NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

ION CHIRICA

BSc in Computer Science

FORMAL VERIFICATION OF A GRAPH LIBRARY

A DEDUCTIVE STUDY ON GRAPHS AND APPLIED ITERATION

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
February, 2024

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

FORMAL VERIFICATION OF A GRAPH LIBRARY

A DEDUCTIVE STUDY ON GRAPHS AND APPLIED ITERATION

ION CHIRICA

BSc in Computer Science

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

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
February, 2024

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. By asserting the correctness of its algorithms and different types of graphs, 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. Moreover, 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 some positive results on formally proving a small collection of graph operations in Why3 and set forth some of our ongoing work towards finding an answer to our question, in the form of a sketched proposal.

Keywords: Formal Verification, Graphs, Iteration, 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 provado correto.

Este trabalho propõe verificar formalmente um subconjunto da biblioteca OCamlGraph. Ao afirmarmos a correção dos seus algoritmos e diferentes tipos de grafos, os seus utilizadores podem sentir-se mais descansados 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 utilização 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 também alguns resultados positivos na verificação formal de um conjunto de operações de grafos em Why3 e elucidamos alguns desenvolvimentos, na forma de esboços, da nossa procura de dar resposta à questão acima apresentada.

Palavras-chave: Verificação Formal, Grafos, Iteração, 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. 10, 12)*

Cameleer A deductive verification tool for OCaml programs. *(pp. i, ii, 3, 10, 11, 16, 33)*

CVC4 SMT solver supporting quantifiers and many theories including equality, arithmetic, datatypes, bitvectors. *(p. 10)*

GOSPEL Generic OCaml SPEcification Language - a behavioral specification language for OCaml. *(pp. i, ii, 3, 9–11, 16, 30, 31, 33)*

OCamlGraph A generic graph library for OCaml. *(pp. i–iii, 2, 7, 14–16, 24, 27)*

Why3 A platform for deductive program verification. *(pp. i, ii, 11–13, 20, 24, 25, 30, 33)*

WhyML Programming language used to prove programs in Why3. *(pp. 3, 10–13, 24–30, 33, 37, 39, 41)*

Z3 Theorem prover from Microsoft Research. SMT solver supporting quantifiers and many theories including equality, arithmetic, datatypes, bitvectors *(p. 10)*

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 [21]. 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 [13]. 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's consider, for instance, Graph Theory. Although this is a well-founded and deeply researched mathematical field, some 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. Notably, in OCaml we find OCamlGraph [22], a graph library providing innumerable algorithms and types of graphs.

1.2 Problem Definition

This work focuses on formally proving 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 question:

> *Is there a sound and reliable way to formally verify higher order iteration?*

To the surprise of a population of zero (rounded to the closest integer), the result is positive. In Rust, we can find work from Bílý et al. [2] and Denis [9]. Managing to successfully come up with compositional reasoning and verification of higher-order iterators. Their work heavily relies on Rust traits and its strong guarantees of non-aliasing memory. Moreover, their work is of the utmost importance, as iterators in Rust can result from the composition of other iterators, achieving a non-trivial feat.

However we should question ourselves how the same can be achieved in environments where we do not have the same guarantees about non-aliasing memory, and so:

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?*

We strongly believe that the effects of answering these questions translate into proofs that are more closely truthful to their actual implementations, without taking shortcuts.

1.3 Goals and Expected Contributions

The contributions of this work are three-fold. Using deductive verification, we seek to prove a subset of the OCamlGraph library. However, we must first devise a feasible way to specify iteration. This task can be decomposed into two sub-tasks:

- Specifying the actual iteration of the collection.
- Attending to different ways of iteration, that its effects are correctly observable after the iteration has terminated.

1.4 Report Structure

- Chapter 2 covers the necessary background to understand the problem, 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 iteration specifications.
- In Chapter 4, we will present some preliminary work and its results. We explore a simple graph specification which we used to prove some graph operations from the library. In this chapter we also devise a sketch for a generic iteration in OCaml using GOSPEL specifications.
- Chapter 5 briefly explains our work plan and how will schedule our time.
- In Appendices A, B and C one should find the full specification for `Cursors`, a graph representation and proofs for some operations in WhyML.

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 [13] and then prove it, is as old as Computer Science itself [31]. Still, with the more recent progress in automated theorem provers, we can formally prove programs more efficiently and concisely [12]. 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 mathematical statements. For which we quickly turn to Hoare Logic [17], where its goal is to provide a construct on how to 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 postconditions. 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*:

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

This is the backward reasoning of the assignment rule, where $A\{\mathsf{x}/E\}$ 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, $\mathsf{s}' = \mathsf{s}\{\mathsf{x} \mapsto E\}$, assuming E has no side-effect. We also know that $A\{\mathsf{x}/E\}$ 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 [27] 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 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 [30]. 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's clear from the pre-condition A, neither what's 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 disjointly 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 the 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 emphasizes 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, and 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 [16, 20].

Functors, on the other hand, are modules parametrized 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 [7], 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 `M.c`, where `M` is a module and `c` a component [7]. 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 parametrized 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 labeled 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 [11]. 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`, respectively conceived 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 in 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 left and right branches of the tree.

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

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`, the former being a degenerate version of the other. 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. Both implementations, following OCaml's Standard Library, for `Lists` are:

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

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

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

2.3 GOSPEL & Cameleer

GOSPEL (Generic OCaml SPEcification Language) is a behavioral specification language for OCaml interfaces [6]. 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 informally document our code. To use them in a meaningful way, we must resort to external verifiers, such as Cameleer [25]. This tool takes GOSPEL annotated OCaml code and translates it to an equivalent WhyML 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 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 `.ml` file, containing the logical function `fib n` and the above specification for the memoized version, to Cameleer with the command: `cameleer fib.ml`. Cameleer translates the input program into an equivalent WhyML representation, launching a graphical interface for the Why3 IDE[2].

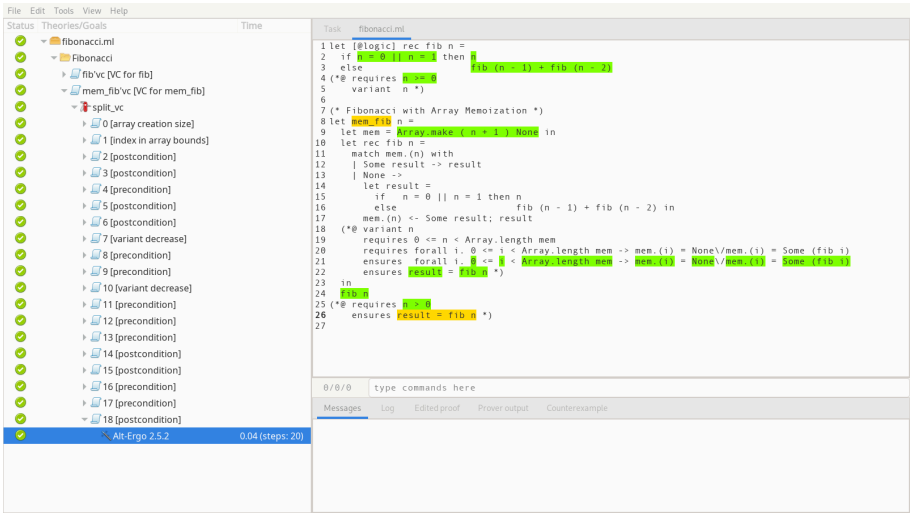


Figure 2.1: Why3 IDE with our *fibonacci* example.

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. Independent of how big

[2]At the time of writing this document, there is ongoing work in extending the pre-processor from GOSPEL's `implementations` branch with a location system as to fix buggy visualizations in the Why3 IDE, and pave the way for clearer future extensions.

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 our 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 introducing 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 [18] or Dafny [19], the Why3 platform relies on multiple external provers, adjusted to the user's needs. It supports both automated and interactive external provers, like Isabelle [24]. 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.

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 be not 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:

```
predicate notfull (s: stack) = length s.view < s.capacity
val push (x: t) (s: stack): unit
   writes   { s } (* modifies stack's view *)
   requires { notfull s }
   ensures  { s.view = Cons x (old s.view) }
```
WhyML

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:

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

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 [14]. 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`.

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

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 [4]. It combines, previously seen, Separation Logic with the powerful interactive theorem prover Coq [8]. 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 [5], let's consider, a push operation in an unbounded Stack.

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

The Stack is modeled as a reference to a List, and note that this does not let us say anything about mutability of type α, which at most is 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 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 4.4 where it will serve as a comparison basis when specifying generic iterations.

2.6 OCamlGraph

OCamlGraph is a generic library featuring a large set of graph data structures and algorithms [7]. 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 see 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 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, let's consider a somewhat real example. Let's consider a simple message transfer protocol that can be represented as an automaton. Let's 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

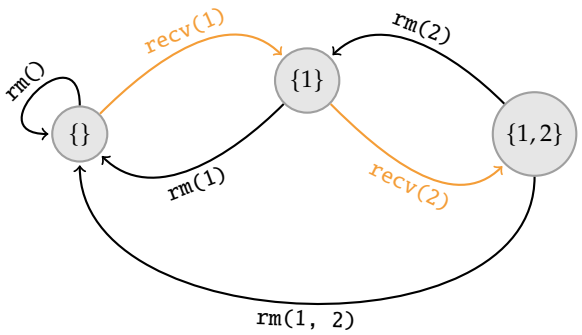
```
open Graph

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 () =                                  (*  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 our 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.

3

State of the Art

3.1 Proving OCamlGraph

Undoubtedly, the closest thing that resembles the work that's here proposed was done by Castanho [3]. The work revolved around proving a subset of the OCamlGraph library using Cameleer. Although this might seem strikingly similar, we propose to complement GOSPEL with Pereira's [23] and Castanho's [3] future work suggestions on higher order iteration. It is with this proposal that we diverge from Castanho's work. Additionally, we propose to continue proving the library to ensure the implementations are correct and to strengthen weaknesses in Cameleer.

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 Pereira [23] 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. 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 [23].

Let us consider an example from [23] 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:

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

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:

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

3.3 Cursors

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

Unlike higher order iteration (HOI), where iteration is done with exhaustion in mind and with no control over how many steps we take[1], `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 [23]. 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 [11]. 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.

[1] Once again, neither early stopping nor stepping is unachievable, but using HOI 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.

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 HOI. The higher-order interface `HO`, is uniquely populated by an iteration function `iter`. A persistent cursor `PC` can be defined by a creation (`iterator`) and a produce (`next`) function.

OCaml

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

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
```

Towards being more modular, we leave their types as abstract, though in order to actually provide a concrete example let's 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 shown 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 l =
      match l with
      | []     → ()               (* exhausted, do nothing *)
      | h :: t → f h; loop f t in (* apply f to head; recurse on the tail *)
    iter f l
end
```

An implementation of a Persistent cursor, over lists, can be expressed as follows:

- Its creation, `iterator l`, returns a new persistent cursor with the full list *l*.
- Subsequent calls to `next it` returns a tuple composed of the *head* of the list and a new *Persistent* cursor containing the remainder .[2]

[2]Do note that any calls following termination will always yield *None*, this is a design choice in the likes of Rust's `FusedIterator` [29], inversely we could have raised an exception indicating termination.

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 it =
    match it with
    | PC [h]      → Some (h, Done)
    | 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 HOI using a Persistent cursor comes at a cost of keeping an internal mutable state of the iterator, essentially defeating the purpose.

OCaml
```
module PC2HO (PC: PC) :
  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 iterators detach themselves from rigorous rules of immutability and accept 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.

```
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 the implementation, creating a list cursor is returning a reference to the input list. The termination test, with `has_next`, compares the iterator with the empty list. Finally, producing the next element is returning the current head and updating the iterator to be the tail.

```
module ListC:                                                OCaml
  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
```

In this section we saw 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.

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 [9] and Bílý et al. [2] build on top of Pereira's [23] 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 [9] 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 $\texttt{a} \overset{\texttt{s}}{\rightsquigarrow} \texttt{b}$. 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 ($\forall\ \texttt{a}.\ \texttt{a} \overset{\varepsilon}{\rightsquigarrow} \texttt{a}$) and transitivity ($\forall\ \texttt{a,b,c}.\ \texttt{a} \overset{v}{\rightsquigarrow} \texttt{b} \land \texttt{b} \overset{w}{\rightsquigarrow} \texttt{c} \implies \texttt{a} \overset{v \cdot w}{\rightsquigarrow} \texttt{c}$). 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 $\exists p, \texttt{initial} \overset{p}{\rightsquigarrow} \texttt{iter}$, provided reflexivity and transitivity are proved.

In order to demonstrate how this applies, and safely borrowing from Denis' ideas [9], let's 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:

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

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 $r.\texttt{start} + 1$. 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 mutable reference enabling access to their environment, and by 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* [9]. As to actually reason about closures themselves, we can decorate them with pre- and post-conditions, forcing closure calls to obey this contract. Illustrating this in Rust, and relying on the `CREUSOT` verification platform [10], 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's consider yet another example from Denis [9], 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 post-condition at some step i must be able to establish the pre-condition at step $i + 1$. With everything presented until now, this can be specified as:

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

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:

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

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 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.

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

These fruitful predicates enable us to correctly reason about closures and their effects in `Map`. We believe that some of these ideas might be helpful in specifying higher-order iteration in OCaml. However, as we are unable to rely on Rust's type system, we have to find an alternative to safely track memory modifications, explored in section 4.4.

4

Preliminary Results

Proving the correctness of a library is not a trivial task as it still requires much manual effort in writing viable sound specifications. Furthermore, since a great deal of the OCamlGraph library uses higher-order iteration, we are faced with a need to find a first-order logic equivalent representation. An initial approach, as proposed in [23], is to consider that iterations, using `folds`, can be equally represented as iterations using `Cursors`. As preliminary work, we explored, resorting to Why3, if that assumption is true for a small collection of graph operations from the library.

4.1 Graph operations

In order to prove that there exists such representation of higher order iteration to first-order logic, we looked at proving that, given a similar graph representation, we can apply `Cursors` to act for its iteration.

Starting by first defining a simple graph, which can be expressed in terms of a `domain` set, our vertices, and an abstract function `succ` that takes a vertex and returns its set of successors.

WhyML

```
type elt
type graph = abstract {
  mutable dom: fset elt;          (* set of vertices *)
  mutable succ: elt → fset elt;   (* function returning successors of some vertex *)
}
① invariant {∀ v1 v2. mem v1 dom → mem v2 (succ v1) → mem v2 dom }
② invariant {∀ v1. not (mem v1 dom) → succ v1 == Fset.empty }
by { dom = Fset.empty; succ = (fun _ → Fset.empty) }
```

Record types can be declared as `abstract`, where their fields are only visible in ghost code/specification. Moreover, their definitions, as in this case, are usually accompanied with a type `invariant` - a logical property imposed on any value of the type. To prevent the introduction of logical inconsistencies, we're required to show the existence of at least one record instance satisfying the invariant. To aid verification, we can also provide an explicit witness using the keyword `by`. If we are successful in proving the type invariant,

then Why3 assumes that any graph satisfies the invariant at any function entry. Possibly, temporarily broken in the middle. And fully restored before reaching the function exit.

Our two type invariants are simple but required to prove more refined graph behaviors. ① states that for any two vertices `v1` and `v2`, if `v1` belongs in the domain of vertices and `v2` is in the successors set of `v1`, then `v2` must also belong in the domain of vertices; ② aims to deal with ghost successors, by stating that for any vertex, if it does not belong in the domain of vertices, then its successors set is empty.

Intuitively, the next step is to specify some operations over our newly defined graph. The first that comes is creating a new graph. Specifying a function that returns an empty graph is remarkably simple:

```
val empty () : (g: graph)                                                    WhyML
  ensures { g.dom = empty }              (* empty set of vertices *)
  ensures { g.succ = (fun _ → empty) } (* function that returns empty for any argument *)
```

Adding a new vertex can be accomplished by adding it to the graph's domain; leaving everything else untouched. Adding a new edge, in a directed environment, from `src` to `dst`, can be attained by updating the successor set of `src` to also include `dst`. We could complement the graph with an extra set of predecessors, where we would have to also add `src` to the set of predecessors of `dst`. We have not given any usage to this set, yet one can imagine it useful, as in the case of computing a graph's transitive closure [28, 32].

```
val add_vertex (ref g: graph) (x: elt) : ()                                  WhyML
  writes { g }
  ensures { g.dom == add x (old g.dom) }
  ensures { ∀ s: elt. g.succ s == (old g.succ) s }

val add_edge (ref g: graph) (src: elt) (dst: elt) : ()
  writes { g }
  ensures { g.dom == old g.dom }
  (* g.succ = [g.succ[src] ← g.succ[src] ∪ {dst}] *)
  ensures { g.succ = Map.set (old g.succ) src (add dst (old g.succ src)) }
```

In these specifications, we are also considering that the input graph is modified in-place, where we could have taken a more persistent approach and returned a new graph with the included modifications. Since our graph was defined as abstract, we are not allowed to access any of its fields in normal code, hence the need to also specify the *read-only* behavior, namely predicates that assess membership or functions returning the state of one of the fields.

```
predicate mem_vertex (g: graph) (v: elt) = mem v g.dom                       WhyML
predicate mem_edge   (g: graph) (src: elt) (dst: elt) = (mem dst (g.succ src))

val successors (g: graph) (v: elt) : fset elt
  ensures { result == g.succ v }
val vertices (g: graph) : fset elt
  ensures { result == g.dom }
```

4.2 Cursor implementation in WhyML

On top of providing the familiar *for* loop expression, WhyML also provides a far richer construct of which we are able to devise more interesting patterns of iteration, focusing on collections. The *for each* loop [15], defined on the left, is syntactic sugar for the code on the right:

```
for p in e1 with S do
  invariant/variant ...
  e2
done
```

```
let it = S.create e1 in
try
  while true do
    invariant/variant ...
    let p = S.next it in
    e2
  done
  with S.Done → ()
```

WhyML

Here `p` is a pattern, `e1` and `e2` are program expressions, and `S` a namespace. Such a construct resembles `Cursors`, where now, instead of having the operation `has_next`, we perpetually query `next` and rely on an exception to indicate termination. Moreover, we find that in order to define a cursor that complies with a `for each` loop, we only need to declare two functions: `create` and `next`, and an exception `Done`.

The most primitive iterations over graphs are either: traversing every vertex in a given graph's domain, or go over the edges of some vertex, be them successors or predecessors. Seeing as all are defined over sets, this poses a need to specify a sound way to iterate over the same data structure. As to not limit iteration to be only over vertices or edges, we need a generic way to do it. Following the work of Pereira [23], we can think to give a generic specification of a `SetCursor` as follows:

WhyML

```
module SetCursor
  type t = private {
    mutable visited : fset elt;
            collection : fset elt;
  } invariant { subset visited collection }
  by { visited = Fset.empty; collection = Fset.empty }

  exception Done

  val next (t: t) : elt
    writes  { t }
    raises  { Done → t.visited = old t.visited ∧ t.visited = t.collection }
    ensures { not mem result (old t.visited) }
    ensures { t.visited = add result (old t.visited) }
end
```

This generic specification still lacks a `create` function seeing as it is the only thing that should really change, and of which we leave up to implementations to define. Any implementation is free to override the cursor type `t`, respecting only that `visited` and `collection` are left untouched. Moreover, we require that any new relevant field of `t`

must be equally accompanied with a unitary witness, or singleton. Exemplifying with a cursor that is defined to traverse the set of vertices of a graph.

WhyML

```
module VertexSetCursor
  type t = private {
    mutable visited : fset elt;
            collection : fset elt;
            structure : graph;
  }
  invariant { subset visited collection }
  invariant { collection == structure.dom } (* define collection w.r.t. structure *)
  by { visited = Fset.empty; collection = Fset.empty;
       structure = graphempty() }           (* empty structure *)

  val create (s: graph) : t
    ensures { result.visited = empty }      (* initial set of visited is empty *)
    ensures { result.structure = s }

  clone export SetCursor with type t        (* clone SetCursor, overriding t *)
end
```

Here, we devised a generic specification for iteration over Sets that can be used by `Cursors`. In earlier versions, we found ourselves with a considerable amount of redundancy. However, the observation that the `create` function is the only function that differs across structures that allowed us to come up with the above-presented specification. Extensive usage of these `Cursors` can be found in Appendix C.

4.3 Case studies

Having covered a way to implement generic `Cursors` for sets in WhyML and a generic specification for graphs, we can move on to prove some functions from the OCamlGraph. We aimed to prove four functions from the `oper.ml` file[1], each with its unique caveat. The intersection and union of two graphs, a graph's complement and its mirror. These case studies are relevant as their success signifies that an equivalent representation of higher-order iteration into first-order is possible.

4.3.1 Complement

The complement of a graph is a new graph, `g'`, that contains the same set of vertices as the original graph, `g`, but the edges are different. In the complement graph, two vertices not adjacent to the original graph are connected by an edge, and two vertices adjacent to the original graph have no direct connection in the complement graph. Formally, the complement of a graph G is denoted as $\overline{G}$. The implementation is relatively simple: for any two vertices (v, v') that do not make up an edge in `g`, add an edge (v, v') in the new graph; otherwise, do nothing.

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

OCaml
```
let complement g =
  G.fold_vertex
    (fun v g' →
       G.fold_vertex
         (fun w g' →
            if G.mem_edge g v w then g'
            else B.add_edge g' v w)
         g g')
    g (B.empty ())
```

WhyML
```
let complement (g: graph) : (g': graph)
  ensures { ∀ src dst: elt. mem_vertex g src → mem_vertex g dst →
                          ¬ mem_edge g src dst ↔ mem_edge g' src dst  }
  ensures { ∀ src dst: elt. mem_vertex g src → mem_vertex g dst →
                            mem_edge g src dst ↔ ¬ mem_edge g' src dst  }
  ensures { g.dom == g'.dom }
  =
  (* find full specification and proof in Appendix C *)
```

Its proof is lengthy, though complete. We are quickly able to prove that the union of any graph G with its complement $\overline{G}$ is a complete graph — there is a direct connection between any two vertices. Moreover, we can also prove that the intersection of G and $\overline{G}$ is a graph with no edges, an empty graph.

WhyML
```
lemma complete: ∀ g: graph, v v': elt.
  mem_vertex g v ∧ mem_vertex g v' →
  mem_edge (union g (complement g)) v v'

lemma empty: ∀ g: graph, v v': elt.
  mem_vertex g v ∧ mem_vertex g v' →
  not mem_edge (intersect g (complement g)) v v'
```

4.3.2 Intersection

The intersection of two graphs, `g1`, and `g2`, is another graph `g'` where every shared vertex and edge between `g1` and `g2` is present in `g'`. The implementation is ingenious, however slow. It goes like this: for every vertex v in `g1`, go over every successor s of v in `g2` and add s to `g'` if it also belongs in `g1`. It might be confusing why we keep on switching graphs, but this allows us to skip some vertices if they do not belong in `g2`, caught by the `Invalid_argument` exception. Nonetheless, some suspicion is raised on why the membership test of s in `g1` resorts to using an `exists` function, which has a time complexity of $O(n)$, rather than `G.mem_edge_e` of complexity $O(log(n))$, since sets are implemented via binary balanced trees and since this operation was available at the time of implementation. This question was raised on an issue[2].

[2]Found here: https://github.com/backtracking/ocamlgraph/issues/136

OCaml
```
let intersect g1 g2 =
  G.fold_vertex
    (fun v g →
       try
         let succ = G.succ_e g2 v in
         G.fold_succ_e
           (fun e g →
              if List.exists (fun e' -> G.E.compare e e' = 0) succ
              then B.add_edge_e g e
              else g)
           g1 v (B.add_vertex g v)
       with Invalid_argument _ →
         (* [v] not in [g2] *)
         g)
    g1 (B.empty ())
```

WhyML
```
let intersect (g: graph) (g': graph) : (g'': graph)
  ensures { ∀ v: elt. g''.succ v == inter ( g.succ v ) (g'.succ v) }
  ensures { g''.dom = inter g.dom g'.dom }
  =
  (* find full specification and proof in Appendix C *)
```

4.3.3 Mirror

The mirror of a graph is a simple function that mirrors any edge from an input graph and reflects those changes in the returning graph. Since there is no point in mirroring an undirected graph, we can return it as is — sustained by the implementation.

OCaml
```
let mirror g =
  if G.is_directed then begin
    let g' = G.fold_vertex              (* make a new graph by copying *)
        (fun v g' → B.add_vertex g' v) (* every vertex from g *)
         g (B.empty ()) in
    G.fold_edges_e                      (* for every edge (v1, v2) in g.. *)
      (fun e g' →
        let v1 = G.E.src e in
        let v2 = G.E.dst e in
        B.add_edge_e g'                 (* ..add edge (v2, v1) to g' *)
          (G.E.create v2 (G.E.label e) v1))
      g g'
    end else
    g                                   (* undirected graph; do nothing *)
```

WhyML
```
let mirror (g: graph) : (g': graph)
  ensures { ∀ src dst: elt. mem_edge g src dst → mem_edge g' dst src }
  ensures { g.dom == g'.dom }
  =
  (* find full specification and proof in Appendix C *)
```

4.3.4 Union

The union of two graphs, g and g', refers to a new graph, g'', that contains all the vertices and edges of both original graphs. When considering the union of two graphs the resulting graph has a domain set that is the union of the domain of g and g', and a successor set for any vertex that is the union of the sets of g and g'. This means that the union of two graphs combines the vertices and edges of the original graphs to form a larger graph.

OCaml

```
let union g1 g2 =
  let add g1 g2 =
    (* add the graph [g1] in [g2] *)
    G.fold_vertex
      (fun v g →
         G.fold_succ_e
           (fun e g → B.add_edge_e g e)
           g1 v (B.add_vertex g v))
      g1 g2
  in
  add g1 (B.copy g2)
```

WhyML

```
let union (g: graph) (g': graph) : (g'': graph)
  ensures { g''.dom = union g.dom g'.dom }
  ensures { ∀ src: elt. g''.succ src == union (g'.succ src) (g.succ src) }
  =
  (* find full specification and proof in Appendix C *)
```

4.4 Proposal for generic iteration specification

In this section we wish to discuss our ideas for a generic iteration specification in GOSPEL. Though this has been tackled in Why3 [23] and implemented in Rust [2, 9], nothing of this nature has been done in GOSPEL. The main problem, which has been alluded to throughout this document and formalized hereinafter, is that there is no viable way to reason, modularly and automatically, about higher-order iteration in OCaml.

Until now, functions that revolved around higher-order iteration, such as iterating over every vertex in a graph or applying a function to every edge, had to be manually deconstructed into recursive functions that mimicked the iteration [3]. It is undesirable for multiple reasons. It is a cumbersome task; we might be prone to introduce previously absent bugs or unmaintainable code; it may have performance drawbacks if not cautious, among other reasons. Reasoning about higher-order iteration is a two-fold combination of reasoning about the iteration and the applied function (and its possible side effects).

We seek to introduce a new type of modularity, a *type class*, more closely related to Rust's *trait*s. We wish to be able to parametrize a type with GOSPEL specifications, which is not yet possible. Building on Pereira's ideas of higher-order iteration in Why3 [23], we developed the following rough sketch, which should be taken broadly, for an `iter` function. Firstly, we define a generic type class, `IteratorClass`, where the predicates are left to be defined and universally quantified.

GOSPEL + OCaml

```
typeclass IteratorClass: sig
  type elt
  type collection

  (*@ predicate inv (v: elt sequence) *)
  (*@ predicate permitted (v: elt sequence) (c: collection) *)
  (*@ predicate complete  (v: elt sequence) (c: collection) *)

  val f : elt → unit
  (*@ f [v: elt sequence, c: collection, M: loc set] x
      requires inv v
      requires permitted (v ++ [x]) c
      requires not complete v c
      modifies M
      ensures inv (v ++ [x]) *)
end
```

The signature function f is decorated with a GOSPEL specification. We introduce ghost arguments to f, that is, arguments that are only visible in the specification. The argument v holds the sequence of visited elements, c the full collection, and M the Set of memory locations potentially mutated by a call to f. Calls to $f\ x$ are required to hold the invariant *inv* for any so far sequence of visited element, x is a valid element of c and calls after completion should not be possible. Additionally, it should ensure that the invariant holds for the new element.

GOSPEL + OCaml

```
(*@ function empty_collection : collection *)
val iter : (elt → unit) → t → unit
(*@ implements IteratorClass
    iter f c
    requires inv empty_collection
    ensures inv c  *)
```

Provided a definition for the empty collection, we wish to state that the higher-order iteration function *iter*, implements `IteratorClass`, and require that the invariant holds for the empty collection and ensure it holds for the full collection one desires to iterate. Leaving everything to be parametrized by the client.

GOSPEL + OCaml

```
let iter f t =
(*@ implements IteratorClass with ~f:f ~collection:t ~elt:elt
                                  ~permitted:(..) ~complete:(..)
                                  ~empty_collection:(..) ~inv:(..)
*)
(...)
```

Though a rough sketch, let's take a step further and see how an iteration over mutable lists can be specified in CFML. This is heavily inspired by Pottier's work on formally verifying a hashtable and its iterators in CFML [26]. Firstly, in OCaml we can define a mutable list as an algebraic data type where its elements are either `Nil` or a tuple of some value and a reference to the next element.

```
type α contents = Nil | Cons of α * α mlist and
     α mlist    = (α contents) ref
```
OCaml

```
let rec iter f p =
  match !p with
  | Nil → ()
  | Cons (x, q) → f x; iter f q
```

Implementing an iteration function `iter` is fairly simple after having previously seen, in section 2.2.3, its inner workings. A specification for the same `iter` function in CFML will be a conglomerate of everything we have seen so far. The predicate `permitted` states that for some lists `V` and `A`, `V` is a valid prefix of `L` if there exists some list `L'`, we can think of this list as the elements yet to be visited, where `V ++ L'` make up `L` in its entirety. We can also state the `complete` predicate holds whenever we have visited the whole list `L`.

CFML

```
Definition permitted (V: list A) L : Prop := ∃ L', V ++ L' = L.
Definition complete  (V: list A) L : Prop := V = L.
Lemma Triple_iter : ∀ (I: list A → hprop) L (f: val) p,
    (∀ x V,
        SPEC (f x)
          Ⓐ PRE (\[ permitted (V&x) L ])
          Ⓑ PRE (\[ ~(complete V L) ])
          Ⓒ PRE (I V)
          Ⓓ POSTUNIT (I (V&x))) →
    SPEC (iter f p)
      Ⓔ PRE (I nil)
      Ⓕ INV (p ~> MList L)
      Ⓖ POSTUNIT (I L).
```

Concerning the specification of the `iter` function, we have to reason about calls to function `f x` (Ⓐ to Ⓓ) and the iteration itself (Ⓔ to Ⓖ). For any value `x` and list of visited elements `V`, its specification can be expressed as follows: Ⓐ require that `V&x` is a prefix of the list `L`; Ⓑ and we have not yet completed the iteration; Ⓒ and the invariant holds for `V`. We must also Ⓓ ensure that the invariant holds too for the new element `x`. As for the specification of the iteration, we Ⓔ wequire the invariant holds for an empty list. Additionally, Ⓕ Seeing as this has to hold in both pre- and post-condition, we can extract it as an invariant: L is a correct representation MList of `p`. Finally, Ⓖ ensure that we end up with the invariant holding for the entire list `L`.

The proof to this lemma in Separation Logic can be done by structural induction on `L2`, the list of elements to be visited. Considering a simplified specification, in which we universally quantify over two lists, if a proof to this lemma is presented, then it implies our specification is also correct.

CFML

```
(∀ L1 L2, L = L1 ++ L2 →
   SPEC (iter f p)
     PRE (I L1)
     INV (p ~> MList L2)
     POSTUNIT (I L)).
```

5

Work plan

Task 1: Preliminary In the first two weeks of March, we plan to select interesting algorithms and types of graphs that will comprise our case studies. We will be looking to prove both imperative and persistent graphs, and algorithms where the graph is static, and where it can change during its execution.

Task 2: Implementation in Why3 Afterwards, and spanning across a month and a half, we will be looking into complementing our implementations and proofs of graphs and iterators in Why3 and the subjects of interest we sought to prove.

Task 3: Translation proposal Over two months, we plan on conceiving a feasible translation of OCaml iteration functions into equivalent WhyML specifications using `Cursors`, and adapt GOSPEL to capture the translation.

Task 4: Proofs & Implementation In this task, we will look at proving the same subjects of interest in Cameleer and their iterators, with eventual extensions to the tool if necessary.

Task 5: Specification proposal Arguably, this is the highest-risk task, where we will propose a final specification for generic iteration.

Task 6: Dissertation We will allocate the whole of September to the writing of the dissertation.

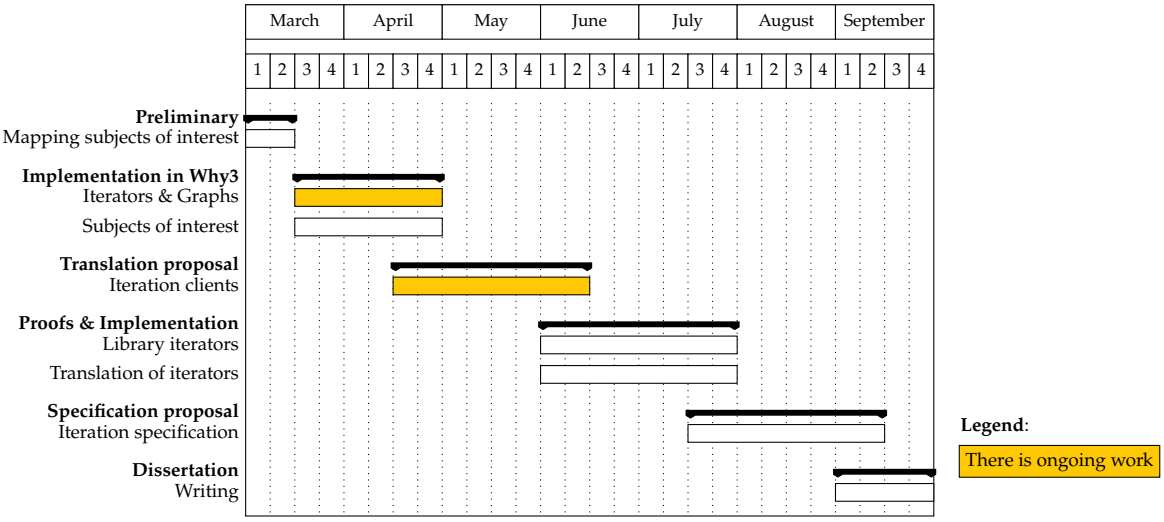


Figure 5.1: Planned Schedule

Bibliography

[1] *OCaml's Reference Manual*. https://v2.ocaml.org/releases/5.1/htmlman/index.html. Accessed: 2024-2-13 (cit. on p. 6).

[2] A. Bílý et al. *Compositional Reasoning for Side-effectful Iterators and Iterator Adapters*. 2022. arXiv: 2210.09857 [cs.LO] (cit. on pp. 2, 20, 30).

[3] D. Castanho and M. Pereira. *Auto-active Verification of Graph Algorithms, Written in OCaml*. 2022. arXiv: 2207.09854 [cs.LO] (cit. on pp. 16, 30).

[4] 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. 13).

[5] A. Charguéraud. *Software Foundations Chapter 6: Separation Logic*. https://softwarefoundations.cis.upenn.edu/slf-current/. Accessed: 2024-1-31 (cit. on p. 13).

[6] 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 p. 9).

[7] S. Conchon, J.-C. Filliâtre, and J. Signoles. "Designing a generic graph library using ML functors". In: (2007-04) (cit. on pp. 7, 14).

[8] *Coq Proof Assitant*. https://coq.inria.fr/. Accessed: 2024-1-31 (cit. on p. 13).

[9] 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. 2, 20–22, 30).

[10] 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. 21).

[11] 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. 8, 17).

[12] 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. 4).

[13] 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, 4).

[14] 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. 13).

[15] *For each in Why3*. https://why3.gitlabpages.inria.fr/why3/syntaxref.html#the-for-loop. Accessed: 2024-1-11 (cit. on p. 26).

[16] *OCaml Functors*. https://ocaml.org/docs/functors. Accessed: 2024-1-06 (cit. on p. 7).

[17] 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. 4).

[18] 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. 12).

[19] 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. 12).

[20] 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. 7).

[21] 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).

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

[23] 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. 16, 17, 20, 24, 26, 30).

[24] 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. 12).

[25] M. Pereira and A. Ravara. "Cameleer: A Deductive Verification Tool for OCaml". In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*. Ed. by A. Silva and K. R. M. Leino. Vol. 12760. Lecture Notes in Computer Science. Springer, 2021, pp. 677–689. DOI: 10.1007/978-3-030-81688-9_31 (cit. on p. 10).

[26] 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 p. 31).

[27] 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. 5).

[28] B. Roy. "Transitivité et connexité". In: *C. R. Acad. Sci. Paris* 249 (1959), pp. 216–218 (cit. on p. 25).

[29] *Rust's FusedIterator*. https://doc.rust-lang.org/std/iter/trait.FusedIterator.html. Accessed: 2024-1-09 (cit. on p. 18).

[30] 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. 5).

[31] 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. 4).

[32] S. Warshall. "A Theorem on Boolean Matrices". In: *J. ACM* 9.1 (1962-01), pp. 11–12. ISSN: 0004-5411. DOI: 10.1145/321105.321107. URL: https://doi.org/10.1145/321105.321107 (cit. on p. 25).

A

Graph - WhyML Specification

WhyML

```
module Graph
  use map.Map
  use set.Fset

  type elt
  type graph = abstract {
    dom: fset elt;
    succ: elt → fset elt;
    pred: elt → fset elt;
  }
  invariant { ∀ v1 v2. mem v1 dom → mem v2 (succ v1) → mem v2 dom }
  invariant { ∀ v1.  ¬ mem v1 dom → succ v1 == Fset.empty }
  by { dom = Fset.empty; succ = (fun _ → Fset.empty); pred = (fun _ → Fset.empty) }

  val graphempty () : (g: graph)
  ensures { g.dom = empty }
  ensures { g.succ = (fun _ → empty) }
  ensures { g.pred = (fun _ → empty) }

  val create_singleton (v: elt): (g: graph)
  ensures { g.dom = Fset.singleton v }

  val constant vv: elt
  val create_singleton_pair (v: elt): (g: graph)
  ensures { g.dom = Fset.singleton v }

  type gvpair = abstract{
    g: graph;
    v: elt;
  }
  invariant { mem v g.dom }
  by { g = create_singleton_pair (vv) ; v = vv }

  val gvpair (g: graph) (v: elt) : gvpair
  ensures { result.g = g }
  ensures { result.v = v }
```

```
  val emptygvpair () : gvpair
  ensures { result.g.dom = empty }
  ensures { result.g.succ = (fun _ → Fset.empty) }
  ensures { result.v = vv }

  val mem_vertex (g: graph) (v: elt) : bool
  ensures { result ↔ mem v g.dom }

  val mem_edge (g: graph) (src: elt) (dst:elt): bool
  ensures { result ↔ (mem dst (g.succ src))}

  val add_vertex (ref g: graph) (x: elt) : ()
  writes { g }
  ensures { g.dom == add x (old g.dom) }
  ensures { ∀ s: elt. g.succ s == (old g.succ) s }
  ensures { ∀ s: elt. g.pred s == (old g).pred s }  (* unchanged predecessors *)

  val add_edge (ref g: graph) (src: elt) (dst: elt) : ()
  writes { g }
  ensures { g.dom == old g.dom }
  ensures { g.succ = Map.set (old g.succ) src (add dst (old g.succ src) ) }

  (* use [g] and set the successors of [g'] [src] : return [graph] *)
  val copy_edges (ref g:graph) (g': graph) (src: elt) : ()
  ensures { g.dom == g.dom }
  ensures { g.succ = Map.set g.succ src ( g'.succ src ) }
  ensures { g.pred = Map.set g.pred src ( g'.pred src ) }
end
```

B

Cursor - WhyML Specification

WhyML

```
module SetCursor
  use set.Fset
  use int.Int
  use export mygraph.Graph
  val eq_elt (x y: elt) : bool
    ensures { result ↔ x = y }
  clone export set.SetApp with type elt = elt, val eq = eq_elt
  predicate subcollection   (v: fset elt) (c: fset elt) = subset v c
  predicate equalcollection (v: fset elt) (c: fset elt) = v == c

  type t = private {
    mutable visited : fset elt;
            collection : fset elt;
  } invariant { subcollection visited collection }
  by { visited = emptyset(); collection = emptyset() }

  exception Done
  val next (t: t) : elt
    writes  { t }
    raises  { Done → t.visited = old t.visited ∧ equalcollection t.visited t.collection }
    ensures { ¬ mem result (old t.visited) }
    ensures { t.visited = add result (old t.visited) }

  function set_variant ( t: t ) : int = (cardinal t.collection) - (cardinal t.visited)
end

module VertexSetCursor
  use set.Fset
  use mygraph.Graph
  use SetCursor

  type t = private {
    mutable visited : fset elt;
            graph : graph;
            collection : fset elt;
  }
```

```
  invariant { subcollection visited collection }
  invariant { collection == graph.dom }
  by { visited = SetCursor.emptyset(); collection = SetCursor.emptyset(); graph = graphempty() }

  val create (s: graph) : t
    ensures { result.visited = empty }
    ensures { result.graph = s }
  clone export SetCursor with type t = t
end

module SuccSetCursor
  use set.Fset
  use mygraph.Graph
  use SetCursor

  type t = private {
    mutable visited : fset elt;
            graph : gvpair;
            collection : fset elt;
  }
  invariant { subcollection visited collection }
  invariant { collection == graph.g.succ graph.v }
  by { visited = SetCursor.emptyset(); collection = SetCursor.emptyset(); graph = emptygvpair() }

  val create (s: gvpair) : t
    ensures { result.visited = empty }
    ensures { result.graph = s }
  clone export SetCursor with type t = t
end

module PredSetCursor
  use set.Fset
  use mygraph.Graph
  use SetCursor

  type t = private {
    mutable visited : fset elt;
            graph : gvpair;
            collection : fset elt;
  }
  invariant { subcollection visited collection }
  invariant { collection == graph.g.pred graph.v }
  by { visited = SetCursor.emptyset(); collection = SetCursor.emptyset(); graph = emptygvpair() }

  val create (s: gvpair) : t
    ensures { result.visited = empty }
    ensures { result.graph = s }
  clone export SetCursor with type t = t
end
```

C

Graph operations - WhyML Proofs

WhyML

```
module Operations
  use ref.Ref

  type elt
  clone mygraph.Graph with type elt = elt
  use mycursor.VertexSetCursor
  use mycursor.SuccSetCursor

  predicate mem_vertex (g: graph) (v: elt) = mem v g.dom
  predicate mem_edge (g: graph) (src: elt) (dst: elt) = ( mem dst (g.succ src) )

  let copy (g: graph): (g': graph)
    ensures {g.dom == g'.dom}
    ensures {g'.succ = fun _ → Fset.empty }
    ensures {g'.pred = fun _ → Fset.empty }
  =
    let ref r = graphempty() in

    for v in g with VertexSetCursor as vc do
      variant { vc.VertexSetCursor.set_variant }
      invariant { ∀ x. mem x vc.VertexSetCursor.visited ↔ mem_vertex r x }
      invariant { r.succ = fun _ → Fset.empty }
      invariant { r.pred = fun _ → Fset.empty }

      add_vertex r v
    done;
    r

  let copy_graph (g: graph): (g': graph)
    ensures {g.dom == g'.dom}
    ensures { ∀ v: elt. mem_vertex g v → g.succ v == g'.succ v}
    ensures { ∀ src dst : elt. mem_vertex g src → mem_edge g src dst ↔ mem_edge g' src dst}
    ensures { ∀ src dst : elt. mem_vertex g src → mem dst (g.pred src) ↔ mem dst (g'.pred src)}
  =
    let ref r = copy g in
```

```
  for v in r with VertexSetCursor as vc do
    variant { vc.VertexSetCursor.set_variant }
    invariant { r.dom == g.dom}
    invariant { ∀ src dst: elt. mem src vc.VertexSetCursor.visited →
                                (mem dst (r.succ src) ↔ mem_edge g src dst) ∧
                                (mem dst (r.pred src) ↔ mem dst (g.pred src)) }
    copy_edges r g v
  done;
  r

let mirror (g: graph) : (g': graph)
  ensures { ∀ src dst: elt. mem_edge g src dst → mem_edge g' dst src }
  ensures { g.dom == g'.dom }
=
  let ref cp = copy g in
  for src in g with VertexSetCursor as vc do
    variant { vc.VertexSetCursor.set_variant }
    invariant { ∀ v e. mem v vc.VertexSetCursor.visited → mem_edge g v e → mem_edge cp e v}
    invariant { cp.dom == g.dom}
    for dst in gvpair g src with SuccSetCursor as ec do
      variant { ec.SuccSetCursor.set_variant }
      invariant { ∀ v e. mem v vc.VertexSetCursor.visited → v ≠ src → mem_edge g v e → mem_edge cp e v}
      invariant { ∀ e.   mem e ec.SuccSetCursor.visited → mem_edge cp e src}
      invariant { cp.dom == g.dom}
        add_edge cp dst src
      done;
  done;
  cp

let intersect (g: graph) (g': graph) : (g'': graph)
  ensures { ∀ v: elt. g''.succ v == inter ( g.succ v ) (g'.succ v) }
  ensures { g''.dom = inter g.dom g'.dom }
=
    let ref cp = graphempty() in
    for src in g with VertexSetCursor as vc do
      variant { vc.VertexSetCursor.set_variant }
      invariant { cp.dom == inter vc.VertexSetCursor.visited g'.dom }
      invariant { ∀ v: elt. mem v vc.VertexSetCursor.visited →
                            mem_vertex g' v → cp.succ v == inter (g.succ v) (g'.succ v) }
      if mem_vertex g' src then (
        add_vertex cp src;
        for dst in gvpair g src with SuccSetCursor as ec do
          variant { ec.SuccSetCursor.set_variant }
          invariant { cp.dom == inter vc.VertexSetCursor.visited g'.dom }
          invariant { ∀ v: elt. mem v (remove src vc.VertexSetCursor.visited) →
                                mem_vertex g' v → cp.succ v == inter (g.succ v) (g'.succ v)  }
          invariant { cp.succ src == inter ( ec.SuccSetCursor.visited ) (g'.succ src) }
            if mem_edge g' src dst then add_edge cp src dst;
        done;)
    done;
    cp
```

```
let complement (g: graph) : (g': graph)
  ensures { ∀ src dst: elt. mem_vertex g src → mem_vertex g dst →
                          ¬ mem_edge g src dst ↔ mem_edge g' src dst }
  ensures { ∀ src dst: elt. mem_vertex g src → mem_vertex g dst →
                             mem_edge g src dst ↔ ¬ mem_edge g' src dst }
  ensures { g.dom == g'.dom }
=
    let ref cp = copy g in

    for src in g with VertexSetCursor as vc do
      variant { vc.VertexSetCursor.set_variant }
      invariant { cp.dom == g.dom}

      invariant { ∀ v v': elt. mem v (diff vc.VertexSetCursor.collection vc.VertexSetCursor.visited) →
                               mem_vertex g v' → ¬ mem_edge cp v v' }
      invariant { ∀ v v': elt. mem v vc.VertexSetCursor.visited → mem_vertex g v' →
                            ¬ mem_edge g v v' ↔ mem_edge cp v v' }
      invariant { ∀ v v': elt. mem v vc.VertexSetCursor.visited → mem_vertex g v' →
                               mem_edge g v v' ↔ ¬ mem_edge cp v v' }
      for dst in g with VertexSetCursor as vc' do
        variant { vc'.VertexSetCursor.set_variant }
        invariant { cp.dom == g.dom}

        invariant { ∀ v v': elt. v ≠ src → mem v vc.VertexSetCursor.visited → mem_vertex g v' →
                                          ¬ mem_edge g v v' ↔ mem_edge cp v v' }
        invariant { ∀ v v': elt. v ≠ src → mem v vc.VertexSetCursor.visited → mem_vertex g v' →
                                            mem_edge g v v' ↔ ¬ mem_edge cp v v' }
        invariant { ∀ v v': elt. v ≠ src → mem v vc.VertexSetCursor.visited →
                                            mem v' vc'.VertexSetCursor.visited →
                                          ¬ mem_edge g v v' ↔ mem_edge cp v v' }
        invariant { ∀ v v': elt. v ≠ src → mem v vc.VertexSetCursor.visited →
                                            mem v' vc'.VertexSetCursor.visited →
                                            mem_edge g v v' ↔ ¬ mem_edge cp v v' }
        invariant { ∀ v: elt. mem v vc'.VertexSetCursor.visited →
                              ¬ mem_edge g src v ↔ mem_edge cp src v }
        invariant { ∀ v: elt. mem v vc'.VertexSetCursor.visited →
                                mem_edge g src v ↔ ¬ mem_edge cp src v }
        invariant { ∀ v v': elt. mem v (diff vc.VertexSetCursor.collection vc.VertexSetCursor.visited) →
                                    mem_vertex g v' → ¬ mem_edge cp v v' }
        invariant { ∀ v: elt. mem v (diff vc'.VertexSetCursor.collection vc'.VertexSetCursor.visited) →
                              ¬ mem_edge cp src v }

        if (mem_edge g src dst) then (assert { ¬ mem_edge cp src dst }; )
        else add_edge cp src dst;

      done;
    done;
    cp
```

```
let union (g: graph) (g': graph) : (g'': graph)
  ensures { g''.dom = union g.dom g'.dom }
  ensures { ∀ src: elt. g''.succ src == union (g'.succ src) (g.succ src) }
=
    let ref cp = copy_graph g' in

    for v in g with VertexSetCursor as vc do
      variant { vc.VertexSetCursor.set_variant }
      invariant { cp.dom == union vc.VertexSetCursor.visited g'.dom }
      invariant { ∀ x: elt. mem x (vc.VertexSetCursor.visited) → mem_vertex g x →
                                  cp.succ x == union (g.succ x) (g'.succ x) }
      invariant { ∀ x: elt. mem x (diff cp.dom vc.VertexSetCursor.visited) → cp.succ x = g'.succ x }

      add_vertex cp v;

      for dst in ( gvpair g v ) with SuccSetCursor as ec do
        variant { ec.SuccSetCursor.set_variant }
        invariant { cp.dom == union vc.VertexSetCursor.visited g'.dom }
        invariant { ∀ x: elt. mem x (diff cp.dom vc.VertexSetCursor.visited) → cp.succ x = g'.succ x }
        invariant { ∀ x: elt. mem x (remove v vc.VertexSetCursor.visited) → mem_vertex g x →
                                    cp.succ x == union (g.succ x) (g'.succ x) }
        invariant { ∀ x: elt. mem x ec.SuccSetCursor.visited → mem_edge cp src dst }
        invariant { cp.succ v == union (ec.SuccSetCursor.visited) ( g'.succ v) }

          add_edge cp v dst;

      done;
    done;
    cp
end
```