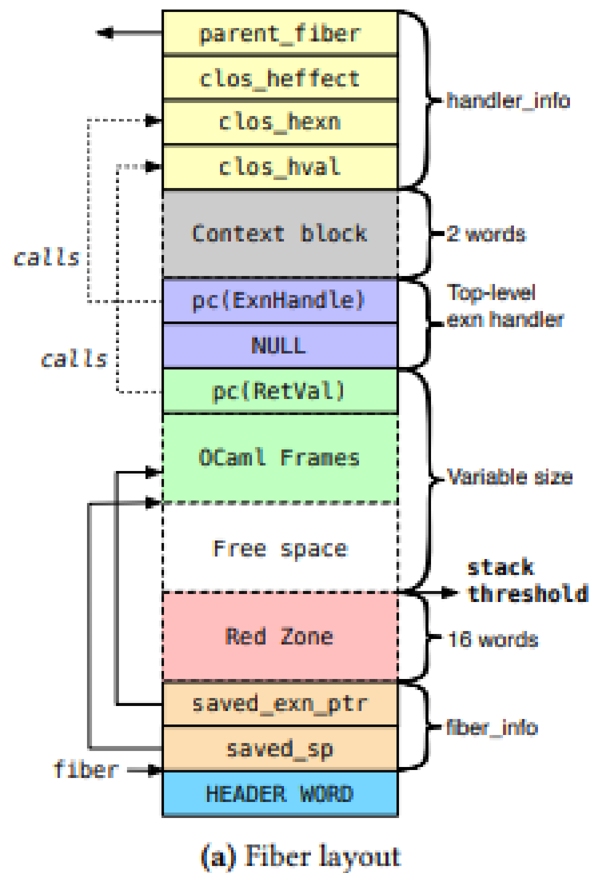
implementation beats the direct style program with almost twice the speed (2.16 seconds vs 5.03 seconds).

This example is a much more realistic, albeit more complicated, use case of effect handlers. Although they can be useful for error recovery, where they truly shine is in asynchronous scheduling. Proving this program would be much more difficult then our previous example: we would have to prove at each step of the loop we are calling the correct stream, model the current state of each stream and prove that the array is sorted. To do this, we will need to reason over the suspended continuation the streams call and then save. One obstacle we have to address is the fact that Why3 only allows using functions as first order values if they do not produce side effects, seeing as these continuations, along with the `add` function, modify the `res_arr` variable, thereby mutating the global state.

2.5 Peeking Under the Curtain : Handler Implementation

Before we further digress into strategies for proving programs with effects, let us journey under the hood and see how the OCaml runtime implements these. The OCaml execution stack is built using *heap fibers* [38], which are made of a list of stack frames and a handler closure, which is itself made of a handler and its environment. Fibers are stored in the C heap and are freed when their execution has completed. This last detail is important: OCaml continuations are *one-shot*, meaning we can only execute them at most once. Therefore, fibers never need to be copied after they finish executing, which allows the runtime to release them from the heap immediately. This makes switching between fibers very fast. When an effect is performed, we pop the top most fibers which in turn become our continuation; when `continue` is called, we push these fibers back on top of the stack.

As we can see in figure 2.2, the base of the fiber is made of a pointer to the parent fiber, the enclosing exception and effect closure (clos_hexn and clos_heffect), which are invoked when the fiber raises an exception or throws an effect. The last frame in the handler_ info section is the clos_hval, which is the closure that will be executed when the fiber finishes its execution naturally and control has been returned to the parent fiber. The next important element in our fiber are OCaml frames: these represent the code that the fiber is currently running; if the size of this region exceeds a certain threshold (stored outside the fiber) then the stack has overflowed and the entire contents of the fiber are copied to a region in memory with twice the size. The red zone is used for optimizing the verification of a stack overflow: if the frame size for a function is smaller then the red zone, then we skip the stack overflow check. At the top, we have the pointer to the closest exception frame and the pointer to the top of the stack. When we switch fibers we must first store the value of these two pointers and load these on to the target stack and resume execution.



(a) Fiber layout

Figure 2.2: An OCaml fiber [38].

2.5.1 Implementing Handlers With Exceptions

Although the OCaml team chose to implement effects by changing the OCaml runtime, there is another strategy: CPS (Continuation Passing Style) and exceptions. CPS is a style of programming where the programmer explicitly controls the flow of a program by having all functions take an extra parameter: a continuation. Instead of returning naturally and delegating control to the program's runtime, we call the continuation and pass as argument the return value of the function. For example, our division interpreter from section 2.4 could be implemented with the following encoding:

OCaml
```
let eval (e : exp) (k : int -> 'a) : 'a =
match e with
|Int x -> k x
|Div l r -> eval l (fun eval_l ->
      eval r (fun eval_r ->
        if eval_r = 0
          then raise (Div_by_zero k)
          else k (eval_l / eval_r)))
```

Although this program would be equivalent to our implementation with effects, applying this kind of transformation to an entire program would render it extremely inefficient. Moreover, since we don't explicitly call functions, we no longer have an explicit call stack, making programs which aim to analyze the contents of the stack mid execution useless. Interestingly, the translation WhyML we present in this thesis also uses exceptions, albeit in a different way (more on this matter in Chapter 5).

2.5.2 Okay, But Why Does This Matter?

There is an argument to be made that the underlying implementation of effect handlers in OCaml is irrelevant for this project, since we are simply trying to model, in an axiomatic context, the behavior of suspended computations. Therefore, how this behavior is implemented in the physical world is irrelevant to our proofs. We disagree, for two basic reasons:

- Understanding how effect handlers are implemented gives us a stronger mental model of their behavior, something that is invaluable when working on program verification.

- It is critical to understand the physical limitations of the programs we are trying to verify. For example, if we were to make a new framework to prove OCaml programs that performed basic arithmetic, and we didn't model the fact that OCaml integers overflow, this hypothetical framework would be useless in verifying programs with arbitrarily large integers. In the case of effect handlers, the only implementation decision we can identify that will make a significant impact in the shape of our proofs is the fact continuations can only be executed once. We will go over this property in more detail in the next chapter.

2.6 Exceptions: What Are They Good For?

As we have seen from the previous examples we have shown, as well as our brief overview of their implementation, performing an effect is syntactically similar to raising an exception while also being a very lightweight operation. This raises the question: what is the use of functions that raise exceptions if we could use an effect and give the caller of the function the option of resuming (using `continue`) or discarding (using `discontinue`) the computation?

Before we answer, we must first understand how exceptions are used in modern programming. There are two common use cases. The first is when we write a function where in some cases there is no logical return value. For example, if we are writing a function that returns the first element of a list, we may raise an exception in case the list is empty

The second use case is when our program has reached an invalid state from which we can not recover and computation must be aborted. For example, if we are implementing a compiler, we might want to throw an exception in case of a syntax error.

Instead of using exceptions to deal with both of these use cases, algebraic effects can be used for the former and exceptions can be used for the latter. Although this might seem like an artificial separation, the choice between effects and exceptions will have important implications in the context of programs with typed algebraic effects. In short, programs that use exceptions are considered to be pure, whereas programs that use effects are considered impure. We will explore this further in the next chapter.

3

State of the Art

In this section, we will dive into some of the work done in regards to the verification of effects and handlers. We will fist go over type systems for programs with effects and the guarantees they can provide. Next, we will look at how we may verify programs using *equational theory*. Finally, we will go over the state of the art in the verification of programs with algebraic effects using Separation Logic

3.1 Typing Programs With Algebraic Effects

The most common type of static verifications programmers use for their programs is type-checking. There are many advantages to using a language with a type system: it can act as a very expressive and straightforward documentation while also statically eliminating a wide class of bugs.

A possible way of extending a type system's reach is by adding an effect system [3]. In normal type systems, we generally have type signatures such as $f : \alpha \to \beta$, where f is a function that receives an argument of type α and returns a value of type β. When using an effect system, type signatures have an additional parameter: a row ρ. A row is simply a list of signatures indicating which effects this function performs. With this typing information, we can determine, at compile time, if a program will perform any unhandled effects, thereby removing yet another common source of errors.

Effect systems can be used to track not only algebraic effects, but any kind of impure behavior such as mutable state and divergence [26]. This means we can imbue within the typing information whether or not a function is pure. Although typing effects is already possible by employing monadic style, effect systems allow us to have our cake and eat it too, letting us write programs in direct style while also having more assurances from the type system. These will be less relevant to our project seeing as Cameleer already has mechanisms for tracking changes in state variables as well as non-terminating functions.

As we mentioned in section 2.6, there is still a place for exceptions in a language with algebraic effects, namely in unrecoverable errors. An unofficial OCaml effect system developed by Jane Street [15] proposed that functions that throw exceptions should be

considered pure and not tracked by the type system. This is mostly so the programmer can still conveniently use `assert false` and other terminating expressions without having to change the entire type signature of their program.

Although effects and handlers have been introduced into an alpha release of Multicore OCaml, there is still no effect system, although there are plans to implement one [2]. Details however, are sparse: we do not know when it will be released or what impure behavior (besides algebraic effects) it will track.

For our tool to be sound in proving OCaml code, it must reject any programs that do not adhere to its type system. This is difficult to do in our case since we don't know how exactly the OCaml type system will evolve. Since we know that the type system will track at least algebraic effects, our tool will also ensure that no effects are unhandled.

3.2 Modelling Algebraic Effects Using Equational Theory

Type systems in ML languages can help catch large swaths of bugs in programs with little to no user written annotations. Regardless, if we wish for a high degree of security that our program does what we wish, there is no better substitute than formal verification.

Although effect handlers are relatively niche, there has been some work in modelling their behavior using equational theory. The vanguard of this approach was Plotkin and Power [34] where they reason over effects as equations of effectful computations. Some research has been done by applying this algebraic approach to effects and handlers. For example, Plotkin and Pretnar [35, 36] developed a logic where the semantics of effects was captured using this approach and handlers were verified by determining if they satisfied a set of equalities. Additionally, some implementations have been developed that embed these reasoning rules into Coq, an interactive theorem prover [28, 42].

The existing body of work that verifies algebraic effects using equational theory has two main setbacks. Firstly, they do not allow for user defined effects. Second, these reasoning rules are embedded into pure languages, meaning they have no support for mutable state and other forms of effectful programming. Additionally, the implementations into Coq fall outside the scope of this thesis seeing as we are focusing on more automated forms of verification.

3.3 Verification of Algebraic Effects with Separation Logic

Instead of equational theory, we will use a more natural form of reasoning: Separation Logic. Although the body of work to draw from is a bit more limited, some research has been done to model the concurrent behavior of effects and handlers into Separation Logic. For instance, Timany and Birkedal [41] developed a framework for the verification of full stack continuations in a language with *call/cc*. Additionally, Hinrichsen, Bengtson, and Krebbers [20] developed an extension of Coq [23] for reasoning over programs with message passing using *protocols* that define the rules that two agents must respect in their

exchange. Drawing inspiration from this research, de Vilhena and Pottier [13] developed a simplified version of protocols for programs with algebraic effects. Their framework interprets effects and effect handlers as a communication between two agents: the one who raised the effect and its handler. To model this communication, they use protocols which describe the values that can be sent in either direction during this communication. The logic that governs these protocols is built on top of Iris, a higher order separation logic encoded in Coq. We will use protocols as described by de Vilhena and Pottier as a foundation for our project.

As is the case of Multicore OCaml, the suspended continuations de Vilhena and Pottier consider are *one-shot*, which means that if a function begins a communication by raising an effect, it can only receive at most one reply. We mentioned in chapter 2 that this decision was made for efficiency's sake. Although this is technically correct, it is incomplete; *one-shot* continuations are also preferable because they are easier to reason about, given that if we allowed continuations to be executed multiple times, we would be permitting functions to be entered once, but exited twice, thereby breaking a fundamental tenant of program reasoning. An example of why such a construct might prove problematic would be the following (taken from [13]):

```
let x = ref 0 in (* initialize x to zero *)
f (); (* f has no access to x, which remains zero *)
x := !x + 1; (* increment x from zero to one *)
assert (!x = 1)
```

In standard OCaml, it is clear that the assertion must succeed, seeing as `x` starts at 0 and is incremented once. Moreover, the call to `f` will never modify `x`, since `x` is defined outside its scope. With *one-shot* effect handlers, this assertion still holds: even if `f` performs an effect, `x` will have the value of 1 when the assertion is reached, seeing as `x` remains at 0 when the computation is resumed . However, in a hypothetical version of OCaml with *multi-shot* continuations, we can no longer make any expectations of what `x`'s value is, since every time we call the continuation, its value is incremented by 1. Since there are no plans to add multi-shot continuations to OCaml in the future, creating proofs for these falls outside the scope of this project.

3.3.1 Language Syntax

The language the authors used in their proofs is a call-by-value λ-calculus with support for mutable references, recursion, and, most importantly, effect handlers. To raise an effect, we use *do* e, which evaluates expression e to a value and performs an effect. Important to note that unlike OCaml's `perform` keyword, which only accepts values of a specific type (effects, which OCaml represents with the internal type `eff`), the *do* keyword accepts any possible value, meaning its effects are unnamed.

To handle effects, we may use $shallow - try$ or $deep - try$, which installs a shallow or deep handler, respectively. Since both are syntactically very similar, we will focus on shallow-try for now: shallow-try e with $\mid h \mid r$, where h and r are both functions, evaluates expression e and, if an effect was performed, calls h by passing it two arguments: the value sent by the effect and the suspended continuation. If e terminated without performing an effect, r is called with the return value of e passed as an argument.

3.3.2 Protocols

We described in chapter 2 how effects work and their implementation in Multicore OCaml. However, if our goal is to deductively verify them, we must map these concepts into a more abstract model. We must then translate this model into standard logical definitions in order to reason over them in an axiomatic fashion.

One of the possible abstract interpretations of effects is as catalysts for a communication between the agent that performs the effect (which we will refer to as *Client*) and the nearest enclosing handler (which we will refer to as *Server*). To describe this, the authors introduce the concept of a protocol step, which is used to describe what requests `Client` will send and the replies that the `Server` will produce. Moreover, these steps can also delegate control of resources and track changes to mutable variables. For a specific example of the type of communications we want to model, let us examine the following example using effects (a similar example was presented in [13]):

```
let xchg  (x : int) : int =
    let old = !p in
    p := x;
    old
```

This is a very simple program which modifies some global reference p by setting it to a value x and returning the old value. The specification for `xchg` would be

$$\{p \rightsquigarrow v\}\ xchg\ x\ \{\lambda r.\ r = v * p \rightsquigarrow x\}$$

Now, let us examine a program where $xchg$ does not write to p directly, but instead delegates this responsibility to the nearest enclosing handler:

```
let xchg (x : int) : int = do x

let main () =
  try xchg x with
  |effect v k ->
    let old = !p in
    p := v;
    k old
```

Note that the handler is irrelevant in the specification of `xchg`, it is merely laid out to give an example of how we would like a hypothetical handler to communicate with the function. We want the pre and postconditions for `xchg` to be the same as before: when it completes, we will have set the value of pointer `p` to `x`. Moreover, the return value of `xchg` must be the old value of p. However, in this program, this behavior is not assured solely by `Client` (`xchg`) but also by `Server` (the nearest enclosing handler). To prove `xchg`'s specification, we must outline rules as to how `Server` and `Client` communicate. Firstly, we must state that, whenever `Client` performs an effect, it must send some integer `x` to `Server`. Moreover, it must allow `Server` to write to variable `p`. Conversely, if `Server` wishes to respond, it must first set `p` to the value `Client` sent. Additionally, the value it returns must be the value `p` pointed to when the effect was performed. Let us assume these set of rules are captured by ψ, a protocol. If `Client` and `Server` both obey ψ, the post condition will naturally be met.

To verify a specification with protocols, we must prove that the values `Client` sends are valid in regards to ψ and that `Client` is prepared to accept any possible reply that `Server` may produce that is also valid under ψ. Finally, it is important to note that a function may exploit its protocol more then once:

```
let xchg_twice (x : int) : int =
    let old = do x in
    let _ = do old in
        old
```

Here, the communication between `Client` and `Server` must have the same behavior we previously outlined, the difference being we perform the effect twice. Therefore, our protocol remains ψ, seeing as the set of rules that a communication must obey has not changed.

3.3.3 Definition of Protocols

By this point, we hope to have conveyed an intuitive grasp on what protocols mean within a specification. We will now outline their formal definition. A protocol has a precondition, which encompass the set of rules `Client` must obey if they wish to perform an effect, and a postcondition, which encompasses the set of rules `Server` must obey if they wish to respond. Finally, in order to give `Client` a wide range of requests to send, and conversely for `Server` to reply, we will also have two binders $\overrightarrow{x}$ and $\overrightarrow{y}$. A protocol step ψ is defined as follows:

$$\psi := !\, \overrightarrow{x} v \, \{P\} . \, ?\, \overrightarrow{y} w \, \{Q\} \tag{3.1}$$

Intuitively, this definition means that `Client` will send v, which may be any of the variables defined in binder $\overrightarrow{x}$, assuming P holds. Similarly, `Server` may reply with w,

which may be any variable defined in binders $\overrightarrow{x}$ or $\overrightarrow{y}$, assuming Q holds. If we wish to define the protocol we used to specify the xchg function, it would be as follows:

$$XCHG \triangleq !x\ x'x\{p \rightsquigarrow x'\}.?\ x'\{p \rightsquigarrow x\} \tag{3.2}$$

The XCHG protocol binds two variables x and x'. These variables are essentially universally quantified, their concrete values will depend on what `Client` will send. The protocol states that `Client` will send x to `Server` and that p points to x'. The assertion $\{p \rightsquigarrow x'\}$ also states that `Server` will have access to pointer p. As for `Server`, it must send x', the old value p pointed to and update p by setting it to x, the value `Client` sent

This definition, however, only encompasses one cycle of request and reply: what if a program exchanges the value stored in pointer p multiple times? Indeed, if we only consider a single exchange between `Client` and `Server`, we would have no means to track such modifications to our program's state. For this reason, protocol steps are interpreted as being repeated any time an effect is performed; this repetition is itself the protocol. By selecting an appropriate protocol step, we can model multiple modifications to mutable variables. To show how one could do this, we will rewrite the `xchg` function to the language defined in section 3.3.1 and specify it. Additionally, we will also specify a function $xchg2$ that calls $xchg$ within the context of a handler and returns the old value of pointer p.

$$xchg\ x \triangleq \lambda p.\ do\ x$$

$$xchg2\ x \triangleq \lambda p.\ deep-try\ in\ xchg\ x\ with\ |\lambda x\ k.\ \lambda old.\ p := \ x;\ k\ old\ !p\ |\lambda\ r.r$$

The specification for $xcgh$ is roughly the same, the only difference is we specify that it obeys protocol $XCHG$

$$\{p \rightsquigarrow v\}\ xchg\ x\ \langle XCHG\rangle\{\lambda r.\ r = v * p \rightsquigarrow x\}$$

With $xchg$ specified, let's move over to $xchg2$: this function calls $xcgh$ within a handler that respects the $XCHG$ protocol, seeing as this handler satisfies the postcondition of protocol $XCHG$. The specification for this function would be

$$\{p \rightsquigarrow v\}\ xchg2\ x\ \langle\bot\rangle\ \{\lambda r.\ r = v * p \rightsquigarrow x\}$$

Important to note that the protocol that $xchg2$ must obey is $\bot$, a special protocol which indicates that the function will never perform an unhandled effect.

3.3.4 Formal Interpretation of Protocols

We have shown an example of a specification which uses a protocol and how we might prove it. However, this was done in a very informal manner, mostly to impart an intuitive understanding of the reasoning rules and definitions surrounding protocols, which we will now formalize. First, let's define what proof obligations a protocol impose on a

program. If a program with the postcondition ϕ uses a protocol ψ and performs the effect $do\ v$ it must prove the predicate ψ $allows\ do\ v\{\phi\}$, whose definition is

$$!\overrightarrow{x}\, v\, \{P\}.\, ?\, \overrightarrow{y}\, w\, \{Q\}\ allows\ do\ v'\ \{\phi\} \dashv\vdash \exists \overrightarrow{x}\ v = v'\ *\ P\ *\ \forall \overrightarrow{y}\ Q \twoheadrightarrow\ \phi w$$

This is the bedrock law regarding the proofs of protocols. When `Client` performs an effect, they must choose the concrete values of $\overrightarrow{x}$ and make a request v', relinquishing control of P. `Server` then chooses how to instantiate $\overrightarrow{y}$, meaning `Client`'s postcondition ϕ must hold regardless of what is sent. As we have previously mentioned, these protocols were made for Iris's logic, which includes a construct known as the *persistence modality* $P\ \square$, which states that assumption P is duplicable. Since this law doesn't have a persistence modality, we conclude that the implication $\forall \overrightarrow{y}\ Q \twoheadrightarrow\ \phi w$ can only be exploited at most once, guaranteeing that our continuations are *one-shot*.

Although not strictly necessary, the authors also included two additional operators to simplify building protocols.

$$\bot\ allows\ do\ v'\ \phi\ \dashv\vdash\ False$$

$$\psi_1 \quad \psi_2\ allows\ do\ v'\ \phi\ \dashv\vdash\ \psi_1\ allows\ do\ v\ \phi\ \vee\ \psi_2\ allows\ do\ v\ \phi$$

The first is the empty protocol $\bot$, which disallows performing effects. Additionally, the + operator gives `Client` the choice between two protocols ψ_1 and ψ_2.

4 Defunctionalization

If we wish to use protocols with Cameleer, we must find a way to encode continuations into Why3. This will be challenging seeing as Why3 does not allow effectful functions to be used as first class values. To get around this limitation, we will use defunctionalization [37], a program transformation technique used to turn a higher order program into a first class equivalent. Although mostly used in compilers, there has been some exploratory work on its usage in proofs involving higher order functions with side effects [40].

4.1 Defunctionalization in a Nutshell

To demonstrate this process, let us first defunctionalize the following simple OCaml program that returns the length of a list, written in CPS.

OCaml

```
let length_cps (l : 'a list) (k : int -> 'a) : 'a =
    match l with
    |[] -> k 0
    |_::tail ->
        length_cps tail (fun tail_length -> k (tail_length + 1))

let length (l : 'a list) : int = length_cps l (fun x -> x)
```

The length_cps function takes two arguments, the list and the continuation. If the list is empty, we call the continuation with the value 0, otherwise we call length_cps passing the tail of the list and a continuation that increments the return value. We also define a length function that calls length_cps with the identity function, thereby returning the length of the list. The first step in defunctionalizing length_cps is to first identify the functions used as higher order values: in this case we have two, the increment function and the identity function. Thus, we will have the following type

OCaml

```
type (_, _) lambda =
    |Ident : ('a, 'a) lambda
    |Inc : (int, 'a) lambda -> (int, 'a) lambda
```

This is a generalized algebraic data type (GADT) with two type parameters, the first representing the function's argument and the second its result. This type has two constructors, the first representing the identity function. Additionally, this constructor has type `('a, 'a) lambda`, meaning it represents a function where the type of the result is the same as the type of the argument. The next constructor represents the lambda we build in `length_cps`, which represents a function that receives an integer and returns some arbitrary type. Unlike `Ident`, this constructor will receive an argument of type `(int, 'a) lambda`, which will represent the free variable `k` in line 5. Now that we are equipped with a first order representation of functions, we may now defunctionalize `length_cps` as such:

```ocaml
1 let length_cps (l : 'a list) (k : lambda int 'a) : int =
2 match l with
3 |[] -> apply k 0
4 |_::tail -> length_cps tail (Inc k)
5
6 let length (l : 'a list) : int = length_cps l Ident
```

This program is equivalent to our previous, except now every first order function has been replaced replaced with our `lambda` type and calls to these use the `apply` function, as we can see in line 3 above, which matches each constructor with its corresponding code, as follows:

```ocaml
let rec apply : type a b. (a, b) lambda -> a -> b =
  fun f arg -> match f with
    | Ident -> let x = arg in x
    | Inc -> let tail_length = arg in
        apply k (tail_length + 1)
```

As was the case in our original program, `KIdent` simply returns its argument and `KInc` returns the application of `k` to the incremented `tail_length`. The full program is shown in figure 4.1.

In short, defunctionalization consists of the following steps: identifying the functions used as first order values, creating constructors for each function using its free variables, transferring the code of these functions into an `apply` function and, finally, replacing every lambda and their call with a constructor of the `lambda` type and the `apply` function, respectively.

4.2 Defunctionalization Coupled with Verification

With defunctionalization briefly covered, let us examine how it can aid us in proving higher order programs. First, we must provide a specification to our non-defunctionalized `length` and `length_cps` functions. For simplicity's sake, we will use GOSPEL instead of

OCaml
```
1  type (_, _) lambda =
2      | Ident : ('a, 'a) lambda
3      | Inc : (int, 'a) lambda -> (int, 'a) lambda
4
5  let rec apply : type a b. (a, b) lambda -> a -> b =
6    fun f arg -> match f with
7      | Ident -> let x = arg in x
8      | Inc k -> let tail_length = arg in
9          apply k (tail_length + 1)
10
11 let rec length_cps (l : 'a list) (k : (int, 'a) lambda) : 'a =
12   match l with
13   |[] -> apply k 0
14   |_::tail -> length_cps tail (Inc k)
15
16 let rec length l =
17   length_cps l Ident
```

Figure 4.1: Fully defunctionalized length.

pure separation logic. The `length` function is simple, we want the result to be equal to the length of the list (we assume that `length_ind` is a logical function that inductively defines the length of a list):

GOSPEL + OCaml
```
let length (l : 'a list) : int = ...
(*@ result = length l
    ensures length_ind l = result *)
```

The post condition for `length_cps` is a bit trickier. Since in CPS we call the function with the result of our computation, we essentially want to prove that the result of our function is equal to the application of the continuation to the length of the list we pass as argument:

GOSPEL + OCaml
```
let length_cps (l : 'a list) (k : int -> int) : int = ...
(*@ result = length_cps l k
    ensures k (length_ind l) = result *)
```

Although this is the most natural translation to the proposed post condition, it is nevertheless unsound: unlike `length_ind`, a logical function, `k` is a function belonging to our program, which means it may produce some impure behavior, such as divergence or modification of mutable variables. In order to reason over it, we will treat `k` as a pair of predicates, its pre and post-conditions, instead of a callable function. To access these, we use the *pre* and *post* predicates, which, for any function f of type $\tau_1 \rightarrow \tau_2$, have the following types:

$$pre\ f \ :\ \tau_1 \rightarrow prop$$
$$post\ f \ :\ \tau_1 \rightarrow \tau_2 \rightarrow prop$$

GOSPEL + OCaml
```
let length_cps (l : 'a list) (k : int -> int) : int =
match l with
|[] -> k 0
|_::tail ->length_cps tail
            (fun (*@ ensures post k (tail_length +1) result*)
                tail_length -> k (tail_length + 1))
(*@ ensures post k (length_ind l) result *)


let length (l : 'a list) : int =
    length_cps l (fun x -> x)
(*@ ensures length_ind l = result *)
```

Figure 4.2: Fully annotated `length`.

Naturally, *pre* holds if the argument we passed satisfies f's precondition and *post* holds if f's post condition is satisfied, assuming the first argument is what pass to f and the second argument is the result. Using this *post* predicate, we can rewrite our previously unsound specification as follows:

GOSPEL + OCaml
```
let length_cps (l : 'a list) (k : int -> int) : int = ...
(*@ ensures post k (length_ind l) result*)
```

We must provide post conditions to the functions we use as first order values. The identity function's specification is trivial, its argument is equal to its result. As for the increment function, its specification must reflect that it calls `k` with the argument it receives plus 1. We present the fully specified GOSPEL annotated program in Figure 4.2. Note that named functions' specifications appear after their definition's end and the anonymous functions' appear in between the `fun` keyword and their argument's name.

To prove this implementation adheres to the given specification, we must defunctionalize this program and convert it into WhyML. Although we will employ a similar strategy as we did before when we defunctionalized `length_cps`, we will use a conversion that doesn't technically use GADTs. This is to avoid having to define `apply` and its specification, meaning we simplify the translation. It is fortunate that this is the simplest option seeing as WhyML doesn't support GADTs. First we create our `lambda` type, the `apply` function and the `post`, all of which will not have an implementation, as follows:

WhyML
```
type lambda 'a 'b
predicate post (lambda 'a 'b) 'a 'b


val apply (f : lambda 'a 'b) (arg : 'a) : 'b
ensures{post f arg result}
```

Now all that is left to convert are the lambdas. To do so, we create a `lambda` value whose `post` predicate matches its GOSPEL specification. This is achieved by creating a function

```
let length_cps l (k : lambda int 'a) : 'a                    WhyML
ensures{post k (length_ind l) result} =
    match l with
    |Nil -> apply k 0
    |Cons _ t ->
      let vc tail_length
        ensures{post k (tail_length + 1) result} =
          apply k (tail_length +1) in

      val f unit : lambda int 'a
        ensures{let f = result in
          forall tail_length result.
              post f tail_length result <->
              post f (tail_length + 1) result} in
      length_cps t (f ())
```

Figure 4.3: Defunctionalized proof of `length_cps`.

that generates a `lambda` with the appropriate `post` predicate instantiation.

```
val f unit : lambda int 'a                                   WhyML
ensures{let f = result in
    forall tail_length result.
        post f tail_length result <->
        post k (tail_length + 1) result} in
```

Important to note that `k` in the post condition is the free variable the lambda calls.

Additionally, we must dispatch verification conditions to prove the body of the function adheres to its specification. To do this, we simply create an auxiliary function `vc` with the same body and specification as the original function. This function is never called an exists only to generate verification conditions. Converting the anonymous increment function from our previous example we would get the following:

```
let vc tail_length                                           WhyML
    ensures{post k (tail_length + 1) result} =
        apply k (tail_length +1)
```

The full translation of `length_cps` is in figure 4.3. All the VCs generated by Why3 for the defunctionalized program are automatically discharged with the Alt-Ergo [22] SMT solver.

4.3 Defunctionalization with State

As mentioned previously, the reason we use the `pre` and `post` predicates is to avoid creating logical inconsistencies by calling impure functions in the pure setting of specifications.

The `length_cps` example, illustrative as it was, contains no side effects and therefore *technically* does not require `post`. To show how we can use this predicate to capture effectful functions, let us look at a simpler example in WhyML:

WhyML

```
val r : ref int

let mk_gen (n : int) = r := 0; fun () -> (r := !r + n; !r)
```

The `mk_gen` function resets the global reference to `0` and returns a function that increments the reference by a fixed value and returns the value the reference holds. Since the returned function modifies a state variable, we will need to extend our *post* predicate to not only encompass the argument and the result, but also the program's state.

$$
\begin{aligned}
pre\ f &: \tau_1 \rightarrow state \rightarrow prop \\
post\ f &: \tau_1 \rightarrow state \rightarrow state \rightarrow \tau_2 \ \rightarrow prop
\end{aligned}
$$

In Why3, we will represent the function's state as a record type that can hold the value of all mutable references. Since we only have one reference to an integer in this program, our record type will hold a single integer.

WhyML

```
type state = {_r : int}
```

The `post` and `apply` will now be as follows

WhyML

```
predicate post (lambda 'a 'b) 'a state state 'b

val apply (f : lambda 'a 'b) (arg : 'a) : 'b
ensures{post arg {_r = old !r} {_r = !r} result}
```

One of the disadvantages of this representation is that all references must be declared in the same scope as the `apply` function, meaning there can be no local state references in our programs. With the necessary modifications in place, we may finally specify the `mk_gen` function as follows:

WhyML

```
val mk_gen (n : int) : (lambda unit int)
ensures{!r = 0}
ensures{let f = result in
    forall old_state state result.
        post f () old_state state result <->
        (old_state._r + n = state._r && result = state._r)}
```

One of the conditions is that the reference `r` is set to `0`. The second states that that each application of this function leads to a state where the `r` reference is equal to its old value plus `n`. Additionally, we also state that the result is equal to the current value of `r`.

5 Reasoning about Effects and Handlers in Why3

Although protocols are a powerful abstraction to reason about effects, they were originally developed for Iris, with a rich Separation Logic embedded in Coq that can handle concurrency and higher order programs. Hence, we translate our OCaml programs into a WhyML embedding that can prove programs annotated with protocols in a mostly-automated fashion.

We present two languages which aim to capture a core fragment of OCaml (as well as GOSPEL) and WhyML. They will be quite similar to one another, since both are ML-style languages featuring type definitions, anonymous functions and mutable references. The main difference is that only the former will have effects and protocols. Although most of our grammar is fairly standard in regards to functional and specification languages, we will briefly discuss some of the more novel aspects, namely protocols and handler specifications.

5.1 GOSPEL Extension

We will first go over the language that aims to approximate GOSPEL annotated OCaml programs. We will focus particularly on our proposed extensions to specify and reason about protocols and effects.

5.1.1 GOSPEL Protocols

In chapter 3, we explained what protocols are on a conceptual level and how we can fit them into Separation Logic. We will now explain our proposed syntax for effects in GOSPEL. Let's imagine a program with some effect `E` that receives and `int` and returns an `int`.

GOSPEL + OCaml

```
effect E : int -> int
```

If we want to specify that:

- A `Client` performing `E` can only send integers greater than zero
- A `Server` to `E` can only send integers smaller then zero
- A `Server` to `E` may modify some pointer `p`

we propose the following protocol, written with our novel GOSPEL extension:

GOSPEL + OCaml
```
(*@ protocol E x :
    requires x > 0
    ensures reply < 0
    modifies p *)
```

The first line of the protocol states that it will refer to the conditions under which effect `E` is performed with argument `x`. The second line is a simple precondition, similar to a standard function, except instead of referring to a condition that must be respected before calling a function, it refers to those that one needs to perform `E`. The next line is a post condition stating that the reply `Server` sends must be smaller then zero. The `reply` variable used in the postcondition is similar to the `result` variable exposed by postconditions in most specification languages. Finally, the last line states that `Server` has write permissions to `p`. Protocols are top-level declarations that can be placed anywhere in the program, after the effect has been defined. This means we cannot reason over local state variable within our protocols. The full grammar for protocol specification clauses is:

$$
\begin{array}{lll}
S_p & := & \text{Protocol Specifications} \\
 & \mid \texttt{requires}\ s & \\
 & \mid \texttt{ensures}\ s & \\
 & \mid \texttt{modifies}\ x & \\
\end{array}
$$

The meta variable s symbolizes GOSPEL terms and x stands for an identifier. The x notation indicates we may have an indefinite amount of identifiers.

5.1.2 Effect Handlers

With this extension of the GOSPEL language tool we will, naturally, provide support for effect handlers. In chapter 2 we have shown some examples of these in real code and now we will provide them with a formal syntax. Effect handlers are, syntactically speaking, very similar to exception handlers; they are `try-with` blocks where every branch corresponds to the instructions to be executed depending on the effect. The only real difference is the fact that handlers also expose a continuation. The syntax for effect handlers is as follows.

$$\texttt{try}\ e\ \texttt{with}\ \overline{\texttt{effect}\ E x\ \texttt{k} \rightarrow e}$$

The meta variable E represents an identifier for an effect.

Interestingly, this syntax is not the same as Multicore OCaml's. In fact, as of writing, OCaml has no syntactic support for effect handlers, only exposing them in the form of

functions from the novel `Effect` module. This is because OCaml doesn't have an effect system, meaning that we cannot determine, at compile time, if all effects are handled. The OCaml developers, therefore, have chosen to only expose handlers syntactically when their effect system arrives. Nevertheless, we are introducing our own syntax (which will likely be very similar to what OCaml will have in the future), since programs written using the current functional approach are very difficult to read. One particular aspect about our handlers is the fact they do not have a branch for a non-exceptional return.

The other major addition to the GOSPEL syntax is handler specifications. At first glance, this addition might seem a little strange: Why do we need to give handlers a specification? Why not simply specify the functions that use them? The reason why this is needed is to be able to give post conditions to the generated continuations. As we previously mentioned, when continuations are called, execution is resumed with the same handler installed. This means when the continuation terminates execution, it will have exited the handler, either via an effect or a normal return. Given this, the postcondition of the continuation will be the same as the handler's postcondition. Therefore, since we can't generate this postcondition automatically, it must be manually inserted by the user. Effectively, what we are actually doing is inserting invariants that must hold every time the handler is exited, whether it be from the function exiting naturally or from an exceptional branch. We will then take these invariants and have the generated continuations use them as postconditions.

Handler specifications have two clauses: `try_ensures`, which encapsulates a post condition of the handler, and a `returns` clause, where we will annotate the return type of this handler. One might wonder why this last clause is necessary as OCaml has automatic type inference. Although OCaml can technically infer the type of these expressions, this would require using the OCaml typed parse tree. For the moment this is out of scope for tools such as GOSPEL and Cameleer, as the typed parse tree implementation in the OCaml compiler is not stable. With the lack of external tools such as ppxlib, the only solution would be to rely on a specific version of the OCaml compiler, which would result in several maintainability issues.

The full grammar for handler specifications is:

S_h	:=		Handler Specifications
	|	`try_ensures` s	
	|	`returns` τ	

5.1.3 Performs Clause

Finally, the last addition to the GOSPEL syntax is the `performs` clause in function specifications. This is simply used to list all the effects this function is allowed to perform. The full grammar for function specifications is:

$$
\begin{array}{lll}
S & := & \text{Specifications} \\
 & \mid Sp & \\
 & \mid \texttt{performs}\ E &
\end{array}
$$

This information isn't strictly necessary for our proofs, seeing as we can statically check which effects, if any, a function performs. Nonetheless, we find it useful to force the user to explicitly annotate this information for the sake of clearer documentation. We will then use this information to check if handlers catch every effect, thereby compensating for the lack of an effect system.

5.1.4 The Remaining Grammar

The remaining of our syntax is quite standard, but we would like to give a few brief overview on some particular aspects on the complete grammar, which can be found in figure 5.1:

- The x metavariable encompasses all valid variable identifiers
- The p metavariable encompasses all valid patterns
- The E metavariable encompasses all valid effect identifiers
- The n metavariable encompasses all valid numbers
- The s metavariable encompasses all valid GOSPEL terms. We do not define a proper grammar for these seeing as it would be mostly identical to a standard first order logic.
- Curried functions are not allowed.
- The only expressions we allow as arguments to `perform` are effect identifiers coupled with their arguments. This means we do not accept more complex expressions such as, for example `perform (if b then E1 else E2)`. We do not believe that this impacts our tool's expressiveness, seeing as it is rare to use effects like this.
- As explained in chapter 2, the only state variables we allow are top level references.
- All functions and state variables must have their types explicitly annotated.

5.2 WhyML Formalization

We will now move on to formalize our target language, which approximates a core fragment of WhyML. There is nothing specially of note regarding the grammar in figure 5.2 since it is rather similar to OCaml's syntax. The only constructs missing are effect handlers, which have been replaced with exception handlers; the `performs` clause, which

e	$::=$	Expressions
	$\mid a$	
	$\mid \texttt{let}\ x = e\ \texttt{in}\ e$	
	$\mid a \oplus a$	
	$\mid !x$	$\mid e\ ;\ e$
	$\mid \texttt{if}\ a\ \texttt{then}\ e\ \texttt{else}\ e$	
	$\mid \texttt{fun}\ \overline{S_f}\ \overline{x\ :\ \tau}\ :\tau \rightarrow e$	
	$\mid \texttt{match}\ a\ \texttt{with}\ \overline{p \rightarrow e}$	
	$\mid \texttt{try}\ e\ \texttt{with}\ S_h\ \overline{\texttt{effect}\ Ex\ \texttt{k} \rightarrow e}$	
	$\mid \texttt{perform}\ Ex$	
a	$:=$	Values
	$\mid x$	
	$\mid n$	
	$\mid$	
	$\mid \texttt{true}$	
	$\mid \texttt{false}$	
Sp	$:=$	Protocol Specifications
	$\mid \texttt{requires}\ s$	
	$\mid \texttt{ensures}\ s$	
	$\mid \texttt{modifies}\ x$	
Sf	$:= S_p$	Anonymous Function Specifications
S	$:=$	Specifications
	$\mid Sp$	
	$\mid \texttt{performs}\ E$	
Sh	$:=$	Handler Specifications
	$\mid \texttt{try_ensures}\ s$	
	$\mid \texttt{returns}\ \tau$	
Ψ	$:= \texttt{protocol}\ Ex\ :\ Sp$	Protocols
G	$:=$	Gospel Declarations
	$\mid \Psi$	
	$\mid \texttt{function}\ \overline{x\ :\ \tau}\ :\ \tau = s?$	
	$\mid \texttt{predicate}\ \overline{x\ :\ \tau} = s?$	
d	$:=$	Declarations
	$\mid \texttt{effect}\ E\ :\ \tau$	
	$\mid \texttt{let}\ ?\texttt{rec}\ \overline{S}\ f\ :\ \tau = e$	
	$\mid \texttt{let v} : \tau\ \texttt{ref} = \texttt{ref e}$	
	$\mid \texttt{type}\ T = t$	
	$\mid G$	
t	$:=$	Types
	$\mid \tau$	
	$\mid \{\overline{x : \tau}\}$	
	$\mid \overline{D\overline{\tau}}$	
pr	$:= \overline{d}$	Program

Figure 5.1: Extended GOSPEL Syntax.

e	$::=$	Expressions
	$\mid a$	
	$\mid \texttt{let}\ x = e\ \texttt{in}\ e$	
	$\mid a \oplus a$	
	$\mid\ !x$	$\mid e\ ;\ e$
	$\mid \texttt{if}\ a\ \texttt{then}\ e\ \texttt{else}\ e$	
	$\mid \texttt{fun}\ \overline{S_f}\ \overline{x\ :\ \tau}\ :\tau \rightarrow e$	
	$\mid \texttt{match}\ a\ \texttt{with}\ \overline{p \rightarrow e}\ \texttt{end}$	
	$\mid \texttt{try}\ e\ \texttt{with}\ \overline{Ex\ \texttt{k} \rightarrow e}\ \texttt{end}$	
	$\mid \texttt{perform}\ Ex$	
a	$:=$	Values
	$\mid x$	
	$\mid n$	
	$\mid$	
	$\mid \texttt{true}$	
	$\mid \texttt{false}$	
S	$:=$	Specifications
	$\mid \texttt{requires}\ \{s\}$	
	$\mid \texttt{ensures}\ \{s\}$	
	$\mid \texttt{raises}\ \texttt{E} \rightarrow \texttt{s}?$	
	$\mid \texttt{modifies}\ \{x\}$	
d	$:=$	Declarations
	$\mid \texttt{effect}\ E\ :\ \tau$	
	$\mid \texttt{let}\ ?\texttt{rec}\ \overline{S}\ f\ :\ \tau = e$	
	$\mid \texttt{let v} : \tau\ \texttt{ref} = \texttt{ref e}$	
	$\mid \texttt{type}\ T = t$	
	$\mid \texttt{function}\ \overline{x\ :\ \tau}\ :\ \tau = s?$	
	$\mid \texttt{predicate}\ \overline{x\ :\ \tau} = s?$	
t	$:=$	Types
	$\mid \tau$	
	$\mid \{\overline{x\ :\ \tau}\}$	
	$\mid \overline{D\overline{\tau}}$	
pr	$:= \overline{d}$	Program

Figure 5.2: WhyML Syntax.

has also been replaced, now by the `raises` clause. Additionally, this language doesn't have protocols.

5.3 Proving an OCaml program with protocols

Before we move on to our general translation scheme, we will give a bird's eye view of the techniques we will use to verify handlers. To illustrate, we will return to the $xchg$ example in Chapter 3. Defining this protocol using our new GOSPEL syntax yields:

GOSPEL + OCaml

```
effect XCHG : int -> int
let p : int ref = ref 0

(*@ protocol XCHG x :
    ensures reply = old !p && !p = x
    modifies p *)
```

First, we define an effect `XCHG` which receives an `int` and returns an `int`. Additionally, we also define `p`, an integer reference. Next, we define a GOSPEL protocol which ensures that, when `Server` returns control to `Client`, the reference will hold the value passed and `Server` will return `p`'s old value. Additionally, we also state that `Server` may modify `p`. Let us now move on to specifying a `Client` that uses this effect.

GOSPEL + OCaml

```
let some_fun (n : int) : int = perform (XCHG n)
(*@ ensures !p = n && result = old !p
    performs XCHG *)
```

When we covered protocols in chapter 3, we explained that programs that use protocols must prove that each request `Client` makes adheres to the protocol's precondition and that `Client`'s postcondition holds for any reply `Server` sends. When translating this program into WhyML, we must do so in a way that it generates verification conditions that prove this. We will start by creating a predicate that encapsulates the protocol's postcondition as well as a function that will mimic performing an XCHG effect. This function will have the same specification as the protocol and have no implementation.

WhyML

```
type state = {_p : int}
let p : ref int = ref 0

(*Predicate that encapsulates the protocol's postcondition*)
predicate post_XCHG (arg : int) (old_state : state)
                    (state : state) (reply : int) =
  let x = arg in reply = old_state._p && state._p = x

(*Special perform function*)
val perform_XCHG (x : int) : int
ensures{post_XCHG x {_p = old !p} {_p = !p} result}
writes{p}
```

Now that we can mimic calls to `perform`, we can translate our client into WhyML, as follows.

WhyML

```
let some_fun (n : int) : int = perform_XCHG n
ensures{!p = n && result = old !p}
```

By using the perform_XCHG function, we can check if the request made satisfies the protocol's precondition and if the function's postcondition holds for any possible reply. The former is proved trivially: seeing as there is no precondition, Client can always make a request. As for the latter, seeing as Server can only reply after setting the p pointer to the sent value and can only return the pointer's old value, the body of some_fun can be proven to adhere to its specification.

Using this embedding, proving a Client is no different than proving functions that call other functions. We will now move on to the difficult part of the verification process: proving each Server reply is correct. Let's assume a fairly standard OCaml handler that correctly follows the proposed protocol:

GOSPEL + OCaml
```
try some_fun 3 with
|effect (XCHG n) k ->
  let old_p = !p in
  p := n;
  continue k old_p
```

With our handler defined we will now give it an invariant. As mentioned in section 5.1.2, the invariant must hold whether the function exits through an exceptional branch or the function ends normally. In the latter case, we can state that the function's postcondition holds. If we reach an exceptional branch, we know that the final instruction was a call to continue. Since this is a deep handler, this handler will remain installed, meaning that if any other effects are performed, continue will keep being called until some_fun terminates. Since continue terminates when sum_fun terminates, we can infer that it's postcondition holds at the end of the effect branch as well. Therefore, the specification for this handler would be:

GOSPEL + OCaml
```
try some_fun 3 with ...
(*@ try_ensures !p = 3 && result = old !p
    returns int *)
```

Although this handler is quite trivial, proving it will require some work, as we have many problems to solve:

- How are we going to translate OCaml's effects into WhyML?
- The Why3 some_fun function we defined doesn't have any kind of exceptional return, it merely calls a function that simulates the handler's behavior. How do we embed this in Why3?
- How do we represent continuations (as well as OCaml's continue function) in Why3?
- What kind of specification should these continuations have?

5.3.1 Translating effects into WhyML

Let us begin by tackling the first problem: in order to represent these effects, we are simply going to use exceptions. More specifically, we create an exception that receives the same arguments as the original effect. In the case of `XCHG`, it receives a single `int`, which means the translated exception will also receive an int:

WhyML
```
exception XCHG int
```

Its return type is irrelevant for the time being. Now that we have some way of representing effects, we can now get to work translating handlers into WhyML using exception handlers. The previous OCaml handler is translated as follows:

WhyML
```
try some_fun 3 with
|XCHG n ->
  let old_p = !p in
  p := n;
  continue k old_p
```

Although we have created a WhyML program that is very similar to our OCaml counterpart, the `k` continuation is undefined, seeing as this is a normal exception handler. But before we get to that, there is another issue: the Why3 `sum_fun` function we defined doesn't throw any exception. To fix this, we simply add the following `raises` clause to the `perform_XCHG` function.

WhyML
```
raises{XCHG}
```

We also replace the `performs` clause of `sum_fun` with this `raises` clause. Note that the original GOSPEL program remains unchanged; all we are doing is changing how we verify the specification using Why3.

5.3.2 Representing continuations in WhyML

Now that we have effectively translated the performs clause into WhyML, we can now get to work on the trickiest part of the proof process: the continuations. The first step is to work out how to represent and call them. Seeing as continuations are essentially just functions, we might be tempted to simply use the the `lambda` type we defined when we introduced defunctionalization in chapter 3. However, continuations have a special property that regular lambdas don't: their *one-shot* check. In order to enforce this, we will introduce the following Why3 type, coupled with a `continue` function:

WhyML
```
type continuation 'a 'b = {
  _k : lambda 'a 'b;
  mutable _valid : bool
}
```

```
meta coercion function _k

type state = {_p : int}

val contin (k : continuation 'a 'b) (arg : 'a) : 'b
  requires{k._valid}
  requires{pre k arg {_p = !p}}
  ensures{post k arg {_p = old !p} {_p = !p} result}
  writes{k._valid, p}
```

Our Why3 continuations are made up of a lambda and a mutable boolean value to determine whether the function has been called or not. Previously, we stated that the only mutable values we allowed are top-level references; these *one-shot* checks are the only exception. The next line is rather cryptic, but essentially it states that if Why3 find a value of type `continuation` when it expected a value of type `lambda`, it will use the continuation's `_k` field instead. This replacement only applies in pure settings such as specifications and is only done to simplify the code.

The next thing we do is define our state. Since our program only has one mutable reference to an `int`, the state will be a record type consisting of a single `int`. Finally, all that's left is to define our `continue` function. This function in Why3 will have to be named `contin` seeing as `continue` is a keyword in Why3. This function will be very similar to our `apply` function from chapter 2, the only difference being that it has as a precondition that the continuation must be valid i.e. it hasn't been called yet. Moreover, it invalidates the continuation. The `pre` and `post` predicates aren't defined, however they have the same meaning as the ones we defined in Chapter 4.

5.3.3 Specifying continuations

With all these different pieces, we can now conclude our WhyML proof by creating a value `k` of type continuation, similarly to how we did for functions in Chapter 4:

WhyML

```
try some_fun 3 with
|XCHG n ->
  val k : continuation int int in
  let old_p = !p in
  p := n;
  contin k old_p
```

We must now give `gen_k` some specification. Naturally, the first thing we must say is that the continuation that `gen_k` creates is valid. To do this, we will assume the following statement.

WhyML

```
assume{k._valid}
```

Next we must define the continuation's precondition. Since when we call the continuation we are surrendering control back to `Client`, the protocol's postcondition must hold. This is done as follows:

WhyML
```
|XCHG n ->
  let old_state = {_p = !p}
  val k : ...
  assume{k._valid}
  assume{
    forall reply state.
      pre k reply state <-> post_XCHG n old_state state reply
  }
```

Before we define the continuation, we create a variable `old_state` that will represent the state at the time the effect was performed. Now we focus on the continuation's postcondition. Since this is a deep handler, the handler remains installed to respond to any further effects. This means this continuation will have the same postcondition as the handler. The handler's postcondition was:

GOSPEL + OCaml
```
try_ensures !p = 3 && result = old !p
```

Where `old !p` refers to p's state before the handler expression. Therefore, the full WhyML handler is the following:

WhyML
```
let init_state = {_p = !p} in
try some_fun 3 with
|XCHG n ->
  ...
  val k : ...
  assume{
    forall arg irrelevant_old_state state result.
      post arg irrelevant_old_state state result <->
      let old_state = init_state in
      state._p = 3 && result = old_state._p}
```

In order to access the state before the handler, we create an `init_state` variable which we then use when defining the continuation's post condition using the `post` predicate. As we mentioned, the `post` predicate receives as argument the state before and after the function is called, but in this instance we ignore the old state. This is because the state in which the continuation is called is only relevant to know if it is allowed to surrender control back to `Client`, which we check in the precondition. In the case of the continuation's postcondition, we only care about the state before the handler.

With the continuation fully specified all that is left is to determine if the handler in fact respects the postcondition. To do this, we simply wrap the WhyML exception handler

GOSPEL + OCaml

```
effect XCHG : int -> int
let p = ref 1

let some_fun (n : int) : int = perform (XCHG n)
(*@ performs XCHG
    ensures !p = x && result = old !x *)

let n =
  try some_fun 3 with
  |effect (XCHG n) k ->
    let old_p = !p in
    p := n;
    continue k old_p
  (*@ try_ensures !p = 3 && result = old !x *)
```

Figure 5.3: Full OCaml program and GOSPEL specification.

in a function with the original GOSPEL postcondition and then immediately call it.

WhyML

```
let handler () : int =
ensures{!p = 3 && result = old !p}
... in handler ()
```

This way, Why3 will check if the body of the handler function respects the postcondition. The WhyML `Client` and handler, along with the original Cameleer program can be found in figures 5.4 and 5.3[1].

5.3.4 Verification of the Program

We will now show what verification conditions Why3 generates in order to prove the correctness of the program. Beginning with `some_fun`, Why3 generates one VC, proving if `some_fun`'s postcondition is valid. Seeing as its postcondition is equivalent to the `perform_XCHG`'s, Why3 easily discharges it.

Finally, we analyze the four VCs generated for the handler. The first one proves if `some_fun` ending non-exceptionally satisfies the handler's invariant. Once again, seeing as the handler's invariant is the same as `some_fun`'s, Why3 proves it as well without an issue. Next, seeing as we call `contin`, Why3 will try to prove its two preconditions: that the continuation is valid and that the protocol's precondition is met. The first one is trivial, seeing as the generated continuation is marked as valid and is never invalidated until `contin` is called. As for the second precondition, since we changed `p` to the value passed in the effect and call the continuation with `p`'s old value, this condition is proved as well. As for the fourth and final condition, it checks if the handler's invariant is valid at the

[1]Although this encoding works for this example, we would run into problems with polymorphic continuations. To remedy this, our actual WhyML translation is slightly more verbose but functionally identical

WhyML

```
exception XCHG int

val perform_XCHG (x : int) : int
ensures{ let reply = result in
          reply = old !p && !p = x}
raises{XCHG}
writes{p}

let some_fun (x : int) : int =
ensures{!p = x && result = old !p}
raises{XCHG}
perform_XCHG x

let n =
  let handler () =
    let init_state = {_p = !p} in
    try some_fun 3 with
    |XCHG n ->
      let old_state = {_p = !p} in
      val k: continuation int int
      ensures{result._valid}
      assume{
        forall reply state.
          pre k reply state <-> reply = old_state._p && state._p = n
      }
      assume{
        forall arg irrelevant_old_state state result.
          post k arg irrelevant_old_state state result <->
          let old_state = init_state in
          state._p = 3 && result = old_state._p} in
      let old_p = !p in
      p := n;
      contin k old_p end in
  handler ()
```

Figure 5.4: WhyML proof.

end of the exceptional branch. Since our last instruction is a call to `contin`, whose post condition is the handler's invariant, Why3 can prove this as well.

What Does This Mean? With all the generated verification conditions met, it is time to step back and reflect on what we can infer from them. In the case of `some_fun`, we can conclude that all requests made are valid and all possible replies satisfy the function's postcondition.

As for the handler, we can prove that all replies are valid and that the continuations are never exploited more then once. Additionally, we can prove that the handler's invariant is respected regardless of whether the handler is exited through an exceptional branch or terminates normally. This proof rests on an underlying assumption that the generated continuations' postcondition is the handler's invariant.

Proving The Continuation's Postcondition To prove that, for any arbitrary handler, its generated continuations' postcondition will be the handler's invariant, we will consider the following OCaml effect handler.

GOSPEL + OCaml

```
try f () with
|effect E1 k -> ...
|effect E2 k -> ...
...
(*@ try_ensures I)
```

As stated in subsection 5.3.4, one of the generated verification conditions is that expression `f ()` satisfies the invariant I. Additionally, we also prove that each of the effect cases satisfies I, assuming a continuation that also respects I. Knowing this, if an effect is performed, calling the generated continuation will have one of two outcomes.

- `Client` resumes and terminates, performing zero effects, leading to the completion of the expression `f ()`. Seeing as this expression must satisfy I, calling the continuation respects the invariant.
- `Client` resumes execution, but will perform n more effects before completion, where $n > 0$. This means the continuation will enter one of the effect cases, generating a new continuation. Seeing as each effect case must respect I, assuming the generated continuation also respects I, calling the continuation also respects I if the generated continuation respects I. To prove this, we can use simple induction, seeing as the generated continuation will resume a `Client` who will perform $n - 1$ effects.

5.4 General Translation Scheme

Although our translation of Multicore OCaml to WhyML is admittedly complex, the underlying ideas are simple:

WhyML

```
type state = {
  _S_x1 : S_τ1;
  _S_x2 : S_τ2;
  ...
  _S_xn : S_τn
}

let S_x1 : ref S_τ1 = ...
let S_x2 : ref S_τ2 = ...
...
let S_xn : ref S_τn = ...
```

Figure 5.5: State Definition.

- Turn effect declarations into exceptions.
- Create special `perform` functions for each effect declaration with the same specification as the corresponding protocol.
- Replace all instances of `perform` with the correct functional versions.
- As for the handlers, create special continuation values with the same precondition as the protocol's postcondition and the same postcondition as the handler's invariant.
- Finally, replace all calls to `continue` with `contin`

Along with these five basic steps, we also defunctionalize all higher order functions using the same process detailed in chapter 2. We will now formalize these ideas into a concrete translation scheme. In Figure 5.5 we have the generalization of the `state` type. In a program with the reference variables S_{x1}; S_{x2} ..., with types `ref` $S_{\tau 1}$; `ref` $S_{\tau 2}$ the state type will be the record type {_S_{x1} : $S_{\tau 1}$; _S_{x2} : $S_{\tau 2}$;...} Note how each field name is equal to a state variable's name with an underscore in the start, this is because WhyML doesn't allow top level variables with the same name as record fields.

Next, we have all the top level definitions we will need to make our transformations valid in figure 5.6. We have already gone over all these definitions in previous sections, so we won't spend too much time on them. The only novel aspect is the S_x symbol, which is just a short hand for the following expression:

WhyML

```
{
  _S_x1 = !S_x1;
  _S_x2 = !S_x2;
  ...
  _S_xn = !S_xn
}
```

WhyML

```
type lambda 'arg 'result

type continuation 'arg 'result = {
   ghost f : lambda 'arg 'result;
   ghost mutable valid : bool
}

predicate pre (lambda 'a 'b) 'a state

predicate post (lambda 'a 'b) 'a state state 'b

val apply (f : lambda 'a 'b) (arg : 'a) : 'b
requires{pre f arg S_x}
ensures{post f arg (old S_x) S_x result}

val contin (k : continuation 'a 'b) (arg : 'a) : 'b
requires{pre k.f arg S_x}
requires{k.valid}
ensures{post k.f arg (old S_x) S_x result}
ensures{not k.valid}
writes{k.valid}
```

Figure 5.6: Top Level Declarations.

Finally, we present the concrete rules to translate OCaml programs to WhyML. We show how to translate each construct presented in 5.1, with the exception of t (types), G (top-level GOSPEL declarations, with the exception of protocols), s (GOSPEL terms), and S_f (function specifications). This is because these are isomorphic to their WhyML counterparts. Additionally, we will not be presenting rules for the translation for top-level references. We assume, before we translate the program, that they have been consumed and the declarations in figure 5.5 have been generated.

Each rule in our translation scheme is of the following form:

$$\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![t_1]\!] \rightsquigarrow t_2 \dashv \Sigma' \circ \Delta' \circ \nu' \circ \mu'$$

meaning translating t_1 under environment $\Sigma \circ \Delta \circ \nu \circ \mu$ produces the term t_2 and a new environment $\Sigma' \circ \Delta' \circ \nu' \circ \mu'$. These environments are made up of four elements,

1. A function Σ that, given the name of an effect, will produce a pair. The first element of that pair is a sequence consisting of the types of the arguments the effect takes. The second element will be the effect's return type. To illustrate, after parsing an effect E of type `t1 -> t2 -> t3 -> ... -> tn`, then the Σ we produce must satisfy $\Sigma E = t_1,\ t_2,\ t_3, ..., t_{n-1},\ t_n$
2. A function Δ that maps function identifiers to a set composed of the state variables it modifies. For example, if our program has a function f that modifies the state

$$\frac{}{\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![\texttt{effect}\ E\ :\ \tau]\!] \rightsquigarrow \texttt{exception}\ E \dashv \Sigma E \rightarrow \mathcal{T}\tau}\ (\textsc{TEffect})$$

$$\begin{aligned}
&\mathcal{T}\ \tau_1 \rightarrow \tau_2 \rightarrow \tau_3 = \\
&\quad let\ s = \mathcal{T}\tau_2 \rightarrow \tau_3\ in \\
&\quad \tau_1 :: \pi_1 s, \pi_2 s \\
\\
&\mathcal{T}\ \tau_1 \rightarrow \tau_2 \rightarrow \tau_3 = \\
&\quad \tau_1 \rightarrow \tau_2 = \tau_1, \tau_2 \\
\\
&\mathcal{T}\ \tau_1 \rightarrow \tau_2 \rightarrow \tau_3 = \\
&\mathcal{T}\ \tau = unit, \tau
\end{aligned}$$

Figure 5.7: Effect Translation Rule.

variables S_1 and S_2, then any Δ must satisfy $\Delta f = \{S_1,\ S_2\}$.

3. A set ν consisting of identifiers that are function parameters. For example, given the following expression `fun x` : $\tau \rightarrow e$, when we translate e, ν must satisfy $x \in \nu$. This will be useful when choosing whether to call functions normally or when to use `apply`.

4. A set μ which consists of the identifiers of all the state variables that are modified by the expression we are evaluating. For example, when translating the expression `x:=e`, the μ we produce must satisfy $x \in \mu$.

One last note about environments: some rules will never modify certain elements of the environment. For example, seeing as we only allow top-level effect declarations, parsing an arbitrary expression will never modify Σ. For this reason, we will adopt a shorthand where if a piece of the environment is not present in the right hand side of the translation, we can assume it hasn't been modified. As an example of this shorthand, the following two statements are equivalent

$$\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![t1]\!] \rightsquigarrow t2 \dashv \Delta' \circ \nu' \circ \Delta'$$
$$\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![t1]\!] \rightsquigarrow t2 \dashv \Sigma \circ \Delta' \circ \nu' \circ \Delta'$$

We now turn to the (TEffect) rule (fig 5.7), which turns effect declarations into WhyML exceptions, while also modifying the environment so that the effect's name maps its type in the aforementioned format. To achieve this we use the $\mathcal{T}$ logical function which takes an OCaml type and returns a sequence with the arguments' type along with the effect's return type. Note that the last case of $\mathcal{T}$ is the special case when the effect has no arguments, in which case we assume it receives a `unit` argument.

The next rule, (TPerform), is quite simple: any instance of `perform` is substituted with one of our automatically generated `perform_E` functions.

$$\frac{}{\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![\texttt{perform E}\overline{x}]\!] \rightsquigarrow \texttt{perform_E}\ \overline{x} \dashv} \ (\text{TPerform})$$

Naturally, before these special `perform` functions can be called they have to be generated. To do so, we use the (TProtocol) rule (fig 5.8), which creates the special $perform$ function for that effect which has write permissions too all variables listed in the modifies clause. Additionally, we also generate two predicates which encapsulate the protocol's pre and postcondition. The names of these predicates will be `pre_` and `post_` followed by the name of the effect. The types of these predicates will be, for any Σ:

$$pre_E : \pi_1 \Sigma E \rightarrow state$$
$$post_E : \pi_1 \Sigma E \rightarrow state \rightarrow state \rightarrow \pi_2 \Sigma E$$

To generate these predicates, we must translate the protocol's $requires$ and $ensures$ clauses and turn the sequence of pre and post conditions of the protocol into a single term using the $\mathcal{S}$ logical function. As mentioned, we will not be translating GOSPEL terms, seeing as the result would be identical to the original program. Nevertheless, there is one small difference in regards to specification clauses: any instance of `!x` is replaced with `state._x` and `old !x` is replaced with `old_state._x`. Seeing as defining a translation scheme for GOSPEL terms just for something this simple would be overkill, we will assume a translation function h that performs this transformation.

Our next rule (TFun) (fig 5.9) pertains to the translation of anonymous functions. When functions are declared using `fun` notation, we assume they will be used as first order values and are therefore defunctionalized using the $\mathcal{D}$ translation function. This is admittedly very *ad hoc*, however this choice was implemented mainly to simplify the translation. To make the translation more robust, we could to do some kind of full program analysis to see what functions need to be defunctionalized.

This function is simply a formalization of the process we described in chapter 2: create a function with the same specification and body to check if there are any inconsistencies between the two; create a function that returns a value of type `lambda` and, finally, call the function. Although this expression is admittedly much longer than the original, it returns a value of type `lambda` and generates the appropriate verification conditions.

Before translating the function's expression, we must add its arguments to ν and have $\mu = S_{vars}$ as such $\Sigma \circ \Delta \circ \nu \cup \overline{x} \circ \emptyset \vdash [\![e]\!] \rightsquigarrow e_t \dashv \mu'$. Although translating this function produces a set of modified variables, it is unused. Since `apply` has write permissions to all state variables, we assume calling a defunctionalized function will modify all state variables.

Next up we have the (TPerformsClause) rule, which turns `performs` clauses into `raises` clauses. Moreover, the `raises` clause also ensures the protocol's precondition

$$\frac{\mathcal{S}\overline{Sp}_{requires} = term_{pre} \qquad \mathcal{S}\overline{Sp}_{ensures} = term_{post}}{\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![\texttt{protocol}\ Ex : \overline{Sp}]\!] \rightsquigarrow \mathcal{P}E,\ x,\ \Sigma,\ term_{pre},\ term_{post},\ Sp_{modifies} \dashv} \text{(TPROTOCOL)}$$

$$\mathcal{S}term :: xs = hterm \wedge \mathcal{S}xs$$
$$\mathcal{S}\epsilon = true$$

$\mathcal{P}E,\ x,\ \Sigma,\ term_{pre},\ term_{post},\ mod =$

WhyML

```
predicate pre_E (arg : π1ΣE) (state : state) =
    let x = arg in
    term_pre

predicate post_E (arg : π1ΣE) (old_state : state)
                  (state : S_τ) (reply : π2ΣE) =
    let x = arg in
    term_post


val perform_E (arg: π1ΣE) : π2ΣE
requires{pre_E arg S_x}
ensures{post_E arg (old S_x) S_x result}
writes{mod}
```

Figure 5.8: Protocol Translation Rule.

$$\frac{\Sigma \circ \Delta \circ \nu \cup \overline{x} \circ \emptyset \vdash [\![e]\!] \rightsquigarrow e_t \dashv \mu' \qquad \mathcal{S}S_{requires} = pre \qquad \mathcal{S}S_{ensures} = post}{\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![\texttt{fun}\ S\ \overline{x : \tau_{arg}} : \tau_{ret} \rightarrow e]\!] \rightsquigarrow \mathcal{D}\ \overline{x : \tau_{arg}}\ \tau_{ret}\ e_t\ S\ pre\ post \dashv} \text{(TFUN)}$$

$\mathcal{D}\ args = x_1 : \tau_1, x_2 : \tau_2, ..., x_n : \tau_n)\ \tau_r\ e_t\ S\ term_{pre}\ term_{post} =$

WhyML

```
let f args S = e_t in
val gen_f () : lambda τ1, τ2, ..., τn τr
ensures{
  forall arg state. pre result arg state <->
    let x1, x2 ... = arg in
    term_pre
}
ensures{
  let f = result in
  forall arg old_state state result.
  post f arg old_state state result <->
  let x1, x2 ... = arg in
  term_post
} in gen_f ()
```

Figure 5.9: Defunctionalization Rule.

using the pre_E predicate generated using rule (TProtocol).

$$\frac{}{\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![\texttt{performs } E]\!] \rightsquigarrow \texttt{raises}\{E\ arg \to E_pre\ arg\ S_x\} \dashv} \text{(TPerformsClause)}$$

Next, we present the rules for the translation of function applications. Since in our translation we have some functions that are defunctionalized and others that are not we need rules to handle both cases: one that uses the apply function and another that calls it normally. We will use the apply rule if the expression $e_0\ e_1$ satisfies one of the following:

1. The e_0 expression is not a function identifier.
2. If the e_0 expression is a function identifier, then it must be a value passed as a parameter, and therefore will be contained in ν

If neither of these conditions are met, we can assume the function has not been defunctionalized and we can apply the function normally. The two rules are as follows:

$$\frac{\forall i\ \Sigma \circ \Delta \circ \nu \circ \mu_i \vdash [\![e_i]\!] \rightsquigarrow e_{it} \dashv \mu_{i1} \qquad e_0 = x \implies x \in \nu}{\Sigma \circ \Delta \circ \nu \circ \mu_0 \vdash [\![e_0\ e_1\ ...\ e_n]\!] \rightsquigarrow \texttt{apply}\ e_0\ e_2, e_3, ..., e_n \dashv S_{vars}} \text{(TAppDefun)}$$

$$\frac{\forall i\ \Sigma \circ \Delta \circ \nu \circ \mu_i \vdash [\![e_i]\!] \rightsquigarrow e_{it} \dashv \mu_{i1} \qquad \neg x \in \nu}{\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![x\ e_0\ e_1\ ...\ e_n]\!] \rightsquigarrow x\ e_{0t}\ e_{1t}\ \dots e_{nt} \dashv \Delta x \cup \mu_{n1}} \text{(TApp)}$$

In the case of (TAppDefun), seeing as we add a call to apply, which has write permissions to all state variables, the set of modified variables it produces is S_{vars}. This set contains the identifiers of all the top level state variables in this program. In the case of (TApp), the set of modified variables is composed of the variables modified by each argument expression, as well as the variables modified by calling x

Finally, we move on to the *crème de la crème* of our translation rules: (TTry) (Fig 5.10). This rule creates an expression which consists of calling a function that consists of an exception handler using the $\mathcal{H}$ translation function. This translation function first creates a snapshot of the state before we enter the handler init_state. Then, we wrap the translated e_{0t} expression in an exception handler. For each handled effect we will create an equivalent exceptional case with the same name and arguments. For each of these cases we create another snapshot of the state eff_state, this time right after control is surrendered to the handler. We then create a function that generates a continuation. This function will ensure the continuation satisfies the following conditions:

1. The continuation's one-shot check is valid.
2. The continuation's precondition is the same as the protocol's postcondition.
3. The continuation's postcondition is the same as the handler's invariant.

4. Any state variable that is not modified by the handler is also not modified by the continuation.

The first three conditions are similar to what we saw in section 5.3. To express the fourth, however, we must state in the postcondition, for each variable x that isn't modified:

WhyML
```
forall arg state_old state result.
  ... state.x = state_old.x
```

To create this condition for all the variables in $S_{vars} \setminus \mu_{n1}$, which contains all state variables except those modified by the handler, we use the logical function $\mathcal{O}$.

$$\mathcal{O}\epsilon = \texttt{true}$$
$$\mathcal{O}x :: t = \texttt{state.}x = \texttt{state_old.}x \land \mathcal{O}t$$

Now that we have fully specified the function that generates the continuation, we then call it and add the translated expression. After doing this for each case we wrap the complete exception handler in a function named `handler` whose postcondition is the handler's invariant.

Although our scheme has a few more rules, they are quite straightforward and are therefore not worthy of a lengthy explanation. The full translation scheme can be found in Figure 5.11

5.5 Limitations

Although our translation scheme can cover an interesting subset of programs with handlers, there are a few limitations on what programs we can verify using this approach:

1. This scheme does not support shallow handlers. More generally, it does not support any handler that generates continuations that produce effects.

2. The only stateful variables allowed are references declared in the top level (with the exception of continuations' *one-shot* check).

3. In case of higher order functions, any functions passed as parameters are only allowed to perform effects explicitly listed in the performs clause. For example:

   GOSPEL + OCaml
   ```
   let f (g : 'a -> 'a) = ...
   (*@ performs E*)
   ```

   when calling `f`, the `g` function we pass as an argument can only perform the `E` effect.

$$\frac{\forall i.0 < i \leqq n \implies \Sigma \circ \Delta \circ \nu \circ \mu_i \vdash [\![e_i]\!] \rightsquigarrow e_{it} \dashv \mu_{i1} \quad \mathcal{S}\overline{Sh_{ensures}} = term_{post} \quad Sh_{returns} = \tau \quad e_{try} = try\ e_0\ with\ Sh\ \overline{\texttt{effect}\ Ex\ k \rightarrow e}}{\Sigma \circ \Delta \circ \nu \circ \emptyset \vdash [\![e_{try}]\!] \rightsquigarrow \mathcal{H}\overline{Ex},\ \overline{e},\ \overline{Sh_{ensures}},\ term_{post},\ \tau,\ \Sigma,\ \mathcal{O}S_{vars} \setminus \mu_{n1} \dashv \mu_n} \text{(TTRY)}$$

$\mathcal{H}e_{0t}, \overline{Ex}, e, \overline{S_{hensures}}, term_{post}, \tau, \Sigma, term_{state}) =$

```
let handler () =                                                    WhyML
  ensures{S_hensures1}
  ensures{S_hensures2}
  ...
      let init_state = S_x in
      try e_0t with
      |...
      |E_n x ->
          let eff_state = S_x in
          val gen_k unit : continuation π_2 ΣE_n τ
          ensures{valid result}
          ensures{
              forall arg state.
              pre result arg state <->
              E_post arg eff_state state result
          }
          ensures{
              let f = result in
              forall arg irrelevant_old_state state result.
              post f arg irrelevant_old_state state result <->
                let state_old = init_state in
                term_post ∧ term_state
          } in let k = gen_k () in e_nt
        |...
  in handler ()
```

Figure 5.10: Translation rule for effect handlers.

4. Our defunctionalized functions are called using `apply`, which is non-divergent. This means that our system rejects programs that need to defunctionalize divergent functions, since these would be called using the non-divergent `apply`, generating an inconsistency.

Although these limitations simply place restrictions on what programs we can verify, they do not introduce any inconsistencies. However, one way this scheme is unsound is in that we cannot prove that `Server` only modifies variables listed in the `modifies` clause of the protocol. Although the other limitations are as a result of some intractable problem our WhyML representation, this one is mainly due to time constraints. We believe that this can be ensured by expanding the continuations' precondition by stating that any variable not listed in the `modifies` clause must not have been altered.

$$\frac{}{\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![\texttt{effect}\ E\ :\ \tau]\!] \rightsquigarrow \texttt{exception}\ E \dashv \Sigma E \rightarrow \mathcal{T}\tau}\ (\textsc{TEffect})$$

$$\frac{}{\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![\texttt{perform E}\acute{\texttt{x}}]\!] \rightsquigarrow \texttt{perform_E}\ \acute{\texttt{x}} \dashv}\ (\textsc{TPerform})$$

$$\frac{\mathcal{S}\overline{Sp}_{requires} = term_{pre} \qquad \mathcal{S}\overline{Sp}_{ensures} = term_{post}}{\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![\texttt{protocol}\ Ex : \overline{Sp}]\!] \rightsquigarrow \mathcal{P}E,\ x,\ \Sigma,\ term_{pre},\ term_{post},\ Sp_{modifies} \dashv}\ (\textsc{TProtocol})$$

$$\frac{\Sigma \circ \Delta \circ \nu \cup \overline{x} \circ \emptyset \vdash [\![e]\!] \rightsquigarrow e_t \dashv \mu' \qquad \mathcal{S}S_{requires} = pre \qquad \mathcal{S}S_{ensures} = post}{\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![\texttt{fun}\ S\ \overline{x : \tau_{arg}} : \tau_{ret} \rightarrow e]\!] \rightsquigarrow \mathcal{D}\ \overline{x : \tau_{arg}}\ \tau_{ret}\ e_t\ S\ pre\ post \dashv}\ (\textsc{TFun})$$

$$\frac{}{\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![\texttt{performs}\ E]\!] \rightsquigarrow \texttt{raises}\{E\ arg \rightarrow E_pre\ arg\ S_x\} \dashv}\ (\textsc{TPerformsClause})$$

$$\frac{\forall i\ \Sigma \circ \Delta \circ \nu \circ \mu_i \vdash [\![e_i]\!] \rightsquigarrow e_{it} \dashv \mu_{i1} \qquad e_0 = x \implies x \in \nu}{\Sigma \circ \Delta \circ \nu \circ \mu_0 \vdash [\![e_0\ e_1\ ...\ e_n]\!] \rightsquigarrow \texttt{apply}\ e_0\ e_2, e_3, ..., e_n \dashv S_{vars}}\ (\textsc{TAppDefun})$$

$$\frac{\forall i\ \Sigma \circ \Delta \circ \nu \circ \mu_i \vdash [\![e_i]\!] \rightsquigarrow e_{it} \dashv \mu_{i1} \qquad \neg x \in \nu}{\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![x\ e_0\ e_1\ ...\ e_n]\!] \rightsquigarrow x\ e_{0t}\ e_{1t}\ \dots e_{nt} \dashv \Delta x \cup \mu_{n1}}\ (\textsc{TApp})$$

$$\frac{\begin{array}{c}\forall i.0 < i \leqq n \implies \Sigma \circ \Delta \circ \nu \circ \mu_i \vdash [\![e_i]\!] \rightsquigarrow e_{it} \dashv \mu_{i1} \\ \mathcal{S}\overline{Sh_{ensures}} = term_{post} \qquad Sh_{returns} = \tau \qquad e_{try} = try\ e_0\ with\ Sh\ \overline{\texttt{effect}\ Ex\ k \rightarrow e}\end{array}}{\Sigma \circ \Delta \circ \nu \circ \emptyset \vdash [\![e_{try}]\!] \rightsquigarrow \mathcal{H}\overline{Ex},\ \overline{e},\ \overline{Sh_{ensures}},\ term_{post},\ \tau,\ \Sigma,\ \mathcal{O}S_{vars} \setminus \mu_{n1} \dashv \mu_n}\ (\textsc{TTry})$$

$$\frac{\Sigma \circ \Delta \circ \nu \circ \emptyset \vdash [\![e]\!] \rightsquigarrow e_t \dashv \mu \qquad e \neq fun\ ... \qquad x \neq \epsilon \implies \Delta' = \Delta f \rightarrow \mu \qquad x = \epsilon \implies \Delta' = \Delta}{\Sigma \circ \Delta \circ \nu \circ \emptyset \vdash [\![\texttt{let}\ S\ f\ x = e]\!] \rightsquigarrow \texttt{let}\ S\ f\ x = e_t \dashv \Delta'}\ (\textsc{TLet})$$

$$\frac{\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![e_1]\!] \rightsquigarrow e_{1t} \dashv \mu' \qquad \Sigma \circ \Delta \circ \nu \circ \mu' \vdash [\![e_2]\!] \rightsquigarrow e_{2t} \dashv \mu''}{\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![\texttt{if}\ a\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!] \rightsquigarrow \texttt{if}\ a\ \texttt{then}\ e_{1t}\ \texttt{else}\ e_{2t} \dashv \mu''}\ (\textsc{TIf})$$

$$\frac{\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![e_1]\!] \rightsquigarrow e_{1t} \dashv \mu' \qquad \Sigma \circ \Delta \circ \nu \circ \mu' \vdash [\![e_2]\!] \rightsquigarrow e_{2t} \dashv \mu''}{\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![e_1\ ;\ e_2]\!] \rightsquigarrow e_1\ ;\ e_2 \dashv \mu''}\ (\textsc{TSeq})$$

$$\frac{\begin{array}{c}x \neq \epsilon \implies \mu = \emptyset \qquad \Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![e_1]\!] \rightsquigarrow e_{1t} \dashv \mu' \qquad e \neq fun\ ... \\ x \neq \epsilon \implies \Delta' = \Delta f \rightarrow \mu' \qquad x = \epsilon \implies \Delta' = \Delta \qquad \Sigma \circ \Delta' \circ \nu \circ \mu' \vdash [\![e_2]\!] \rightsquigarrow e_{2t} \dashv \mu''\end{array}}{\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![\texttt{let}\ S\ f\ x = e_1\ \texttt{in}\ e_2]\!] \rightsquigarrow \texttt{let}\ S\ f\ x = e_{1t}\ \texttt{in}\ e_{2t} \dashv \mu''}\ (\textsc{TLetIn})$$

$$\frac{\forall i.0 \leqq i < n \implies \Sigma \circ \Delta \circ \nu \circ \mu_i \vdash [\![e_i]\!] \rightsquigarrow e_{it} \dashv \Sigma \circ \Delta \circ \nu \circ \mu_{i1}}{\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![\texttt{match}\ a\ \texttt{with}\ \overline{p \rightarrow e_i}]\!] \rightsquigarrow \texttt{match}\ a\ \texttt{with}\ \overline{p \rightarrow e_{it}} \dashv \Sigma \circ \Delta \circ \nu \circ \mu_n}\ (\textsc{TMatch})$$

$$\frac{}{\Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![\texttt{modifies}\ s]\!] \rightsquigarrow \texttt{writes}\{s\} \dashv}\ (\textsc{TModifies}) \qquad \Sigma \circ \Delta \circ \nu \circ \mu \vdash [\![\epsilon]\!] \rightsquigarrow \epsilon \dashv (\textsc{Tempty})$$

$$\frac{\Sigma \circ \Delta \circ \nu \circ \emptyset \vdash [\![d]\!] \rightsquigarrow d_t \dashv \Sigma' \circ \Delta' \qquad \Sigma' \circ \Delta' \circ \nu \circ \emptyset \vdash [\![d]\!] \rightsquigarrow d_{tl} \dashv \Sigma'' \circ \Delta''}{\Sigma \circ \Delta \circ \nu \circ \emptyset \vdash [\![d :: d]\!] \rightsquigarrow d_t :: d_{tl} \dashv \Sigma'' \circ \Delta''}\ (\textsc{TDecl})$$

Figure 5.11: Inductive Translation Rules.

6

Case Studies

We will now put our scheme to the test and prove four different programs. We start with the division interpreter we presented in 2.4. We then present a program which implements mutable state using algebraic effects. Next, we show a program which turns an iterator into a generator; essentially a simpler version of the joining streams example from section 2.4.2. Finally, we show an example created by hand that shows how we may prove functions with shallow handlers using Why3 as well as going over the limitations which prevented us from generalizing this approach.

All these examples, with the exception of the one with shallow handlers, were proved using an extension of Cameleer that implements the translation rules we showed in section 5.4. The complete implementation is publicly available[1].

6.1 Cameleer Implementation

Before we go over our verified case studies, we will briefly discuss some of the decisions we made when our implementation of the translation scheme. As we mentioned, the target language we present in Chapter 5 is an approximation of a core fragment of the OCaml language. Nevertheless, some of the syntactic constructs presented do not exist as of yet in OCaml, the most notable of which being effect handlers. Although these technically exist in the alpha release of OCaml 5.0, they are exposed with functions and therefore have no syntactic support. To illustrate how one might write a handler in OCaml, we present, in Figure 6.1 a handler for the `xchg` function from section 5.3.

All case studies detailed in this chapter were implemented and proved using this notation. However, for the sake of maintaining legibility, we will present them using the syntax from Chapter 5.

[1] `https://github.com/mrjazzybread/cameleer/tree/effects`

OCaml
```
let _ =
    try_with (fun x -> xchg x) 3
    {effc =
        (fun (type a) (e : a Effect.t) ->
            match e with
            |XCHG n -> (Some (fun (k : (a, _) continuation) ->
                let old_p = !p in
                p := n;
                continue k old_p))
            |_ -> None)}
```

Figure 6.1: Handler for the `xchg` function implemented in OCaml.

GOSPEL + OCaml
```
effect Div_by_zero : int

let rec eval (e : exp) : int = match e with
    |Int x -> x
    |Div(l, r) ->
    let eval_l = eval l in
    let eval_r = eval r in
        if eval_r = 0
            then perform Div_by_zero
            else eval_l / eval_r

let main e =
    try eval e with
    |effect Div_by_zero k -> continue k max_int
```

Figure 6.2: Division Interpreter.

6.2 Division Interpreter

We start with the division interpreter. The program, showed in figure 6.2 is exactly as we detailed in section 2.4.

6.2.1 Interpreter Specification

The first step in writing our specification should be finding an appropriate protocol. When replying to an effect, `Server` is giving meaning to divisions by zero. Although `Server` can choose any value it wants, this value must remain consistent. For example, if we call `eval` and reply to an effect with the value 1000, every other effect that `Client` performs must also be replied with this same value. To enforce this, we will create a ghost variable that will save the value that `Server` chose for divisions by zero. Since this variable will only be relevant for our specifications, it will be defined (and later modified) within a GOSPEL comment:

GOSPEL + OCaml
```
(*@ val v0 : int *)
```

Next, we define a protocol stating that `Server` must reply with `v0`:

GOSPEL + OCaml

```
(*@ protocol Div_by_zero :
    ensures result = !v0)
```

Note how this protocol does not have a `modifies` clause. This means that `Server` cannot choose a new value for `v0`.

Now that we have a protocol defined, let us now consider the specification for `eval`. In section 2.4, we defined a logical function `eval_ind` to describe how we evaluate an expression. We will define a similar function where we explicitly handle the case of a division by zero by returning a default value `v0`.

GOSPEL + OCaml

```
(*@ function eval_ind (exp : exp) (v0 : int) : int =
    match exp with
    |Const n -> n
    |Div exp1 exp2 ->
        if eval_ind exp2 v0 = 0
          then v0
          else div (eval_ind exp1 v0) (eval_ind exp2 v0)
*)
```

With this function in place, we may now specify `eval` as follows:

GOSPEL + OCaml

```
let eval (e : exp) = ...
(*@ ensures eval_ind e !v0
    performs Div_by_zero*)
```

If we wish to write a handler for this function, we need only to set the `v0` variable with the value with which we want to define divisions by zero before calling. In this case, we chose the value 1000. The postcondition will be roughly the same as the postcondition for `eval`

GOSPEL + OCaml

```
let main (e : exp) =
  try v := 1000; eval e with
  |effect Div_by_zero k -> continue k 1000
(*@ try_ensures eval_ind e 1000
    returns int *)
```

The full program (without any GOSPEL declarations) is in figure 6.3

6.2.2 Translating The Interpreter

We will now move over the interpreter's translation. This will be the only case study (barring the shallow handler) in which we detail the WhyML translation. This is mostly due to the fact that none of the translations have any major structural differences. First,

GOSPEL + OCaml

```
let rec eval (e : exp) : int =
   match e with
   |Const n -> n
   |Div(e1, e2) ->
      let eval_l = eval e1 in
      let eval_r = eval e2 in
      if eval_r = 0
         then perform Div_by_zero
         else eval_l / eval_r
(*@
   ensures eval_ind e !v0 = result
   performs Div_by_zero
   variant e
*)

let main (e : exp) =
    try v := 1000; eval e with
    |effect Div_by_zero k -> continue k 1000
(*@try_ensures eval_ind exp 1000 = result
      returns int *)
```

Figure 6.3: Specified Division Interpreter with Handler.

we must define the `state` type. Since there is only one reference in this program (`v0`), the state is represented by the following record type:

WhyML

```
type state = {_v0 : int}
```

Next, we must set up the `Div_by_zero` effect. Naturally, we will need an exception with the same name. Additionally, we will need a specialized `perform` function with the same specification as the protocol. We will assume two predicates `pre_Div_by_zero` and `post_Div_by_zero` that encapsulate the postconditions of the protocol, as follows.

WhyML

```
exception Div_by_zero


val perform_Div_by_zero () : int
ensures{post_Div_by_zero (old Sx) Sx result}
```

In this case, S_x is shorthand for the expression `{_v0 = !v0}`. The translation of the specification and the body of the `eval` function will be identical to its GOSPEL and OCaml counterparts. The one difference is that the call to `perform` will be replaced by the `perform_Div_by_zero` function. Additionally, the `performs` clause will be replaced by a `raises` clause. The full specification is as follows:

WhyML

```
ensures{eval_ind (!curr_exp) = result}
raises{Div_by_zero}
variant{e}
```

6.2.3 Verifying the Interpreter

Verifying the `eval` function mostly amounts to discharging a series of verification conditions we would normally find for this kind of recursive function (i.e., proving termination, proving the base case and the inductive case respect the function's postcondition, and proving the `raises` clause) . There are a few verification conditions, nevertheless, that are specific to programs with protocols.

This verification condition is validated. We must also prove that the postcondition of this function is met after we perform the effect. Since we only perform `Div_by_zero` when the right hand side of the division evaluates to 0 and `Server` always replies with `v0`, Why3 is able to prove that this function respected the `eval_ind` predicate

6.2.4 Translation of the Handler

Translating the handler into WhyML will be quite verbose, but not dissimilar to what we have seen thus far. Therefore, we will focus specifically on the creation and specification of the continuation. Firstly, as always, we must specify that the continuation is valid:

```
val k : continuation int int
assume{k._valid}
```

Next, we must state that the precondition of the continuation is the protocol's postcondition.

```
assume{forall reply state.
    pre k reply state <->
    reply = state._v0}
```

Finally, we must ensure that the postcondition of the continuation is the handler's invariant.

```
assume{ forall reply old_state state result.
    post k reply old_state state result <->
    eval_ind exp old_state._v0}
```

6.2.5 Verifying the handler

This handler generates four VCs. The first states the handler's invariant holds in case of a non-exceptional return. Since the handler's invariant is equivalent to `eval`'s postcondition, this is proved trivially. Next, in case of an exceptional return, we must prove that the continuation is valid and that the continuation's precondition (that is, the protocol's postcondition) holds. For the former, since the continuation starts valid and we haven't called it, Why3 easily verifies this. As for the latter, since we set `v0` to equal `1000`, Why3 can also prove this. Finally, we must prove that in case of an exceptional return, the invariant still holds. Since the exceptional branch ends with a call to the continuation,

whose postcondition is the invariant, this is also proved. All of these conditions are proved using the Alt-Ergo prover [22]

6.3 Mutable Reference

Our next example will involve the creation of an ambient state using algebraic effects. In short, we will simulate the behavior of a single mutable integer reference by having two effects. A `Get` effect that will retrieve the value of the reference and a `Set` effect that will receive as argument a single integer and change the value of the reference. Important to note that the implementation of the OCaml program will be written without any mutable values.

6.3.1 Client implementation and Specification

As stated, our program will have the following effects.

GOSPEL + OCaml

```
effect Get : int
effect Set : int -> unit
```

The `Get` effect will simply return the current value of the ambient reference. The `Set` effect will receive the integer and set the value of the reference. As we have stated, there is no explicit mutable value. However, to model this behavior, we will have a ghost GOSPEL reference that will hold the value of this abstract state.

GOSPEL + OCaml

```
(*@ val ghost r : int ref *)
```

Using this ghost reference, we can create a protocol for both effects. We then create two functions that simply call these two effects.

GOSPEL + OCaml

```
(*@ protocol Get :
    ensures result = !r *)

(*@ protocol Set x :
    ensures !r = x
    modifies r*)

let set (n : int) : unit = perform (Set n)
(*@ performs Set
    ensures !r = n*)

let get () : int = perform Get
(*@ performs Get
    ensures result = !r*)
```

Since the `get` and `set` functions are essentially just reading and writing from a ghost reference, proving a `Client` that uses these would be no different from proving one that uses normal references. Nevertheless, we will give a small example to give the reader an idea of the kinds of specifications a `Client` of this nature might have:

GOSPEL + OCaml

```
(*@ predicate fun_post (old_r : int) (r : int) (result : int) =
    result = r * r && r = old_r * 2  *)

let some_fun () : int =
    set(get () + get ()); get() * get()
(*@ ensures fun_post (old !r) !r result
    performs Get
    performs Set*)
```

Verifying this function is quite simple: it produces a single VC proving that `some_fun` satisfies `fun_post`, seeing as neither `Get` nor `Set` have preconditions.

6.3.2 Handler Implementation and Specification

Although writing and proving `Client` was very simple, this handler will be the most complex out of all the ones we have seen. Instead of calling the continuation, it will return a function that (potentially) calls a continuation. The returned function will be of type `int -> int`. This will be the first time we see a handler that doesn't have the same return value as `Client`. The argument it receives is the initial value the reference will holds. Its result is the result of `some_fun` assuming that the initial value of the ambient reference is the argument of the function.

To achieve this, we first consider the function we return after `some_fun` completes. Since `some_fun` has terminated, the function we will return will ignore the argument passed and simply return whatever `some_fun` returned.

GOSPEL + OCaml

```
try
    let ret = some_fun () in
    fun _ -> ret
with ...
```

Now, we must specify this anonymous function. Seeing as the specification of `some_fun` is `fun_post` applied to the initial and final values of `r` as well as the result. Therefore, the specification is as follows:

GOSPEL + OCaml

```
try let ret = some_fun in
    (*@ let final_r = !r in)
    fun (*@ ensures fun_post init_state._r final_r result*) _ -> ret
with ...
```

A reminder that the `init_state` variable represents the state before the handler began execution. Our next example will be the case of a `Get` effect. Here, we want to return a function that calls the continuation by passing the current value of the reference. In other words we call the continuation using the argument of the `fun` we return.

GOSPEL + OCaml
```
|effect Get k ->
    fun n -> continue k n
```

This call to `continue`, however, will return another function, since that is the return type of the handler. This function also receives as argument the current value of the state. Seeing as `Get` effects do not modify the reference, we simply call the function with the argument, just like we did with the continuation.

GOSPEL + OCaml
```
|effect Get k ->
    fun n ->
        let env = continue k n in
        env n
```

As for this function's specification, its postcondition will, once again, be the same as `some_fun`. It will, nonetheless, require a precondition stating that the argument this function receives is equal to the ghost reference. Moreover, we will also state that the value of the ghost reference hasn't been changed since the this effect was called. The full specification would be:

GOSPEL + OCaml
```
fun
    (*@ requires n = !r
        ensures fun_post init_state._r !r result *)
    n -> ...
```

Finally, we move on to the last case, which is the `Set` effect. The general structure will be similar to the previous lambda. The main difference will be in the arguments we pass. Since `Set` has type `int -> unit`, the continuation can only be resumed by passing unit. Moreover, the argument we will pass to the function the continuation returns is the value passed when the effect was performed.

GOSPEL + OCaml
```
|effect (Set n) k ->
    fun (*@ ensures fun_post init_state._r !r result *) _ ->
        let env = continue k () in
        env n
```

The full handler can be found in figure 6.4. All that is left now to give the handler its specification. Specifying the handler will be equivalent to specifying the function it returns. Regardless of which branch the handler will fall into, each function has the same postcondition. Therefore, one of the invariants will be:

GOSPEL + OCaml
```
try
    let res = some_fun in
    (*@ let final_r = !r in)
    fun (*@ ensures fun_post init_state._r final_r result*) _ -> res
with
|effect Get k ->
    fun
    (*@ requires n = !r
        ensures fun_post init_state._r !r result *)
    n ->
        let env = continue k n in
        env n
|effect (Set n) k ->
    fun (*@ ensures fun_post init_state._r !r result *) _ ->
        let env = continue k () in
        env n
```

Figure 6.4: Complete `sum_fun` handler.

GOSPEL + OCaml
```
try_ensures
    let g = result in
    forall arg state_old state result.
        post g arg state_old state result ->
        fun_post (old !r) state._r result
```

Note that `old !r` refers to the value `r` holds at the start of the handler expression.

Moving on to the function's precondition, we would like to state that it can only be called with the current value of the `r` reference. Additionally, we would like to state that it can only be called once. As we mentioned in section 5.5 however, we cannot directly prove the one-shot rule for functions that call continuations. To work around this, we will create a new ghost reference `va` that will track if the continuation has been called or not.

GOSPEL + OCaml
```
(*@ val va : bool ref *)
```

We then add the following invariant.

GOSPEL + OCaml
```
try_ensures forall arg state.
    state._va && state._r = arg ->
    pre result arg state
```

Although the natural reading of a condition of type (... `-> pre f arg state`) *f's precondition is ...* what we are actually saying is *An equivalent or stronger condition relative to f's postcondition is ...*. This means that even though the function returned when `some_fun` ends doesn't have a precondition, it still satisfies this invariant.

We will however, have to change the two functions returned in case of an effect, seeing as the function's returned by the continuation require the `va` flag to be set to `true`. In short,

we add a precondition to both functions simply stating `!va`. The full handler specification is:

GOSPEL + OCaml

```
(*@
    try_ensures forall arg state.
            state._va && state._r = arg ->
            pre result arg state
    try_ensures let g = result in
        forall arg state_old state result.
        post g arg state_old state result ->
            fun_post (old !r) state._r result
    returns int -> int
*)
```

6.3.3 Verification of the Handler

We will now go over the VCs this program generates. The first VC states the body of the function returned after `some_fun` ends adheres to its specification. Since its postcondition is:

GOSPEL + OCaml

```
fun (*@ ensures fun_post init_state._r final_r result*) _ -> res
```

and it simply returns the value returned by `some_fun`, this VC is validated. The next two state the returned function respects the two invariants. The first one is simple, seeing as this function has no precondition, in other words `pre f arg state <-> true`.

As for the next invariant, since it technically states

`fun_post (old !r) state._r result`

and the postcondition for the function states

`fun_post init_state._r final_r result`,

Why3 can't prove that

`state._r = final_r`.

In simpler terms, we cannot prove the final value that `r` holds at the end of the returned function equals `final_r`. Since `r` is simply a ghost variable to aid in our specification, we can use it in whatever way makes our proofs easier, as long as we don't introduce anything that breaks the underlying logic of what we are trying to model. Therefore, we will modify the function slightly so that Why3 can reach the conclusions we want:

GOSPEL + OCaml

```
fun (*@ ensures fun_post ... && !r = final_r *)
_ -> (*@ r:= final_r; *) ret
```

With this new postcondition, Why3 can now prove that this function satisfies both invariants.

The next two verification conditions refer to the `Get` effect. We wish to prove the preconditions for calling `continue`. Namely if the continuation is valid and if the protocol's postcondition is met. The former is verified trivially. The latter is verified using the precondition stating that the argument passed must be equal to `!r`.

Next we prove the precondition for the `env` function that the continuation returns. According to the first invariant, the precondition of this function can be assured if the `va` variable is set to true and the `r` variable equals the argument. Both of these facts can be proven from the function's precondition.

We must also prove the body of this function respects the postcondition. Seeing as its postcondition is the same as the postcondition of the function returned by the continuation, this is easily proved.

Finally, we have to prove this lambda term respects the invariants. Since its precondition and postcondition are equivalent to what is presented in the invariants, Why3 proves this trivially.

There are also the VCs generated for the `Set` effect. We will skip these since they are effectively the same as those generated for `Get`.

6.4 Generators

In section 2.4.2, we showed how effects can be used to control the flow of the iteration of a stream of integers. This case study will be a simpler version of that example where we have a higher order function that applies a function received as argument to a stream of integers. We will use an algebraic effect so as to turn this iteration over a stream of integers into a function that generates integers one by one.

6.4.1 Client Specification

To specify the original iterator, we use a reference to a ghost list of integers.

GOSPEL + OCaml

```
(*@ val ghost l : (list int) ref *)
(*@ predicate permitted (l : list int) (n : int) *)
(*@ predicate complete (l : list int)*)
```

This list represents the values which the iterator has already enumerated. We will also have two predicates `permitted` and `complete`. The former is used to verify if `n` is a valid next element, given a list of iterated values `l`. The later is used to check if iteration has completed. Given these predicates, our client will have the following specification.

GOSPEL + OCaml

```
let iter (f : int -> unit) : int = ...
(*@ requires forall arg s. permitted s._l arg -> pre f arg s
    ensures complete !l *)
```

The precondition of `iter` states that the strongest precondition possible for `f` is that it only allows values that are permitted next elements in the iteration. Furthermore, the

GOSPEL + OCaml

```
(*@ val ghost l : (list int) ref *)
(*@ predicate permitted (l : list int) (n : int) *)
(*@ predicate complete (l : list int)*)

effect Yield : int -> unit

(*@ protocol Yield x :
    requires permitted !l x
    ensures !l = x::(old !l)
    modifies l*)

let iter (f : int -> unit) : int = ...
(*@ requires forall arg s. permitted s._l arg -> pre f arg s
    ensures complete !l
    performs Yield *)
```

Figure 6.5: Client Specification.

postcondition ensures that once `iter` terminates, `l` will satisfy `complete`, which is another way of saying the iteration has ended.

All that is left is to include a `performs` clause. Ideally, we would like to state that this function performs any effect that `f`, its argument, may perform. However, translating this kind of assertion into WhyML would be very difficult due to certain limitation regarding specifying higher-order functions that perform effects. We will go over these limitations in section 6.5. Therefore, we will not give `iter` a proper implementation and instead add the following `performs` clause

GOSPEL + OCaml

```
performs Yield
```

where `Yield` is the following effect.

GOSPEL + OCaml

```
effect Yield : int -> unit
```

The basic idea is very similar to the `join_stream` example: we use the `Yield` effect to stop the iteration and expose the current value. The `l` list should be updated to reflect that we have just iterated another element. With this in mind, the protocol for `Yield` is as follows:

GOSPEL + OCaml

```
(*@ protocol Yield x :
    requires permitted !l x
    ensures !l = x::(old !l)
    modifies l*)
```

The full program can be found on figure 6.5

6.4.2 Handler Implementation

We will now show how we implement our handler so as to create a generator. This function will have type `unit -> int option`. The expected behavior of our generator function is as follows:

1. If the iteration has completed and there are no more elements to generate, the function should return `None`.

2. If there is at least one x left to iterate, the generator should return `Some x`. Additionally, the generator should update itself so that the next time it is called, it produces the next element in the iteration (or `None`, potentially).

In order to implement a function that updates itself, we will need the following top-level variable:

GOSPEL + OCaml
```
let gen : (unit -> int option) ref = (fun () -> None)
```

Since `Server` modifies this variable, we must add it to the `modifies` clause of the protocol.

GOSPEL + OCaml
```
(*@ protocol Yield x :
...
    modifies l, gen *)
```

We are now ready to begin implementing the handler. As usual, we start with the expression between the `try_with`. Naturally, we begin with a call to `iter`. This call will be done using a function that simply performs a `Yield` effect. Once `iter` has completed, there are no more elements to return. This means the `gen` function must be modified so as to return `None`:

GOSPEL + OCaml
```
try
    iter (fun x -> perform (Yield x));
    gen := (fun () -> None)
with ...
```

Interestingly, this handler returns `unit`. In other words, it has no meaningful return value, all it does is update `gen`. As for the case where `Yield` is performed, we update `gen` with a function that updates the `l` reference, returns the yielded value and calls the continuation. This will resume iteration and update `gen` once more.

GOSPEL + OCaml
```
|effect (Yield x) k ->
    gen := (fun () ->
        l := x::!l; continue k (); Some x)
```

6.4.3 Handler Specification

We start by specifying the function we pass when calling `iter`. We state this function performs `Yield` and its precondition is that the argument it receives must be a valid next step in the iteration.

GOSPEL + OCaml
```
fun (*@ requires permitted !l x
        performs Yield*) x -> ...
```

The specification of this handler will be rather complex, since we are dealing with a function that updates itself. To specify this handler, we stae the postcondition of the generator functions it produces will have as their postcondition the `inter_inv` predicate. This predicate must hold after calling any generator this handler may produce. It will receive the state of the program before and after the generator was called as well as the result.

GOSPEL + OCaml
```
(*@ predicate iter_inv
    (s_old : state) (s : state) (result : option int) *)
```

Now we move on to defining this predicate. In case the result is `None`, we want to state that the iterator is `complete` and that there was no modification to any state variable:

GOSPEL + OCaml
```
(*@ predicate iter_inv ... =
    match result with
    |None -> complete s._l && s_old = s
    |...  *)
```

As for the case where the result is `Some x`, we want to state `x` is a valid next step in the iteration. Moreover, we want to state that the `l` list was modified by adding `x` to the head.

GOSPEL + OCaml
```
predicate iter_inv
    (s_old : state) (s : state) (result : option int) =
match result with
|None -> ...
|Some x -> permitted s_old._l x && s._l = x::s_old._l && ...
```

We also want to state that after the generator function is called, the `gen` reference will be updated with a new function. This function must also have as its postcondition the `iter_inv` predicate. We would also like to state that each new function can only be called at most once, due to the one-shot rule. Unfortunately, as we mentioned in section 5.5, this cannot be done for functions that call continuations

GOSPEL + OCaml
```
forall old_next_state next_state next_result.
    post s._gen () old_next_state next_state result ->
    iter_inv old_next_state next_state result
```

```
(*@                                                              GOSPEL + OCaml
axiom iter_inv1 :
  forall state_old state.
  iter_inv state_old state None <-> complete state._l && state = state_old
*)

(*@
axiom iter_inv2 :
  forall state_old state r. (
    permitted state_old._l r
    && state._l = r::state_old._l
    && forall state_old1 state1 result.
      (post state._gen () state_old1 state1 result ->
          iter_inv state_old1 state1 result)
      ) <-> iter_inv state_old state (Some r)
*)
```

Figure 6.6: Axiomatized Definition of `iter_inv`

Alas, this predicate is rejected by Why3 during our translation. Since this predicate is recursively defined, Why3 tries to prove its termination, to no avail. One way around this is to define `iter_inv` using axioms as shown in figure 6.6.

With this predicate defined, we can now specify the lambdas and the handler. As for the former, their specifications are quite simple, we will simply state that they satisfy the `iter_inv` predicate with the state before and after the function is called:

```
fun (*@ ensures iter_inv (old S_x) S_x result *) -> ...        GOSPEL + OCaml
```

The handlers specification simply states that the function the `gen` reference points to satisfies the `iter_inv` predicate.

```
(*@ try_ensures                                                GOSPEL + OCaml
    forall s_old s result.
        post !gen () s_old s result ->
        iter_inv s_old s result*)
```

The full handler can be found in figure 6.7

6.4.4 Verification

The first generated VC for this example state that the call to `perform` satisfies the precondition of the protocol, which is `next !l x`, where `x` is the yielded value. This is proved using the lambda's precondition, which is equivalent to the protocol. Next, Why3 proves that the function we pass as a parameter when calling `iter` satisfies its precondition

```
requires forall arg s. next s._l arg -> pre f arg s            GOSPEL + OCaml
```

GOSPEL + OCaml

```
try
    iter (fun (*@ requires permitted !l x
                  performs Yield*)
               x -> perform (Yield x));
    gen :=  (fun (*@ ensures iter_inv (old S_x) S_x result *)
                 () -> None)
with
|effect (Yield x) k ->
    gen := (fun (*@ ensures iter_inv (old S_x) S_x result *)
                 () -> l := x::!l; continue k (); Some x)
(*@ try_ensures
    forall s_old s result.
        post !gen () s_old s result ->
        iter_inv s_old s result*)
```

Figure 6.7: Handler for the `iter` function.

Since the precondition of `f` is the same as the precondition for the lambda we pass, this condition is also proven.

We may now move on to the next instruction where we set `gen` to be equal to a function that returns `None`. First, we prove that the function satisfies the `iter_inv` predicate. Since this function always returns `None`, does not modify the state and is only generated when the iteration has completed, Why3 is able to prove such condition. Next, we prove the handler's invariant when the handler terminates without an effect. Since we've stated that `gen` contains a function that satisfies `iter_inv`, this is proven trivially.

Coming up to the `Yield` case, we first prove that the precondition for `continue` (in other words, the protocol's postcondition) is met. Since the protocol's postcondition is `!l = x::(old !l)` and modify `l` by adding `x` to the head, this condition is met. Next, we prove the continuation's one-shot check. Finally, we prove that this function satisfies the invariant. This implies proving three things:

1. The result is a valid next step in the iteration

2. The `l` list has been updated with the newly yielded value

3. The `gen` function still holds a function that satisfies the `iter_inv`

The first is proved from the protocol's precondition. The second one is proven knowing we update the list with `x` before returning. The final condition is proven knowing that the continuation's postcondition (the handler's invariant) is that `gen` will hold a function that satisfies `iter_inv`. Finally, Why3 proves the handler's invariant for this case, which is proven trivially just like the base case.

6.5 Shallow Handlers

As we have mentioned in previous chapters, our translation scheme only works for deep handlers. Although it is technically possible to specify and verify programs with effects with Why3 using a similar strategy, there are a few limitations that we will go over that make this approach inviable. The case study we will present in this section is somewhat trivial, we are mostly using it to demonstrate some of the difficulties in encoding effectful continuations.

6.5.1 Extending The Grammar

Before delineating our case study, we first present how we will extend OCaml's GOSPEL's syntax to encompass shallow handlers. Like with deep handlers, OCaml does not yet have a proper syntax for these. Nevertheless, the syntax we are presenting will most likely be similar to what will be available in OCaml in the future. The basic syntax will be almost identical to deep handlers:

$$\texttt{resume}\ k,\ e\ \texttt{with}\ \overline{\texttt{effect}\ Ex\ \texttt{k} \rightarrow e}$$

There are a few differences, the most notable of which is that instead of `try`...`with` we have `resume`...`with`. Another difference is that instead of an e in between the `resume`...`with` we have k, e, a tuple consisting of a shallow continuation (that is, a continuation with no handler) and its argument. If k doesn't evaluate to a continuation or e's type is not valid, the program is rejected.

A shallow handler `resume` $k,\ e$ `with`... will evaluate e and call k using the return value of e. The rest is quite similar to what we have seen, with the exception that the continuations this handler generates will not reinstall the handler and may perform effects. Important to note that there is no $continue$ function for shallow continuations, the only way to call these is with a handler. Moreover, shallow continuations are also *one-shot*

The final difference is that these handlers do not have an invariant clause. As we explained in section 5.1.2, we needed these invariants to give the generated continuations a postcondition. However, this is unnecessary in the context of shallow handlers. This is because when we resume some continuation k_1 and generate a new continuation k_2, the postcondition of k_2 will be equivalent to k_1's. Therefore, we do not need the user to manually insert an invariant.

We will also introduce a function $fiber$ with the following type:

$$fiber\ :\ \alpha\ \rightarrow \beta \rightarrow continuation\ \alpha\ \beta$$

All this function does is turn a function into a continuation. This is necessary since to use a handler we need a continuation and, without this function, we would have no way of generating one.

Lastly, we will go over how we extend GOSPEL so we may reason over shallow continuations. To reason over the effects a function may perform, we will introduce the `may_perform` binary predicate. Stating `f may_perform E1, E2, E3` means that the function (or continuation) `f` is allowed (although doesn't strictly need) to perform effects `E1` `E2` and `E3`.

6.5.2 Shallow to Deep

We will now present a program which implements a deep handler for an effect `E` using shallow handlers. Let us consider the following annotated OCaml program:

GOSPEL + OCaml

```
effect E : unit

let rec deep (k : ('a, 'b) continuation) (arg : 'a) : 'b =
    resume k, arg with
    |effect E -> deep k ()
(*@ requires k may_perform E
    ensures post k arg result *)

let handler (f : 'a -> 'b) (arg : 'a) : 'b = deep (fiber f) arg
(*@ requires f may_perform E
    ensures post f arg result *)
```

The `deep` function is very simple: it calls `k` using `arg` and if any effects are performed, it calls itself recursively, reinstalling the handler. The `handler` function receives a normal function, turns it into a continuation and then calls `deep`.

The specifications for these two functions allow `f` and `k` to perform the effect `E`. Therefore, we are not allowed to call these functions by passing lambdas that perform other effects. Additionally, both these functions ensure the postcondition of their lambda.

We will not provide any protocol for `E` seeing as we are only interested in proving if each effect is handled correctly. This is much harder then when we were dealing with deep handlers since now we have to prove programs with higher order functions that perform effects. Additionally, since we have no state variables, we will assume the versions of $post$ without the state arguments. Since we have no protocol, we will also be dismissing the pre predicate for this example.

Before moving on to the WhyML translation, we will assume our program also has the following effect defined:

`effect Wildcard : unit` This effect will not be used by our case study and will only be here to highlight a limitation that we will discuss onwards.

6.5.3 Translation

The first step in proving a shallow handler is, similarly to their deep counterparts, turning all effects into exceptions:

WhyML
```
exception E
exception Wildcard
```

Next, we translate the deep function. We will translate shallow handlers using, much like our previous case studies, exception handlers. The transformation will be slightly different, however: we turn

GOSPEL + OCaml
```
resume k, arg with ...
```

into

WhyML
```
try continue k arg with ...
```

This way k is called with arg and its *one-shot* check is invalidated. The full translation of deep is as follows:

WhyML
```
let deep (k : continuation 'a 'b) (arg : 'a) : 'b
try contin k arg with
|E ->
    val new_k : continuation unit 'b in
    deep new_k ()
end
```

This program, aside from being incomplete (we haven't specified gen_k nor deep), is also invalid, seeing as we wrapped a call to contin, a function that throws no exceptions, with an exception handler. To fix this, we must give contin permission to throw any arbitrary exception. First, we will create a new predicate $throws$ similarly to pre and $post$.

$$throws\ f\ :\ \tau_1 \rightarrow exn \rightarrow prop$$

Where exn is the type of exception constructors. Using this predicate, we can state the conditions under which a function f may perform an effect (which we codify as exceptions in WhyML). For example, if we say

$$\forall arg\ e.\ throws_f\ arg\ e$$

We allow f to throw any exception for any argument it may receive. Using this new predicate, we would like to add the following clause to the contin function.

WhyML
```
predicate throws (lambda 'a 'b) 'a exn

val contin (k : continuation 'a 'b) (arg 'a) : 'b
    raises{e -> throws k arg e}
```

This clause states that `k` can throw a generic exception `e` as long as it satisfies the `throws` predicate This, however, is not allowed in Why3, seeing as exceptions can't be used as normal values. Exceptions in WhyML are identifiers that can only be used in three specific instances: raising an exception, catching an exception and in `raises` clauses. We cannot use them in predicates nor reason over them with in a generic way.

To get around this limitation, we will have to create our own `exn` type which will have one constructor for each effect in our program.

```
type exn = E | Wildcard
```
WhyML

Note that we still keep the exceptions we declared at the start of this section. Next, we will add the following clauses to `contin`

```
val contin (k : continuation 'a 'b) (arg : 'a) : 'b
...
    raises{E -> throws k arg E}
    raises{Wildcard -> throws k arg Wildcard}
```
WhyML

Now that `contin` throws exceptions, we can resume the translation of `deep`. Our next step is to translate the specification into WhyML. To recap, the GOSPEL specification was.

```
requires k may_perform E
ensures post k arg result
```
GOSPEL + OCaml

The postcondition is quite simple and can be translated directly. The precondition will be a bit trickier. Naturally, we will need to use our new `throws` predicate, but in a bit of an unnatural way. Since we are saying that `k` may perform `E`, we know that we cannot pass continuations that perform any effect other then `E`. Therefore, the translation of the GOSPEL `requires` clause to WhyML would be:

```
requires{forall arg e. e <> E -> not throws k arg e}
```
WhyML

Note how we didn't state that `throws k arg E` is true, since we allow passing continuations that don't perform any effects. All that is left is to specify the generated continuation. As usual, we will start by saying that the continuation is valid.

```
val new_k : continuation unit 'b in
assume{valid result};
```
WhyML

Next, we state the postcondition of the generated continuation. Since this is a shallow handler, it will simply resume where `k` left off, meaning its postcondition will be the same as `k`'s.

```
assume{forall result. post new_k () result <-> post k arg result};
```
WhyML

Finally, we say that the generated continuation throws the same exceptions as k

```
assume{forall exn. throws new_k () exn <-> throws k arg exn}
```
WhyML

```
let rec deep (k : continuation 'a 'b) (arg : 'a) : 'b =                WhyML
  requires{forall arg e. e <> E -> not throws k arg e}
  ensures{post k arg result}
    try contin k arg with
    |E ->
        val new_k : continuation unit 'b in
          assume{valid new_k};
          assume{forall arg1 result.
            post new_k arg1 result <-> post k arg result};
          assume{forall exn.
            throws new_k () exn <-> throws k arg exn};
        deep new_k ()
    end
```

Figure 6.8: Partial translation of a shallow handler.

The full program thus far can be found in figure 6.8

Although this is quite close to what we want, there are still a few unresolved issues. The easiest one to solve is the fact that, when we call `contin`, Why3 has no way to verify if the continuation is valid. To this end, we will insert the following precondition.

```
requires{valid k}                                                       WhyML
```

This should be something that Cameleer adds automatically, not something that a logician should be expected to insert into their GOSPEL specifications, since there is no reason to pass a continuation as an argument if we assume it to be invalid.

Another issue is the fact that, since `contin` may throw exceptions other than `E`, we must add an additional exceptional clause to our handler, stating that this will never happen by calling `contin` with `k`. What would be most natural is to use a wildcard pattern such as:

```
try ... with                                                            WhyML
|E -> ...
|_ -> absurd
```

As we have mentioned, nonetheless, Why3 doesn't allow using exceptions in this way, meaning we must add an exceptional branch for each known effect. In this case, since there is only one other (`Wildcard`), we simply add:

```
try ... with                                                            WhyML
|E -> ...
|Wildcard -> absurd
```

The final problem left to solve is divergence. Since this program calls itself recursively anytime there is an exception, Why3 tries to prove its termination. Intuitively, we know the program eventually terminates, since, as we have stated in 5.5, we assume all functions passed as parameters are non-divergent. Since when we call `deep` recursively the generated continuation is simply resuming `k`, we know that it is also non-divergent. However, there

WhyML
```
let rec deep (k : continuation 'a 'b) (arg : 'a) : 'b =
  requires{valid k}
  requires{forall arg e. e <> E -> not throws k arg e}
  ensures{post k arg result}
  diverges
    try contin k arg with
    |E ->
        val new_k : continuation unit 'b in
          assume{valid new_k}
          assume{ forall result.
            post new_k () result <-> post k arg result}
          assume{forall exn.
            throws new_k () exn <-> throws k arg exn}
        deep new_k ()
    |Wildcard -> absurd
    end
```

Figure 6.9: Complete translation of the `deep` function.

is no way for Why3 to piece together this proof with the encoding we are using, since there is no notion of the continuation's "progress". Therefore, we will simply add a `diverges` clause to our complete program (Fig. 6.9).

Seeing as the `handler` function's specification will be identical to `deep`, we will not go over its translation. The only aspect worth noting is the fact that it uses the `fiber` function which turns functions into continuations. In order to use this in Why3, we will use the following logical function:

WhyML
```
let function fiber (f : lambda 'a 'b) : continuation 'a 'b =
    {_k = f; _valid = true}
```

The `handler` function's specification is identical to `deep`'s, and its body is unmodified. The full program is found in figure 6.10.

6.5.4 Verification

We will first go over the verification conditions generated by `deep`. The call to `contin` generates a verification condition proving that its *one-shot* check is valid, which is trivially proven from the precondition we inserted. Next, we must prove that the postcondition holds in the case of a non-exceptional exit. Since the postcondition for `contin` is the same as `deep`'s, this is also proven trivially.

Next, the exceptional branch generates three VCs, the first two being the preconditions for the recursive call to `deep`. We must prove that the continuation we pass is valid and that it does not throw any exceptions other than `E`. The former is easily proven seeing as one of the postconditions to `gen_k` is that the generated continuation is valid. The later is proven by knowing that the continuation raises the same exceptions as the continuation. Since

```
val contin (k : continuation 'a 'b) (arg : 'a) : 'b                    WhyML
  requires{k.valid}
  ensures{post k arg result}
  ensures{not k.valid}
  writes{k.valid}
  raises{E -> throws k arg E}
  raises{Wildcard -> throws k arg Wildcard}

let function fiber (f : lambda 'a 'b) = {_k = f; valid = true}

let rec deep (k : continuation 'a 'b) (arg : 'a) : 'b =
  requires{valid k}
  requires{forall arg e. e <> E -> not throws k arg e}
  ensures{post k arg result}
  diverges
    try contin k arg with
    |E ->
        val new_k () : continuation unit 'b in
          assume{valid new_k};
          assume{forall result.
            post new_k () result <-> post k arg result};
          assume{forall exn.
            throws new_k () exn <-> throws k arg exn};
        deep new_k ()
    |Wildcard -> absurd
    end

let handler (f : lambda 'a 'b) (arg : 'a) : 'b =
  requires{forall arg e. e <> E -> not throws f arg e}
  ensures{post f arg result}
  diverges
    deep (fiber f) arg
```

Figure 6.10: Fully Translated Program.

we know that the continuation never throws any exceptions other then `E`, this condition is also met.

The final condition is that the `Wildcard` exception is never thrown. If we reach this case by calling `contin`, the following condition is met `throws k arg Wildcard`. However, since we know that, for any exception besides `E`, this predicate cannot by satisfied, it is impossible for this branch to be reached. The fact we generate a verification condition for each possible exception `contin` could throw was the main reason we chose to focus on deep handlers. If we have programs with n effects, any time `contin` would be called, n verification conditions would be generated, leading to an incredibly cumbersome proof to interpret. The only way to work around this problem would be to extend Why3's type system so that we could reason over exceptions in a more general sense.

As for the handler function, it generates a verification condition proving that `deep`'s

precondition is met and that the call to `deep` satisfies its precondition. Since these two functions have identical specifications, this is easily proven. Additionally, Why3 also generates a VC proving the continuation passed as argument is valid. This is done knowing that the continuation returned by `fiber` is valid. All of these verification conditions are proved using the Z3 prover [30]

6.6 Overview

In this chapter, we were able to show how our encoding can be used to verify several non trivial programs which encompass some of the most common use cases for effects and handlers. Although these show promise for this approach, there are still problems which hold it back from matching the level of expressiveness and modularity of other methods. The main one is the lack of support for hidden state. Not only does it prevent us from covering more complex use cases, it also means our case studies are forced into a style that only permits top level references. This leads to code that, although technically correct, is unlike real world OCaml code. Another weakness is in how we represent functions in specifications. The usage of the pre and post predicates in GOSPEL leads to specifications that are difficult to read and unnatural to write. These two points feel like the two most important to address to make this approach practical.

7

Conclusions

We developed an extension of the GOSPEL specification language that allows one to declare protocols, effect handlers and handler specifications. We also created a WhyML encoding that simulates algebraic effects and handlers without extending Why3. Finally, we developed a series of translation rules that transform GOSPEL annotated OCaml programs into the aforementioned encoding, as well as implementing these rules into the Cameleer tool.

7.1 Contributions

Extension to GOSPEL. We extended GOSPEL so users could have the expressive power to specify programs that use effects. We did this by adding a `performs` clause, which allows the user to specify what effects a function is allowed to perform. Additionally, we also added handler specifications where users can define a series of conditions that the handler must respect whenever it is exited. The most notable contribution to the GOSPEL language, however, was the addition of protocols which allow us to define the rules we must respect when performing and handling effects.

Important to remember that GOSPEL is a standalone tool (the G stands for generic), even though we mostly used it in this project in the context of Cameleer. We believe that the aforementioned additions are sufficient for modelling the behavior of algebraic effects and handlers, regardless of what context we wish to use GOSPEL.

Since the current version of OCaml 5.0 was released without an effect system, the `performs` clause is useful even when divorced from a verification context, seeing as it is a piece of helpful documentation marking which effects a function may perform. However, when we eventually have this information in the type system, this clause will be redundant and most likely deprecated, barring some fundamental shift with OCaml's approach to typing effects.

Verifying With Defunctionalization. One of the challenges we faced with this project was how we would represent continuations in a system that did not allow stateful functions

to be used as first order values. With defunctionalization, we were able not only to represent continuations, but any arbitrary higher order function. Although we had already explored this approach [40], this thesis further refined it, solving a number of limitations such as a lack of support for polymorphism and an inability to prove divergence.

Automated Verification framework. The main contribution of this thesis is the development of an automated verification framework for programs with algebraic effects and handlers. As mentioned in Chapter 3, the most common approaches for the verification of effects and handlers (and other forms of non-parallel concurrency) were done using interactive provers, namely Coq. Nevertheless, we showed how protocols, as defined by de Vilhena and Pottier [13] can be encoded within an automated proof assistant such as Why3.

We did this by presenting a translation scheme that converts programs with effects and handlers into a proof aware language without these constructs by means of an embedding using exceptions and defunctionalization. Although this translation scheme was made with GOSPEL annotated OCaml and WhyML in mind, we believe that the ideas presented are general enough that they could be applied to other automated theorem provers with a similar logic to Why3. We also showed how our translation scheme could be adapted to translate shallow handlers if Why3 could reason over exceptions in a more general way.

One limitation of our scheme stands out. Namely, we must always specify the handler if we wish for the continuations to have any meaning in the verification process. This is because neither Cameleer nor WhyML are able to automatically generate specifications for the used continuations.

Framework Implementation. In addition to the theoretical formalization of our embedding, we also implemented this translation scheme into Cameleer. We then tested this implementation (and, by extension, our translation scheme) by attempting to prove three case studies: a division interpreter, a state reference implemented with effects and a generator created from an iterator. These case studies, in our view, are not only distinct but encompass the most common use cases for effects and handlers (resumable exceptions, creation of an ambient environment and granular control of iteration). One use case we would like to have proved but were ultimately unable was a scheduler for asynchronous tasks. The main limitation that held us back was our inability to capture hidden state, which is invaluable when implementing this kind of program.

This is, in our opinion, the most important part of our work: we have shown that our translation scheme is robust enough that it can prove complex, real world OCaml programs in a completely automated fashion.

7.2 Future Work

As mentioned, one of our biggest limitations is our inability to capture hidden state. We made various attempts to axiomatize a completely general Why3 `state` type that would allow us to reason over dynamically generated state variables. In spite of our best efforts, we always ran into a brick wall that would leave us short of our goal. Although there might be some reasonably simple solution we have not considered, we are almost certain that some Why3 extension will be required to fully verify handlers.

Another limitation is the fact that our continuations cannot perform any effects. This means we cannot represent shallow handlers, nor deep handlers that don't handle all effects that `Client` may perform. This stems from a fundamental lack of generalization power that Why3 has in regards to exceptions. One possible solution would be to add extendible types [17] to Why3 and attempt to implement exceptions in that way. The only solution that does not involve an overhaul to Why3's exception system is to translate effects into some kind of monadic encoding, although such a transformation might complicate the resulting program in such a way that Why3 would be unable to verify it.

Even though we tested our approach with a series of non trivial test cases, we would like to verify even more complex examples. A good target for verification would be the *eio* library, which will likely be OCaml's standard API for concurrent code. This would show how robust our Cameleer extension is when put up against real world OCaml code. Additionally, it would also be a significant contribution to the OCaml community to verify a library that will likely be very popular in the future.

Finally, we believe that the `pre` and `post` predicates, although invaluable in our WhyML encoding, might be too low level for a language like GOSPEL. In the future, we would like to be able to nest GOSPEL specifications as follows.

GOSPEL + OCaml

```
let f (g : int -> int) ...
(*@ requires forall arg r.
        (*@ r = g a
            requires ...
            ensures r = 0 *)
    ensures ... *)
```

This way, we could state the function that `f` can only receive functions that return 0. Naturally, we would then create a translation scheme that would turn these nested specifications into an equivalent representation using `pre` and `post`

Bibliography

[1] *2020 Developer survey*. https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-languages-wanted. Accessed: 2022-1-15 (cit. on p. 4).

[2] Avsm et al. *Multicore ocaml: September 2021, Effect Handlers Will be in ocaml 5.0!* 2021-10. URL: https://discuss.ocaml.org/t/multicore-ocaml-september-2021-effect-handlers-will-be-in-ocaml-5-0/8554 (cit. on pp. 10, 21).

[3] A. Bauer and M. Pretnar. "An Effect System for Algebraic Effects and Handlers". In: *Algebra and Coalgebra in Computer Science*. Ed. by R. Heckel and S. Milius. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–16. ISBN: 978-3-642-40206-7 (cit. on p. 20).

[4] A. Bauer and M. 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 (cit. on p. 2).

[5] F. Bobot et al. "Why3: Shepherd Your Herd of Provers". In: *Boogie 2011: First International Workshop on Intermediate Verification Languages* (2012-05) (cit. on p. 2).

[6] J. I. Brachthäuser, P. Schuster, and K. Ostermann. "Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism". In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020-11). DOI: 10.1145/3428194. URL: https://doi.org/10.1145/3428194 (cit. on p. 2).

[7] A. Charguéraud. "Characteristic Formulae for the Verification of Imperative Programs". In: 46.9 (2011-09), pp. 418–430. ISSN: 0362-1340. URL: https://doi.org/10.1145/2034574.2034828 (cit. on p. 8).

[8] A. Charguéraud. "Separation Logic for Sequential Programs (Functional Pearl)". In: *Proc. ACM Program. Lang.* 4.ICFP (2020-08). DOI: 10.1145/3408998. URL: https://doi.org/10.1145/3408998 (cit. on p. 6).

[9] A. Charguéraud and F. Pottier. "Temporary Read-Only Permissions for Separation Logic". In: *Programming Languages and Systems*. Ed. by H. Yang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 260–286. ISBN: 978-3-662-54434-1 (cit. on p. 7).

[10] A. Charguéraud et al. "GOSPEL — Providing OCaml with a Formal Specification Language". In: *Formal Methods - The Next 30 Years - Third World Congress*. Vol. 11800. Lecture Notes in Computer Science. Springer, 2019, pp. 484–501. URL: 10.1007/978-3-030-30942-8%5C_29 (cit. on pp. 2, 7, 8).

[11] *Companies using OCaml*. https://ocaml.org/learn/companies.html. Accessed: 2022-1-15 (cit. on p. 4).

[12] L. Convent et al. "Doo bee doo bee doo". In: *Journal of Functional Programming* 30 (2020-03). DOI: 10.1017/S0956796820000039 (cit. on p. 2).

[13] P. E. de Vilhena and F. Pottier. "A Separation Logic for Effect Handlers". In: *Proc. ACM Program. Lang.* 5.POPL (2021-01). DOI: 10.1145/3434314. URL: https://doi.org/10.1145/3434314 (cit. on pp. 2, 22, 23, 81).

[14] Dijkstra, Edsger W. "The humble programmer". In: *Commun. ACM* 15.10 (1972), pp. 859–866 (cit. on p. 1).

[15] *Effective programming: Adding an effect system to OCAML*. URL: https://www.janestreet.com/tech-talks/effective-programming/ (cit. on p. 20).

[16] J.-C. Filliâtre. "Deductive Software Verification". In: *International Journal on Software Tools for Technology Transfer (STTT)* 13.5 (2011-08), pp. 397–403. ISSN: 1433-2779. URL: 10.1007/s10009-011-0211-0 (cit. on p. 1).

[17] B. R. Gaster and M. P. Jones. *A polymorphic type system for extensible records and variants*. Tech. rep. Citeseer, 1996 (cit. on p. 82).

[18] J. Hickey, A. Madhavapeddy, and Y. Minsky. *Real World OCaml*. 2014. ISBN: 144932391. URL: http://www.worldcat.org/isbn/144932391 (cit. on p. 4).

[19] D. Hillström, S. Lindely, and R. Atkey. "Effect handlers via generalised continuations". In: *Journal of Functional Programming* 30 (2020), e5. DOI: 10.1017/S0956796820000040 (cit. on p. 2).

[20] J. K. Hinrichsen, J. Bengtson, and R. Krebbers. "Actris: Session-Type Based Reasoning in Separation Logic". In: *Proc. ACM Program. Lang.* 4.POPL (2019-12). DOI: 10.1145/3371074. URL: https://doi.org/10.1145/3371074 (cit. on p. 21).

[21] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (1969-10), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: https://doi.org/10.1145/363235.363259 (cit. on p. 5).

[22] M. Iguernelala. "Strengthening the Heart of an SMT-Solver: Design and Implementation of Efficient Decision Procedures". Thèse de Doctorat. Université Paris-Sud, 2013-06 (cit. on pp. 31, 61).

[23] R. Jung et al. "Iris From the Ground Up: A Modular Foundation For Higher-order Concurrent Separation Logic". In: *Journal of Functional Programming* 28.e20 (2018). DOI: 10.1017/S0956796818000151. URL: https://hal.archives-ouvertes.fr/hal-01945446 (cit. on pp. 8, 21).

[24] O. Kammar, S. Lindley, and N. Oury. "Handlers in Action". In: *SIGPLAN Not.* 48.9 (2013-09), pp. 145–158. ISSN: 0362-1340. DOI: 10.1145/2544174.2500590. URL: https://doi.org/10.1145/2544174.2500590 (cit. on p. 11).

[25] D. Kroening and O. Strichman. *Decision Procedures - An Algorithmic Point of View, Second Edition*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016. ISBN: 978-3-662-50496-3. DOI: 10.1007/978-3-662-50497-0. URL: https://doi.org/10.1007/978-3-662-50497-0 (cit. on p. 1).

[26] D. Leijen. "Koka: Programming with Row Polymorphic Effect Types". In: *Electronic Proceedings in Theoretical Computer Science* 153 (2014-06). DOI: 10.4204/EPTCS.153.8 (cit. on pp. 2, 20).

[27] R. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *16th International Conference, LPAR-16, Dakar, Senegal*. Springer Berlin Heidelberg, 2010-04, pp. 348–370. URL: https://www.microsoft.com/en-us/research/publication/dafny-automatic-program-verifier-functional-correctness-2/ (cit. on p. 5).

[28] T. Letan et al. "Modular Verification of Programs with Effects and Effect Handlers in Coq". In: *FM 2018 - 22nd International Symposium on Formal Methods*. Vol. 10951. LNCS. Oxford, United Kingdom: Springer, 2018-07, pp. 338–354. DOI: 10.1007/978-3-319-95582-7_20. URL: https://hal.inria.fr/hal-01799712 (cit. on p. 21).

[29] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (cit. on pp. ii, iv).

[30] L. M. de Moura and N. Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Ed. by C. R. Ramakrishnan and J. Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24. URL: https://doi.org/10.1007/978-3-540-78800-3%5C_24 (cit. on p. 79).

[31] M. Pereira and A. Ravara. "Cameleer: A Deductive Verification Tool for OCaml". In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*. Ed. by A. Silva and K. R. M. Leino. Vol. 12760. Lecture Notes in Computer Science. Springer, 2021, pp. 677–689. DOI: 10.1007/978-3-030-81688-9_31 (cit. on p. 2).

[32] M. J. P. Pereira. "Tools and Techniques for the Verification of Modular Stateful Code". PhD thesis. University of Paris-Saclay, France, 2018. URL: https://tel.archives-ouvertes.fr/tel-01980343 (cit. on p. 8).

[33] G. Plotkin and J. Power. "Algebraic Operations and Generic Effects". In: *Applied Categorical Structures* 11 (2003-02), pp. 69–94. DOI: 10.1023/A:1023064908962 (cit. on pp. 1, 9).

[34] G. Plotkin and J. Power. "Computational Effects and Operations: An Overview". In: *Electronic Notes in Theoretical Computer Science* 73 (2004). Proceedings of the Workshop on Domains VI, pp. 149–163. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2004.08.008. URL: https://www.sciencedirect.com/science/article/pii/S1571066104050893 (cit. on p. 21).

[35] G. Plotkin and M. Pretnar. "A Logic for Algebraic Effects". In: *Proceedings of the 2008 23rd Annual IEEE Symposium on Logic in Computer Science*. LICS '08. USA: IEEE Computer Society, 2008, pp. 118–129. ISBN: 9780769531830. DOI: 10.1109/LICS.2008.45. URL: https://doi.org/10.1109/LICS.2008.45 (cit. on p. 21).

[36] G. Plotkin and M. Pretnar. "Handlers of Algebraic Effects". In: *Programming Languages and Systems*. Ed. by G. Castagna. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 80–94. ISBN: 978-3-642-00590-9 (cit. on p. 21).

[37] J. C. Reynolds. "Definitional Interpreters for Higher-Order Programming Languages". In: *Proceedings of the ACM Annual Conference - Volume 2*. ACM '72. Boston, Massachusetts, USA: Association for Computing Machinery, 1972, pp. 717–740. ISBN: 9781450374927. DOI: 10.1145/800194.805852. URL: https://doi.org/10.1145/800194.805852 (cit. on pp. 2, 27).

[38] K. Sivaramakrishnan et al. "Retrofitting Effect Handlers onto OCaml". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 206–221. ISBN: 9781450383912. DOI: 10.1145/3453483.3454039. URL: https://doi.org/10.1145/3453483.3454039 (cit. on pp. 2, 16, 17).

[39] J. Smans, B. Jacobs, and F. Piessens. "Implicit Dynamic Frames". In: *ACM Trans. Program. Lang. Syst.* 34.1 (2012-05). ISSN: 0164-0925. DOI: 10.1145/2160910.2160911. URL: https://doi.org/10.1145/2160910.2160911 (cit. on pp. 5, 8).

[40] T. Soares and M. Pereira. "Verificação de Programas OCaml Imperativos de Ordem Superior, através de Desfuncionalização". In: *INForum 2021*. 2021-09. URL: INForum_2020_paper_45.pdf (cit. on pp. 2, 27, 81).

[41] A. Timany and L. Birkedal. “Mechanized Relational Verification of Concurrent Programs with Continuations”. In: *Proc. ACM Program. Lang.* 3.ICFP (2019-07). DOI: 10.1145/3341709. URL: https://doi.org/10.1145/3341709 (cit. on p. 21).

[42] L.-y. Xia et al. “Interaction Trees: Representing Recursive and Impure Programs in Coq”. In: *Proc. ACM Program. Lang.* 4.POPL (2019-12). DOI: 10.1145/3371119. URL: https://doi.org/10.1145/3371119 (cit. on p. 21).

2022 Handle with Care and Confidence – Extending Cameleer with Algebraic Effects and Effect Handlers Tiago Soares

NOVA
NOVA SCHOOL OF SCIENCE & TECHNOLOGY