NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

ION CHIRICA

BSc in Computer Science

UNFOLDING ITERATORS

SPECIFICATION AND VERIFICATION OF HIGHER-ORDER ITERATORS, IN OCAML

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
September, 2024

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

UNFOLDING ITERATORS

SPECIFICATION AND VERIFICATION OF HIGHER-ORDER ITERATORS, IN OCAML

ION CHIRICA

BSc in Computer Science

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

Examination Committee

Chair: João Carlos Antunes Leitão
Associate Professor, School of Science and Technology
NOVA University Lisbon
Rapporteur: Jean-Christophe Filliâtre
Principal Investigator,
Université Paris-Saclay
Adviser: Mário José Parreira Pereira
Assistant Professor, School of Science and Technology
NOVA University Lisbon

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
September, 2024

Unfolding Iterators
Specification and Verification of Higher-Order Iterators, in OCaml

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.1.7) [23].

Acknowledgements

To whom this work would not be, in the slightest, possible: Mário, I cannot express to the full extent my gratitude in words. You took me under your wing and welcomed me into this fascinating field. Throughout this last year, there was no moment of doubt that I had made the right choice to work with you. For all the moments of self-doubt that you quickly rubbed off, for constantly pushing me to be the best version of myself, and for all the encouragement, patience, and unprecedented availability to reply to my foolish questions. For all the teachings and those to come. To the inhabitants of the π lab: who have made this last year and so more tolerable. To Tiago for being the best unofficial co-advisor. To Ana, who, even in her bad days, always pushed me to work and do the best I could. To my friends: be they from my past life or my newest. To Clube de Leitura who have made me laugh more than I could handle and for all the great moments. To Jaime Raimundo and Miguel Oliveira, my oldest friends, who serve me a great deal of inspiration. To Miguel Santos for always being available to hear me brainstorm. To Mariana Alves for always matching my humor. To Daniel Cavalheiro, Hugo Pereira, and Vasco Ramos for always being there when I needed it. To my family: whom they do not understand what I am doing but still support me with everything. To my father, who shaped me to be the man I am today. To my mother for her unconditional love and support. To my sister, what a woman you turned out to be. To my grandmother, who once said that an idiot is someone who has a lot of ideas. And to my feline companions. To the members of the Gospel project for their availability and great suggestions that have made this work possible. To Jean-Christophe Filliâtre who agreed to be the rapporteur of this thesis. Finally, to those who wish me wrong, as though I am a fallacy, I can make things right.

Mulţumesc! Muito obrigado! Thank you! Miau! Merci!

”

“Pensez-vous que l’hiver sera rude?”

— Mr. Fox, upon seeing a wolf

Abstract

When it comes to software development, programmers find themselves hardly implementing anything from scratch, relying on internal or third-party libraries with pre-written code. Besides providing genericity, abstraction, and performant features, by encapsulating everything in a library, we are more keen to standardize code that has been formally proved correct.

This work aims to formally prove a subset of the OCamlGraph library, with special concern on algorithms that employ higher-order iteration. By asserting the correctness of its algorithms, its users can feel safer knowing that the library is not error-inducing. We will base ourselves on GOSPEL specifications that can be consumed by the verification framework Cameleer. As most graph algorithms in the OCamlGraph library employ some sort of higher-ordered iteration, we seek to answer the question: *"How to soundly and reliably formally verify implementations and clients of OCaml higher-order iteration, using mostly automated proof tools?"*.

In this document, we outline some theoretical and practical background concerning deductive verification in the functional paradigm and available techniques for specifying and verifying higher-order iteration. We also present our methodology for the specification and verification of higher-order iterators in OCaml using GOSPEL specifications. We complement this methodology with a collection of case studies that sustain our work.

Keywords: Formal Verification, Higher-Order Iteration, Graphs, OCaml, Why3, GOSPEL, Cameleer

Resumo

No que toca ao desenvolvimento de software, os programadores raramente implementam muita coisa de raiz, dando uso a bibliotecas, quer internas ou de terceiros, com código pré-escrito. Para além da genericidade, da abstracção, e de componentes com um alto desempenho que costumam oferecer, a encapsulação de código numa biblioteca permite a sua normalização, se este for provado correto.

Este trabalho propõe verificar formalmente um subconjunto da biblioteca OCamlGraph, com especial interesse em algoritmos que empreguem iteração de ordem superior. Ao afirmarmos a correção dos seus algoritmos, os seus utilizadores podem sentir-se mais confiantes ao saber que, pelo menos, a biblioteca não induzirá em erros. O trabalho vai ser baseado em especificações na linguagem GOSPEL que, por sua vez, serão consumidas pela plataforma de verificação Cameleer. A observação de que grande parte dos algoritmos da biblioteca OCamlGraph utiliza iteração de ordem superior, leva-nos a procurar responder à questão: *"Como verificar formalmente, de modo completo e fiável, implementações e o consumo da iteração de ordem superior em OCaml, recorrendo a ferramentas de prova automática?"*.

Neste documento, vamos abordar algum fundamento teórico e prático no que diz respeito a verificação dedutiva no paradigma de programação funcional e algumas técnicas para especificação e verificação de iteração de ordem superior. Apresentamos ainda a nossa metodologia para a especificação e verificação destes mesmos iteradores, em OCaml, com o uso de especificaçãos na linguagem GOSPEL. Complementamos a nossa metodologia com uma coleção de casos de estudos que sustêm o nosso trabalho.

Palavras-chave: Verificação Formal, Iteração de Ordem Superior, Grafos, OCaml, Why3, GOSPEL, Cameleer

Contents

Glossary

Alt-Ergo SMT-based theorem prover supporting quantifiers, polymorphic sorts, and various theories including equality, linear and non-linear arithmetic over integers or rational numbers, arrays, records, enumerated types. *(pp. 11, 12, 46, 49, 58)*

Cameleer A deductive verification tool for OCaml programs. *(pp. vii, ix, 2, 3, 11, 12, 25, 36, 39–42, 44, 46, 65)*

CFML Tool for the interactive verification of OCaml programs, using Coq and Separation Logic. *(p. 26)*

CVC4 SMT solver supporting quantifiers and many theories including equality, arithmetic, datatypes, bitvectors. *(p. 11)*

GOSPEL Generic OCaml SPEcification Language - a behavioral specification language for OCaml. *(pp. vii, ix, 2, 3, 10–12, 23, 25, 33, 36–42, 44–62, 65, 66)*

OCamlGraph A generic graph library for OCaml. *(pp. vii, ix, xi, 2, 8, 15, 16, 25, 49, 58, 65, 66)*

Why3 A platform for deductive program verification. *(pp. vii, ix, xiii, 12–14, 23, 28, 29, 34, 35, 42, 44, 46, 48, 51, 55, 65)*

WhyML Programming language used to prove programs in Why3. *(pp. 2, 11–14, 33–36, 40–44, 63, 65)*

Z3 Theorem prover from Microsoft Research. SMT solver supporting quantifiers and many theories including equality, arithmetic, datatypes, bitvectors *(p. 11)*

1

Introduction

1.1 Motivation

The innate human tendency to make ~~erros~~ errors is unavoidable. Human error manifests in various forms and can have different effects in different fields. We cannot think of comparing the impact of a mistake made by a doctor during a critical surgery to that of a romanticist forgetting a comma in a novel. In the medical field, an oversight or misstep during surgery can have life-altering consequences for the patient. Meanwhile, imperfections may not have immediate life-or-death repercussions in the domains of literature and art. Still, they can substantially impact how people perceive and interpret the work. In Computer Science, we enroll as both artists and doctors; we take pride in writing elegant and concise pieces of software but also be prudent that its flaws can be razor-sharp. In this work, we will tie ourselves to flaws that condemn our software to produce incorrect results.

Indisputably, the most common way of checking if a program produces incorrect results is by testing. Although fast and cheap, it remains a mere way of showing what works and alone should not be the foundation of an argument for correctness. Testing can provide full coverage; however, this is only observed in code with low cyclomatic complexity [24]. In reality, code tends to be very complex, and exhaustive testing can be not only cumbersome but also unfeasible.

If our goal is a rigorous argument of correctness, we must resort to deductive verification [15]. It is an ambitious alternative where we transform the correctness of a program into a mathematical statement and then prove it, resorting to a computer. From a practical standpoint, one must conceive a logic specification, which is nothing short of a mathematical program description. In conjunction with this description, the program is transformed into a mathematical statement, coined Verification Condition, expressing that the implementation is per the mathematical model. Finally, the Verification Condition is fed to Theorem Provers, tasked with proving the formula's validity.

Let us consider, for instance, Graph Theory. Although this is a well-founded and deeply researched mathematical field, some software implementations might not be loyal

to their theoretical model through some overlooked detail or misinterpretation. Moreover, its applications are ubiquitous in almost every other field of Computer Science. As foolish as it may sound, we can represent nearly any problem and structure as a graph. We usually find these implementations in libraries, collections of pre-written pieces of code that provide the functionality to be used by other programs.

The world of functional programming languages has many of these libraries [12, 8, 25]. Notably, in OCaml, an industrial-grade functional language, we find OCamlGraph [26, 8], a graph library providing innumerable algorithms and types of graphs.

1.2 Problem Definition

This work focuses on formally verifying a subset of the OCamlGraph library to tame eventual vulnerabilities, derived from incorrect implementations, its users might be exposed to. Furthermore, as most of the algorithms defined in the OCamlGraph library employ higher-ordered iteration functions we should first aim to answer the following questions:

1. *How to soundly and reliably specify higher-order iteration idioms written in OCaml?*
2. *How to soundly and reliably formally verify implementations and clients of OCaml higher order-iteration, using mostly-automated proof tools?*

1.3 Goals and Contributions

In this work, we seek to generically specify and verify higher-order iterators in OCaml. To that effect, we will use GOSPEL [7] specifications to define what an iteration is and what is its behaviour. This in turn can be used by Cameleer [29] to translate into a WhyML [17] program that can be deductively verified. Our contributions are the following:

- Extend GOSPEL to support higher-order iteration specifications.
- Extend Cameleer to translate the higher-order iteration into a first-order iteration using cursor clients.

The results of this work were partially presented in two peer-reviewed conferences:

- Ion Chirica and Mário Pereira with *Unfolding Iterators: Especificação e Verificação de Iteradores de Ordem Superior, em OCaml*
  INForum 2024, the Portuguese Symposium in Computer Science (awarded **Best Student Paper** in the Software Science track).
- Tiago Soares, Ion Chirica and Mário Pereira with *Static and Dynamic Verification of OCaml Programs: The GOSPEL Ecosystem* [33]
  ISoLA 2024, the International Symposium on Leveraging Applications of Formal Methods, Verification and Validation.

1.4 Report Structure

- Chapter 2 covers the necessary background to understand the problem at hand, such as a theory overview on formal verification, an audit on the OCaml language, on the tools that will be used, and the target library of our verification.
- In Chapter 3, we will explore some existing work on library verification using Cameleer and foundational work towards the specification of iteration.
- In Chapter 4 we present our methodology to specify and verify higher-order iterators in OCaml, using GOSPEL specifications. This chapter presents an in-depth explanation on the extensions to the GOSPEL language and Cameleer tool.
- In Chapter 5 we find some case studies that sustain our work. Namely, the specification and verification of higher-order iterators over *sequences*, *trees* and *graphs*.
- Chapter 6 concludes with some closing remarks and possible future work.

1.5 Conventions

The following document is based on a few presentation conventions.

Code listings. Throughout this document, the reader will stumble across an unorthodox way to present listings. This was done on purpose. The listings are not numbered nor accounted for. We wish to provide a linear reading experience being possible to follow the code presented with intermediate text explaining it, similar to the classical approach of literate programming. The environment can be found on the top right corner of a listing. Down below, we find a piece of OCaml code with an intriguing function.

OCaml

```
let rec quick_sort l =
  match l with
  | [] → []
  | x::xs →
    let smaller, larger = List.partition (fun y → y < x) xs in
    let x = (quick_sort smaller) and y = (x::quick_sort larger) in
    x @ y
```

During most code excerpts the reader is able, and encouraged, to ignore the text and consider reassembling a program in the corresponding environment.

Sequences. We use a horizontal bar to represent sequences of elements. For instance, $\overline{\alpha}$ represents a sequence from α_1 to α_n.

This page is intentionally left blank.

2

Background

2.1 Formal Verification

The idea of deductive verification, where we aim to express the correctness of a program as a mathematical statement [15] and then prove it, is as old as Computer Science itself [35]. Still, with the more recent progress in automated theorem provers, we can formally prove programs more efficiently and concisely [14]. In this section, we will lay down some theoretical background that has made the field reach that point.

2.1.1 Hoare Logic

In deductive verification, we generally aim to express the correctness of a program as a set of mathematical statements. For which we quickly turn to Hoare Logic [19], whose goal is to provide logical system reason about the behaviour of a program. It is based on contracts, otherwise called Hoare Triples, that can be expressed using the notion of pre- and post-conditions. A Hoare Triple, noted as $\{P\}\ \mathrm{C}\ \{Q\}$, is deemed valid:

> *If program C starts in a state that satisfies P then, if it terminates, the resulting state satisfies Q.*

This relation is also called *partial correctness* where termination is not guaranteed, however if it does we deem it correct. Additionally, Hoare Logic also provides a set of syntactic proof rules that specify how to build valid Hoare triples. One of such rules is called *assignment rule*:

$$\frac{}{\{A\{\mathrm{x}/E\}\}\ \mathrm{x} := E\ \{A\}}\ \text{assignment}$$

This is the backward reasoning of the assignment rule, where $A\{\mathrm{x}/E\}$ states that x, a variable, is replaced by the value of the expression E. An informal proof of its soundness can be expressed as follows:

Let s be the state of our pre-condition and s′ of our post-condition. So, $\mathsf{s}' = \mathsf{s}\{\mathrm{x} \mapsto E\}$, assuming E has no side-effect. We also know that $A\{\mathrm{x}/E\}$ holds in s if and only if A holds in s′, because:

1. Every variable, except x, has the same value in s and s'.
2. $A\{x/E\}$ has every x in A replaced by E.
3. A has every x evaluated to E in s, via $s' = s\{x \mapsto E\}$.

Innocently enough, the constraint we made about E not having any side-effect renders Hoare Logic practically unusable when reasoning about programs that access and mutate data structures or in concurrent environments.

2.1.2 Separation Logic

Building on the foundations of Hoare Logic, Separation Logic [31] extends Hoare triples with the notion of *separation conjunction*. A triple of the form $\{P_1 \star P_2\}\ C\ \{Q_1 \star Q_2\}$ is valid if a program C is executed in a state where P_1 and P_2 are valid on *separated* portions of the memory, and the execution terminates in a state where Q_1 and Q_2 are valid conditions on *separated* portions of memory. Memory access assertions $L \mapsto V$, in Separation Logic, hold in the state where the memory location denoted by L contains the value denoted by V. The underlying model assumes that dynamic memory is a set of memory locations (L) storing contents (V) [34]. The *assignment rule* in separation logic is as follows:

$$\frac{}{\{x \mapsto V\}\ \ x := E\ \{x \mapsto E\}}\ \text{assignment}$$

Atop of being more straightforward, it is significantly more efficient, as the pre-condition now refers precisely to the part of the memory used by the fragment, following the *frame rule*. This rule allows us to preserve information about the "rest of the world" and locally reason about the effects of a program that only manipulates a given piece of state.

$$\frac{\{A\}\ P\ \{B\}}{\{A \star C\}\ P\ \{B \star C\}}\ \text{frame rule}$$

Moreover, there is no need to specify what is modified, seeing that it is clear from the pre-condition A, neither what is left unchanged, framed away, as in C. Additionally, and most importantly, this allows us to safely reason about memory where aliasing is not possible, as memory is either dis-jointly captured in A or unchanged in C.

2.2 OCaml

Over the last few years, functional languages have been on a notorious rise in popularity, sparked by many popular languages, namely Rust, adopting key features such as iterators and closures onto their own. The misconception that functional programming languages are a niche to academic research still haunts the common programmer to a point of despise. Furthermore, its steep learning curve and small footprint in the industry only seem to serve as fuel.

OCaml is one success story among many others. It began as an academic research project and is now a general-purpose, industrial-grade functional programming language [1]. Although less popular than other languages such as C or Java, mostly due to the different nature of the paradigms, OCaml is stable and mature enough to be used in industry. The best example is Jane Street, a top Wall Street firm focused on developing critical mathematical models and algorithms to make trading decisions, which has been actively using OCaml as its preferred language.

2.2.1 Language overview

Akin to most introductions to functional programming, we will use the classic function to compute the *n-th* element in the *Fibonacci* sequence. OCaml shines on simplicity here as it differs no more than a few keywords from the mathematical definition of the recursive function.

OCaml

```
let rec fib n =
  if   n = 0 || n = 1 then n
  else                     fib (n - 1) + fib (n - 2)
```

More often than not, recursive functions go over the same computation multiple times until they reach a base case, and towards being more performant, we usually rely on *memoization*, a technique used to cache values that are repeatedly re-computed; we first check a look-up table for a target value and either return the previously stored result or take steps in computing and store it for future use. If we adapt our previous example to include memoization now, we observe that the underlying implementation of the function was left untouched; however, it is now encapsulated with our definition of memoization.

OCaml

```
let fib_memoized n =
  let memo = Array.make ( n + 1 ) None in
  let rec fib n =
    match memo.(n) with
    | Some result → result    (* previously cached value *)
    | None →                  (* unseen value *)
      let result =            (* definition of the fibonacci function *)
        if   n = 0 || n = 1 then n
        else                     fib (n - 1) + fib (n - 2)
      in
      memo.(n) ← Some result; (* store newly computed value *)
      result                  (* also return it *)
  in
  fib n
```

2.2.2 Modules and Functors

Modular programming is an approach to software design that emphasises separating a program into independent, interchangeable modules, such as functions and classes. We can achieve modular programming in OCaml by using modules containing values, functions, and types. They can be structured using features such as structures (`struct`), signatures (`sig`), and functors. The module system in OCaml is designed to support parametric, strongly typed, and separately compilable programming, allowing for code reuse and abstraction over types and values [18, 22].

Functors, on the other hand, are modules parameterised by other modules, allowing parametrization of types by value, something not directly possible in OCaml without functors. The best way to see how modular programming can be used in OCaml is to see how it is employed in our subject of proof. Let us present a summary introduction to modules and functors adapted from [8], which will help understand their relevancy and shed some light on how modular and generic the OCamlGraph library is.

Modules can be defined with the `struct...end` construct and the optional `module` binding to give them a name. Externally, components from modules can be accessed via dot notation: `M.c`, where `M` is a module and `c` a component [8]. For example, a module packing together a type for a graph data structure and some operations can be expressed in the following way:

OCaml

```
module Graph = struct
  type label = int                    (* edge label *)
  type t = (int * label) list array   (* adjacency list *)

  let create n = Array.create n []
  let add_edge g v1 v2 l = g.(v1) ← (v2,l) :: g.(v1)
  let iter_succ g f v = List.iter f g.(v)
end
```

On the other hand, signatures are the type of a module and can be used to hide implementation details. Signatures can be defined using the `sig...end` construct, and the optional `module type` to give them name. A signature for the previous example could be the following:

OCaml

```
module type GRAPH = sig
  type label
  type t

  val create : int → t
  val add_edge : t → int → int → label → unit
  val iter_succ: t → (int * label → unit) → int → unit
end
```

We can define modules parameterised by other modules through the module system's functions. Then, they can be applied once or several times to specific modules with the expected signature. For example, we can implement Dijkstra's shortest path algorithm for any graph implementation where edges are labelled with integers.

OCaml
```
module type S = sig
  type label
  type t
  val iter_succ: t → (int * label → unit) → int → unit
end
module Dijkstra (G: S with type label = int ) =
struct
  let dijkstra g v1 v2 = (* ... *)
end
```

The `with type` annotation is used to indicate that the `label`'s type is instantiated with type `int`. We can also notice that the signature `S` contains only what is necessary to implement the algorithm. Although any other signature could have been used, requiring only to be a subtype of `S`.

2.2.3 Higher Order Iterators

Iterators in modern programming languages provide a sweet way to iterate, usually in an exhaustive manner[1], over the elements of an abstract collection, provided by the data structure [13]. Some iterators are of higher-order, taking closures or functions as a parameter. A well-known example is the `fold` iterator, which folds every element of a collection c, by applying a function f, accumulating the result acc, into a single value. Iteration using `fold`s can be done from left to right, using a `fold_left`, or inversely using a `fold_right` conceived, respectively, as:

$$f\ (\ldots\ (f\ (f\ \text{acc}\ c_1)\ c_2)\ \ldots)\ c_n \qquad f\ c_1\ (f\ c_2\ (\ldots\ (f\ c_n\ \text{acc})\ \ldots))$$

Implementing the `fold_left` function for a `list`, and more generally for any collection, is usually defined via recursion, with the base case indicating termination and the recursive branch applying a function to the accumulator and an element of the collection.

OCaml
```
let rec fold_left f acc = function
| []     → acc                      (* exhausted the collection *)
| h :: t → fold_left f (f acc h) t  (* recursively fold the collection *)
```

In OCaml, if we wish to sum all the elements of a `list`, assuming an adequate summation function of the data-type, we could do so by using a `fold` iterator.

OCaml
```
let sum list = List.fold_left (+) 0 list
```

[1] Early-stopping is unusual but can be achieved. For example, in OCaml, by raising an exception.

We can take a step further and define a more elaborate example, where we now wish to do the same sum of elements, however now over a *tree* algebraic data-type (ADT).

OCaml
```
type α tree =
| Leaf
| Node of α tree * α * α tree
```

To define a `fold_tree`, we should employ the same logic as previously shown; there is a need for a base case where we reached a `Leaf` indicating exhaustion and another which recursively applies an argument function to both the left and right branches of the tree.

OCaml
```
let rec fold_tree f acc = function
| Leaf           → acc
| Node (l, x, r) → let left  = fold_tree f acc l in
                   let right = fold_tree f acc r in f x left right
```

In the end, we are left with an iterator that can apply a function f to our very own tree, and in the spirit of following our previous example we can get the sum of a tree by supplying the adequate sum function.

OCaml
```
let sum tree = fold_tree (fun x l r → x + l + r) 0 tree
```

Two other iterators of special interest are `iter` and `map` . At their core, both functions apply a function, *f*, individually to every element in a collection *c*. However, `map` collects the results into the same type of the input collection. Their implementation is the following:

OCaml
```
let rec iter f = function
| []   → ()              (* base case, return nothing *)
| a :: l → f a; iter f l  (* apply f to head; recurse on the tail *)

let rec map f = function
| []     → []            (* base case, return empty list *)
| h :: t → let r = f h in (* evaluate (f h) *)
           r :: map f t   (* append r to the recursive call *)
```

Although one can conceive functions with a more elaborate behaviour for iteration, the functions before presented suffice our needs for what is to come, graph iterations.

2.3 GOSPEL & Cameleer

GOSPEL (Generic Ocaml SPEcification Language) is a behavioural specification language for OCaml interfaces [7]. It is a contract-based, strongly typed language with formal semantics defined employing translation into Separation Logic. The main goal of GOSPEL is to provide a concise and accessible specification language for OCaml interfaces, which can be used for various purposes. However, with a strong emphasis on verification with the use of tools that translate GOSPEL annotated OCaml into languages understandable by

automated theorem provers. GOSPEL specifications are added as comments beginning with `@`, `(*@ ... *)`, at the end of the function definition.

Since GOSPEL only provides means of specifying OCaml code, it can only take us so far as to formally document our code. To use them in a meaningful way, we must resort to external verifiers, such as Cameleer [29]. This tool takes GOSPEL annotated OCaml code and translates it to an equivalent WhyML [17] representation, which can be verified using automated theorem provers, such as CVC4, Alt-Ergo, or Z3, among many others.

2.3.1 Example

Recalling the *Fibonacci* function presented in Section 2.2.1, we might be quick to regard it as correct and not question what could go wrong. However, the function is only defined for non-negative arguments, and failing to comply will incur in an infinite recursion. We can specify this pre-condition in GOSPEL with a `requires` clause. Furthermore, to prove that the recursion converges to a halting state, we must provide a `variant`, a value that strictly decreases at each recursive call and has a lower bound. In our case, the argument n of the function is either decremented by 1 or 2 and is lower-bounded by the pre-condition, free of charge. To instruct Cameleer to consider `fib` a logical function, we add the OCaml attribute `[@logic]`, allowing us to use it in specification clauses and regular OCaml code.

GOSPEL + OCaml

```
let [@logic] rec fib n =
  if   n = 0 || n = 1 then n
  else                     fib (n - 1) + fib (n - 2)
(*@ requires n ≥ 0
    variant  n *)
```

Based on our observations of the inefficiency of *Fibonacci* function, we can move onto prove that its memoized and more efficient version, from Section 2.2.1, will still yield the same results as the purely recursive form. Seeing that the memoized function, `mem_fib`, in itself has another recursive function, we must provide suitable contracts for both functions. We can sufficiently define the contract for the inner function as follows:

- Ⓐ State that the strictly decreasing value is still the function's argument n.
- Ⓑ Require that n can only assume values in the range of $[0, ||\texttt{mem}||)$, taking care of `IndexOutOfBounds` errors.
- Ⓒ Require that any value `mem.(i)` in the array is either `None`, in the case of uncomputed values, or `Some (fib i)`, the *i-th Fibonacci* number.
- Ⓓ Ensure that the previous statement remains true at a function exit.
- Ⓔ Ensure the function yields the same result as a call to the logic function *fib n*.

Furthermore, concerning the correctness of the `mem_fib` function, we must also provide it a fitting contract: Ⓕ requiring that the function argument n is a positive integer, and Ⓖ ensuring that its result yields the same as a call to the logical function *fib n*.

GOSPEL + OCaml

```
let mem_fib n =
  let mem = Array.make (n + 1) None in
  let rec fib n =
    match mem.(n) with
    | Some result → result
    | None →
        let result = if n = 0 || n = 1 then n else fib (n - 1) + fib (n - 2) in
        mem.(n) ← Some result; result
    (*@ (A) variant n
        (B) requires 0 ≤ n < Array.length mem
        (C) requires ∀ i. 0 ≤ i < Array.length mem → mem.(i) = None ∨ mem.(i) = Some (fib i)
        (D) ensures  ∀ i. 0 ≤ i < Array.length mem → mem.(i) = None ∨ mem.(i) = Some (fib i)
        (E) ensures result = fib n *)
  in
  fib n
(*@ (F) requires n ≥ 0
    (G) ensures result = fib n *)
```

We can feed a `fibonacci.ml` file, containing the logical function and the above specification for the memoized version, to Cameleer with the command: `cameleer fibonacci.ml`. Cameleer translates the input program into an equivalent WhyML representation, launching a graphical interface for the Why3 IDE.

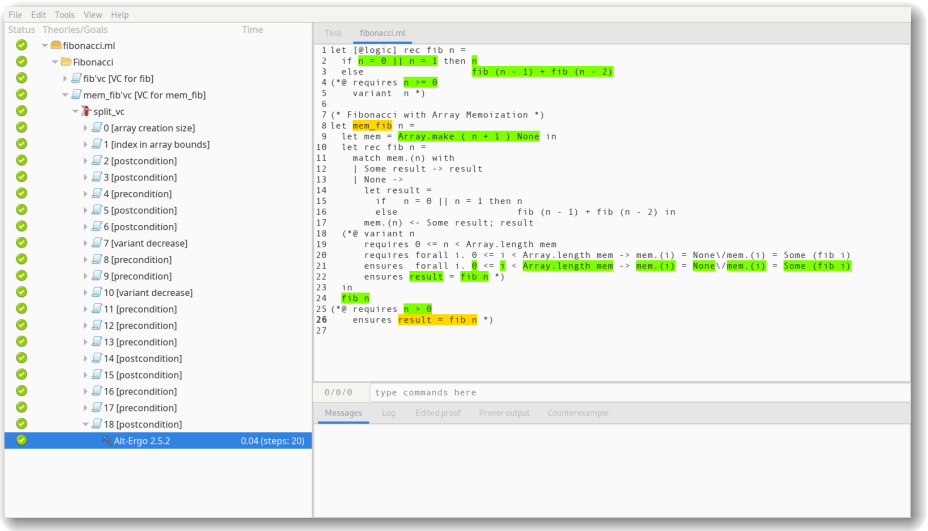


Figure 2.1: Why3 IDE with our *fibonacci* example.

The Why3 graphical interface allows us to individually prove VCs (Verification Conditions), by decomposing a compound task into smaller, more digestable, VCs — this can be done with `split_vc` in the dedicated command line. Irrespective of how big the task is, either a compound or an irreducible verification condition, we can dispatch external provers by selecting a VC and explicitly choose a specific prover in Tools » Provers, or collectively call every prover in Tools » Strategies. In this case, Alt-Ergo quickly discharges the postcondition proof of `mem_fib`, taking 0.04 seconds and 20 mechanized proof steps.

2.4 Why?

We will devote this section to introduce the platform Why3 and its accompanying language WhyML, and for lack of a better title, we raise the question "Why?". Unlike other tools and languages aimed at deductive verification, such as VeriFast [20] or Dafny [21], the Why3 platform relies on multiple external provers, adjusted to the user's needs. It supports both automated and interactive external provers, like Isabelle [28] and Coq. This provides for a more flexible and ergonomic proof session as some provers might be more efficient than others, leading to a faster and sometimes sole, discharge of a proof. Furthermore, Why3 allows for local reasoning of verification conditions, by assuming everything else true, leading to better pinpointing erroneous specifications.

Why3 also comes with its own programming language, WhyML, a dialect of the ML family. Offering features commonly found in functional languages, like pattern-matching, algebraic types and polymorphism, but also imperative constructions, such as records with mutable fields and exceptions. Programs written in WhyML can also be annotated with contracts, using the keywords `requires` and `ensures`, modifications to mutable data using `writes`, and signal exceptions and their implications with `raises`. The code can be annotated with loop `invariants` or termination measures, `variants`, for loops and recursive functions. It is also possible to add intermediate `assertions` to ease automatic proofs.

Let us consider the example of a `push` function on a capacity-bounded Stack. A bounded Stack can be defined using a record with two fields, a mutable `view` as its representation, and `capacity` as the number of elements it can store. We say that this record is `abstract`, which means that it is a private record and its fields are *ghost*.

WhyML

```
module Stack
  use int.Int
  use list.ListRich (* import all list theories in one *)

  type t  (* stack's element type *)
  type stack = abstract {
    mutable view: list t;  (* stack's representation *)
            capacity: int  (* number of elements it can hold *)
  }
end
```

A specification for the `push` function, residing in the same module `Stack`, can be done in two different ways: by requiring the Stack to not be full or by means of an exception that indicates that it reached its capacity. Following the former, the `push` function's contract is as follows:

WhyML

```
  predicate not_full (s: stack) = length s.view < s.capacity
  val push (x: t) (s: stack): unit
     writes   { s.view }
     requires { not_full s }
     ensures  { s.view = Cons x (old s.view) }
```

We defined a predicate that compares the size of the Stack and its capacity, and a function `create c`, which we ommit for simplicity reasons, that returns a new stack with capacity `c` and an empty `view`. One way to see that this specification will lead to an expected behavior is to consider a Stack of capacity one and try to push multiple elements onto it:

WhyML
```
let push_fail (x: t) =
  let stack = create 1 in
  push x stack;   (* this call to push is valid *)
  push x stack;   (* this one is not *)
```

The call to the second push is unable to satisfy the pre-condition, as now the Stack is full. This leads to Why3, and really anyone, being unable to prove the second push valid.

Abstraction barriers, which allow implementation details to be hidden behind an opaque interface, also apply to deductive program verification [16]. They come most handy by allowing a proof of an abstract interface to be also true for its implementations. In the previous example, we left the Stack's type `t` abstract so that subsequent instantiations can use its once-proven properties. Through WhyML's module system, we can consider an instantiation of our Stack, `StackInt`.

WhyML
```
module StackInt
  clone Stack with type t = int
end
```

As this module is only an instantiation of Stack, it has no new proof obligations and is considered as correct as the module it clones. However, if we were to extend with type-specific specifications, we would introduce new verification conditions, which would, in turn, have to be proven.

2.5 CFML

Characteristic Formulae for ML (CFML) is yet another tool for interactive verification of OCaml programs [5]. It combines, previously seen, Separation Logic with the powerful interactive theorem prover Coq [9]. CFML is able to work directly with OCaml code, by extracting OCaml functions and embedding them directly into Coq function definitions. Formal proofs in CFML are mainly defined via lemmas, which in turn have to be manually proved. Adapting from [6], let us consider, a push operation in an unbounded Stack.

OCaml
```
type α stack = (α list) ref
let push p x = p := x :: !p
```

The Stack is modeled as a reference to a list. Note that this does not let us say anything about mutability of its elements, which, at most, are immutable. In CFML, we can define a Stack as a heap predicate that is captured by a location p containing a list L, with ~~>.

CFML
```
Definition Stack T (L:list T) (p:loc) : hprop :=
  p ~~> L. (* pointer [p] contains a list [L] *)
```

The specification that correctly maps the function's behaviour, *i.e.*, an element x is pushed onto a Stack, is rather simple. Universally quantifying, for any memory location p, any value x and list L, the precondition states that L is a Stack representation of p, using ~>. And the post-condition holds in a state where p points to location in memory that can be logically expressed as x appended to the head of L.

CFML

```
Lemma push_spec : ∀ T (p:loc) (x:T) (L:list T),
  SPEC (push p x)
    PRE (p ~> Stack L)
    POSTUNIT (p ~> Stack (x :: L)).
```

Although a ridiculously minute introduction to CFML, its relevancy will be evermore clear as we reach Section 3.3 where it will serve as a comparison basis with related work on the verification of higher-order iterators.

2.6 OCamlGraph

OCamlGraph is a generic library featuring a large set of graph data structures and algorithms [8]. By exploiting functors it is able to provide 19 types of different graphs, resulting of different combinations of 4 properties:

1. *directed* or *undirected* graph.
2. *labeled* or *unlabeled* edges.
3. *persistent* or *imperative* data structures.
4. *concrete* or *abstract* type for vertices.

One way to see how graphs can be persistent or imperative is to consider their signatures and check their return values in functions such as `add_vertex` or `add_edge`:

OCaml

```
module type P = sig
  include G
  (* return a new graph *)
  val add_vertex : t → V.t → t
  val add_edge   : t → V.t → V.t → t
  ...
end
```

```
module type I = sig
  include G
  (* return unit, modifications are in-place *)
  val add_vertex : t → V.t → unit
  val add_edge   : t → V.t → V.t → unit
  ...
end
```

Moreover, to see how one might use the library, in a somewhat realistic example, let us consider a simple message transfer protocol that can be represented as an automaton. Let us also say that state transitions are protocol operations, and the state itself is the current set of messages. We want to see if it is possible to get from the initial state, the empty set of messages, to one where we have messages from both clients 1 and 2. We can represent this automaton as a graph and perform a path check algorithm of the two states.

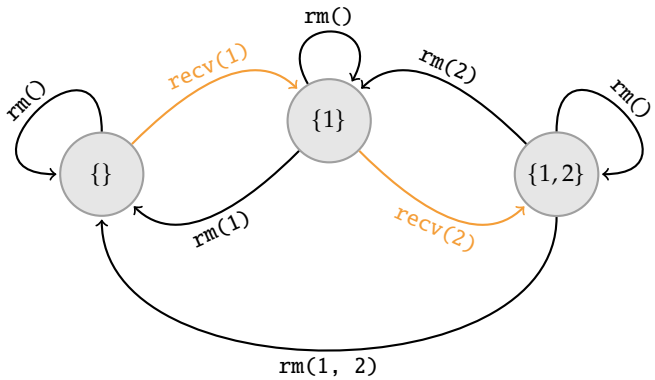
OCaml

```
open Graph

module MSet = struct
  include Set.Make(struct
    type t = int
    let compare = compare
  end)
  let hash = Hashtbl.hash
end
module Oper = struct
  type t = string
  let compare = String.compare
  let default = String.empty
end
module G = Persistent.Digraph.ConcreteLabeled(MSet)(Oper)
module Path = Path.Check(G)

let g = G.empty
let g = G.add_vertex g MSet.empty
let g = G.add_edge_e g (G.E.create (MSet.empty) "recv(1)" (MSet.singleton 1))
let g = G.add_edge_e g (G.E.create (MSet.singleton 1) "recv(2)" (MSet.of_list [1; 2]))
let g = G.add_edge_e g (G.E.create (MSet.of_list [1; 2]) "rm(1,2)" (MSet.empty))
let g = G.add_edge_e g (G.E.create (MSet.of_list [1; 2]) "rm(2)" (MSet.singleton 1))
let g = G.add_edge_e g (G.E.create (MSet.singleton 1) "rm(1)" (MSet.empty))
let g = G.add_edge_e g (G.E.create (MSet.empty) "rm()" (MSet.empty))
let g = G.add_edge_e g (G.E.create (MSet.singleton 1) "rm()" (MSet.singleton 1))
let g = G.add_edge_e g (G.E.create (MSet.of_list [1; 2]) "rm()" (MSet.of_list [1; 2]))

let () =                                    (*  initial         final       *)
  let path = Path.check_path (Path.create g) MSet.empty (MSet.of_list [1; 2]) in
  if  path then print_endline "There is a path"
  else          print_endline "There is no path";
```



The OCamlGraph library provides everything to solve this problem: a correct graph representation and a path-checking algorithm. We are only required to define the types of our vertices, in this case, a set of messages and edges that can be represented as strings. We also model the graph as persistent, directed, and labeled with concrete vertices. After populating the graph with states and their transitions, we can call a path-checking function provided by the library and conclude that there is indeed a path from the initial empty set to a state where we have messages from both clients.

Although a simplified version of a real problem, one can imagine that if critical decisions are based on the existence of a path between two states, then its underlying implementation must be correct. Worry not; we will see later on that the implementation of this function is indeed correct.

2.7 Depth-First Sequences

In this section, we wish to present a pedagogical example that also serves the purpose of introducing *lazy lists*, also called *cascades* or *sequences*. We will tackle another type with the same name but different semantics. For the remainder of the section, consider the former.

These sequences are lists that produce elements on demand. That is, the client is given control over the production of elements. The OCaml standard library gives the following definition for its type:

```
type +α node =                                                    OCaml
| Nil
| Cons of α * α t

and α t = unit → α node
```

First of all, why is α positive? The plus sign next to the type indicates that the type α is *covariant* with the type `node`. In other words, if a function works with α `node`, it will also work with β `node` if α is a subtype of β.

Second of all, we note that this type is mutually recursive. It is essentially the definition of a mathematical list, but the second element of the pair of the `Cons` branch is a function that, when called, returns the next element in the list. So, for instance, to compute the length of a sequence, we can follow a classical approach:

```
let rec length_aux acc xs =                                       OCaml
  match xs () with
  | Nil          → acc
  | Cons (_, xs) → length_aux (acc + 1) xs

let length x = length_aux 0 x
```

Pay attention to how the match is done; we call `xs` to produce the element and then pattern match on it. This allows us to halt the iteration over sequences at any point, as it is the consumer that has control of the iteration.

Let us now consider an interesting example. A Depth-First Search (DFS) on a graph defined through adjacency sequences and with integer nodes. The common miss-conception that a Depth-First Search is a Breadth-First Search where the queue is replaced with a stack works for trees but not for arbitrary data structures. For that reason, we need an auxiliary set to store visited nodes. The implementation of this algorithm is the following:

```
let dfs g s =                                                     OCaml
  let stack = Stack.create () in
  let visited = ref Visited.empty in
```

We start by pushing the source `s` onto the stack:

```
  Stack.push s stack;                                             OCaml
```

And while we have elements, to visit, in the stack, we pop the head:

```
while not (Stack.is_empty stack) do
  let v = Stack.pop stack in
```
OCaml

If this element was not visited, we say that it is now:

```
  if not (Visited.mem v !visited) then
    visited := Visited.add v !visited;
```
OCaml

This is followed by exposing the sequence of successors of `v`, with the function `uncons`. If the sequence is empty, we do nothing. However, if it has any successors, we push them all onto the stack:

```
    Seq.iter (fun x → Stack.push x stack) (Array.get g v)
done
```
OCaml

How wonderful, the higher-order function `iter` pushed all the successors at once. This is rather boring. It does not let us grasp the potential of sequences. One alternative is to push one successor at a time. This is essentially the same, but we do not consume the whole sequence at once; we only push one. This requires updating the array of successors as `uncons` does not actually consume the element as `iter` does. Let us see how this works:

```
let dfs g s =
  let visited = ref (Visited.singleton s) in
  let stack = Stack.create () in
  Stack.push s stack;
  while not (Stack.is_empty stack) do
```
OCaml

This time, we do not directly pop from the stack, this is only the case when we have visited all its successors — the `None` branch of the pattern match.

```
    let top = Stack.top stack in
    match Seq.uncons (Array.get g top) with
    | None → let _ = Stack.pop stack in ()
    | Some (x, xs) →
```
OCaml

Otherwise, we update the adjacency list of the top-most stack element with the rest of its, to-be visited, successors. If the top-most element has not yet been visited, we mark it as such and push it onto the stack.

```
      g.(top) ← xs;
      if not (Visited.mem x !visited) then
        visited := Visited.add x !visited;
        Stack.push x stack
done
```
OCaml

This version also yields a DFS traversal but the order might be slightly different. This is a natural observation as in the first case we pushed every successor of a node, whilst in the second we only pushed the top-most successor. But as we have seen, sequences are essentially lists with a caveat. They allow for potentially infinite streams of elements, producing only when needed.

2.8 An audit on iteration paradigms

Albeit a reasonable stance on iteration, recursion is often hard to conceptualize and may lead to undesirable behaviour, such as infinite recursion or poor performance. Sidelining to more conventional types of iteration we find `Cursors`.

Unlike Higher-Order Iteration, where iteration is done with exhaustion in mind and with no control over how many steps we take[2], `Cursors` give full control to the client on how it chooses to do define the iteration. They are defined using two functions, `next()` and `has_next()`, which returns the next element in the iteration, and tests whether the iteration has completed, respectively. We occasionally find `Cursors` in languages such as Java or C++, with the "for each" loop construct `for (E x: ...)` being nothing short of syntactic sugar for a client cursor that reaches exhaustion through controlled single steps [27]. A client cursor over a collection c, typically looks like:

Syntax

```
cursor ← create_cursor(c)
while has_next(cursor) do
  x ← next(cursor)
  ...
```

Iterators can be classified as *persistent*, where a call to `next()` returns a newly produced value and a new iterator with a possibly updated state, or *imperative*, where the iterator keeps an internal mutable state and a call to `next()` returns the newly produced value [13]. The former follows a more functional approach to iteration where values are immutable. Although this gives us a history of side-effect-free iterators, conceiving exhaustive iterations without mutability is challenging and impractical.

To demonstrate this, let us consider a desire to adapt a persistent cursor to exhaustively traverse a collection and apply a function f, as in the case of Higher-Order Iteration. The interface `HO` for a higher-order iteration, is uniquely populated by the function `iter`.

OCaml

```
module type HO = sig
  type elt                                    (* element type *)
  type collection
  val iter: (elt → unit) → collection → unit (* iteration function *)
end
```

A persistent cursor is defined by a creation (`iterator`) and a produce (`next`) function.

OCaml

```
module type PC = sig
  type elt                          (* element type *)
  type collection
  type it                           (* iterator *)
  val iterator: collection → it     (* create function *)
  val next: it → (elt * it) option (* next step in iteration *)
end
```

[2]Once again, neither early stopping nor stepping are unachievable, but using Higher-Ordered Iteration for those purposes seems counter-intuitive. Achieving early-stop requires adding error handling hacks, and stepping requires keeping track of the current internal state of the iterator, all while taking single steps requires redundant re-iterations.

Towards being more modular, we leave their types abstract, though in order to actually provide a concrete example let us also consider that our element type is `int` and our collection `int list`. An implementation of a higher order iteration, over lists, boils down to defining the recursive function that traverses a list and individually applies a function to each of its elements, as tackled in Section 2.2.3.

OCaml

```
module ListHO:
  HO with type elt        = int and
          type collection = int list
= struct
  type elt = int
  type collection = int list

  let iter f l =
    let rec loop f = function
    | []     → ()              (* exhausted, do nothing *)
    | h :: t → f h; loop f t in (* apply f to head; recurse on the tail *)
    loop f l
end
```

An implementation of a persistent cursor, over lists, can be expressed by two functions, one that creates a cursor, `iterator`, and one that produces elements `next`. The former returns a new persistent cursor with the initial collection l. On the other hand, the latter returns an option where non-terminated iterations produce a pair of the next element and a new persistent cursor with the remainder.[3]

OCaml

```
module ListPC:
  PC with type elt        = int and
          type collection = int list
= struct
  type elt = int
  type collection = int list
  type it = Done | PC of collection

  let iterator l = PC l
  let next = function
  | PC []       → None
  | PC (h :: t) → Some (h, PC t)
  | Done → None
end
```

The module `PC2HO` adapts a persistent cursor to work as a higher order iterator, implementing the signature of `HO` by only using functions that `PC` exposes. Observing the adapter's implementation, we find that achieving Higher-Order Iteration using a persistent cursor comes at a cost of keeping an internal mutable state of the iterator, essentially defeating the purpose of being persistent.

[3] Do note that any calls following termination will always yield *None*, this is a design choice in the likes of Rust's `FusedIterator` [32], inversely we could have raised an exception indicating termination.

```
module PC2HO (PC: PC) :                                              OCaml
  HO with type elt        = PC.elt and
          type collection = PC.it
= struct
  type elt = PC.elt
  type collection = PC.it

  let iter f it =
    let cursor = ref it in    (* mutable iterator *)
    let cond = ref (PC.next !cursor ≠ None) in
    while !cond do
      let next = PC.next !cursor in
      match next with
      | None → cond := false   (* finished iteration, stop *)
      | Some (x, next_cursor) →
        cursor := next_cursor;  (* save next cursor *)
        f x;                    (* apply f to x *)
    done
end
```

Imperative cursors detach themselves from rigorous rules of immutability and accept the fact that as long as side-effects are kept internally, they are allowed to keep a reference to the iteration — a mutable index in the case of iterations over arrays, for example. This has become the norm for iteration, as it brings the comfort of simpler and more efficient conceptions of iteration but at the cost of being significantly harder to reason on as we are dealing with possibly aliased memory.

Its signature must define a creation function, one that tests its termination and another that produces the next element. The produced element is an option, where it could be instead just `elt`. However, this would require having an exception that would be thrown when calling `next` after the iteration has finished.

```
module type C = sig                                                  OCaml
  type elt
  type collection
  type it
  val iterator: collection → it (* creation *)
  val has_next: it → bool       (* test termination *)
  val next: it → elt option     (* retrieve the next element *)
end
```

In a similar fashion as other examples, defining a cursor over lists is now far simpler. In order to create a cursor over a list l, we can return a reference to it. Testing the termination of this cursor, with `has_next`, can be obtained by comparing the iterator, which in this case is a list, with the empty list. Finally, producing elements entails returning the current head and updating the iterator with the tail, on the case where the iteration has not yet finished, or return `None` where it has.

OCaml
```
module ListC:
  C with type elt = int and
         type collection = int list
= struct
  type elt = int
  type collection = int list
  type it = collection ref

  let iterator l = ref l
  let has_next it = !it ≠ []
  let next it =
    if has_next it then
      let res = List.hd !it in (* keep current head *)
      it := List.tl !it;       (* iterator is now the tail *)
      Some res                 (* return head *)
    else None
  end
```

Following from our previous example, if we now wish to adapt an imperative cursor to behave as a higher-order iterator, we would need to implement the function `iter` by means of only what the cursor exposes. The module `C2HO` defines the `iter` function as a loop that while the iterator has elements to produce, we produce it with `next` and apply it the argument function `f`:

OCaml
```
module C2HO (C: C) :
  HO with type elt = C.elt and
          type collection = C.it
= struct
  type elt = C.elt
  type collection = C.it
  let iter f it =
    while C.has_next it do
      f (Option.get (C.next it));
    done
end
```

This is very much less verbose and intuitive than its persistent counter-part, and although this one introduces the notion of mutability, with an internal reference to the collection, it is not a big problem as this one is kept inside the cursor and with no possible way of introducing *memory leaks* or *aliasing*.

In this section we presented different ways of iteration. Higher-order iterators prefer to give more control to producers, all while hiding implementation details that programmers usually do not care about. On the other hand, cursors allow us to take single steps, giving more control to consumers. We found imperative cursors to be more favourable over the functional approach of persistent due to their expressiveness and low-overhead to define a higher-order iterator. This last remark motivates us to, further on, use these cursors to express higher-order iteration.

2.9 A motivating example

Let us consider the sum of a finite and deterministic enumeration, the other semantics for *sequences*, to illustrate the different iterations currently specifiable. Currently, we can solve this problem using two different approaches: recursion or a loop. As we progress, we will also explore two other methods: using cursors and, finally, using higher-order iteration.

From a recursive standpoint, the problem is solved by a function that iterates from an `upper` bound down to a `lower` bound, while summing up the elements of the sequence indexed by `upper`. This serves as our logical definition of a sum. It is similar to the definition in the Why3's Standard Library, but here we fix the consumer function to the binary sum. We provide this function with a contract that states the decreasing argument is `upper` and requires both `lower` and `upper` to be confined in the range of the sequence `l`.

GOSPEL + OCaml

```
let [@logic] rec sum l lower upper =
  if   upper ≤ lower then 0
  else                    l[upper - 1] + sum l lower (upper - 1)
(*@ r = sum l lower upper
    variant upper
    requires 0 ≤ lower ≤ upper ≤ length l *)
```

From a cycle standpoint this can be solved by introducing two mutable variables, `index` to get the *n-th* element of a sequence, and `acc` the accumulator of the sum. In this cycle, we iterate over the range of the length of `l` and successively add its elements:

GOSPEL + OCaml

```
let sum_cycle l =
  let (index, acc) = (ref 0, ref 0) in
  while !index < (length l) do
```

We have to provide this loop with a specification. Stating its termination measure, which in this case is the convergence of the `index` to the length of the sequence. And an invariant which states that this index is bounded in the limits of the sequence's length and that the accumulator holds the current sum of elements up until the value of `index`.

GOSPEL + OCaml

```
    (*@ variant (length l) - !index
        invariant 0 ≤ !index ≤ length l ∧ !acc = sum l 0 !index *)
    acc := !acc + l[!index];
    index := !index + 1;
  done;
  !acc
```

Finally, we state that the result of this function equals the sum of the whole sequence.

GOSPEL + OCaml

```
(*@ r = sum_cycle l
    ensures r = sum l 0 (length l) *)
```

This specification and subsequent verification is entirely possible because the integer sum is a pure function, *i.e.*, it has no side-effects. Whereas if we considered any other function with writes in memory, for instance, this would not be enough. We will see later on, that this is possible.

This page is intentionally left blank.

3

State of the Art

In this chapter, we present some work in regards to the specification and verification of higher-order iteration and some work on the formal verification of OCamlGraph.

3.1 Proving OCamlGraph

Undoubtedly, the closest thing that resembles the work that's here proposed was done by Castanho [4]. The work revolved around proving a subset of the OCamlGraph library using Cameleer. Although this might seem strikingly similar, we complement GOSPEL with Pereira's [27] and Castanho's [4] future work suggestions on higher order iteration. It is with this that we diverge from Castanho's work.

3.2 Iteration

As previously mentioned, iteration will play a big role in the development of this thesis, through the way it is used in the OCamlGraph library and how we choose to reason about it. Reasoning about iterators is not novel: prior work conducted by Filliâtre and Pereira [27] proposes a modular way to reason about iterators using two predicates per iterator, a `permitted` predicate that defines the produced values and a `completed` predicate that states its eventual completion, in the case of finite iterations. Let us consider an example from [27] regarding the iteration over an array a, from left to right. The predicate tied to the enumeration of the visited elements, `permitted`, is as follows:

$$\texttt{permitted}(v, a) \triangleq ||v|| \leq \texttt{length}(a) \wedge \forall i. 0 \leq i \leq ||v|| \implies v[i] = a[i]$$

This means that for an array a, the sequence of visited elements, v, is a prefix of a. And the `completed` predicate can be expressed as the comparison of both lengths:

$$\texttt{completed}(v, a) \triangleq ||v|| = \texttt{length}(a)$$

Borrowing from these ideas, we can specify an iteration over trees resorting to an auxiliary `order` function which will state the order we will visit the nodes in the tree, and claim

that the `permitted` predicate entails the sequence of visited elements is a prefix of the order by which we will visit the nodes in the tree.

$$\texttt{permitted}(v, \mathit{tree}) \triangleq \texttt{prefix}(v, \texttt{order}(\mathit{tree}))$$

And similarly, reasoning about completion can be done by comparing the length of the visited sequence and the number of nodes the tree has:

$$\texttt{completed}(v, \mathit{tree}) \triangleq ||v|| = \texttt{count}(\mathit{tree})$$

Arguably the most important result of this work is that iteration, be it finite or infinite, deterministic or non-deterministic, can be reasoned upon using these predicates. A secondary result is that iteration is not necessarily the traversal of a data structure as it can be, for instance, the result of an algorithm [27].

3.3 Verification of Higher-order Iterators

When it comes to interactive proofs, the work of Pottier [30] is the first instance where we find the use of the pair `permitted` and `complete` to reason about iteration. This work was conducted in CFML to formally verify an implementation of the `Hashtbl` from the OCaml Standard Library. A key aspect of this work that very much aligns with ours is how the author verifies the iteration over the data-structure. We will focus on Pottier's general form of an iteration via a fold.

First of all, they fix a fold to be a function with three arguments `f`, `c`, and `acc`, in that order. The first is the consumer function, that when applied an element and an accumulator, must return a new accumulator. The second one is the collection being iterated, *i.e.*, where the iteration gets its elements. The third is the initial value of the accumulator.

We can start by encoding all the necessary declarations of logical entities, or `Variables`. The type of `fold` is an abstract logical function, `func`.

```
Variable fold : func.
```
CFML

We also note that the signature of `call` is what we expect its arguments to be: a given function, some type `A`, which is the type of elements in a collection, followed by `B`, the type of our accumulator. The notation for the return of `call`, `~~B`, is analogous to `hprop → (B → hprop) → Prop`.

```
Variables A B C : Type.
Variable call : func → A → B → ~~B.
```
CFML

This means that `call f x acc` should be applied to a pre- and postcondition and the return value of this parameter is a proposition where both conditions are met.

The parameters `permitted` and `complete` are typed as one would expect them to be: given a list of `A`, *i.e.*, list of elements from the collection, return a proposition.

```
Variables permitted complete : list A → Prop.
```
CFML

The parameter `I` represents an invariant. This one takes a list of elements with type `A`, an accumulator of type `B` and returns a heap proposition.

```
Variable I : list A → B → hprop.
```
CFML

Finally, the parameters `S` and `S'` capture the permission to access some collection `C`. They are two different parameters as we need to access the collection twice: once on a `call` of a function and another on an application of fold.

```
Variables S S' : C → hprop.
```
CFML

The definition Fold encapsulates the behavior of a fold function, which processes a list by recursively applying a function to elements of type `A` and accumulating a result in an accumulator of type `B`. This definition states that on the premise that the consumer function is well-behaved, then `fold` also behaves appropriately. Let us breakdown this definition:

```
Definition Fold :=
```
CFML

This is our premise. Given a function `f` and some collection `c` if some list `(xs & x)` of elements is permitted then we can call `f` with a produced element `x` and the accumulator.

```
  ∀ f c,
    ( ∀ x xs accu,
        permitted (xs & x) →
        call f x accu
```
CFML

This call must respect a precondition where we claim ownership of the collection and the invariant is true for the so-far visited elements `xs` and for any given accumulator. Additionally, in the postcondition we claim once more ownership of the collection and state that the invariant now holds for the so-far visited elements and the current element `x`, given an accumulator.

```
          PRE (S' c \* I xs accu)
          POST (fun accu ⟹ S' c \* I (xs & x) accu) ) →
```
CFML

Now for the specification of an application of fold, we claim ownership of `c` and state that the invariant must hold for the empty list of visited elements and for the initial value of the accumulator. As a postcondition, there must exist a list where the invariant is true for it, with some accumulator, and where the same list makes the iteration complete.

```
    ∀ accu,
      app fold [f c accu]
        PRE (S c \* I nil accu)
        POST (fun accu ⟹ Hexists xs, S c \* I xs accu \* \[ complete xs ]).
```
CFML

This general and concise specification allows us to formally capture the iteration using a fold. Nonetheless, verifying, for instance, a *read-only* iteration, where the consumer cannot

modify the table, requires around 60 lines of Coq script, or around 14 for an iteration where the consumer function has no access at all to the table. The proof for the latter is based an instantiation of the former. For sake of simplicity, let us present the specification for `fold_spec`, where the consumer cannot access the table.

The `permitted` predicate states that there must exist some dictionary `M'` where removing the elements `kxs` from some dictionary `M` yields `M'`. With this approach, we take care of some potential non-determinism that could be a result of trying to enumerate, in some order, the table.

```
Definition permitted kxs := ∃ M', removal M kxs M'.
```
CFML

The `complete` predicate is rather straightforward. Given some list of elements `kxs`, the iteration is complete if the resulting dictionary from the removal of `kxs` from `M` is empty.

```
Definition complete kxs  := removal M kxs empty.
```
CFML

A permissionless specification is given as follows: The parameter `S` is instantiated with `(fun h ⟹ h ~> Table M)`, that is, the `fold` has full access to the table. Contrary to the consumer function `f`, where `S'` is instantiated with `(fun h ⟹ \[])`, an empty heap assertion.

```
Theorem fold_spec:
  ∀ M B I,
  Fold MK.fold
    (fun f kx (accu : B) ⟹
       app f [(fst kx) (snd kx) accu])
    (permitted M) (complete M) I
    (* [fold] requires and preserves this: *)
    (fun h ⟹ h ~> Table M)
    (* [f] does not have access to the table: *)
    (fun h ⟹ \[]).
Proof.
 (* find the full proof in the original paper [30] *)
Qed.
```
CFML

This work presents a general and well founded specification for a higher-order iteration via a fold. Nonetheless, the proofs are done in Coq which requires alot of human effort and expertise in the matter, especially when combined with *Separatin Logic*. In our work, we wish to mechanize this effort by establishing a general specification in Why3 and delegate the work to SMT solvers, capable of quickly proving its validity.

3.4 Side-effectful Iteration

In this section we wish to present some recent work done on iteration parametrized with closures, which can modify their captured state as a side effect. The work was done in Rust, though we believe that some ideas are transversal to any language, provided strong guarantees about non-aliasing memory. Denis [10] and Bílý et al. [3] build on

top of Pereira's [27] specification of iterators in Why3, however, and while the work was concurrent, they take different approaches.

Although Rust's *borrowing system* provides safe memory control, the handling of mutable references `&mut T` requires extra caution seeing that they represent a temporary *borrow* of ownership of a memory location. Denis [10] found that, to correctly model the propagation of mutations, a mutable reference `r: &mut T` should be represented as a pair `{*r, ^r}`, the current value pointed to by that reference and a value the reference will point to when the borrow ends, respectively.

Now iterators are specified as state machines, where a value of an iterator is a state. The predicate `produces(a, s, b)` defines the transition of `a` to `b` through `s`, denoted $\mathtt{a} \overset{\mathtt{s}}{\rightsquigarrow} \mathtt{b}$. The `completed` predicate gives the set of final states. With the added notion of state transition, we are also required to prove the laws of reflexivity ($\forall\ \mathtt{a}.\ \mathtt{a} \overset{\varepsilon}{\rightsquigarrow} \mathtt{a}$) and transitivity ($\forall\ \mathtt{a,b,c}.\ \mathtt{a} \overset{v}{\rightsquigarrow} \mathtt{b} \land \mathtt{b} \overset{w}{\rightsquigarrow} \mathtt{c} \implies \mathtt{a} \overset{v \cdot w}{\rightsquigarrow} \mathtt{c}$). Moreover, we are able to say that an iterator at the i-th iteration is the result of calling `next` i times, and existentially quantifying, we can state our invariant, which will hold for any iteration state as $\exists p, \mathtt{initial} \overset{p}{\rightsquigarrow} \mathtt{iter}$, provided reflexivity and transitivity are proved.

In order to demonstrate how this applies, and safely borrowing from Denis' ideas [10], let us consider a `Range` iterator that iterates over a range of integers. This is a fairly simple example, especially since mutations are kept internally and side effects should not be of our concern, yet. `Range` can be defined as a record with two fields, `start` and `end`. We can define the `produces` and `completed` as follows:

$$r \overset{v}{\rightsquigarrow} r' \triangleq |v| = r'.\mathtt{start} - r.\mathtt{start} \land r'.\mathtt{end} = r.\mathtt{end} \land$$
$$|v| > 0 \implies r'.\mathtt{start} \leq r'.\mathtt{end} \ \land \forall i \in [0, |v| - 1], v[i] = r.\mathtt{start} + i$$
$$\mathit{completed}\,(r) \triangleq {*r} = {\hat{}r} \land (*r)\,.\mathtt{end} \geq (*r)\,.\mathtt{start}$$

The transition of one iteration state r to r' is defined via v, which its size indicates how many steps in the iteration were taken. Atop of trivial properties, we must ensure that everything we visited still holds, and this is done by stating that for any i in $[0, |v| - 1]$, $v[i]$ is equal to $r.\mathtt{start} + 1$. In the `completed` predicate, we claim termination is achieved by expecting the value of the reference `r` is equal to the value of the reference when the borrow ended, and that `end` is at least `start`.

Another key feature of Rust are closures: anonymous functions that capture their environment, with the added ability to specify how one wishes to capture state. Rust provides three types of closures: `FnOnce`, `FnMut` and `Fn`. Describing how a call should pass state: via ownership and only callable once, by a mutable reference enabling access to their environment, or by an immutable reference. Essentially, allowing us to reason about higher-order code with mutable state by only relying on Rust's mutable value semantics, avoiding the need for *Separation Logic* [10]. As to actually reason about closures themselves, we can decorate them with pre- and postconditions, forcing closure calls to obey this contract. Illustrating this in Rust, and relying on the `CREUSOT` verification platform [11], the most restrictive closure `FnOnce` can be defined as:

```rust
pub trait FnOnce<Args> {
  #[predicate] fn precondition (self, a: Args) → bool;
  #[predicate] fn postcondition_once (self, a: Args, res: Self::Output) → bool;

  #[requires(self.precondition(args))]
  #[ensures(self.postcondition_once(args, result))]
  fn call_once(self, args: Args) → Self::Output;
}
```
Rust

Seeing as a call to `FnOnce` consumes the closure, it is not desirable when trying to conceive iteration. Conversely, `FnMut` allows mutable closures to be called multiple times, for which trait, that also extends `FnOnce`, can be defined as:

```rust
pub trait FnMut<Args> :FnOnce<Args> {
  #[predicate] fn unnest(self, _: Args) → bool;
  #[ensures(self.unnest(self))]
  #[law] fn_unnest_refl(self);

  #[requires(self.unnest(b) && self.unnest(c))]
  #[ensures(self.unnest(c))]
  #[law] fn unnest_trans(self, b: Self, c: Self);

  #[predicate] fn postcondition_mut(&mut self, _: Args, _: Self::Output) → bool;
  #[requires((*self).precondition(arg))]
  #[ensures(self.postcondition_mut(args, result))]
  fn call_mut(&mut self, args: Args) → Self::Output;
}
```
Rust

In order to link closure states across multiple calls, we need to introduce an unnesting predicate. This property is captured by `F::unnest(a, b)`, where it states that the prophecies in the state of `F` have not changed from `a` to `b`. Without it, we would lose track of the history of mutable borrows. Furthermore, as this property should hold throughout the lifetime of the closure, we must impose that this predicate is reflexive and transitive.

To see how this can be applied, let us consider yet another example from Denis [10], where we wish to specify a higher-order iterator, `Map`. As previously mentioned, `Map` iterates a collection and individually applies a n-ary function to each element, collecting the results, and in this case lazily generating values with a yield instruction.

```rust
fn map<I: Iterator, B, F: FnMut( I::Item ) → B>( iter: I, func: F ) {
  for a in iter { yield (f)(a) }
}
```
Rust

Since we have introduced a loop, we need to an invariant that can satisfy subsequent iterations, *i.e.*, the postcondition at some step i must be able to establish the precondition at step $i + 1$. With everything presented until now, this can be specified as:

$$it \overset{s \cdot e_1 \cdot e_2}{\rightsquigarrow} i' \wedge pre\left(*f, \mathsf{e}_1\right) \wedge post\left(f, \mathsf{e}_1, r\right) \implies pre\left(\hat{}f, \mathsf{e}_2\right)$$

If we produce an element $\mathtt{e}_1$ that satisfies the precondition of the initial closure $*f$ and combine it with the postcondition of f, we can establish the precondition for the final closure $\hat{}f$ with the following element $\mathtt{e}_2$. The predicates `permitted` and `completed` for `Map` can be expressed as:

$$\begin{aligned} it \overset{v}{\rightsquigarrow} it' \triangleq \exists\, v', fs, |v'| = |fs| = |v| \wedge it.\mathtt{iter} \overset{v'}{\rightsquigarrow} it'.\mathtt{iter} \wedge\ & \mathit{unnest}\,(it.\mathtt{func}, it'.\mathtt{func}) \\ \wedge\ & \left(it.\mathtt{func} = *fs[0] \wedge \hat{}fs[0] = *f[1] \wedge \ldots \wedge \hat{}fs[n] = it'.\mathtt{func}\right) \\ \wedge\ & \forall i \in [0, |v| - 1], \mathit{pre}\left(*fs[i], v'[i]\right) \wedge \mathit{post}\left(fs[i], v'[i], v[i]\right) \\ \mathit{completed}\,(r) \triangleq\ & \mathit{completed}\,(it.\mathtt{iter}) \wedge (*it).\mathtt{func} = (\hat{}it).\mathtt{func} \end{aligned}$$

In the state transition $it \overset{v}{\rightsquigarrow} it'$, we existentially quantify over the sequence of produced values by the iterator v' and the sequence of *mutable reference* of states fs. We then require that fs forms a chain, *i.e.*, the current value of the element $*fs[i]$ being the same as the result of previous state $\hat{}fs[i-1]$. We also universally quantify over every iteration and require their closure pre- and postconditions. Finally, we require that the first and last states meet the unnesting relation. The *completed* predicate states the iteration has completed and the closure state has not changed.

We are now left with a specification for `Map` that allows us to specify code such as `x.map(|a: u32| a + 5)`, where we add 5 to every element of the collection `x`. This is a fairly simple example, which we think suffices our needs, however as we previously mentioned Rust allows composition of iterators, where the results of one iterator are fed to the next, raising a need for a more robust specification. We can meet it with additional *ghost* information about the values that have been produced by the iteration so far, *ghost* sequence `produced`. The state transition predicate is extended to consider the additional information with the conjunct stating that $it'.\mathtt{produced} = it.\mathtt{produced} \cdot v'$, and also relaying the *ghost* sequence to both pre- and post- conditions. The *completed* predicate states that after the borrow ends, no more values will be produced. These fruitful predicates enable us to correctly reason about closures and their effects in `Map`.

$$\begin{aligned} it \overset{v}{\rightsquigarrow} it' \triangleq\ & \exists\, v', fs, |v'| = |fs| = |v| \wedge it.\mathtt{iter} \overset{v'}{\rightsquigarrow} it'.\mathtt{iter} \\ & \wedge\ it'.\mathtt{produced} = it.\mathtt{produced} \cdot v' \wedge\ \mathit{unnest}\,(it.\mathtt{func}, it'.\mathtt{func}) \\ & \wedge\ \left(it.\mathtt{func} = *fs[0] \wedge \hat{}fs[0] = *f[1] \wedge \ldots \wedge \hat{}fs[n] = it'.\mathtt{func}\right) \\ & \wedge \forall i \in [0, |v| - 1],\ \mathit{pre}\left(*fs[i], v'[i], it.\mathtt{produced} \cdot v'[0..i]\right) \wedge \\ & \quad \mathit{post}\left(fs[i], v'[i], it.\mathtt{produced} \cdot v'[0..i], v[i]\right) \\ \mathit{completed}\,(r) \triangleq\ & \mathit{completed}\,(it.\mathtt{iter}) \wedge (*it).\mathtt{func} = (\hat{}it).\mathtt{func} \wedge (\hat{}it).\mathtt{produced} = \varepsilon \end{aligned}$$

This work presents a methodology to specify higher-order iterators in Rust, accounting for their composability. However, due to the inherent nature of Rust, where there is no type for anonymous function, the user needs to specify all three kinds of closures `Fn`, `FnMut` and `FnOnce`, which tends to verbose. In OCaml we do not suffer the same problem, and with our work, we are able to support any given consumer function.

This page is intentionally left blank.

4

Generic Specification for Iteration

In this chapter we focus on presenting the contributions of this thesis. It is structured as follows: first, in Section 4.1, we build a foundation on the definition of cursors and their clients in WhyML; afterwards, in Section 4.2, we show how one might specify Higher-Order Iteration in GOSPEL; and lastly, in Sections 4.3 to 4.5, we bridge the gap between these two languages through translation schemas.

4.1 A first-order specification

Using the work of Fillâtre and Pereira [27] we can define a generic cursor, in WhyML, as a record with 3 fields. Each field maps to the ideas that were the result of their findings.

WhyML

```
type α cursor = abstract {
  mutable visited  : seq α;          (* sequence of visited elements *)
          permitted: seq α → bool; (* permitted predicate *)
          complete : seq α → bool; (* complete  predicate *)
} invariant { permitted visited }
by { visited = empty ; permitted = (fun _ → true); complete  = (fun _ → true); }
```

The first field holds the so-far visited elements of an iteration employed by cursors. The other two are the predicates `permitted/complete`. They take a sequence of elements as input and return a Boolean if it meets their body. Additionally, we decorate this type with an invariant, saying that the sequence of so-far visited elements is always `permitted`. This is a property that must be valid at a function's entry point, potentially and temporarily broken in the middle, but must be fully restored at its exit. This property must also be respected by every type-inhabitant, at creation-time. Type-invariants are introduced as axioms in the local proof context, however, as to not incur in a contradiction we must show that there always exists an inhabitant that suffices the invariant. The generic aspect of it, apart from being a polymorphic data structure, is its ability to express iterations over a collection without imposing a particular type of elements.

In order to use cursors in a meaningful way, we have to define the functions that make it possible to iterate over a collection. We can further generalise the functions `next` and `has_next`. The specification of `next` is as follows:

```
val next (c: cursor α) : α                                          WhyML
  requires { not (c.complete c.visited) }
  ensures  { c.visited = snoc (old c).visited result }
  writes   { c.visited }
```

In order to produce elements, we must be in a state where the iteration of a cursor `c` is not yet finished, encoded in the pre-condition. And as a post-condition, we ensure that the sequence of visited elements is now the affix of its old state and some newly visited element, `result`. We are not particularly interested in knowing what this `result` is, but we know that it must respect the `permitted` predicate, by our type-invariant. Finally, since this results in a modification of the `visited` sequence, we have to be explicit with this effect in memory with the `writes` clause.

The function `has_next` follows a more simple specification. Ensuring, by a post-condition, that the result of calling this fiction is `true` if and only if the sequence of visited elements is not `complete`, as follows:

```
val has_next (c: cursor α) : bool                                   WhyML
  ensures  { result ↔ not (c.complete c.visited) }
```

We can factor these functions and the cursor type into a WhyML module and let us call it `CursorLib`, which can be later used, leaving only the `create` function to be declared and specified. Why3 already has a module for this data-structure but we argue that we made it more generic as we do not have to relate the collection that is being iterated in any other function other than `create`, the only function that differs from one cursor to another.

One way to use this module is to consider an iteration over a polymorphic WhyML sequence. We first clone the type and the two previously introduced functions, `next` and `has_next` and give the declaration of the `create` function a specification:

```
module CursorSeq                                                    WhyML
  clone export cursor.CursorLib

  val create (s: seq α) : (c : cursor α)
    ensures { c.visited   = empty }
    ensures { c.permitted = (fun v → length v ≤ length s ∧
                                ∀ i. 0 ≤ i < length v → v[i] = s[i]) }
    ensures { c.complete  = (fun v → length v = length s) }
end
```

The first post-condition is rather natural, at creation we ensure a cursor with an empty sequence of visited elements. Afterwards we give body to the predicates `permitted`, with the definition of `v` being a prefix of the collection to be iterated, and the `complete` given by the comparison of both lengths. One direct observation that we can conclude is that, we can define all kinds of iteration in WhyML, by only specifying the cursor creation function.

In order to demonstrate how one might use these constructs to sum a sequence of integers, as shown in Section 4.2, let us consider the following example, `sum_seq`.

WhyML

```
  let sum_seq (s: int seq) : int
Ⓐ   ensures { result = sum (fun i → s[i]) 0 (length s) }
  =
    let acc = ref 0 in
    let cursor = CursorSeq.create s in
Ⓑ   while CursorSeq.has_next cursor do
Ⓒ       variant { length s - length cursor.visited }
Ⓓ     invariant { !acc = sum (fun i → cursor.visited[i]) 0 (length cursor.visited) }
Ⓔ     let x = CursorSeq.next cursor in
      acc := !acc + x;
    done;
    !acc
```

First, Ⓐ we ensure that the result of calling this function is equal to the sum of all elements of `s` (given by the function that indexes the *i-th* element of the sequence) in the range of $[0, |s|)$. The sum function is defined in the Why3 standard library under the module `Int`. Afterwards, we can initialise an accumulator as a reference to zero and worry not, mutability is only kept inside the scope of this function, followed by the creation of a cursor over the sequence `s`. The mechanics of the while loop are rather straightforward: Ⓑ while we have elements to produce, Ⓔ produces the next element, and update the accumulator with the application of the consumer function, in this case, the sum of two integers, with its current value and the newly produced element, `x`. After the while-loop, we return the value of the accumulator, `!acc`.

For Why3 to prove the post-condition of this function, we have to give a specification to the while-loop, Ⓒ to Ⓓ. Firstly, as a measure of termination, we argue that this is given by the convergence of the elements to be visited to 0. Secondly, the loop-invariant, is given by the accumulator holding the current sum of all the visited elements. This specification suffices to prove the post-condition, though this is only possible by the fact that the iteration is `complete` and `permitted`, which means that the `visited` sequence is extensionally equal to `s`, and the loop-invariant ends up implying the post-condition.

This simple example allows us to understand that first-order iteration using cursor clients tends to follow a pattern. In the following sections, we will see what is possible to generalise under a common translation schema.

4.2 A higher-order specification

Now consider the following module `Sum` in OCaml. This module takes as an argument another module, `Seq`, that defines a polymorphic type `seq` and a higher-order function `fold`. This function takes as input another function, some type α, a sequence of β and returns an α. Conversely, in the structure of the module `Sum`, we find a function `sum_seq` that calls `fold` with the consumer function being the sum of two elements; an initial value of 0 for the accumulator and the iteration collection being `s`.

```
module Sum (Seq : sig                                                    OCaml
  type α seq
  val fold : (α → β → α) → α → β seq → α
end) =
struct
  let sum_seq (s : α Seq.seq) = Seq.fold (fun a x → a + x) 0 s
end
```

In the previous section, we saw that it is possible to generalise an iteration by cursors by only defining what its `create` function is. We noted that iteration is a two-fold task of defining what it means to iterate something, and what should be the behaviour of the iteration. The first is encoded in an OCaml interface whilst the second in the implementation. In this section we will see how we can express this using GOSPEL specifications, followed by our extensions to the language to correctly parse them and how they are translated to WhyML using Cameleer. We focused on supporting two iteration patterns, iter and fold.

4.2.1 An interface overview

Considering the former example let us see how we can define an iteration over a sequence, using our extension to the GOSPEL language, in terms of `permitted` and `complete`.

```
  val fold: (α → β → α) → α → β seq → α                                  OCaml
```

One can add a GOSPEL specification by first giving names to its result and arguments:

```
  (*@ r = fold func acc col                                              GOSPEL
```

Our extension to the language starts by defining over which kind of iteration pattern this cursor is going to be used, and since in this case we are dealing with a fold, we state this with the keyword `foldspec`:

```
    foldspec                                                   Extended GOSPEL
```

After which we find the body of the `permitted` and `complete` predicates, both of which take the sequence of visited elements. For the user to specify the collection that is being iterated in the predicates, it must use the keyword `collection`:

```
    ~permitted:(fun v → length v ≤ length collection ∧          Extended GOSPEL
                  ∀ i. 0 ≤ i < length v → v[i] = collection[i])
    ~complete:(fun v → length v = length collection)
```

At this point, as we do not have any kind of information about the types, we have to be explicit about some of them: the type of the structure we are iterating over, `structure`, the type of elements that the cursor produces, `elt`, and as auxiliary information, what is the name of the `accumulator` in the header of the specification:

```
    with structure = (β seq), elt = β, accumulator = acc *)    Extended GOSPEL
```

This fairly simplistic specification lets us capture everything we need to define a cursor over a polymorphic sequence. The fact that we are dealing with an untyped AST may

seem that it hinders specification but note that this is not the case. We do, in fact, have to be explicit about what we are iterating over, as from the arguments, we cannot directly infer what type of collection is being iterated. We believe that by being explicit about the type of `structure`, we can account for the freedom that OCaml allows its users to define the order of the arguments and that the collection will not be just one argument but rather multiple; we will see examples of this later on. One could, heuristically (for example, by paying close attention to the types of the consumer function), determine the type of produced elements. However, we delegate some of the typing weight to the user to avoid ill-typing. Lastly, it might seem unclear why we need to identify the `accumulator`, but this is so we know in which position it occurs in a call, and thus get its initial value.

4.2.2 An implementation overview

Having covered how one might define a generic cursor in an interface, let us now see how one might go onto specify its behaviour. This is only possible to do when we call `fold`, in an implementation. Going back to our module `Sum`, recall the function `sum_seq`:

```
let sum_seq (s: Seq.seq) = Seq.fold (fun a x → a + x) 0 s
```
OCaml

This function employs a higher-order iteration function `fold`. However, its implementation is opaque to the user. They do not know how the iteration takes place, but they expect the consumer function be applied to every element of `s`, with an initial accumulator 0. We argue that by hiding how the iteration takes place in logic, we only ask the user to specify the essentials. Lets now see how one might specify the behaviour of this function.

They first start by stating the kind of iteration pattern they expect, in this case is fold:

```
(*@ foldspec
```
Extended GOSPEL

Afterwards, and in no particular order, they let us know what is the iterated collection:

```
    ~collection:s
```
Extended GOSPEL

The termination measure, and in the case of finite and deterministic enumerations, tends do follow a pattern, the convergence of to-be visited elements to 0:

```
    ~convergence:(fun c v → length c - length v)
```
Extended GOSPEL

And lastly, the iteration invariant:

```
    ~inv:(fun a v → a = sum (fun i → v[i]) 0 (length v)) *)
```
Extended GOSPEL

This new syntax may have introduced a lot of new ideas at once. Let us make them clear. The first keyword that stands out is `foldspec`. This keyword lets us know what to generate during translation. All the other parametric fields are GOSPEL logical terms. Note how the parameter `convergence` takes two arguments, the collection that is being iterated and the sequence of visited elements. Both of these arguments are automatically fed by our translation. The `inv`, in the case of a fold, which introduced the notion of accumulator, is also parameterised with two arguments, the accumulator and the sequence of visited elements. They too are automatically fed.

IS	::=	Iteration Specification
	| `iterspec` A^* L	
	| `foldspec` A^* L	
A	::=	Iteration Specification Arguments
	$\sim i : t$	
T	::=	Iteration Specification Types
	$i = p$	
L	::=	Iteration Specification Level
	| ε	
	| `with` $\overline{T}$	

Figure 4.1: Extensions to the GOSPEL grammar.

4.3 Parsing

We had to extend the GOSPEL parser and pre-processor to digest the previously seen specifications. These extensions can be observed in Figure 4.1. They are four new parsing rules with an entry-point in IS. Note that both `iterspec` and `foldspec` are two new keywords. Some aspects of this grammar are not present in the figure but they essentially are:

- The i meta-variable captures all identifiers.
- The t meta-variable captures all valid GOSPEL logical terms.
- The p meta-variable captures patterns.

Both specifications make use of the same parsing rule, IS, but their differ on the level, L. We do not directly enforce the user to be specific at which level its specification is (implementation or interface). Furthermore, note how we refer to types and arguments as identifiers. We did this as to not introduce any new keywords that might pollute the language with its broad meaning, e.g., `complete`. The verification for these names is done in the translation phase, which is arguably late, but we looked at trying to capture both specification levels at the same time, with a short grammar.

For the full, and expanded, syntax rules, see Figure 4.2. These rules are what we expect during translation. Once again, the motivation behind this was to not introduce any new, unnecessary, keywords.

4.4 Quick intermission

Before we tackle how we translated OCaml code annotated with GOSPEL, we will shed some light into how these two languages co-exist. The OCaml AST defines two type of expressions: *program* (Pexp) and *signature* (Sexp). The former makes up regular OCaml found in implementation, while the latter is found in the interface. Both types of expressions

IS	::=	Iteration Specification
	| PS	
	| SS	
PS	::=	Program Specification
	| `iterspec` $PA*$	
	| `foldspec` $PA*$	
SS	::=	Signature Specification
	| `iterspec` $SA*$ L	
	| `foldspec` $SA*$ L	
PA	::=	Program Arguments
	| `~collection` : t	
	| `~inv` : t	
	| `~convergence` : t	
SA	::=	Signature Arguments
	| `~permitted` : t	
	| `~complete` : t	
T	::=	Iteration Specification Types
	| `structure` = p	
	| `elt` = p	
	| `accumulator` = p	
L	::=	Iteration Specification Level
	| `with` $\overline{T}$	

Figure 4.2: Extended GOSPEL syntax during translation.

can have attributes, which is how GOSPEL embeds itself in the OCaml AST. For example, consider the following declaration and specification:

GOSPEL + OCaml

```
val f: int → unit
(*@ f x
    requires x > 0 *)
```

They are first fed to the GOSPEL pre-processor which places the specification where it can be considered as an attribute for the respective expression:

OCaml

```
val f : int → unit
[@@gospel {| f x
             requires x > 0 |}]
```

After the GOSPEL pre-processor turns the specification into an OCaml attribute, GOSPEL relies on the OCaml parser to generate the OCaml AST, and only after that it parses the specifications from the generated AST.

We also note that although it is possible to type-check GOSPEL specifications in the interface, it is unfeasible to type-check the implementation side. This would require keeping up-to-date with the OCaml typed AST. It is for that reason that Cameleer works with the untyped AST.

4.5 Found in Translation

In previous sections we have explored how we can tell an iteration tale, by defining generic cursors and perform first-order iteration using their clients, in WhyML. Afterwards we saw how we can specify a higher-ordered iteration in GOSPEL. We do, however, need a bridge between these two languages. For such, we will present our extension to Cameleer through two translation schemas.

4.5.1 Fold

The first translation schema is concerned with a higher-order function declaration. These can be found in OCaml interface files or as arguments of a functor. The general specification form for a fold is the following:

Extended GOSPEL + OCaml

```
val fold: (α → β → α) → α → β t → α
(*@ r = fold func acc col
    foldspec
    ~permitted: (fun v → term_p) ~complete: (fun v → term_c)
    with structure = τ_s, elt = τ_e, accumulator = acc *)
```

And as we saw before, `permitted` and `complete` are defined using GOSPEL logical terms; `structure` and `elt` are patterns but during translation are redefined as types; and finally, the `accumulator` gets its name from the header.

We are now in a position to be able to translate the specification into a creation of a cursor. In Cameleer, we replace the function declaration with a `scope`, *i.e.*, a namespace. The name of this scope is the capitalised name of function:

WhyML

```
scope Fold
```

We start by introducing the module `seq.Seq`, used to initialise the sequence of visited elements as `empty`, cloning the module `cursor.CursorLib` to expose the generic parts of a cursor, that is, the type and the functions `next` and `has_next`, as such:

WhyML

```
  use seq.Seq
  clone export cursor.CursorLib
```

As we have seen previously, we only need to define the `create` function declaration, which we do so. This function takes a `collection` as argument of type τ_s, this comes from the `structure` argument, and returns a cursor that produces τ_e, taken from the argument `elt`. Afterwards, we translate the GOSPEL logical terms, `permitted` and `complete`, into WhyML terms and these are the corresponding fields in the resulting cursor, as follows:

WhyML

```
  val create (collection: τ_s) : cursor τ_e
    ensures { result.visited = empty }
    ensures { result.permitted = (fun v → term_p) }
    ensures { result.complete  = (fun v → term_c) }
end
```

The translation schema on the implementation side required some gymnastics. We noted that we only had to exploit one node out of the OCaml AST, the `Pexp_apply` node. This node is, essentially, the application of a function, which is what we are ultimately doing. With this insight, we can look for its attributes and try to parse the, possibly existing, specification with our newly introduced rules.

```
fold (fun a e → ...) col x₀                                    Extended GOSPEL + OCaml
(*@ foldspec ~inv: (fun a v → term_i) ~collection: term_c'
    ~convergence: (fun c v → term_v) *)
```

We extend Cameleer to translate the previous specification into a first-order iteration using a cursor client. Let us describe such a translation process in what follows.

Our translation schema starts by initialising a variable that is going to hold the accumulation of of the repeated calls to the consumer function, `acc`. The value of this variable is extracted from the OCaml expression and since we are able its position in the list of arguments, we can get its initial value, x_0:

```
let acc = ref x₀ in                                                        WhyML
```

It is also here that we first use our generated cursor from the interface, using the `create` function from the previously defined `scope`, we create a cursor that iterates over the collection, $term_{c'}$:

```
let cursor = Fold.create term_c' in                                        WhyML
```

Here we find the while-loop, conditioned by the `has_next` function. Note that if this loop ends, then when it does, the cursor predicate `complete` holds:

```
while Fold.has_next cursor do                                              WhyML
```

The variant of the while-loop is $term_v$, where we apply, under the hood, $term_{c'}$, the collection that is being iterated, and the current sequence of visited elements:

```
    variant { term_v term_c' cursor.visited }                              WhyML
```

On the other hand, the invariant of the while-loop is $term_i$, where we apply, the current value of the accumulator[1] followed by sequence of visited elements:

```
  invariant { term_i !acc cursor.visited }                                 WhyML
```

We produce the next element in the iteration and store it in `x`. Followed by the application of the consumer function to the accumulator and the produced value, `x`.

```
  let x = Fold.next cursor in                                              WhyML
  acc := (fun a e → ...) !acc x;
done;
```

Alas, we return the contents of the `accumulator`:

```
!acc                                                                       WhyML
```

[1]This is only the case of a fold. If we are dealing with an iter, we only apply the sequence of visited elements.

4.5.2 Iter

Quoting Jean-Christophe Filliâtre, one might consider iter a degenerate version of a fold [13]. That is why we decided to present first the translation schema for a fold, rather than the other way around.

These differ in some minor details but which ensure their expected behaviour. Firstly, the specification to a higher-order iteration function that applies a consumer function to all elements, is almost the same as in the previous case. However, the iteration pattern is described with the keyword `iterspec` and this kind does not need an accumulator:

Extended GOSPEL + OCaml

```
val iter: (α → β) → α t → unit
(*@ r = iter f l
    iterspec ~permitted: (fun v → term_p) ~complete: (fun v → term_c)
    with structure = τ_s, elt = τ_e*)
```

Again, this is translated into a `scope` that declares and specifies the `create` function:

WhyML

```
scope Iter
  use seq.Seq
  clone export cursor.CursorLib
  val create (collection: τ_s) : cursor τ_e
    ensures { result.visited = empty }
    ensures { result.permitted = (fun v → term_p) }
    ensures { result.complete  = (fun v → term_c) }
end
```

Conversely, a call to `iter` is very much identical to the previous case, but note how the invariant, `inv`, is a function that only expects, `v`, the sequence of visited elements:

Extended GOSPEL + OCaml

```
iter (fun e → ...) col
(*@ iterspec ~inv: (fun v → term_i) ~collection: term_c' ~convergence: (fun c v → term_v) *)
```

Cameleer translates this into the following WhyML code. As before we start by creating our cursor over a collection $term_{c'}$:

WhyML

```
let cursor = Iter.create term_c' in
```

The while-loop head and its specification are also pretty similar to the one presented before. Note, once more, that the invariant is only applied with the visited sequence:

WhyML

```
while Iter.has_next cursor do
    variant { term_v col cursor.visited }
  invariant { term_i cursor.visited }
```

The biggest difference between fold and iter lies in what the we do with the consumer function. As before, we produce an element `x`, but when applying the consumer function to it, we have to force Why3 to call it with a *let-binding*:

WhyML

```
  let x = Iter.next cursor in
  let _ = (fun e → ...) x in
  ()
done
```

And as expected, the body of the while-loop is of type unit. After-which we are done with the iteration. These translations, for fold and iter, allows us to capture both the reasoning about their iteration and their expected behaviour as WhyML programs.

4.5.3 What about nesting?

One natural question that might arise is what about nested iteration? By which we mean: what if the consumer function also employs some sort of higher-order iteration?

Following classical deductive reasoning, we know that if we find a while-loop inside one another, the inner-most must preserve the invariant of the outer-most. Consider the following example, adapted to our translation:

```
while has_next cursor do                                    WhyML
    variant { V c v }
 Ⓐ invariant { I !a v }
  ...
  while has_next cursor' do
      variant { V' c' v' }
   Ⓑ invariant { I' !a' v' }
    ...
  done;
  ...
done
```

The matter that concerns us the most is how can we preserve the invariant Ⓐ, parameterised with the current value of its accumulator, `!a`, and the so-far visited elements, `v`, inside the second while-loop, Ⓑ. We gave this problem some thought. We could, for instance, add a copy of the invariant Ⓐ in the inner-loop, which would preserve it. However, this approach would require us to keep track of past invariants and, more importantly, restrain the user from defining its own invariants. Our solution does not keep track of the whole invariant, but rather its arguments: for folds the value of the accumulator and the visited elements, and for iters only the latter. While translating, we propagate these arguments to the invariants in inner levels. That said, the invariants would have to be:

```
Ⓐ invariant { I !a v }                                      WhyML
Ⓑ invariant { I' !a' v' !a v }
```

Or as a formula, append the necessary arguments of each level above the current, l:

$$\texttt{invariant \{ I } \left(\bigoplus_{i=0}^{l} \texttt{(!a}_i\texttt{)? (v}_i\texttt{))}\right) \texttt{ \}} \qquad \textit{WhyML}$$

One could provide more information, through partial applications of the invariant and these would naturally precede the iteration luggage. And so we get the following general form, where ∂ is a partially applied argument:

$$\texttt{invariant \{ I } \overline{\partial}\ \left(\bigoplus_{i=0}^{l} \texttt{(!a}_i\texttt{)? (v}_i\texttt{))}\right) \texttt{ \}} \qquad \textit{WhyML}$$

4.5.4 A slight optimisation

Throughout the development of our translation schema, we noticed Why3 is not a big fan of anonymous functions applications. There is almost always an extra step during a proof, with the computation of the goal or in-lining the VC. We decided to lift some weight on our side. If we find an anonymous function, we perform a *lambda lift*:

$$(\lambda x_1 \ldots x_n.E)\ a_1 \ldots a_n \longrightarrow \texttt{let}\ x_1 = a_1\ \texttt{in}\ \ldots\ \texttt{let}\ x_n = a_n\ \texttt{in}\ E$$

In our specification, we usually find them as terms, such as, `permitted`, `complete`, `convergence` and `inv`. It were these last two who proved to be troublesome and needed extra work. This rather innocent looking optimisation turned out to have a significant impact in the proof time, and proof steps necessary, on some case studies.

4.5.5 A digression on the development

Before we started our development, we noticed that the Cameleer was based on a branch (`implementations`) of GOSPEL which was very divergent from the main branch. We merged both branches and proposed a pull request to the official repository. Our local clone is based on the re-based repository. Cameleer was not directly compatible with the main branch of GOSPEL. We had to accommodate the tool with the new way that GOSPEL defines *type-invariants*. Prior to the rebase, and considering a type-invariant that models positive integers, they would be defined as follows:

GOSPEL + OCaml
```
type t
(*@ model n : int
    invariant n ≥ 0 *)
```

And in turn, translated into the following WhyML:

GOSPEL + OCaml
```
type t = abstract {
  ghost n : int
} invariant { n ≥ 0 }
```

However, GOSPEL added the `with` keyword which introduces a name for the type being specified in the invariant:

GOSPEL + OCaml
```
type t
(*@ model n : int
    with t invariant t.n ≥ 0 *)
```

This cannot be directly translated into WhyML as this one does not need to refer to the type being specified. This was an easy fix. We looked at removing the prefix of *terms*, such as `t.n`, and replace it with just `n`. Which would now translate into correct WhyML code.

5

Case studies

In this chapter we focus on showcasing some case studies that we explored to consolidate our hypothesis on translating higher-order iteration to first-order cursor clients. Our case studies are over *sequences*, *binary trees* and *graphs*.

5.1 Sequences

We like to think about sequences as an abstract concept. Every linear enumeration falls in this category. So whenever we are talking about lists, arrays or matrices, they can all be thought as sequences.

In this section we will present some case studies about sequences. We have seen, previously, how to specify higher-order iteration on sequences, which might help the reader follow better.

From our recipe follows that the first step is to specify a declaration of a higher-order function. We can do this all in a single `.ml` file, through *functors*, under a module `Sequences`:

Extended GOSPEL + OCaml

```
module Sequences ( S: sig
    type α seq
    val fold: (α → β → α) → α → β seq → α
    (*@ r = fold f acc s
      foldspec ~permitted:(fun v → length v ≤ length (collection) ∧
                           ∀ i. 0 ≤ i < length v → v[i] = (collection)[i])
      ~complete:(fun v → length v = length (collection))
      with structure = (β seq), elt = β, accumulator = acc *)

    val iter: (α → β) → α seq → unit
    (*@ r = iter f c
      iterspec ~permitted:(fun v → length v ≤ length (collection) ∧
                           ∀ i. 0 ≤ i < length v → v[i] = (collection)[i])
      ~complete:(fun v → length v = length (collection))
      with structure = (α seq), elt = α *)
    end ) =
struct
```

As a first example, consider a function that creates a *queue* out of a sequence. This function iterates every element of `s` and enqueues in `queue`. This is attained as follows:

```
let queue_of_seq s =                                                    OCaml
  let queue = Queue.create () in
  S.iter ( fun x → Queue.add x queue ) s
```

A `Queue` is logically modeled through a sequence of elements called `view`. This model is included as a specification to the OCaml Standard Library in Cameleer. We state our invariant as the current `view` be extensionally equal to the prefix of `s` up until the length of the visited sequence `v`. This could have, in turn, be stated as the `view` be equal to `v`, but we want to evince that both definitions are valid. Termination is trivially specified.

```
(*@ iterspec ~convergence:(fun c v → length c - length v)      Extended GOSPEL + OCaml
    ~inv:(fun v → queue.Queue.view == s[..(length v)]) ~collection:s *);
```

After-which, we return the queue:

```
  queue                                                                 OCaml
```

The post-condition to this function is a resulting queue with all the elements of `s`:

```
(*@ r = queue_of_seq s                                                  GOSPEL
    ensures r.Queue.view = s *)
```

It may not be clear but this function has effects. The consumer function calls `Queue.add`, which performs the modification of the queue. Nonetheless, as this function was given a well-fitted contract, Why3 is able to introduce it in the context of the proof.

Its sister data-structure, the `Stack`, follows a similar specification. But in this particular case, we have to `reverse` the `view`, seeing as elements are inserted at the beginning. The function `reverse` introduces extra definitions in the logical context of the proof, which slightly hinders its proof time. Its full implementation and specification are as follows:

```
let stack_of_seq s =                                          Extended GOSPEL + OCaml
  let stack = Stack.create () in
  S.iter ( fun x → Stack.push x stack ) s
  (*@ iterspec ~inv:(fun v → Seq.reverse stack.Stack.view == s[..(length v)])
      ~collection:s ~convergence:(fun c v → length c - length v) *);
  stack
(*@ r = stack_of_seq s
    ensures Seq.reverse r.Stack.view = s *)
```

This function also has effects. The function `push` performes modifications in-place. This is, once more, not a problem. We have not stated why, but let us do now. Recall that for an iter, we call the consumer function in a *let-binding* but discard its value, unit. This "forces" Why3 to introduce the contract of the function in the local context of the proof, which in the case of `push` is an in-place append to the head of the `view`.

Nonetheless, both functions were fully dispatched by the SMT solver Alt-Ergo in less than one second. And finally, we can close the module, and consequently this section:

```
end                                                                     OCaml
```

5.2 Binary trees

Shifting our attention to the second batch of examples we find *binary trees*. These examples highlight that we can model iteration through Abstract Data Types (ADTs), although we cheat a little bit. We will see two examples of iteration over trees: through an iter we will count how many elements are greater than some integer node in a tree, and through a `fold` over levels, we will compute the height of a tree. Let us consider in the following module, `Tree`, we start by defining what a tree is:

OCaml
```
module Tree ( T: sig
    type α tree =
    | E
    | Node of α tree * α * α tree
```

The first thing we have to think about is how are we going to traverse the tree. We chose to do in a *in-order* fashion. We can define a function that receives a polymorphic tree and returns the a sequence where the elements appear in that order:

GOSPEL
```
    (*@ function inorder (t: α tree) : α seq = match t with
        | E  → []
        | Node (l, e, r) → (inorder l) @ (e :: inorder r) *)
```

We will do a little bit of cheating here. We will call the function `in-order` to flatten the `collection` tree into a sequence and reason about it as such:

Extended GOSPEL + OCaml
```
    val iter: (α → β) → α tree → unit
    (*@ r = iter func c
      iterspec
      ~permitted:(fun v → length v ≤ length (elements collection) ∧
                    ∀ i. 0 ≤ i < length v → v[i] = (elements collection)[i])
      ~complete:(fun v → length v = length (elements collection))
      with structure = (α tree), elt = α *)
```

As to model an iteration over levels, we can employ a similar train of thought. We define the function `order_level` that returns a sequence with all the nodes at some distance `l` from the root:

GOSPEL
```
    (*@ function order_level (t: α tree) (l: int) : α seq = match t with
        | E  → []
        | Node (left, e, right) →
          if l = 0 (* we are the l-th level, return [e] *)
            then (e :: [])
          else     (* explore the level below the current *)
            (order_level left (l - 1)) @ (order_level right (l - 1)) *)
```

Additionally, lets also define a function that actually computes the height of a tree, this is trivially given by adding one to the greatest height from either the left branch or right:

GOSPEL
```
    (*@ function height (t: α tree) : int = match t with
        | E → 0
        | Node (l, _, r) → 1 + max (height l) (height r) *)
```

Finally, in terms of `permitted` and `complete` we combine both `order_level` and `height`, where we, once more, follow the same logic as before: turn the structure into a flattened sequence and reason about as one, as follows:

Extended GOSPEL + OCaml

```
val fold_level: (α → β → β) → α tree → β → β
(*@ r = fold_level func c acc
  foldspec
  ~permitted:(fun v → length v ≤ height collection ∧
                ∀ i. 0 ≤ i < length v → v[i] = (order_level collection i))
  ~complete:(fun v → length v = height collection)
  with structure = (α tree), elt = (α seq), accumulator = acc *)
end ) =
struct
```

Note how now we produce elements of type `(α seq)`. And by our definition of `permitted`, the `visited` sequences holds, at each position `i`, all the elements of the tree at level `i`.

Our first example counts how many integer nodes are greater than some threshold. This is similar to another case study that does the same computation over sequences. However, here we do it via an iter that traverses the tree with a consumer function that compares the elements with a threshold and increments a counter when true, as such:

Extended GOSPEL + OCaml

```
let gt_tree tree t =
  let count = ref 0 in
  T.iter (fun e → if e > t then count := !count + 1) tree
  (*@ iters ~collection:tree ~convergence:(fun c v → length (T.inorder c) - length v)
     ~inv:(fun v → !count = numof (fun i → (T.inorder t)[i] > t) 0 (length v)) *);
  !count
```

To correctly express this behavior in logic terms we will use a function provided by the Why3 Standard Library, the function `numof`. This function counts the number of elements in a given range that satisfy a given predicate `p`, by this definition:

$$\sum_{i=l}^{u} \begin{cases} 0 & \text{if } \neg \texttt{p(i)} \\ 1 & \text{if } \texttt{p(i)} \end{cases}$$

In our case, the predicate `p`, is the comparison of the *i-th* element of the tree with the threshold. The specification for the function `gt_tree` is given in a similar fashion:

GOSPEL

```
(*@ r = gt_tree tree t
  ensures r = numof (fun i → (T.inorder tree)[i] > t) 0 (length (T.inorder tree)) *)
```

One other approach would be to consider all the sorted permutations of the visited sequence and find the index where all the elements less or equal to the threshold are to the left, in which case the count should be the length sub-sequence to the right, or if there is no such index, then the count should be 0. We leave this exercise to the reader.

Our second example computes the height of a tree. To do this, we can iterate by levels and count how many are there. This is given by increasing the accumulator, initially 0, at each iteration step. The invariant of this iteration is given as the accumulator being the length of the visited sequence. Termination is trivially given.

```
let tree_height t =                                          Extended GOSPEL + OCaml
  T.fold_level (fun a e → a + 1) t 0
  (*@ foldspec ~inv:(fun a v → a = length v)
      ~collection:t ~convergence:(fun c v → T.height c - length v) *)
(*@ r = tree_height t
    ensures r = T.height t *)
```

Alt-Ergo fully dispatches both, taking less than 10 milliseconds for `tree_height` and close to 40 seconds for `gt_tree`. We can now close the module and therefore this section:

```
end                                                                          OCaml
```

5.3 Graphs

This is our *crème de la crème*. When we started this thesis, we were motivated to further prove a subset of the OCamlGraph library. Still, we found ourselves having to extend our toolbox to reason about iteration correctly. Despite that, our primary case studies were about graphs, specifically the module `Oper`[1] from the OCamlGraph library. In this section, we will present them and, as an extra, a refactored proof of the `check_path` function, which was previously proven correct by Castanho [4]. However, contrary to the mentioned work, we do not alter the original OCaml implementation as to avoid higher-order.

5.3.1 A logical model

First of all, what is a graph? The mathematical definition of a graph is through sets, and for our purposes these will be finite. It is modeled using a domain, the set of vertices that make up a graph, and a function that takes a vertex and returns a finite set vertices, these will be our successors of a vertex. In other words, the set of edges in the graph. The module `Oper` starts by defining the type of a vertex and a graph:

```
module Oper (G : sig                                                         OCaml
  type vt (* arbitrary vertex type *)
  type gt (* arbitrary graph type *)
```

The type graph is annotated with a logical model. It is made up of a domain `dom` and the map `suc`. Additionally, we give this type an invariant stating that the graph's domain is closed under the map `suc`, and for every element that is not in the domain, its set of successors is empty — this statement is rather strange but it is helpful in the proof of a graph's `complement`, which we shall present later in this section.

```
(*@ model dom: vt fset                                                       GOSPEL
    model suc: vt → vt fset
    with t invariant (∀ v1, v2. mem v1 t.dom ∧ mem v2 (t.suc v1) → mem v2 t.dom) ∧
                     (∀ v1. ¬ (mem v1 t.dom) → (t.suc v1) == empty) *)
```

[1] See here: `https://github.com/backtracking/ocamlgraph/blob/master/src/oper.ml`

An empty graph is not particularly interesting. We can define two functions that grow the graph in its domain and its successors. These functions are `add_vertex` and `add_edge`. The first one takes a graph and a vertex as input and returns a new graph. The second takes a graph and two vertices, the latter being the successor of the former; this function also returns a graph. The specification for the first functions states that the domain of the returning graph, `g'`, is the addition of the vertex `v` to the domain of `g`, and for every entry in the map of successors, they remain unchanged — note that we cannot say that its map of successors is empty as this vertex could have already been in the domain.

GOSPEL + OCaml

```
val add_vertex : gt → vt → gt
(*@ g' = add_vertex g v
    ensures g'.dom == add v (g.dom)
    ensures ∀ v. (g'.suc v) == (g.suc v) *)
```

The specification for the addition of an edge is as follows: We ensure that both domains of `g` and `g'` are the same, and the successor map of `g'` is the map of `g` with the, possibly changed, entry for `v` with the addition of `v'`:

GOSPEL + OCaml

```
val add_edge : gt → vt → vt → gt
(*@ g' = add_edge g v v'
    ensures g'.dom == g.dom
    ensures g'.suc = Map.set g.suc v (add v' (g.suc v)) *)
```

To test membership over a graph we defined two functions, one that tests the existence of some vertex in a graph, `mem_vertex`, and another that tests the membership of some vertex in the successor set of another, `mem_edge`. Recall the `[@logic]` attribute for these functions to be considered in both program code and specification. Their specification can be thought as queries and are follows:

GOSPEL + OCaml

```
val [@logic] mem_vertex: gt → vt → bool
(*@ r = mem_vertex g v
    ensures r ↔ mem v g.dom *)

val [@logic] mem_edge: gt → vt → vt → bool
(*@ r = mem_edge g v v'
    ensures r ↔ mem v' (g.suc v) *)
```

Finally, we consider two functions that are used to build graphs as a whole. One that returns an empty graph, `empty`. And another that returns a copy of one, `copy_graph`. Their specification are rather simple:

GOSPEL + OCaml

```
val empty : unit → gt
(*@ g = empty ()
    ensures g.dom = empty ∧ g.suc = (fun _ → empty) *)

val copy_graph: gt → gt
(*@ g' = copy_graph g
    ensures ( g.dom == g'.dom )
    ensures ( ∀ v. g.suc v == g'.suc v) *)
```

We have covered how to build graphs and how to make queries over one. But we are yet to mention how to iterate over one. The module `Oper` uses three functions to iterate over a graph, `fold_vertex`, `fold_succ` and `fold_edges`. Logically, they all iterate over finite sets. Let us see how we can specify an iteration over vertices with the first function:

```
val fold_vertex : (vt → α → α) → gt → α → α
```
OCaml

In order to produce elements, the visited set has to be a subset of the graph's domain; and have to be all distinct as a sequence. The iteration is complete when the have visited all elements of the graph's domain. From the types of arguments we can conclude that we are iterating over a graph and producing vertices.

```
(*@ r = fold_vertex func graph acc                                    Extended GOSPEL
  foldspec
  ~permitted:(fun v → subset (Set.of_seq v) collection.dom ∧ Seq.distinct v)
  ~complete:(fun v → Set.of_seq v == collection.dom)
  with structure = gt, elt = vt, accumulator = acc *)
```

One other that iterates over a graph is `fold_succ`. This function takes as argument a graph and some vertex and iterates over its successors. Its signature is as follows:

```
val fold_succ : (gt → α → α) → α → gt → vt → α
```
OCaml

This iteration is also over sets, however our structure of iteration is a pair (`gt`, `vt`) seeing as it does not make much sense to iterate over a vertex without a graph nor over successors of which we do not know their origin. Both `permitted` and `complete` are in regard to the set of successors of some vertex `s` of some graph `g`. This iteration produces vertices.

```
(*@ r = fold_succ func acc pair                                       Extended GOSPEL
  foldspec
  ~permitted:(fun v → let (g, s) = collection in
                      subset (Set.of_seq v) (g.suc s) ∧ Seq.distinct v)
  ~complete:(fun v → let (g, s) = collection in
                     cardinal (Set.of_seq v) = cardinal (g.suc (s)))
  with structure = (gt * vt), elt = vt, accumulator = acc *)
```

Note how now we are less expressive in the `complete` predicate. We only say that they have to be equal in their cardinality but not in the enumeration of their elements, as was in the previous case. This is also fine, a lemma in the `Cardinal` module under the Why3 Standard Library, `subset_eq`, states that if a set `s1` is a subset of `s2` and their cardinality is the same, then they are the same set.

Last but not least, to iterate over all the edges of a graph, we find the function `fold_edges`. We found it easier to flatten the graph into a set of edges. We consider edges a pair of vertices. This can be obtained with a fuction `flatten_graph`. However we not do give it a body but rather axiomatize its ideal behaviour:

```
type edge = vt * vt                                                   Extended GOSPEL + OCaml
(*@ function flatten_graph (g: gt) : edge fset *)
(*@ axiom flat: ∀ g r v v'. r = flatten_graph g → mem v' (g.suc v) → mem (v, v') r *)
```

Or in other words, for any given graph `g` the result of calling `flatten_graph g` is a finite set with all the edges, these being pairs of vertices where the first element in the pair is the source and the second the destination.

Finally, we can specify the fuction `fold_edges` with the `permitted` being an iteration over sets, in this case the set of edges and where the `complete` compares the cardinality of both sets.

Extended GOSPEL + OCaml

```
val fold_edges : (edge → α → α) → α → gt → α
(*@ r = fold_edges func acc graph
  foldspec
  ~permitted:(fun v → subset (Set.of_seq v) (flatten_graph collection) ∧ Seq.distinct v)
  ~complete:(fun v → cardinal (Set.of_seq v) = cardinal (flatten_graph collection))
  with structure = gt, elt = edge, accumulator = acc *)
 end) =
struct
```

This is everything we need to implement and specify the `Oper` module. In this module we will seek to prove four operations: the intersection and union of two graphs, the complement and the mirror of one. The first three employ higher order iteration inside the consumer function of another, this is great as we will see how we deal with nesting. We will showcase these case studies by first showing their specification, followed by the loop invariants.

5.3.2 Mirror

The first example is the simplest. Given a graph `g`, we wish to mirror its edges. This means that for any given edge (v, v'), the resulting graph should have (v', v). This can be done by first creating an empty graph with the vertices of `g` and iterate every edge and adding the mirror of that edge to the accumulating graph.

First, let us define a function that copies the vertices of a graph, `copy_vertices`. This function iterates over all the vertices of a graph and adds them in an accumulator graph, that starts empty. The invariant is rather simple, we state that for any visited vertex, this is contained in the domain of the accumulator graph and its set of successors is empty:

Extended GOSPEL + OCaml

```
let copy_vertices (g: G.gt) =
  G.fold_vertex (fun g' v → G.add_vertex g' v) g (G.empty())
  (*@ foldspec ~collection:g ~convergence:(fun c v → cardinal c.G.dom - length v)
    ~inv:(fun a v → ∀ e. (mem e v ↔ mem e a.G.dom) ∧ (a.G.suc = (fun _ → empty))) *)
(*@ r = copy_vertices t
    ensures r.G.dom == t.G.dom
    ensures r.G.suc = (fun _ → empty) *)
```

Now to get the mirror of a graph, we can employ a `fold_edges` over `g`, and adding the mirror of an edge to the accumulator graph that has the vertices of `g`. Note how we are partially applying a new argument to the invariant, the iterated graph `g`. Termination is trivially obtained, by the convergence of the cardinal of set of edges to be visited to zero.

We ensure that calling this function returns a graph with the same domain and all edges of g mirrored.

Extended GOSPEL + OCaml

```
let mirror g =
    G.fold_edges
      (fun e g' →
         let (v1, v2) = e in
         G.add_edge g' v2 v1) (copy_vertices g) g
(*@ foldspec ~inv:(mirror_inv g) ~collection:g
  ~convergence:(fun c v → cardinal (G.flatten_graph c) - length v)*)
(*@ g' = mirror g
    ensures ( g.G.dom == g'.G.dom )
    ensures ( ∀ v1 v2. G.mem_edge g v1 v2 → G.mem_edge g' v2 v1 ) *)
```

The iteration invariant follows almost the same nature of the above OCaml implementation. For every visited edge, we say that its mirror exists in the accumulator graph. And the domain on the accumulator is equal to the domain of the graph being iterated.

GOSPEL

```
(*@ predicate mirror_inv (g: G.gt) (visited: G.edge seq) (acc: G.gt) =
   ( ∀ v. mem v (Set.of_seq visited) →
      let (v1, v2) = v in G.mem_edge g v1 v2 → G.mem_edge acc v2 v1 )
 ∧ ( acc.G.dom == g.G.dom ) *)
```

5.3.3 Union

The union of two graphs is a very interesting case study. A union of two graphs g1 and g2 is a new graph that is the union of both domains and every successor set of both graphs. This function iterates over vertex s of g1 and inside the consumer function over every successor of s while adding both the vertex s and its successors in a copy of g2, the accumulator. We ensure, as a postconditon, that calling this function results in a new graph with the properties previously stated:

Extended GOSPEL + OCaml

```
let union g1 g2 =
  G.fold_vertex
    (fun g s →
      G.fold_succ (fun e g → G.add_edge g s e) (G.add_vertex g s) g1 s
      (*@ foldspec ~inv:(union_inner g1 g2 s) ~collection:(g1,s)
          ~convergence:(fun (g, s) v → cardinal (g.G.suc s) - length v ) *) )
    g1 (G.copy_graph g2)
(*@ foldspec ~inv:(union_outer g1 g2) ~collection:g1
    ~convergence:(fun c v → cardinal c.G.dom - length v) *)
(*@ g'' = union g g'
    ensures g''.G.dom == union g.G.dom g'.G.dom
    ensures ∀ src. (g''.G.suc src) == (union (g'.G.suc src) (g.G.suc src)) *)
```

As we are dealing with two higher order iterators we need to give both an iteration invariant. The first, `union_outer` is the invariant of the outer loop, `fold_vertex`. This invariant states that the domain of the accumlator graph is the union of the set of visited elements, *i.e.*, the visited domain of g1, and the domain of g2. Addionally, for every visited

element `v`, its successors in `acc` is the union between its successor set in `g1` and `g2`. Finally, for every element to be visited `x`, its successor set is equal to the successor set in `g2`.

GOSPEL
```
(*@ predicate union_outer (g1: G.gt) (g2: G.gt)
                          (visited: G.vt seq) (acc: G.gt) (* outer iteration *) =
  ( acc.G.dom == union (Set.of_seq visited) g2.G.dom )
∧ ( ∀ v. mem v (Set.of_seq visited) → acc.G.suc v == union (g1.G.suc v) (g2.G.suc v))
∧ ( ∀ x. mem x (diff acc.G.dom (Set.of_seq visited)) → acc.G.suc x == g2.G.suc x ) *)
```

The inner invariant `union_inner` is a tad bit tricky. This invariant has the addded resposability of preserving the `union_outer` predicate. This predicate is the first that showcases how we automatically expose the information of the outer invariant. In order to correctly specify this invariant, we need information about both graphs and the vertex that is being iterated at the moment. Its singature is as follows:

GOSPEL
```
(*@ predicate union_inner (g1: G.gt) (g2: G.gt) (src: G.vt)
                          (visited': G.vt seq) (acc': G.gt) (* inner iteration *)
                          (visited: G.vt seq) (_acc: G.gt)  (* outer iteration *) =
```

Our first statement preserves part of the previous invariant. The domain of the accumulator graph, `acc`, is the union of the set of visited vertices and the domain of `g2`:

```
    ( acc'.G.dom == union (Set.of_seq visited) g2.G.dom )
```

For every vertex to be visited, its successor set in the accumulator graph is equal to that from the graph `g2`:

```
  ∧ ( ∀ v. mem v (diff acc'.G.dom (Set.of_seq visited)) → acc'.G.suc v == g2.G.suc v )
```

For every visited vertex different from the current one, its set of successors is the union of those in the graphs `g1` and `g2`:

```
  ∧ ( ∀ v. v ≠ src → mem v (Set.of_seq visited) →
          acc'.G.suc v == union (g1.G.suc b) (g2.G.suc b) )
```

And finally, the set of successors for the current element is the union between the set of so far visited successors and the successors in `g2`:

```
  ∧ ( acc'.G.suc src == union (Set.of_seq visited') (g2.G.suc src) ) *)
```

Note that we did not use the second accumulator. This is due to the accumulator `acc` already being an updated version of `_acc`. However this is not always the case, one can imagine a case where the accumulator of the inner iteration is not related to accumulator of the outer.

5.3.4 Intersection

The intersection of two graphs `g1` and `g2` is a new graph where all vertices and successors exist in both graphs. As was in the previous example, this function also employs the same kind of iteration but we start accumulating in an empty graph. First we iterate over the vertices of `g1` and check they belong in `g2`, if that is the case we iterate over its successors in `g1` and check if they too belong in `g2`, adding that edge if that is the case.

Extended GOSPEL + OCaml

```
let intersect g1 g2 =
  G.fold_vertex
    (fun g v →
      if G.mem_vertex g2 v then
        G.fold_succ
           (fun e g →
              if G.mem_edge g2 v e
              then G.add_edge g v e
              else g) (G.add_vertex g v) g1 v
      (*@ foldspec ~inv:(intersect_inner g1 g2 v) ~collection:(g1, v)
          ~convergence:(fun (g, s) v → cardinal (g.G.suc s) - length v ) *)
      else g) g1 (G.empty())
(*@ foldspec ~inv:(intersect_outer g1 g2) ~collection:g1
    ~convergence:(fun c v → cardinal c.G.dom - length v) *)
```

This function returns a new graph `g` where its domain is the intersection of the domains of `g1` and `g2`, and their successors sets:

GOSPEL

```
(*@ g = intersect g1 g2
    ensures g.G.dom == inter g1.G.dom g2.G.dom
    ensures ∀ v. g.G.suc v == (inter (g1.G.suc v) (g2.G.suc v)) *)
```

The outer invariant is relatively simple. Actually, both invariants are. We state that the accumulator graph has a domain which is the result of the intersection of the set of visited vertices with the domain of `g2`. And for every visited vertex, its successor set in `acc` is the intersection of the respective sets in `g1` and `g2`:

GOSPEL

```
(*@ predicate intersect_outer (g1: G.gt) (g2: G.gt)
                              (visited: G.vt seq) (acc: G.gt)  (* outer iteration *) =
   ( acc.G.dom == inter (Set.of_seq visited) g2.G.dom )
 ∧ ( ∀ v. mem v (Set.of_seq visited) → acc.G.suc v == inter (g1.G.suc v) (g2.G.suc v) ) *)
```

The inner invariant starts by preserving the first statement of the outer invariant, followed by a partial preservation of the second statement up until `src`. This is because we have not finished the iteration of `src`, which unables us to say anything about its successors. This is sustained by the last statement, the successor set of `src` in the accumulator is the intersection of the set of visited successors and the respective set in `g2`:

GOSPEL

```
(*@ predicate intersect_inner (g1: G.gt) (g2: G.gt) (src: G.vt)
                              (visited': G.vt seq) (acc': G.gt) (* inner iteration *)
                              (visited: G.vt seq) (_acc: G.gt)  (* outer iteration *) =
   ( acc'.G.dom == inter (Set.of_seq visited) g2.G.dom )
 ∧ ( ∀ v. v ≠ src → mem v (Set.of_seq visited) →
             acc'.G.suc v == inter (g1.G.suc v) (g2.G.suc v))
 ∧ ( acc'.G.suc src == inter (Set.of_seq visited') (g2.G.suc src) ) *)
```

It is worth mentioning that the three previous functions over sets, `union`, `inter` and `diff` were provided by the module `Set` from the Why3 Standard Library.

5.3.5 Complement

The complement of a graph `g` is another graph `g'` where all the edges present in `g` are not present in `g'` and all the edges not present in `g` are present in `g'`. The domains are the same. This innocent looking function has a very verbose specification — or at least in terms of invariants. To compute the complement of a graph we iterate over the domain of the graph squared; if some edge does not exist in `g`, we add it to the accumulator graph.

Extended GOSPEL + OCaml

```
let complement g =
  G.fold_vertex
    (fun g' v →
       G.fold_vertex
         (fun g' w →
           if G.mem_edge g v w then g'
           else G.add_edge g' v w) g g'
    (*@ foldspec ~inv:(complement_inner g v) ~collection:g
        ~convergence:(fun c v → cardinal c.G.dom - length v) *) )
  g (copy_vertices g)
(*@ foldspec ~inv:(complement_outer g) ~collection:g
    ~convergence:(fun c v → cardinal c.G.dom - length v) *)
```

The postcondition for this function is a graph where for any two vertices `(src, dst)` in the domain of `g`, if an edge from `src` to `dst` does not exist in `g` it exists in the resulting graph and the other way round. Additionally, their domains are the same.

GOSPEL

```
(*@ g' = complement g
    ensures ( ∀ src dst. G.mem_vertex g src → G.mem_vertex g dst →
                      ¬ G.mem_edge g src dst ↔ (G.mem_edge g' src dst)  )
    ensures ( ∀ src dst. G.mem_vertex g src → G.mem_vertex g dst →
                        G.mem_edge g src dst ↔ ¬ (G.mem_edge g' src dst)  )
    ensures ( g.G.dom == g'.G.dom ) *)
```

As both invariants are very verbose let us talk about them one statement at a time. The signature of the outer invariant is the following:

GOSPEL

```
(*@ predicate complement_outer (g: G.gt)
                               (visited: G.vt seq) (acc: G.gt) (* outer invariant *) =
```

We start by stating that the accumulator has the same domain as `g`:

```
    ( acc.G.dom == g.G.dom )
```

Afterwards, by stating that every vertex visited, `v`, if some other vertex `v'` that belongs to the graph then if there is not an edge from `v` to `v'` in `g` then that edge exists in the accumulator graph. Conversely, if some edge `v` to `v'` exists in `g` then it cannot exist in `acc`:

```
  ∧ ( ∀ v v'. mem v (Set.of_seq visited) → G.mem_vertex g v' →
        ¬ G.mem_edge g v v' ↔ G.mem_edge acc v v' )
  ∧ ( ∀ v v'. mem v (Set.of_seq visited) → G.mem_vertex g v' →
        G.mem_edge g v v' ↔ ¬ G.mem_edge acc v v' )
```

Finally, we state that every vertex yet to be visited has no outgoing edges:

```
    ( ∀ v v'. mem v (diff g.G.dom (Set.of_seq visited)) → ¬ G.mem_edge acc v v' ) *)
```

We have seen that the inner invariant tends to be lengthier than the outer. And this one is no exception. Its signature is as follows:

```
(*@ predicate complement_inner (g: G.gt) (s: G.vt)                    GOSPEL
                               (visited': G.vt seq) (acc': G.gt) (* inner iteration *)
                               (visited: G.vt seq) (_acc: G.gt)  (* outer iteration *) =
```

We start by stating that their domains are the same:

```
  ( acc'.G.dom == g.G.dom)
```

For any visited vertex in the outer loop, if it is different than the one we are currently iterating over, preserve the complement property:

```
∧ ( ∀ v v'. v ≠ s → mem v (Set.of_seq visited) → G.mem_vertex g v' →
      ¬ G.mem_edge g v v' ↔     G.mem_edge acc' v v'  )
∧ ( ∀ v v'. v ≠ s → mem v (Set.of_seq visited) → G.mem_vertex g v' →
      G.mem_edge g v v' ↔ ¬ G.mem_edge acc' v v'  )
```

And for every vertex to be visited in the outer loop, it has no successors:

```
∧ ( ∀ v v'. mem v (diff g.G.dom (Set.of_seq visited)) → G.mem_vertex g v' →
      ¬ G.mem_edge acc' v v' )
```

Now for the vertex we are currently visited, the complement property must be respected:

```
∧ ( ∀ v. mem v (Set.of_seq visited') → ¬ G.mem_edge g s v ↔ G.mem_edge acc' s v  )
∧ ( ∀ v. mem v (Set.of_seq visited') → G.mem_edge g s v ↔ ¬ G.mem_edge acc' s v  )
```

And finally, for any other vertex to be visited in the inner loop, there is not an edge in the accumulator graph:

```
∧ ( ∀ v. mem v (diff g.G.dom (Set.of_seq visited')) → ¬ G.mem_edge acc' s v ) *)
```

We have reached the end of this module. All four algorithms are fully dispatched by the SMT solvers under 10 seconds. In the following section we will see three interesting by-products that come from the conjuction of some specifications: We can now close this module and the section.

```
end                                                                   OCaml
```

5.3.6 By-products

The complement of a graph in itself is very interesting, but when we *complement* it with other operations, things are truly compelling. Consider, for instance, the intersection of a graph with its complement. This new graph should be empty with the same domain but no edges. By decorating the functions with `[@logic]` attributes, our specifications fully capture this, and the lemma `empty_graph` is quickly fully dispatched:

```
(*@ lemma empty_graph: ∀ g g'. g' = intersect g (complement g) →       GOSPEL
    ∀ v v'. G.mem_vertex g v → not G.mem_edge g' v v' *)
```

Or, for instance, the union of a graph with its complement results in a complete graph where all there exists is an edge between all vertices that are contained in the domain:

```
(*@ lemma complete_graph: ∀ g g'. g' = union g (complement g) →                GOSPEL
    ∀ v v'. G.mem_vertex g v → G.mem_vertex g v' → G.mem_edge g v v' *)
```

And finally, the involution of the complement operation. The complement of the complement of some graph `g` is `g`.

```
(*@ lemma involution: ∀ g g'. g' = complement (complement g) →                 GOSPEL
    (g.G.dom = g'.G.dom) ∧
    ∀ v v'. (G.mem_vertex g v ∧ G.mem_vertex g v') →
            (G.mem_edge g v v' ↔ G.mem_edge g' v v') ∧
            (¬ G.mem_edge g v v' ↔ ¬ G.mem_edge g' v v') *)
```

These lemmas are very quickly dispatched by the SMT solver Alt-Ergo.

5.3.7 Path checking

Although an interesting module, graph operations are rarely used in meaningful algorithms. The only concrete usage of a graph operation found in OCamlGraph is for the *mirror* of a graph used to compute the connectivity in a strongly connected directed graph[2].

On the other hand, one can think that formally verifying an algorithm that checks the existence of a path between two vertices might be more relevant. The following specification was devised in prior work conducted by Castanho and Pereira [4]. We follow their work with our ideas to specify higher-order iteration using first-order cursor clients. We observed that the number of VCs was reduced by more than half (160 to 70).

This algorithm makes use of a `Queue` and a `Hashtbl`. The elements of the latter must define a set of functions of comparison, hashing, and an equality function. These functions can be encapsulated in a module `COMPARABLE`, as follows:

```
module type COMPARABLE = sig                                                  GOSPEL
  type t

  val [@logic] compare : t → t → int
  (*@ axiom pre_order_compare: is_pre_order compare*)

  val [@logic] hash : t → int

  val equal : t → t → bool
  (*@ b = equal t1 t2
      ensures b ↔ t1 = t2 *)
end
```

The module `Check` takes as input another module, `G`. This one defines the vertices through the module `COMPARABLE`, and the type of a vertex is now `V.t`. Additionally, the module `G` must provide a higher-order iteration function over the set of successors of some vertex `v`, `iter_succ`. This function is specified as we have previously shown. We model the graph the same as we did before, while its type-invariant is less expressive.

[2]See here: `https://github.com/backtracking/ocamlgraph/blob/master/src/components.ml`

```
module Check (G : sig                                          Extended GOSPEL + OCaml
  module V : COMPARABLE (* The vertices are COMPARABLE *)
  type gt
  (*@ model dom: V.t fset
      model suc: V.t → V.t fset
      with x invariant ∀ v1, v2. mem v1 x.dom → mem v2 (x.suc v1) → mem v2 x.dom *)

  val iter_succ : (V.t → unit) → gt → V.t → unit
  (*@ iter_succ f graph vertex
    iterspec
    ~permitted:(fun v → let (g, s) = collection in
                        subset (Set.of_seq v) (g.suc s) ∧ Seq.distinct v)
    ~complete:(fun v → let (g, s) = collection in
                        cardinal (Set.of_seq v) = cardinal (g.suc s))
    with structure = (gt * V.t), elt = V.t *)
  end) =
struct
```

The algorithm that checks a path between two vertices follows a classical approach. It employs a Breadth-First Search from the first vertex and, during traversal, it looks out for the second. This algorithm uses one `Hashtbl` to keep a track of marked vertices and, in logic, another, as a *ghost* table to keep track of fully visited vertices, *i.e.*, vertices that have been popped from the queue. Additionally, to make the search, we use a `Queue` to enqueue elements to be visited. The specification for this program is very lengthy, for 44 lines of code (LoC) there are 103 lines of specification (LoS). Let us start by explaining the algorithm and we will then follow with the specification for the higher-order iteration.

We start by defining a module `H`, our `Hashtbl` over vertices. The `Make` functor takes a module as input which must adhere to the signature `COMPARABLE`.

```
module H = Hashtbl.Make(G.V)                                   OCaml
let check_path graph v1 v2 =
  let marked = H.create 97 in
  let [@ghost] visited = H.create 97 in
  let q = Queue.create () in
```

The core of the algorithm lies in this function, `loop`. If the queue is empty this means that we exhausted the graph and `v2` was not found.

```
  let rec loop () =                                            OCaml
    if Queue.is_empty q then false (* exhausted graph and no path was found *)
    else
```

Otherwise, there are still elements to be traversed. In this case, we pop an element from the queue and compare it with the vertex we are looking for. If this is true, then we have found a path to `v2`.

```
      let v = Queue.pop q in                                   OCaml
      if G.V.compare v v2 = 0 then true (* path found *)
      else begin
```

If that is not the case, we add its successors to the queue and mark them in the consumer function of the higher-order iteration `iter_succ`.

```
H.add visited v ();                                                    OCaml
G.iter_succ
  (fun v' →
    if not (H.mem marked v') then begin H.add marked v' (); Queue.add v' q end)
  graph v
```

This is where our specification comes in. We parameterize the arguments as follows: the collection we are iterating over is the successors of `v`, the vertex we are currently visited; the termination is given by the convergence of to-be visited successors to zero; and the loop invariant is some predicate `iter_inv` parameterized with arguments that are useful to capture its behavior. We will tackle this predicate later on.

```
    (*@ iterspec ~collection:(graph, v)                  Extended GOSPEL + OCaml
        ~convergence:(fun (g, s) v → cardinal (g.G.suc s) - length v)
        ~inv:(iter_inv visited marked (old marked) v v1 v2 q graph) *);
    loop ()
  end
(* very long specification for the recursive function *)
in
```

This is our starting point. We enqueue and mark `v1` and call the recursive function `loop`.

```
H.add marked v1 () ;                                                   OCaml
Queue.add v1 q;
loop ()
```

This function requires `v1` to be a member of the graph's domain and ensures the result being equal to there being a path between `v1` and `v2` in `g`:

```
(*@ b = check_path g v1 v2                                             GOSPEL
    requires mem v1 graph.G.dom
    ensures b ↔ has_path g v1 v2 *)
```

The predicate `has_path` logically captures the property of there being a path between two vertices. This predicate is an existential of there being a path `p` that satisfies another predicate, `is_path`, that tests if `p` is a valid path from `v1` to `v2`. A valid path is one such that if it is empty then `v1` and `v2` must be equal, else there is an edge from `v1` to the first element of the path and its last element is `v2`, all while there are edges between all consecutive links in the path. This is encoded as such:

```
(*@ predicate is_path (v1 : G.V.t) (l : G.V.t seq) (v2 : G.V.t) (g : G.gt) =   GOSPEL
      let len = Seq.length l in
      if len = 0 then v1 = v2
      else edge v1 l[0] g && l[len - 1] = v2 && mem v1 g.G.dom &&
           ∀ i. 0 ≤ i < len - 1 → edge l[i] l[i+1] g *)

(*@ predicate has_path (v1 v2 : G.V.t) (g : G.gt) = ∃ p. is_path v1 p v2 g *)
```

Now, let us breakdown the loop invariant, statement by statement. Some allow us to correctly capture others, however we find some that preserve not only *correctness* but also *completeness*. The signature for this predicate is as follows. We expect three tables, the *ghost* visited, the current and old state of the marked table; three vertices: the current successor and both source and sink; the queue of elements; the graph; and finally, the sequence of visited successors:

```
(*@ predicate iter_inv (visited marked old_marked: unit H.t) (s v1 v2: G.V.t)
                       (q: G.V.t Queue.t) (graph: G.gt) (v: G.V.t seq) =
```
GOSPEL

We start by stating that `v2` was not visited. This would imply that we already found a path but for some reason are still processing.

```
    (not (mem v2 visited.H.dom))
```

All the elements in the queue are distinct:

```
  ∧ (distinct q.Queue.view)
```

The marked vertices are a subset of the domain of the graph:

```
  ∧ (subset marked.H.dom graph.G.dom)
```

All marked vertices: were fully processed, *i.e.*, they have left the queue, or still waiting to be processed, *i.e.*, they are in the queue.

```
  ∧ (∀ v'. mem v' marked.H.dom → mem v' visited.H.dom ∨ mem v' q.Queue.view)
```

There cannot exist elements in both states:

```
  ∧ (disjoint q visited)
```

All processed and to-be processed elements are also marked:

```
  ∧ (∀ v'. mem v' visited.H.dom → mem v' marked.H.dom)
  ∧ (∀ v'. mem v' q.Queue.view  → mem v' marked.H.dom)
```

The source vertex `v1` was marked and visited:

```
  ∧ (mem v1 marked.H.dom) ∧ (mem v1 visited.H.dom)
```

Already marked vertices maintain their status as marked:

```
  ∧ (∀ v'. mem v' (old_marked).H.dom → mem v' marked.H.dom)
```

Every visited element during the iteration was marked:

```
  ∧ (∀ v'. mem v' v → mem v' marked.H.dom)
```

And finally, every marked vertex is reachable from the source `v1`:

```
  ∧ (∀ v'. mem v' marked.H.dom → has_path v1 v' graph) *)
```

Although this monolith invariant captures the correct behavior of the iteration, it is only part of the full specification of the function. The other part is concerned with the core recursive function. On top of maintaining some properties that we have already shown above, it also concerned with three crucial properties.

Safety holds by the fact that the exception `Empty` is never thrown. This is guarded by the first *if* statement in the body of the `loop` function.

```
raises  Queue.Empty → false
```
GOSPEL

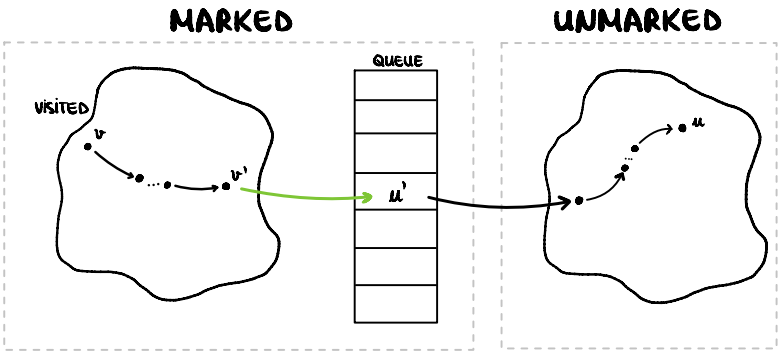


Figure 5.1: Visual representation of the `intermediate_value` lemma.

Correctness means if `check_path` returns true, then there is indeed a path between `v1` and `v2`. This is achieved by maintaining the invariant, here expressed as a pair of *requires-ensures*, that there is always a path from `v1` to a marked vertex, which includes all vertices in the queue and the `visited` table. This is captured as follows:

```
requires ∀ v. mem v marked.H.dom → has_path v1 v graph                    GOSPEL
ensures  ∀ v. mem v marked.H.dom → has_path v1 v graph
```

Completeness means that if there is not a path, then we have indeed fully explored the graph. In other words, while the traversal is not finished, if there is a path from `v1` to `v2` then such a path must go through an intermediate vertex `w` from the queue:

```
requires has_path v1 v2 graph → ∃ w. mem w q.Queue.view ∧ has_path w v2 graph   GOSPEL
```

As all things in life, there is no free lunch. This last property is not trivially solved by SMT solvers. For it, we have to define an auxiliary lemma, `intermediate_value`. This lemma is visually represented in Figure 5.1. Or, in words, if there is a path from a marked node `v` to an unmarked node `u`, then it must be the case that there is an intermediate edge crossing the set of marked elements to the set of unmarked elements. In our case, we need to retrieve the bridge between the `visited` set and the queue, colored in green in Figure 5.1. This lemma is proved by instantiating a recursive traversal from `v` to `u`, until a green arrow is found. Or, in GOSPEL, it is encoded as follows:

```
(*@ lemma intermediate_value : ∀ p, u, v, s, g.                          GOSPEL
    p u → not p v → is_path u s v g → ∃ u' v' s1 s2.
    p u' ∧ not p v' ∧ is_path u s1 u' g ∧ is_path v' s2 v g ∧ edge u' v' g *)
```

We universally quantify the property `p`. This is due to the generic nature of this lemma. But for our purposes this property is the membership of some vertex in the visited table.

This concludes this case study. As mentioned before, we were able to reduce the number of Verification Conditions from 160 to 70. This is a natural consequence as we replaced a recursive function that mimicked the iteration and which needed to embody invariants through pairs of *requires-ensures* with a loop, able to capture the pair with an invariant. Finally, we can close the module and, ergo, this section:

```
end                                                                      OCaml
```

Iteration client	# VCs	LoC / LoS	Time (s)	Iteration	Effects
Sequences	15	9 / 13			
`sum_seq`	6	1 / 2	10.45	`fold`	-
`stack_of_seq`	1	3 / 2	3.01	`iter`	✓
`queue_of_seq`	1	3 / 2	1.13	`iter`	✓
`gt_seq`	6	1 / 5	3.41	`fold`	-
`noccs_of_seq`	1	1 / 2	0.90	`fold`	-
Graphs	275	124 / 286			
`intersect`	13	12 / 15	3.13	`fold + fold`	-
`union`	11	4 / 17	4.40	`fold + fold`	-
`complement`	12	8 / 23	5.80	`fold + fold`	-
`mirror`	6	5 / 8	2.51	`fold`	-
`copy_vertices`	1	1 / 2	1.64	`fold`	-
`check_path`	71	44 / 103	47.79	`iter`	✓
`check_path`†	161	50 / 118	105.07	-	✓
Binary trees	14	5 / 6			
`sum_tree`	5	1 / 2	10.51	`fold`	-
`height_tree`	1	1 / 2	1.02	`fold`	-
`gt_tree`	8	3 / 2	37.05	`iter`	✓

Table 5.1: Summary of verified iteration case studies.

5.4 Summary

In this chapter we presented a collection of case studies, and Table 5.1 summarises them with some statistics. The case studies defined with an `iter` all have side-effects. In all examples, the user never had to reason about the generated WhyML code.

Sequences. The sum of a sequence, `sum_seq`, has more VCs than the others within the same module. This is due to a `split_vc` and some `in_line` goals that an automatic strategy does. The function `gt_seq` was relatively quick to be dispatched. As we have stated before, although `queue_of_seq` and `stack_of_seq` are very similar, the specification of the latter considers the reverse of the sequence of visited elements, which is a logical computation that slightly hinders the replay time of the proof.

Graphs. The graph operations are all fully automatic. The slowest between these is the complement of a graph. This might be due to its verbosity as there are no real computations that can realistically hinder its performance. This is observable by paying close attention to the number of LoS of each case study and relating them to their proof time, which tends to grow linearly. Regarding the `Check` module, we can compare the version of `check_path` prior to our work, `check_path`† in the table, where we see improvements in the number of VCs and the replay time of the proof. The latter is a direct consequence of the halving of the VCs, achieved by our methodology to specify the function `iter`.

Binary Trees. The sum of a tree suffers the same problem as the sum of a sequence. The sum is a recursive function that computes $\sum_{\texttt{i=l}}^{\texttt{u}}$ `f(i)`, from a given lower bound `l` to some upper bound `u`. The height of a tree is very quick to be dispatched as it is a direct logical consequence from our specification of an iteration by levels. The function `gt_tree` does the same computation as `gt_seq`, however, as this one has to consider the flattening of a tree, it seems that this had a significant impact in its overall performance.

This page is intentionally left blank.

6

Conclusion and Future Work

In this document, particularly in Chapter 4, we presented our work on the specification and verification of higher-ordered iterators in OCaml. We sustained our work with relevant case studies that targeted different aspects of our methodology, as per Chapter 5. In this chapter, we will present some reflections on the development of this thesis, in chronological order. Afterwards, we present some ideas for future work.

6.1 Reflections

As a stepping stone, we conducted, primarily in WhyML, the proofs of graph operations from the OCamlGraph library. In these proofs, the iteration was through first-order cursor clients. Ultimately, this allowed us to take inventory of every generic aspect of first-order clients, which we used to conceive our translation schemas.

As it happens, spending long hours looking at code makes you wonder about its subtle details, and you start to question some of its implementation choices. This happened with a function of the OCamlGraph, the function `intersect`. This function would test the membership of some edge in a graph by linearly looking for it in a list of successors of a given vertex. This could have been, in turn, a logarithmic query to the hash-set of successors. Fortunately, we reported this rather minor overlooked detail and was fixed.

We developed an extension to GOSPEL focused on the specification of higher-order iterators. We also developed an extension to Cameleer that could translate these specifications into regular WhyML code that could, in turn, be deductively verified using Why3.

Additionally, we devised a collection of case studies that are able to showcase various approaches to how one might use our work to verify higher-order iterators, through the `permitted` and `complete` predicates. We presented three different ways to flatten structures, all of which present distinct approaches to iteration problems: the axiomatization of a flattened graph, an in-order iteration of a tree and an iteration over the rings of a tree.

Finally, the work on the specification and verification of higher-order iterators was subject to a peer-reviewed submission to the Portuguese Symposium in Computer Science.

6.2 Future work

Though we believe we have successfully achieved what we sought to do, we argue that there is plenty of room for future work and we wish to present some avenues to that effect.

Generic specification for patterns. Our generic iteration specification captures patterns that any client or iterator must follow, regardless of their concrete implementation. We believe this pattern-based specification could be applied to other common OCaml language functions, such as the higher-order functions `map` or `filter`. This specification would introduce a concept similar to *typeclasses* or *traits* in GOSPEL, as it would only be necessary to describe the general abstractions and provide concrete instances for each specific use.

Relational equivalence. The correctness of our translation schemas is based on an informal argument that a higher-order iterator can be converted in a cursor that plays the iteration part. One possible extension to our methodology could be to generate a skeleton of a proof, for example, in Coq that captures such equivalence. For such, it would be natural to resort to binary logic, *e.g.*, Relational Hoare Logic [2], allowing us to reason about the equivalence of two OCaml programs.

Case studies. Here, we propose to go back to our primary goal, the formal verification of the OCamlGraph library. We are now equipped with tools to formally verify higher-order iterations, which was not possible prior to our work. There is still a lot of future work in this regard, though we believe we helped pave the way. One interesting case study would be to consider an iteration where the collection that is being iterated is not static but rather dynamic. For instance, an iteration where the collection is an ever-growing graph and would, most likely, require a reference to the graph on the creation of a cursor.

Bibliography

[1] *OCaml's Reference Manual*. https://v2.ocaml.org/releases/5.1/htmlman/index.html. Accessed: 2024-2-13 (cit. on p. 7).

[2] N. Benton. "Simple relational correctness proofs for static analyses and program transformations". In: *SIGPLAN Not.* 39.1 (2004-01), pp. 14–25. ISSN: 0362-1340. DOI: 10.1145/982962.964003. URL: https://doi.org/10.1145/982962.964003 (cit. on p. 66).

[3] A. Bílý et al. *Compositional Reasoning for Side-effectful Iterators and Iterator Adapters*. 2022. arXiv: 2210.09857 [cs.LO] (cit. on p. 28).

[4] D. Castanho and M. Pereira. *Auto-active Verification of Graph Algorithms, Written in OCaml*. 2022. arXiv: 2207.09854 [cs.LO] (cit. on pp. 25, 49, 58).

[5] A. Charguéraud. "Characteristic formulae for the verification of imperative programs". In: *SIGPLAN Not.* 46.9 (2011-09), pp. 418–430. ISSN: 0362-1340. DOI: 10.1145/2034574.2034828. URL: https://doi.org/10.1145/2034574.2034828 (cit. on p. 14).

[6] A. Charguéraud. *Software Foundations Chapter 6: Separation Logic*. https://softwarefoundations.cis.upenn.edu/slf-current/. Accessed: 2024-1-31 (cit. on p. 14).

[7] 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, 10).

[8] S. Conchon, J.-C. Filliâtre, and J. Signoles. "Designing a generic graph library using ML functors". In: (2007-04) (cit. on pp. 2, 8, 15).

[9] *Coq Proof Assitant*. https://coq.inria.fr/. Accessed: 2024-1-31 (cit. on p. 14).

[10] X. Denis and J.-H. Jourdan. "Specifying and Verifying Higher-order Rust Iterators". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Ed. by S. Sankaranarayanan and N. Sharygina. Vol. 13994. Lecture Notes in Computer Science. ETAPS. Paris, France: Springer, 2023-04, pp. 93–110. DOI: 10.1007/978-3-031-30820-8_9. URL: https://hal.science/hal-03827702 (cit. on pp. 28–30).

[11] X. Denis, J.-H. Jourdan, and C. Marché. "Creusot: a Foundry for the Deductive Verification of Rust Programs". In: *ICFEM 2022 - 23th International Conference on Formal Engineering Methods*. Lecture Notes in Computer Science. Madrid, Spain: Springer Verlag, 2022-10. URL: https://inria.hal.science/hal-03737878 (cit. on p. 29).

[12] M. Erwig. "Inductive Graphs and Functional Graph Algorithms". In: *Journal of Functional Programming* 11 (2001-04). DOI: 10.1017/S0956796801004075 (cit. on p. 2).

[13] J.-C. Filliâtre. "Backtracking Iterators". In: *Proceedings of the 2006 Workshop on ML*. ML '06. Portland, Oregon, USA: Association for Computing Machinery, 2006, pp. 55–62. ISBN: 1595934839. DOI: 10.1145/1159876.1159885. URL: https://doi.org/10.1145/1159876.1159885 (cit. on pp. 9, 19, 42).

[14] J.-C. Filliâtre. *Deductive Program Verification with Why3: A Tutorial*. https://why3.lri.fr/ssft-16/notes-why3.pdf. Accessed: 2024-1-08 (cit. on p. 5).

[15] J.-C. Filliâtre. "Deductive software verification". In: *International Journal on Software Tools for Technology Transfer* 13 (2011-10), pp. 397–403. DOI: 10.1007/s10009-011-0211-0 (cit. on pp. 1, 5).

[16] J.-C. Filliâtre and A. Paskevich. "Abstraction and Genericity in Why3". In: *ISoLA 2021 - 9th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Vol. 12476. Rhodes, Greece, 2021-10. DOI: 10.1007/978-3-030-61362-4_7. URL: https://inria.hal.science/hal-02696246 (cit. on p. 14).

[17] J.-C. Filliâtre and A. Paskevich. "Why3 — Where Programs Meet Provers". In: *Programming Languages and Systems*. Ed. by M. Felleisen and P. Gardner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 125–128. ISBN: 978-3-642-37036-6 (cit. on pp. 2, 11).

[18] *OCaml Functors*. https://ocaml.org/docs/functors. Accessed: 2024-1-06 (cit. on p. 8).

[19] 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).

[20] B. Jacobs, J. Smans, and F. Piessens. "A Quick Tour of the VeriFast Program Verifier". In: *Programming Languages and Systems*. Ed. by K. Ueda. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 304–311. ISBN: 978-3-642-17164-2 (cit. on p. 13).

[21] 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. 13).

[22] X. Leroy. "A modular module system". In: *J. Funct. Program.* 10.3 (2000), pp. 269–303. DOI: 10.1017/S0956796800003683. URL: https://doi.org/10.1017/s0956796800003683 (cit. on p. 8).

[23] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on p. i).

[24] T. McCabe. "A Complexity Measure". In: *IEEE Transactions on Software Engineering* SE-2.4 (1976), pp. 308–320. DOI: 10.1109/TSE.1976.233837 (cit. on p. 1).

[25] A. Mokhov. "Algebraic graphs with class (functional pearl)". In: *SIGPLAN Not.* 52.10 (2017-09), pp. 2–13. ISSN: 0362-1340. DOI: 10.1145/3156695.3122956. URL: https://doi.org/10.1145/3156695.3122956 (cit. on p. 2).

[26] *OCamlGraph's Github Repository*. https://github.com/backtracking/ocamlgraph. Accessed: 2024-2-13 (cit. on p. 2).

[27] M. J. Parreira Pereira. "Tools and Techniques for the Verification of Modular Stateful Code". Theses. Université Paris Saclay (COmUE), 2018-12. URL: https://theses.hal.science/tel-01980343 (cit. on pp. 19, 25, 26, 29, 33).

[28] L. C. Paulson. "Isabelle: The Next 700 Theorem Provers". In: *CoRR* cs.LO/9301106 (1993). URL: https://arxiv.org/abs/cs/9301106 (cit. on p. 13).

[29] 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 pp. 2, 11).

[30] F. Pottier. "Verifying a hash table and its iterators in higher-order separation logic". In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*. CPP 2017. Paris, France: Association for Computing Machinery, 2017, pp. 3–16. ISBN: 9781450347051. DOI: 10.1145/3018610.3018624. URL: https://doi.org/10.1145/3018610.3018624 (cit. on pp. 26, 28).

[31] J. Reynolds. "Separation logic: a logic for shared mutable data structures". In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817 (cit. on p. 6).

[32] *Rust's FusedIterator*. https://doc.rust-lang.org/std/iter/trait.FusedIterator.html. Accessed: 2024-1-09 (cit. on p. 20).

[33] T. L. Soares, I. Chirica, and M. Pereira. "Static and Dynamic Verification of OCaml Programs: The Gospel Ecosystem". In: *Leveraging Applications of Formal Methods, Verification and Validation. Specification and Verification*. Ed. by T. Margaria and B. Steffen. Cham: Springer Nature Switzerland, 2025, pp. 247–265. ISBN: 978-3-031-75380-0 (cit. on p. 2).

[34] B. Toninho. *Lectures Notes on Separation Logic*. http://ctp.di.fct.unl.pt/~btoninho/teaching/cvs-22/seplogic.pdf. Accessed: 2024-1-31 (cit. on p. 6).

[35] A. Turing. "Checking a Large Routine". In: *The Early British Computer Conferences*. Cambridge, MA, USA: MIT Press, 1989, pp. 70–72. ISBN: 0262231360 (cit. on p. 5).