AlgoCameleer – Deductive Verification of Classic Algorithms, in Cameleer

Daniel João Nunes Castanho - 55078
Supervisors: António Ravara and Mário Pereira

NOVA School of Science and Technology

Abstract. In this report, we describe the steps, difficulties, and results of this project. Our goal was to develop a small certified library of OCaml code, by taking a subset of the popular OCamlgraph library and proving it correct with Cameleer, a software for deductive verification of OCaml code. The selected parts of the library were two structures and five algorithms, one of which we decided to use for proof through certificate instead of the usual proof of implementation. We report on the successful, mostly-automatic proof of such case studies. Cameleer proved to be a robust, easy to use tool that allows one to tackle the verification of realistic OCaml implementations. The GOSPEL language and the Why3 backend also proved to be key ingredients to the success of this project: on one hand, we are able to write concise and readable specification to our OCaml code; on the other hand, when needed, we were able to conduct lightweight interactive proving directly inside the Why3 IDE

Keywords: Deductive Verification · OCaml · OCamlgraph · Cameleer

1 Introduction

Graphs are important structures in today's computer science since, besides being able to store data, they are useful in representing and solving problems. From simple problems, like Sudoku solvers, to complicated ones, such as finding the fastest way to the nearest pizza shop.

So it comes as no surprise that there are graph code libraries in several languages, like *JGraphT*[1] for Java or *graphlib*[2] for JavaScript, and the one we focused on, *OCamlgraph* for OCaml.

These libraries are of course not iron-clad and nothing assures us that using them will provide the correct results. The fact that they have widespread use gives us some sense of safety, because if hundreds successfully used them, then they probably work. Unfortunately, this is not always true, and a great example is Java's default library sorting algorithm, TimSort, which was found to break occasionally [1]. OCamlgraph itself has been shown to not always work, as demonstrated by the issue reports on the official Github page[3].

[1] `https://jgrapht.org/`
[2] `https://github.com/dagrejs/graphlib`
[3] `https://github.com/backtracking/ocamlgraph`

This raises the question, how can we be confident our program does what we want it to do? The usual way is through testing. We submit the program to an intense test battery and if it passes everything then it is a good indication that it works. However, like Dijkstra once said: "testing can be used to show the presence of bugs, but never to show their absence."

Formal proof of programs was first suggested by Alan Turing himself [2], consisting of assigning to a program a set of assertions that it had to check. Robert Floyd [3] followed up on him, introducing the idea of defining a program's meaning through a set of predicates associated with it, regardless of its programming language. Tony Hoare applied these same ideas to his work, eventually creating what we today call *Hoare Logic* [4]. With it, we specify each instruction of a program with a precondition and a post-condition, creating a *judgment*. These triples (precondition, instruction, post-condition) form the basis of deductive verification of programs. Unfortunately, the way these judgments were defined by Hoare, means we have to specify them for every instruction of our program, which is not only tedious but also quite prone to the very errors we are trying to prevent. In response to this problem, Edsger Dijkstra came up with the *weakest precondition calculus* [5]. This allowed proofs to be done a lot easier, conditions and assertions must be placed at specific parts of the code, and the rest can be inferred with this process. The combination of all the assertions is called the program's specification. After defining it, it is necessary to prove that the program adheres to it, and as such, theorem provers are used. If we successfully prove that a program adheres to its current specification, we have proven the program (this means that a program's correctness is only as strong as its specification).

With all of this in mind, we decided to take a subset of the OCamlgraph library and try to formally prove it using Cameleer [6], a tool for proof of programs written in OCaml, creating a small *certified* library. Given that Cameleer is still in development, this project doubled as a stress test for this tool.

Contributions. Through this project we were forced to add more features to Cameleer, to accommodate for the OCamlgraph code. Such features include initialization of functors and the addition of standard library structures such as Maps, Sets and Hashtbls. The complete proofs of our selected case studies are useful and important towards the OCaml and software verification communities. We also offered a suggestion to OCamlgraph itself through a pull request[4], to add a function to their library (issue explained further). Submission of a paper to the Working Conference on Verified Software: Tools, Techniques and Experiments[5].

Report structure. This report is split into several sections. In Section 2, we provide a short description of Cameleer, as well as an example of a small proven function. In Section 3, we go a bit more into detail about Ocamlgraph, showing a small example of how to use it. In Section 4 we present our proven code, alongside some explanations, changes and proof examples. In Section 5, we refer to the

[4] `https://github.com/backtracking/ocamlgraph/pull/116`
[5] `http://verify.inf.usi.ch/VSTTE21`

work related to our project, mentioning similar undertakings and inspirations. The final part of the report, Section 6, offers a conclusion and some planned work for the future. All the files (examples, proofs or proof sessions) used in this project can be found in the project's GitHub page[6].

2 Cameleer in a nutshell

As mentioned before, Cameleer is a code verification tool, specifically designed to prove OCaml code. It is built on top of Why3, a set of tools to perform deductive verification of programs. Why3 comes with its own logic language, called **Why**, which is an extension of first-order logic (e.g. adds recursive definitions, algebric data types, inductive predicates), and its own programming language, called **WhyML**. A distinctive feature provided by this set of tools is that it allows the use of external solvers in its proofs.

To use Cameleer, we attach GOSPEL [7] notation comments to our OCaml code and feed it to the tool. GOSPEL is a behavioral specification language for OCaml and it is designed to enable modular verification of data structures and algorithms. These comments represent the specification of our code and can contain constructs such as preconditions, post-conditions, variants, invariants. We can also define predicates and functions using GOSPEL comments, which come in handy during our proofs. Here is a small example of a function written in OCaml, with GOSPEL comments, which was proved using Cameleer:

```
1  (*@ open Power *)
2
3  let rec fast_exp x n =
4    if n = 0 then 1
5    else let r = fast_exp x (n / 2) in
6      if n mod 2 = 0 then r * r else r * r * x
7  (*@ r = fast_exp x n
8        requires 0 <= n
9        variant  n
10       ensures  r = x ^ n *)
```

The function exemplified above shows a recursive and efficient way to compute a fast exponential, which uses some tricks to calculate the exponential using only $O(log(n))$ multiplications instead of $O(n)$.

Notice the comments started with a `@` in the above code. These are the GOSPEL comments mentioned before and define the function's specification, which in this case involves one precondition (the `requires` clause), one post-condition (the `ensures` clause), and one variant. We can also use external modules in our proofs (though these modules can only be used in specifications) such as the `Power` module opened at the start of the proof, which allows the use of the `^` symbol.

[6] `https://github.com/dCastanho/algocameleer`

Here, we tell our user, that if `n` is equal or greater than 0, then the result will be `x` to the power of `n`. Besides indicating what the purpose of the function is, we must assure that it is safe - as in, it does not get stuck in an infinite loop (recursive or otherwise). For this purpose we add the variant, a value that decreases with each call until it reaches a point where the loop ends.

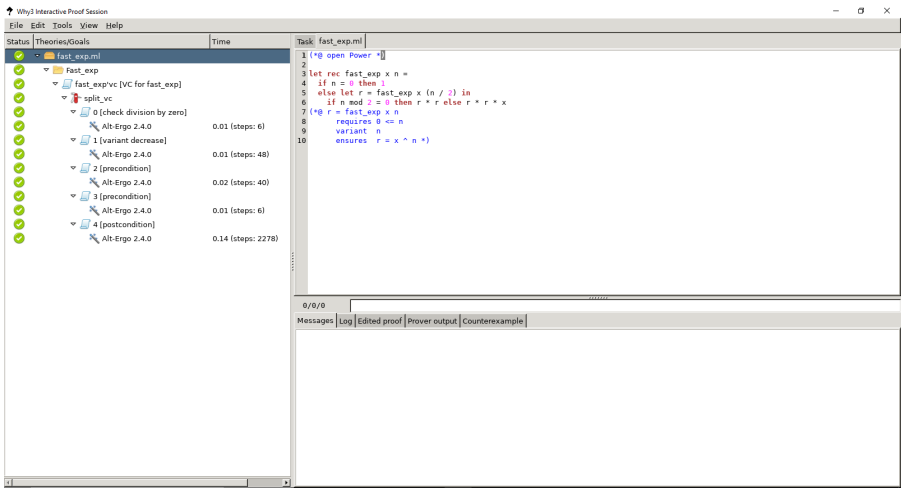

Fig. 1. Visualization of the `fast_exp.ml` proof in the **Why3 IDE**

A small thing to keep in mind is that post-conditions can only be assumed to be true if the preconditions are also true when the function is called. For example, if we were to call `fast_exp` with a negative `n`, we could not expect the result to be valid, since our call does not respect the precondition.

Feeding a file to our tool is simple: after installing Cameeler one must simply call the command `cameleer <filename>.ml`. A graphical interface will pop up, in which all the methods with GOSPEL specification will be listed and can be selected to be proven. When selected, it is also necessary to choose a theorem prover already installed. In the case of our example we used Alt-Ergo[7], which failed to complete the proof right away. Sometimes, it is necessary to split the conditions of a specification using the command `split_vc`, which separates the various conditions to try and prove them one by one, assuming the rest are true. After this, Alt-Ergo quickly dispatched the proof of our little example. See **Figure 1** for a visualization of this example in the **Why3 IDE**.

Cameleer is still in development at this point, and so it lacks compatibility with some aspects of the core language, such as higher-order functions with side effects or the inclusion of modules in other modules.

[7] `https://alt-ergo.ocamlpro.com/`

3 OCamlgraph

As aforementioned, OCamlgraph is a graph library written in OCaml. The first notable aspect of this library is that it "does not provide a single data structure for graphs but many of them, enumerating all possible variations (19 altogether)—directed or undirected graphs, persistent or mutable data structures, user-defined labels on vertices or edges, etc.—under a common interface" [8]. This is possible due to OCaml's flexible and powerful module system.

This system offers special modules, called functors[8] which are important for the second notable feature of this library: the separation of algorithms and data structures, meaning that algorithms are built independently from the underlying data structure. This makes it so, for instance, it does not matter how the vertices and edges are stored, as long as the necessary functions for using the algorithm are supplied in the functor's argument.

OCamlgraph provides a large level of abstraction with all its modules and makes using the library itself incredibly easy, especially since all the graphs end up having very similar signatures. However, analyzing the code behind it proved to be quite difficult because of this abstraction. Not only that, the existence of higher-order functors - functors that instead of returning a module, return another functor - was quite troublesome, not just for us who were inspecting the code, but for Cameleer itself, that was not prepared to handle things of the sort, as mentioned before. Because of this, we were forced to dial down the abstraction, creating a single functor that represented one structure, for each structure analyzed.

```
include Graph
include Path

module IntComparable = struct
  type t = int
  let compare a b = if a = b then 0 else if a < b then -1 else 1
  let equal a b = a = b
  let hash a = a
  let default = 0
end

module DP = Persistent.Digraph.
    ConcreteLabeled(IntComparable)(IntComparable)

module CheckPathDP = Path.Check(DP)
let () =
  let g = DP.empty in
  let g = DP.add_vertex g 1 in
  ...
```

[8] https://ocaml.org/manual/moduleexamples.html#s%3Afunctors

```
let g = DP.add_edge g 2 3 in
let finder = CheckPathDP.create g in
if CheckPathDP.check_path finder 1 3
    then print_string "Path exists"
    else print_string "Path does not exist"
```

In the simple example shown above, we defined a module (`IntComparable`) that represents an integer, adhering to the `COMPARABLE` and `ORDERED_TYPE_DFT` signatures, so it can be used as our vertices and edge labels. Afterward, we instantiate our graph's functor using the module we just defined. Since this instantiated module adheres to the signature necessary for `Path.Check`, we can simply feed it as an argument to the functor. With our modules ready, we can start building our graph. Notice how each call to a function of `DP` returns a new graph, this is what we call a *persistent* structure in OCamlgraph. There are also *imperative* structures in OCamlgraph, which alter the structure in memory and return `unit`. If we were to change the type of graph to be used in this example, we would most likely not have to change anything in our code besides the instantiating of the module `DP` - so long as the type of structure remained the same - because the signatures across graphs are extremely similar.

4 Formally verified case studies

Over the course of the project, we proved four different modules. Two of them represent structures: the first is an imperative, unlabeled digraph[9] and the second is a persistent, labeled graph [10]. The other two consist of algorithms over graphs: the first finds a cycle in a given graph[11], and the other checks whether a path between two given vertices exists. We provide two implementations of this second algorithm: the first is the original implementation[12], and the second is an altered version, which was easier to prove and is slightly more efficient[13]. The two structures were selected to try and cover as many of the available options for graphs (enumerated in Section 3) as possible, both of which use concrete (and not abstract) values in their vertices and edges for simplicity's sake.

In this section, we will present small segments of code regarding each module and explanations about them. For our proofs, we used the following theorem provers: Z3[14], Alt-Ergo, CVC4[15] and EProver[16]. Eprover is a slightly different prover than the rest but proved itself to be extremely useful, as some proofs could only be discharged by it. It is also possible to find some statistics regarding the execution of the proofs in the Appendix.

[9] https://github.com/dCastanho/algocameleer/blob/main/proofs/imperative_unlabeled_digraph.ml
[10] https://github.com/dCastanho/algocameleer/blob/main/proofs/persistent_labeled_graph.ml
[11] https://github.com/dCastanho/algocameleer/blob/main/proofs/find_cycle.ml
[12] https://github.com/dCastanho/algocameleer/blob/main/proofs/path.ml
[13] https://github.com/dCastanho/algocameleer/blob/main/proofs/altered_path.ml
[14] https://github.com/Z3Prover/z3
[15] https://cvc4.github.io
[16] https://wwwlehre.dhbw-stuttgart.de/~sschulz/E/E.html

4.1 Persistent Labeled Graph

As explained before, persistent structures are those where each function creates a new version of the original structure and returns such version with the desired change. One of the proved structures is of such kind. It also differs from the other structure in its edges, where the other is unlabeled and directed. This one has generic labels on its edges and they are undirected, which means they go both ways. This structure is incorporated in the following module:

```
module PersistentLabeledGraph(Vertex: COMPARABLE)
                              (Edge: ORDERED_TYPE_DFT)
```

The vertices and their operations are defined with the `COMPARABLE` signature. However, the labels of edges also need to be represented, and for such, we use a functor with two arguments like shown above. The second argument is what we will use to represent our edges and their operations, through the `ORDERED_TYPE_DFT` signature, as follows:

```
module type COMPARABLE = sig
  type t
  val[@logic] compare : t -> t -> int
  (*@ axiom pre_order : is_pre_order compare *)
  val hash : t -> int
  val equal : t -> t -> bool
end

module type ORDERED_TYPE_DFT = sig
    type t
    val [@logic] compare : t -> t -> int
    (*@ axiom pre_order_compare: is_pre_order compare*)
    val [@logic] default : t
end
```

These signatures both contain a type and a way to compare elements from it, and any module that implements either one must have the represented functions. Notice the `[@logic]` tag associated to the `compare` and `default` functions - this tag means that these functions can be used in specifications. The `compare` function here is supposed to define an order between the elements of its signature. We guarantee this through the axiom inside GOSPEL comments, which indicates us that the `compare` function does represent an order and does not return random numbers (the predicate `is_pre_order` can be found in the standard GOSPEL library). This assurance is important because, without it, we could not assume anything related to the `compare` function. The definition and invariants of the graph itself are interesting subjects to present:

```
module HM = Map.Make(Vertex)

```

```
module VE = struct
  type t = Vertex.t * Edge.t
  let [@logic] compare (x1, y1) (x2, y2) =
    let cv = V.compare x1 x2 in
    if cv != 0 then cv else Edge.compare y1 y2
  (*@ axiom pre_order_compare: is_pre_order compare*)
end

module S = Set.Make(VE)

type t = { self : S.t HM.t }
(*@ invariant forall v1. Set.mem v1 self.HM.dom ->
    forall e. Set.mem e (self.HM.view v1).S.dom ->
    let (v2, l) = e in
      Set.mem v2 self.HM.dom /\
      Set.mem (v1, l) (self.HM.view v2).S.dom *)
```

Here, we introduce a module `VE` which is used for hashing of vertex and edge pairs. The graph itself is represented as an adjacencies list - each vertex has a set associated to it, representing its outgoing edges. Our sets contain pairs `(Vertex.t, Edge.t)` which store the destination and the label of the edge. This means parallel edges are allowed, as long as they have different labels.

The type takes on the format of a record, however, originally, it was represented as an alias type, `type t = S.t HM.t`. Because type invariants can only be applied to record types, we altered it. The invariant guarantees two things: that all the successors of a vertex also belong to the graph and that if there is an edge `(v2, l)` in `v1`'s adjacency list, then the edge `(v1, l)` exists in `v2`'s adjacency list. This captures the undirectness of the graph.

In both our structures, we could have chosen to keep the graph type an alias type and not have assigned it an invariant. But to make sure these important invariants were maintained, we would have had to add them as pre and post-conditions to every function that manipulates the graph. We chose to add the invariant for two reasons: first, it is safer, as we do not run the risk of forgetting to add these conditions; secondly, it was a good opportunity to demonstrate how invariants are used and see how Cameleer would handle them in this situation.

Most proofs in the structure were automatically discharged by at least one of our provers. However some times we were forced to interactively help it along in some more complicated assertions. An example of one such proof was in the case of the `find_all_edges` function, presented in **Figure 2**. No prover was able to conclude that every member of the list `S.elements (HM.find v1 g.self)` also belonged to the graph. This list represents the successors of the vertex `v1` and so, according to our invariant, they must belong to the graph. By instantiating the invariant with the graph `g` and vertex `v1` as well as the destruction of some logical implications (splitting the premise and the condition) Cameleer managed to reach this same conclusion.

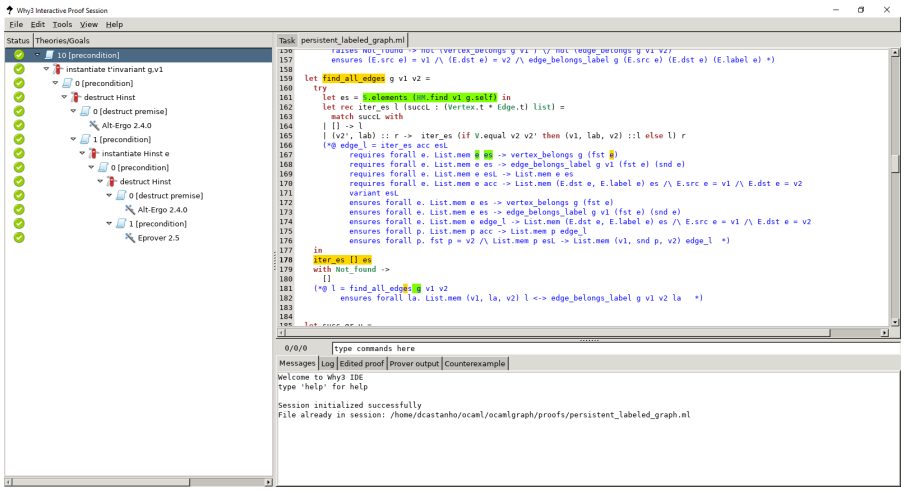

Fig. 2. Visualization of the interactive proof for in the **Why3 IDE**

Despite many of the proofs being automatic, we had to add some lemmas that look very simple, because some times the provers were having hard time reaching those simple conclusions amidst the proof. When generalized, as a lemma, automatic theorem provers were quick to prove them correct. A small example is the `eq_seq_list` shown ahead, which simply states that all elements of a list are also represented on the sequence generated from it.

```
(*@ lemma eq_seq_list : forall l :  (Vertex.t * S.t) list, e.
                        List.mem e l -> Seq.mem e (of_list l) *)
```

4.2 Cycle finding

One of OCamlgraph's cycle detection routines is a function called `has_cycle`, which returns a Boolean, whether or not the graph has a cycle. Interestingly, one of the tasks on OCamlgraph's to-do list, since around 2015, is updating this function to return the cycle that was found, instead of simply a Boolean.

After a brief discussion with one of the developers, via the project's Github, we decided to add a new function called `find_cycle`, inspired by `has_cycle`, which returned the cycle found in the form of a list. Given the complexity of the function, we decided to use it for a different kind of proof, one we call *proof through a certificate*. This kind of proof is useful when the function or program we are trying to prove is complex, but given the output of it, we can check whether that output is correct or not. And so, instead of proving the function itself, we construct a new support function, a *certificate generator* or *witness validator* so to say, which receives an output of the first function and tests whether it is correct, creating a certificate for it. Then we prove this support function is correct.

This is exactly what we did. After adding the new `find_cycle` function, we created a function that given a list and a graph, checks if that list represents a cycle in the graph. Finally, we proved this function did exactly that. An interesting feature of this kind of certificate proof, is that our support function is largely independent of the implementation of the cycle finder used, so long as the returned list is in the format we expect. As mentioned previously, all our algorithms are independent of a specific graph implementation, relying on receiving it as a module that adheres to a certain signature, as follows:

```
module type T = sig
  val is_directed : bool
  module V : COMPARABLE
  type gt
  (*@ model dom: V.t fset
      model suc: V.t -> V.t fset
      invariant forall v1, v2. Set.mem v1 dom /\
        Set.mem v2 (suc v1) -> Set.mem v2 dom*)
   val [@logic] succ : gt -> V.t -> V.t list
   (*@ l = succ g v
        requires Set.mem v g.dom
        ensures forall v'. List.mem v' l -> Set.mem v' g.dom
        ensures forall v'. List.mem v' l <-> Set.mem v' (g.suc v) *)
  val all : gt -> V.t list
  (*@ l = all g
        ensures forall v. List.mem v l <-> Set.mem v g.dom *)
end
```

Here, that signature requires the following: a module `V` that adheres to `COMPARABLE` and is used to represent the vertices of the graph; a type `gt`, which represents the type of our graph; a function `succ`, which given a graph and a vertex, returns its list of successors, and a function `all`, which given a graph returns a list of all its vertices. Originally, both functions provided higher-order iterators to iterate over these lists, but due to Cameleer's current incompatibility with these, we changed them to return the lists so we could iterate over them manually.

For our specification to make sense of what the `gt` type represents, we assign a model to it. In this case, our model consists of a finite set (`fset`) `domain`, which stores the vertices that belong to the graph and a `succ` function, that given a vertex, returns its successors set. This model is used to provide a specification of the functions of the signature.

```
let is_path v1 l v2 g =
    let rec is_succ v = function
        | [] -> false
        | v' :: vs -> G.V.equal v' v || is_succ v vs
    (*@ b = is_succ v l
```

```
        variant l
        ensures b <-> List.mem v l *)
    in let rec iter_path = function
        | [] -> assert false
        | v' :: v'' :: vs -> is_succ (v'') (G.succ g v') &&
                iter_path (v'' :: vs)
        | v' :: [] -> G.V.equal v' v2
    (*@ b = iter_path l
        requires l <> []
        requires forall v. List.mem v l -> Set.mem v g.G.dom
        variant l
        ensures let s = of_list l in
          b <-> (forall i. 0 <= i < List.length l - 1 ->
           edge s[i] s[i+1] g) /\ s[List.length l - 1] = v2 *)
    in match l with
        | [] -> G.V.equal v1 v2
        | x :: _ -> is_succ x (G.succ g v1) && iter_path l
    (*@ b = is_path v1 l v2 g
        requires Set.mem v1 g.G.dom
        requires Set.mem v2 g.G.dom
        requires forall v. List.mem v l -> Set.mem v g.G.dom
        ensures b <-> is_path v1 (of_list l) v2 g *)
```

Cycles, by definition, are paths from a vertex to itself. As such, the simplest way to define a cycle is to start by defining a path. The function demonstrated above receives a list and two vertices and tests whether the list represents a path from `v1` to `v2`. The list itself, does not contain `v1` but its last element must be `v2`. We make the verification by first checking whether the head of the list is a successor of `v1`. Then, we take each vertex from the list and check if the one that follows it (in the list) is one of its successors (this is done by the `iter_path` function). If the list is empty, then it means there are no other vertices involved, so it returns true if `v1` and `v2` are the same vertex.

Our function has several preconditions, which can be summed up by saying that all vertices involved must belong to the graph, otherwise there is no doubt that it is not a path. The function assures us that the returned Boolean is equivalent to the one returned by the predicate of the same name, defined as follows:

```
(*@ predicate edge (v1 : G.V.t) (v2 : G.V.t) (g : G.gt) =
        Set.mem v2 (g.G.suc v1) *)

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

```
10                0 <= i < len - 1 -> edge l[i] l[i+1] g *)
```

It is easy to see a similarity between the two functions, however, the predicate can use the logical quantifier `forall` as well as operate directly over the model defined in the signature. This function was proved fully automatically by Cameleer, though it is still not the certificate creator we want. The certificate generator was defined as follows:

```
let is_cycle l g =
    let rec get_last = function
    | x :: [] -> x
    | x :: xs -> get_last xs
    | [] -> assert false
    (*@ x = get_last l
          variant l
          requires l <> []
          ensures List.mem x l
          ensures x = (of_list l)[List.length l - 1] *)
    in not (is_empty l) &&
    let v = get_last l in is_path_func v l v g
(*@ b = is_cycle_func l g
      requires forall v. List.mem v l -> Set.mem v g.G.dom
      ensures b <-> is_cycle (of_list l) g *)
```

Now we have to decide how a cycle would be stored in a single list. We ended up by choosing the simplest possible representation: given a list, it is a path from the last vertex to itself (though technically, there is a path from any of its vertices to itself). Given this way of storing cycles, an empty list represents the non-existence of a cycle (as in, empty lists cannot be cycles). The function above does exactly this verification, once again requiring all its members to be part of the graph.

```
(*@ predicate is_cycle (l : G.V.t seq) (g : G.gt) =
     let len = Seq.length l in
     len <> 0 /\ is_path l[len - 1] l l[len - 1] g*)
```

The predicate used to represent this is once again very similar to the actual function, simply calling `is_path` with the correct values. All these functions were proved to adhere to their specifications by Cameleer, meaning that given any function which *supposedly* returns a cycle, we can check whether their result is correct or not.

Since this function is something the OCamlgraph developers have had in their to-do list for a while, we made a pull request to add our implementation of it to the library. We also raised an interesting point regarding cycle finding - in the case where a cycle is not found, returning a *correct* topological sort of the graph can also serve as proof that there are no cycles.

4.3 Path checking

For simple path checking between two vertices, OCamlgraph has a type and a function, where the type serves as a cache for previously searched values and the function uses breadth-first search to find the path between two given vertices, `v1` and `v2`. In this section we will be explaining the original version of the code[17].

The implementation is simple but requires two support structures: a queue (called `q`) for vertices to check and a Hash table (called `visited`) for checked vertices. We start the algorithm by adding `v1` to `q`. At each execution of the algorithm, we check if the queue is empty, if it is, we stop, returning false - as there are no more reachable vertices from `v1` - if it is not, we pop the first value, `v`. If the popped value is equal to `v2`, we have found a path and return true, if not, we add `v` to `visited` and iterate over its successors, adding them all to the queue.

There were several conditions to be treated as invariants over the execution of the algorithm, but due to the use of recursive functions instead of loops, these were adapted as pre and post-conditions. The definition of a path is defined just like in the previous subsection regarding cycles. However this time we will provide a more general definition, which instead of being asked if a given sequence is a path, simply checks if there exists such a path.

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

Proving the path was found when the function returns true is easy, thanks to the invariant that states that the vertices in `q` are all connected to `v1`, which means if `v2` is one of them (when popped), then there is a path between the two.

```
forall v'. Seq.mem v' q.Queue.view -> has_path v1 v' pc.graph
```

Proving that when the function returns true, then there is a path, refers to the *correctness* of it, while returning false (instead of unsure, for example) when there is no path, refers to its *completeness*. This last one proved to be much more difficult than the first.

In our algorithm, if `q` is empty when we start a loop, it means there are no more reachable vertices, and so, there is no path between `v1` and `v2`. From a programming point of view, this train of thought is natural but logically proving it is a different story. The following invariant is how we prove that if the queue is empty, then there is no path:

```
has_path v1 v2 pc.graph ->
    exists w. Seq.mem w q.Queue.view /\ has_path w v2 pc.graph /\
      not (Set.mem w visited.HV.dom)
```

Because if there is a path, then there must be an intermediate *unvisited* vertex in the queue and if there are no more vertices in the queue, this can't happen. With such a property in hand, we were able to prove the completeness of the function.

[17] `https://github.com/backtracking/ocamlgraph/blob/master/src/path.ml`

However, proving this specific condition was not done automatically and before we did so we needed an extra invariant.

```
forall v. Set.mem v visited.HV.dom ->
   forall s. edge v s pc.graph ->
       Seq.mem s q.Queue.view \/ Set.mem s visited.HV.dom
```

This means that if a vertex is visited, then all its children have either been visited or are in the queue. Cameleer easily proved this condition and we proceeded to the proof of the previous one, doing so through a case by case analysis:

1. If `v2` is in the queue, then it is easily proven, as it is the vertex we are looking for. The condition `has_path v2 v2` trivially holds.
2. If `v2` is not in the queue, then we know `v1` is in `visited`, because `v1` is the first vertex to be popped. So we know that the there is a path from inside the `visited` set (`v1`) to outside of it (`v2`) . This path takes an edge `v`→`w`, where `v` is in `visited` and `w` is in the queue, but not in visited, and is exactly the vertex we're looking for.

To prove this last case we required the aid of the following ghost function:

```
let [@ghost] [@logic] rec intermediate_value p
    (u : G.V.t) (v : G.V.t) ( s : G.V.t list) ( g : G.gt)  =
      match s with
      | [] -> assert false
      | x :: xs -> if not (p x) then (u, x, [], xs) else
       let (u', v', s1, s2) = intermediate_value_func p x v xs g
       in (u', v', x::s1, s2)
(*@ (u', v', s1, s2 ) = intermediate_value p u v s g
    requires p u /\ not (p v) /\ is_path u (of_list s) v g
    variant s
    ensures p u' /\ not (p v') /\ is_path u s1 u' g
    ensures is_path v' s2 v g /\ edge u' v' g *)
```

Given a certain property `p`, which takes the form of a Boolean function, two vertices, a list representing their path, and the graph where they belong to, it will find the edge in the path which starts in a vertex that respects `p` and ends in a vertex which does not.

If we instantiate this function with our values `v1`, `v2`, our graph, the path used by `has_path` and `p = fun x -> Set.mem x visited.HM.dom`, it will search for the vertex `w` referred in the second case. We know this vertex will be in the queue, because we if `v` (from the edge `v`→`w`) is visited, then all its children are either visited or in the queue. If `w` is not visited, then it *must* be in the queue.

Because our path definition uses sequences and not OCaml lists , to be able to use this function in our proof correctly (it uses lists and not sequences because sequences are not an OCaml construct but a GOSPEL one), we had to create the following auxiliary lemma.

```
(*@ lemma intermediate_value :
forall p : (G.V.t -> bool), u, v : G.V.t, s : G.V.t seq, g : G.gt.
  p u -> not p v -> is_path u s v g ->
    exists u' v' s1 s2. p u' /\ not p v' /\
       is_path u s1 u' g /\ is_path v' s2 v g /\ edge u' v' g *)
```

This lemma was what ultimately helped us conclude the proof of completeness and the function explained before was used to prove this lemma. See **Figure 3** for a visualization of the proof of the condition in the **Why3 IDE**.

An alternative method. The way this function is written makes it so the queue and the `visited` table have overlapping vertices, which forces us to refer to the unvisited but reached vertices by the difference of these two. Not only that, it makes it so the queue might have repeated values and can possibly grow to have more vertices than there are in the graph. We propose an alternative implementation that avoids both of these issues: we made `visited` a ghost structure and added in a new Hash table by the name of `marked`; then, instead of adding successors to the queue with no verification, we first check whether that vertex has been marked or not. If it has, we simply move on to the next successor; if it hasn't, we mark it and add it to the queue; on pop vertices are still added to visited. This means that `visited` and `q` are disjoint and that all the values in `q` are distinct. This implementation was also fully proved in Cameleer, with a simpler completeness proof, given the separation of the values in `visited` and `q`

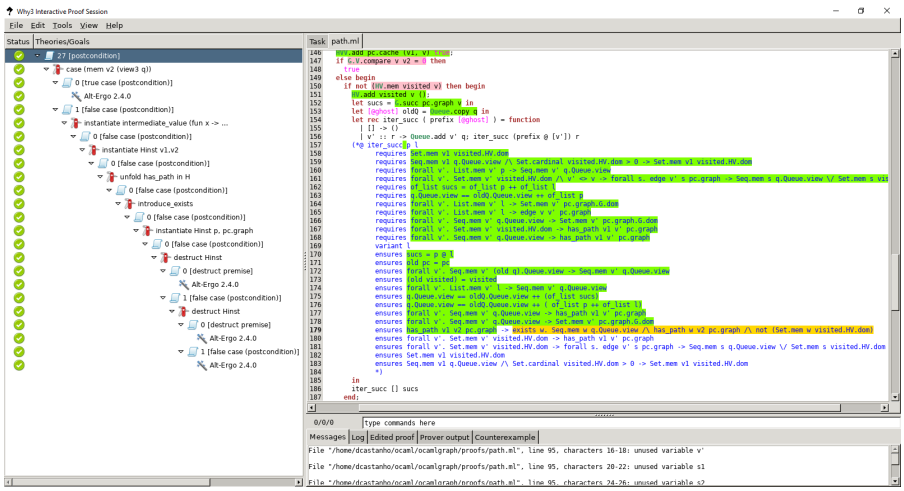

Fig. 3. Visualization of the proof of completeness in the **Why3 IDE**

5 Related work

Given the importance of graphs, it makes sense that formal verification of algorithms over graphs is not new. For example, the Why3 gallery of verified programs contains a small section specifically for graph algorithms[18], including a breadth-first search such as the one we proved. There are differences, however, such as the fact that their proof was made in WhyML, Why3's programming language, and ours was made in OCaml. Not to mention that our proof was taken from already written code, instead of a proof-oriented implementation.

Another example of verified graph algorithms is [9] in which the authors prove some classic algorithms written in C. The implementations proved in this particular work were taken from textbooks and included the verification of heaps and other support structures, along with the rebuttal of some common notions like how *Prim's Algorithm* requires a connected graph.

In [10] we can find proofs for Tarjan's strongly connected components algorithm, proved with three different provers: Isabelle/HOL, Coq and Why3.

Alkassar et al. [11] adjust graph algorithms to produce witnesses that can be then used by verified validators to check whether the result is correct, similarly to how we create a certificate and validators for our cycle finding.

Several other works about formal and informal proof of graph algorithms and/or programs exist [12,13,14,15,16,17,18,19]. Some focus on the correct treatment of pointers or concurrent structures while other are part of larger libraries.

As far as we are aware, there is no OCaml graph library currently verified and we found no works on the correctness of graph *structures*, only their algorithms.

6 Conclusion and Future work

During the execution of this project, we managed to accomplish the goals we set out to do. We verified structures and algorithms from the OCamlgraph library and forced Cameleer to add features that had not yet been thought about. However, in the grand scheme of things, the subset of the library we proved is rather small and Cameleer still requires more features to be fully functional. As such, our plan for the future involves expanding on both of these topics. Proving more algorithms, like the classic *Prim's algorithm*, and structures, like graphs that utilize abstract values instead of concrete ones, will increase the verified subset of the OCamlgraph library. By adding the ability to handle higher-order iterators to Cameleer, we will fix one of its major flaws and biggest source of incompatibility with real, already written code. This last one is complicated, but not impossible, since these iterators can be easily compared to the classic `for each` cycle, and possibly be converted into them. Following such a route, would allow us to modularly reason about iteration, as presented by Filliatre and Pereira [20]. From a practical point of view, this means extending Cameleer with the ability to identify higher-order iterators and convert them to a piece of WhyML code that uses the enhanced for loop.

[18] `http://toccata.lri.fr/gallery/graph.en.html`

References

1. S. de Gouw, J. Rot, F. S. de Boer, R. Bubel, and R. Hähnle, "Openjdk's java.utils.collection.sort() is broken: The good, the bad and the worst case," in *Proceedings of the 27th International Conference on Computer Aided Verification*, vol. 9206 of *Lecture Notes in Computer Science*, pp. 273–289, Springer, 2015.
2. F. L. Morris and C. B. Jones, "An early program proof by alan turing," *IEEE Ann. Hist. Comput.*, vol. 6, no. 2, pp. 139–143, 1984.
3. R. W. Floyd, *Assigning Meanings to Programs*, pp. 65–81. Springer Netherlands, 1993.
4. C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969.
5. E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, no. 8, pp. 453–457, 1975.
6. M. Pereira and A. Ravara, "Cameleer: A Deductive Verification Tool for OCaml," in *Computer Aided Verification - 33rd International Conference, CAV 2021, July 20-23, Proceedings, Part II* (A. Silva and K. R. M. Leino, eds.), vol. 12760 of *Lecture Notes in Computer Science*, pp. 677–689, Springer, 2021.
7. A. Charguéraud, J. Filliâtre, C. Lourenço, and M. Pereira, "GOSPEL - providing ocaml with a formal specification language," in *Proceedings of the Third World Congress - The Next 30 Years - Formal Methods.*, vol. 11800 of *Lecture Notes in Computer Science*, pp. 484–501, Springer, 2019.
8. S. Conchon, J. Filliâtre, and J. Signoles, "Designing a generic graph library using ML functors," in *Proceedings of the Eighth Symposium on Trends in Functional Programming*, vol. 8 of *Trends in Functional Programming*, pp. 124–140, Intellect, 2007.
9. A. Mohan, W. X. Leow, and A. Hobor, "Functional correctness of c implementations of dijkstra's, kruskal's, and prim's algorithms," in *Computer Aided Verification* (A. Silva and K. R. M. Leino, eds.), pp. 801–826, Springer International Publishing, 2021.
10. R. Chen, C. Cohen, J. Lévy, S. Merz, and L. Théry, "Formal proofs of tarjan's algorithm in why3, coq, and isabelle," *CoRR*, vol. abs/1810.11979, 2018.
11. E. Alkassar, S. Böhme, K. Mehlhorn, and C. Rizkallah, "A framework for the verification of certifying computations," *J. Autom. Reason.*, vol. 52, no. 3, pp. 241–273, 2014.
12. C. M. Poskitt and D. Plump, "Hoare-style verification of graph programs," *Fundam. Informaticae*, vol. 118, no. 1-2, pp. 135–175, 2012.
13. I. Wegener, "A simplified correctness proof for a well-known algorithm computing strongly connected components," *Inf. Process. Lett.*, vol. 83, no. 1, pp. 17–19, 2002.
14. P. Lammich and R. Neumann, "A framework for verifying depth-first search algorithms," *Arch. Formal Proofs*, vol. 2016, 2016.
15. F. Mehta and T. Nipkow, "Proving pointer programs in higher-order logic," *Inf. Comput.*, vol. 199, no. 1-2, pp. 200–227, 2005.
16. I. Sergey, A. Nanevski, and A. Banerjee, "Mechanized verification of fine-grained concurrent programs," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015* (D. Grove and S. M. Blackburn, eds.), pp. 77–87, ACM, 2015.
17. A. Raad, A. Hobor, J. Villard, and P. Gardner, "Verifying concurrent graph algorithms," in *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings* (A. Igarashi, ed.), vol. 10017 of *Lecture Notes in Computer Science*, pp. 314–334, 2016.

18. A. Hobor and J. Villard, "The ramifications of sharing in data structures," in *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013* (R. Giacobazzi and R. Cousot, eds.), pp. 523–536, ACM, 2013.
19. R. Chen and J. Lévy, "A semi-automatic proof of strong connectivity," in *Verified Software. Theories, Tools, and Experiments - 9th International Conference, VSTTE 2017, Heidelberg, Germany, July 22-23, 2017, Revised Selected Papers* (A. Paskevich and T. Wies, eds.), vol. 10712 of *Lecture Notes in Computer Science*, pp. 49–65, Springer, 2017.
20. J.-C. Filliâtre and M. Pereira, "A modular way to reason about iteration," in *8th NASA Formal Methods Symposium* (S. Rayadurgam and O. Tkachuk, eds.), vol. 9690 of *Lecture Notes in Computer Science*, (Minneapolis, MN, USA), Springer, June 2016.

A Statistics

A.1 Computer and Environment specs

- 16 GB of RAM
- AMD Ryzen 5 1600 Six-Core Processor 3.20 GHz
- Executed inside WSL 2.0 in Windows 10 64x, with an Ubuntu 20.04 LTS distribution
- Why3 setup to have up to 5 processes running at once

A.2 imperative_unlabeled_digraph.ml

Number of root goals. **Total:** 22 **Proved:** 22

Number of sub goals. **Total:** 79 **Proved:** 79

Replay time. 7 executions timed, removed maximum and minimum. Statistics bellow are from the leftover 5 tests.

Minimum	**Maximum**	**Average**
0m15.768s	0m16.276s	0m16.001s

Statistics per prover.

Prover	**Number of Proofs**	**Minimum (s)**	**Maximum (s)**	**Average (s)**
Eprover 2.5	27	0.03	2.44	0.24
Alt-Ergo 2.4.0	75	0.00	4.67	0.15
Z3 4.8.7	27	0.01	4.47	0.92
CVC4 1.6	40	0.06	0.25	0.12

A.3 persistent_labeled_graph.ml

Number of root goals. **Total:** 28 **Proved:** 28

Number of sub goals. **Total:** 219 **Proved:** 219

Replay time. 7 executions timed, removed maximum and minimum. Statistics bellow are from the leftover 5 tests.

Minimum	**Maximum**	**Average**
0m24.661s	0m24.898s	0m24.769s

Statistics per prover.

Prover	**Number of Proofs**	**Minimum (s)**	**Maximum (s)**	**Average (s)**
Eprover 2.5	67	0.02	6.46	0.60
Alt-Ergo 2.4.0	179	0.01	4.71	0.13
Z3 4.8.7	55	0.01	0.22	0.07
Z3 4.8.7 (noBV)	30	0.01	0.13	0.05
CVC4 1.6	101	0.07	4.33	0.29

A.4 find_cycle.ml

Number of root goals. **Total:** 9 **Proved:** 9

Number of sub goals. **Total:** 78 **Proved:** 78

Replay time. 7 executions timed, removed maximum and minimum. Statistics bellow are from the leftover 5 tests.

Minimum	**Maximum**	**Average**
0m18.572s	0m19.298s	0m18.809s

Statistics per prover.

Prover	**Number of Proofs**	**Minimum (s)**	**Maximum (s)**	**Average (s)**
Eprover 2.5	2	0.13	0.15	0.14
Alt-Ergo 2.4.0	66	0.01	1.73	0.07
CVC4 1.6	6	0.11	23.71	6.86

A.5 path.ml

Number of root goals. **Total:** 14 **Proved:** 14

Number of sub goals. **Total:** 223 **Proved:** 223

Replay time. 7 executions timed, removed maximum and minimum. Statistics bellow are from the leftover 5 tests.

Minimum	**Maximum**	**Average**
0m22.797s	0m22.933s	0m22.860s

Statistics per prover.

Prover	**Number of Proofs**	**Minimum (s)**	**Maximum (s)**	**Average (s)**
Eprover 2.5	70	0.02	3.22	0.24
Alt-Ergo 2.4.0	165	0.01	3.85	0.19
Z3 4.8.7	54	0.01	0.13	0.05
CVC4 1.6	75	0.04	5.33	0.29

A.6 altered_path.ml

Number of root goals. **Total:** 15 **Proved:** 15

Number of sub goals. **Total:** 325 **Proved:** 325

Replay time. 7 executions timed, removed maximum and minimum. Statistics bellow are from the leftover 5 tests.

Minimum	**Maximum**	**Average**
0m29.238s	0m30.095s	0m29.565s

Statistics per prover.

Prover	**Number of Proofs**	**Minimum (s)**	**Maximum (s)**	**Average (s)**
Eprover 2.5	104	0.03	4.27	0.25
Alt-Ergo 2.4.0	230	0.01	2.00	0.13
Z3 4.8.7	90	0.01	4.10	0.21
CVC4 1.6	97	0.07	3.56	0.25