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 which we will be taking for the completion of our research project during the APDC course, as well as explain some decisions made, such as the chosen programming language and other tools. For the actual project, we will select a few algorithms from the popular OCamlgraph library and write a formal mathematical specification for each of them, which will be used by Cameleer, a software for deductive verification of OCaml code, to prove them. We believe this is important work, as having a widely spread library be formally proved is extremely beneficial for its users. It will also serve as a good way to stress-test the Cameeler tool.

Keywords: Deductive Verification · OCaml · OCamlgraph · Cameleer

1 Introduction

Although often overlooked, graphs are exceptional data structures. They can be used for a data structure's usual purpose of storing information, but their importance stems from their capacity to represent and solve real problems. A simple, yet eye-opening, example is the one given by Jean-Christophe Filliâtre *et al* [1] in which they show how to solve a Sudoku problem through a graph and a 9-coloring algorithm of that graph.

Graphs are such useful structures that it is possible to find code libraries about these in almost every language. To name a few, there is *JGraphT* for Java, *graphlib* for Python, *graphlib* for JavaScript (yes they have the same name) and the one we will be focusing on, *OCamlgraph* for OCaml.

Unfortunately, like all written code, these libraries are still prone to errors and bugs, though to a lesser extent due to their widespread use. A classic example is the TimSort algorithm from the default `java.util` library, which was found to break in certain situations[2]. Ocamlgraph is no exception and despite being in development since 2006, 15 years to today, there is still the occasional bug. This is evidenced by the recent issue reports on the official Github page.

Because of this, we seek a way to know for sure that our libraries will do what they are supposed to do. To do so, we turn ourselves to formal proof of programs, a method through which we can guarantee that the behavior of our code is the one we want.

With all of this in mind, we aim to create a *certified* (formally proved) subset of the OCamlgraph library, and for that end we will be using Cameleer, a tool for formal proof of OCaml programs. We will add the correct specifications to prove that the selected algorithms are correct, changing none of the executable code unless there is a bug. As such, this will be a good way to test Cameleer with real, already written code.

1.1 Formal proof of programs

Errors in computer programs are usually referred to as *bugs*, and there are several types, from run-time crashes to incorrect results. To avoid them, the most common approach is to submit the program to an intense test battery. If it performs as expected, it is a good indication that there *probably* are not any bugs. Sadly, like Edsger Dijkstra once said "testing can be used to show the presence of bugs, but never to show their absence". This leaves us with the question: is there a way to guarantee that our program is bug-safe? The answer is, fortunately, yes. Through the formal proof of programs, we can prove they are error-free, as long as the tools used are correct.

Formal proof of programs usually revolves around a two-step process. First, we assign our code a mathematical specification of what it is *supposed* to do. Second, we prove that the code adheres to the written specification. If it does, we have successfully demonstrated that our program does what it is supposed to do.

There are several formal methods for proof of programs, which we can be loosely split in three types: static (like abstract interpretation[3] or type systems[4]), dynamic (like model checking[5] or monitoring), or deductive (like Hoare style or type theory based provers) analysis. In this project, we will be using deductive program verification to prove our written code.

"Deductive program verification is the activity of turning the correctness of a program into a mathematical statement, and then proving it."[6] One of the biggest hurdles of this method is how to convert our code into mathematical expressions. This is achieved through a set of assertions related to the code. Utilizing *Hoare Logic*[7], we represent the specification of our program through a *judgment*, which consists of a triple, with two logic expressions and a program, taking the form of:

$$\{P\}C\{Q\}$$

The logic originally used in these expressions was the classic first-order logic, however, different tools may use different logics. Here, P represents a *pre-condition*, meaning, the program C will only work properly in states where P is true. Q indicates the program's *postcondition* which means, that if C terminates, the state it ends in will satisfy the predicate Q. This is the definition of *Hoare Triple*, the basis for deductive verification of a program's partial correctness.

From just this definition, we have to provide pre- and postcondition for every instruction of our program. There are several *inference rules* defined by Hoare Logic which specify how to build these triples and allow us to validate a program's Hoare Triples. One of these rules is called *consequence rule*:

$$\frac{P \Rightarrow P' \quad \{P'\}\ C\ \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\ C\ \{Q\}}$$

This rule essentially states that given a Hoare Triple, we can weaken its precondition and strengthen its postcondition while still maintaining its validity.

"The way one uses Hoare logic to verify the correctness of a program makes it hard to scale to the proof of larger programs. To build a valid triple for a whole program, one must provide the intermediate triples for each and every individual expression."[6] This is, obviously, not a good process, as it is lengthy and quite prone to errors. However, Edsger Dijkstra presented a solution to this bothersome process[8]. He proved that in reality the programmer must only declare assertions at certain points of the program and the rest can be automatically inferred. He proposed that given a program C and its postcondition Q (provided by the user) we can compute the weakest precondition, noted as wp(C,Q), such that if a state satisfies wp(C, Q) and C executes and terminates, then Q will be true in the final state. This means that the following Hoare Triple is valid:

$$\{wp(C, Q)\}C\{Q\}$$

As mentioned before, defining the mathematical specification of a program is only the first step of deductive verification. The second step is about proving that specification. We could do this by hand, following the inference rules defined by Hoare Logic. This is however a bothersome process, and so we turn to *theorem provers*, which are tools used to write and/or check formal proof of programs. *Automatic theorem provers* are a subset of theorem provers that facilitate a lot of the deductive reasoning which would normally have to be done by the programmer. "The family of SMT (*Satisfability Modulo Theory*) solvers are nowadays a crucial ingredient in automatic proof of programs." [6] and are the ones we will be using the most in our proofs. Z3[1], Alt-Ergo[2] and CVC4[3] are the three main provers used in the project.

1.2 OCaml

"OCaml is an industrial-strength programming language supporting functional, imperative and object-oriented styles."[4] We chose it as our main programming language for this project for several reasons. First, it is statically typed with type inference, which in itself is a way to prevent many errors. Secondly, OCaml is being more and more prominent in innovating companies like Facebook or Docker. Last but not least, OCaml offers a complex and versatile way to separate, organize and even hide code through the use of modules and signatures. Modules

[1] `https://github.com/Z3Prover/z3`
[2] `https://alt-ergo.ocamlpro.com`
[3] `https://cvc4.github.io`
[4] `https://ocaml.org/`

aggregate functions, types and even exceptions, which can later be used. OCaml's own default library is built on these principles.

Signatures can be looked at as interfaces, they simply provide *signatures* for functions and possibly declare *unimplemented* types. For example, a signature may state that a type t exists, but not declare how it is implemented. Signatures can then be bound to modules, in which case, that module must implement everything declared on the signature, including types and functions. From an outside perspective, that module will now be seen as the signature, meaning the client code cannot access its implementation.

The most complex part of this system, but also the most useful, is the existence of functors. Functors are modules, that take other modules as an argument. This is a strange idea, but for example, if we wish to create a module for sorted lists, we need to have a way to compare the elements of our list. OCaml provides a way to declare polymorphic functions, that can take any type as an argument or return value, but the default polymorphic comparison operators are inefficient and sometimes even unsafe. The solution for this issue is taking in a module as an argument, a module that declares a type, which will be the one we will be storing in our list, and operations to compare it. Most times, we declare a signature in our functor, meaning any module adhering to it, can be used as input.

Functors and modules are extremely useful because they offer a large degree of versatility and allow for highly generic types. They are also crucial to OCamlgraph, allowing the user of this library to implement very modular solutions.

2 Tools

During the course of the project we will be using a couple of pre-existing tools and resources, briefly explained in this section.

2.1 Why3

Why3 is a set of software tools to make *deductive verification* of software easier. It provides a logic language called **Why**, which "is an extension of first-order logic with polymorphism, algebraic data types, and inductive predicates" [9]. There is also a programming language included, by the name of **WhyML** which is described as "a first-order ML-like language with imperative features, pattern matching, and exceptions"[9]. The two coexist in this environment, where the logic is used to *specify* the programs (through pre-conditions, postconditions, loop invariants, etc.). Why3 uses calculus of the weakest pre-condition to extract a program's verification conditions, which are then transformed before being forwarded to one of the many available external provers.

There are three main ways to use the Why3 platform: for its logic only, to verify algorithms and data structures or as an intermediate language for programs written in different programming languages. The third way is the one we will be using the most, as we will be converting OCaml code into WhyML for Why3 to analyze.

2.2 Cameleer

Cameleer is a verification tool for programs written directly in OCaml[5]. It works by having the programmer specify the OCaml code using the GOSPEL language[10]. The code is then translated to WhyML and supplied to Why3, that proves it like a normal WhyML program, including using any of its many available off-the-shelf theorem solvers. The task of translating code from one language to another is much harder than it seems on paper. Despite their similarities, there are several key differences between the two programming languages and due to the size of the OCaml language, at this point in time, only a subset of it is allowed by Cameleer, roughly corresponding to *caml-light*[11].

2.3 OCamlgraph

As aforementioned, OCamlgraph is a graph library for 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"[1]. The second aspect is that all the provided algorithms over graphs, are themselves independent from any implementation of graphs.

The amount of provided data structures would make one think that the code behind it is extensive and unmanageable when in fact that is quite far from the truth. OCamlgraph manages to implement all of the 19 structures in roughly 1000 lines of code, and due to the module system provided by OCaml, these implementations are well organized and easy to expand if necessary.

The decoupling of algorithms and graph implementations is useful for two main reasons: it means these algorithms can be used even with user-implemented graphs and that adding an algorithm to the library is extremely simple since nothing else needs to be altered. This separation is possible with the use of functors, where we program our algorithm, focusing only on the required functionalities the graph must have to use said algorithm.

3 Related Work

To the best of our knowledge, there is no verified graph library written in OCaml. However, formal proof of programs has been around for a while, and over the last few years, it has grown in importance along with the increase of tools to aid in it. As such, it is no surprise that there are in fact certified code bases, for example, *WhyMP*[12], which is a formally verified arbitrary-precision integer library in C. Another great example is *HACL**[13], a verified *cryptographic* library written in F* and then compiled to C. Finally, we should also mention the VOCaL Project[14], which aims to develop a general purpose certified OCaml library.

[5] `https://github.com/ocaml-gospel/cameleer`

Another important related work comes from the examples provided by the developers of Cameleer[6]. These examples show how to specify certain complex algorithms and functions that manipulate data structures. Despite not being related to graphs, these examples will be useful in serving as a source of inspiration as well as a good reference for when we must write our own specifications.

4 Preliminary examples

In this section, we give two examples of features we will deal with in the project: the first shows an implementation of a depth first search that is independent from the graph's implementation, similarly to how it is done in OCamlgraph. The second briefly shows how annotations in GOSPEL look like and explains some of them, by implementing a simple directed graph with a signature very similar to the ones in the OCamlgraph library. Files can be found in the Github repository[7].

4.1 Depth First Search

We start by defining a module for a depth first search algorithm, in the form of `module DFS = struct...end`. Inside the module is where we will have the actual implementation of our algorithm, as well as the signature a graph must adhere to so it can use depth first search. Let's start by defining the signature:

```
module type G = sig
    type t
    type elt
    val iterAll : t -> (elt  -> unit) -> unit
    val iterAdj : t -> elt  -> (elt -> unit) -> unit
    val v : t -> int
end
```

The signature G declares two types, **t** which represents the graph, and **elt** which represents the type of the elements stored in the graph. We also need a function to iterate over all vertices, one to iterate over the adjacent vertices of a given vertex and one to get the number of vertices (though this one could possibly be removed). Note how we receive the graph as an argument of the functions.

The actual implementation of our depth first search will take the form of a functor, that receives a module implementing G as its argument and contains a single function, which implements the search:

```
module Make( Graph : G ) = struct
    let dfs g f =
        let marked = Hashtbl.create (Graph.v g) in
        let st = Stack.create () in
```

[6] https://github.com/ocaml-gospel/cameleer/tree/master/examples
[7] https://github.com/dCastanho/algocameleer/tree/main/examples

```
        let mark e = Hashtbl.add marked e e in
        let addStack e = Stack.push e st in
        let aFunc e =
            addStack e ;
            while not (Stack.is_empty st) do
                let curr = Stack.pop st in
                if not (Hashtbl.mem marked curr) then begin
                    Graph.iterAdj g curr addStack ; f curr ; mark curr
                end done in
        Graph.iterAll g aFunc
end
```

4.2 Cameleer specification

During the explanations of our graph's code, we start by describing what it should do and then we present the logic annotations that guarantee that our code does what we described.

```
module type COMPARABLE = sig
  type t

  val equal : t -> t -> bool
  (*@ b = equal x y
        ensures b <-> x = y *)
end
```

This signature declares a type **t**, which will later be used as our vertices. It also provides a function for comparing two elements of this type, returning true if they are equal and false otherwise. Notice the comment starting with `@`: this is how specifications are declared with GOSPEL. The notation *ensures* defines a post-condition that any implementation of this signature must follow: that the result of the function is *logically* equivalent to equality of the two arguments.

With the elements defined, we now write the definition for our graph. To keep it simple, we implemented it with a functional map, this means, we store our vertices in the form of a function, which receives a vertex and returns the appropriate adjacency list. Given this way of storing vertices, we also need to somehow store the domain of our function, to know which vertices are valid.

```
module Concrete (V: COMPARABLE) = struct
  type vertex = V.t

  type t = {
    succ : vertex -> vertex list;
    dom  : vertex list;
  } (*@ invariant forall x. List.mem x dom -> forall y. List.mem y (succ x) ->
          List.mem y dom *)
```

We declare that our vertices are of type V.t, with V being a module that implements the signature defined before. We also declare our graph (type t) as a record with two fields: the functional map mentioned before and the domain, which takes the form of a list (though it is technically a set). Alongside our definition we declare a *type invariant*. This is a predicate that must always be true for any element of our type. In this case, for every vertex in the domain, all the vertices in its adjacency list must also belong to the domain.

```
  let empty =
    { succ = (fun _ -> []); dom = [] }
```

This is the starting point of any graph created with our module and is simple to understand. Our succ defines that any vertex it receives will return the empty list and our domain is also empty.

```
let rec mem_list v = function
    | [] -> false
    | x :: r -> V.equal v x || mem_list v r
(*@ b = mem_list l
         variant l
         ensures b <-> List.mem v l *)
```

Since we are manipulating lists, but using our own definition of equality for two members, it is a good idea to define a function like the one above to find whether a member of a list is equal to a provided v. Notice that despite being an auxiliary function, it must also be proved sound, since other functions make use of it. We have to define a *variant* so we can prove function termination. A variant's role is to ensure termination, so it points to something that decreases over the execution of our function, in this case, every time we recursively call `mem_list` we do so with a smaller list than before. The result of this function must be true if v is inside l and false if it is not, and this is represented through the postcondition. We can use `List.mem` here because we are in logic and not programming, where equality is not as strict.

```
let[@logic] mem_vertex g v = mem_list v g.dom
(*@ b = mem_vertex g v
        ensures b <-> List.mem v g.dom *)
```

This function receives a graph and a vertex and returns whether or not the vertex is a member of the graph. We do this by checking whether the given vertex is inside the given graph's domain. The `[@logic]` keyword means that this function can also be used inside specifications, to help in proofs. Functions with this notation are often programmed similarly to their specification. The result of the function must be equivalent to checking if the given vertex is in fact a member of the graph's domain.

```
let[@logic] mem_edge g x y = mem_vertex g x && mem_list y (g.succ x)
 (*@ b = mem_edge g x y
        ensures b <-> mem_vertex g x && List.mem y (g.succ x) *)
```

We also want to be able to check whether two vertices (x and y) are connected, in which case we check if x is a member of the graph and if y is one of its successors. We do not need to check if y is also a member of the graph, because if it is not, then we know `mem_list y (g.succ x)` is false. For this function to be correct, its result has to be equivalent to what was described before, the vertex must be a member of the graph and y must be a member of its successors list.

```
  let update g x v =
    (fun y -> if V.equal x y then v else g.succ y)
```

This function is how we update the graph's map. We return a new function that expects to receive a value y. If this value is equal to x, then we return the list v. If not then, we will ask the succ function of the graph we received.

```
let add_vertex g v =
    if mem_vertex g v then g
    else { dom = v :: g.dom; succ = update g v [] }
(*@ r = add_vertex g v
        ensures mem_vertex r v
        ensures forall x. List.mem x g.dom -> List.mem x r.dom *)
```

To add a vertex, we receive it and the graph we want to add it to. If the vertex is already a member, then we do not change anything, simply returning the graph as is. If it is not, then we return a new graph. Its domain is equal to the previous one, but with v added to it, and succ is updated to assign the empty list to v. The resulting graph must guarantee two things, that v is a member of it and that every vertex that existed before, is still part of its domain.

```
let add_edge g v1 v2 =
    if mem_edge g v1 v2 then g
    else
      let g = add_vertex g v1 in
      let g = add_vertex g v2 in
      { g with succ = update g v1 (v2 :: g.succ v1) }
(*@ r = add_edge g v1 v2
        ensures  mem_edge r v1 v2 *)
```

To add an edge, we receive a graph and two vertices. If the edge already exists, we simply return the graph without altering it. Otherwise, we add the vertices v1 and v2 (if they already exist nothing changes). Then, we return an altered version of g, updating succ to have v2 as a successor of v1. At the end of the function, we must have guaranteed that there is now an edge between v1 and v2.

5 Goals and Planning

In the figure below, we show a simple Gaant chart, containing the planning for our project splitting up the work in tasks which are spread out over weeks:

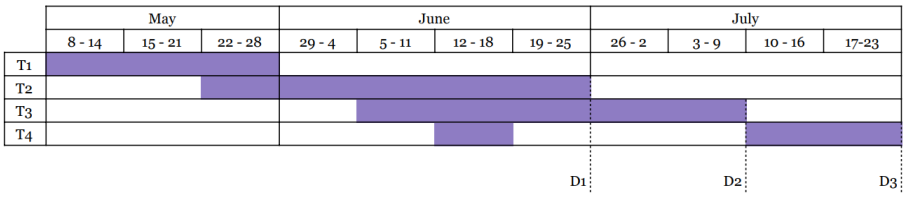

Fig. 1. Planning.

T1: Features Selection. The first task consists of selecting the algorithms and data structures from OCamlgraph which we will be proving in the next tasks. While we do this selection, we will be taking the chance to have a better understanding of how the library works.

T2: Specification. The second task is where we will start writing the specification of our selected algorithms and structures. We will begin with a relaxed specification, that is not supposed to actually run in Why3 or Cameleer but will be properly written before the end of the task. This is to facilitate finding all the necessary components before starting to actually write the specifications. There is a small overlap between T1 and T2 for choice corrections.

T3: Proof Effort. The third task is where we will actually begin proving our choices by using Cameleer and our specifications written in T2. Since specification and proof go hand in hand, it makes sense that there is a large overlap between the two, and corrections to the specifications may be made over the course of this task.

T4: Final Report. The last task is focused on writing the final report to deliver at the end of the project. The report will have an initial stage near the end of T2, where we will start justifying and showcasing our choices for T2 and T1.

Deliverables. There will be three main deliverables in the project, represented in our planning by D1, D2 and D3. **D1** will be a set of algorithms and their specifications. **D2** will be the respective proofs of those algorithms, and possibly the correction of some specifications. **D3** represents the final report, to be delivered at the end of the project.

6 Conclusion

Graphs are incredibly useful for representing and solving problems. Because of this, having a generic and versatile graph library like OCamlgraph is a huge asset to any programmer. Unfortunately, like all code, this library is susceptible to bugs, which can completely break any system dependent on it. To assure ourselves of its correctness, we turn to formal proof of programs and tools to help us, which brings us to Cameleer, a software made to prove OCaml code. With all of this in mind, we have decided to tackle this project, in which we aim to prove a subset of OCamlgraph, taking the opportunity to test Cameleer with already written code and complex algorithms and data structures.

References

1. 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.
2. 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.
3. F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer, 1999.
4. B. C. Pierce, *Types and programming languages*. MIT Press, 2002.
5. E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, eds., *Handbook of Model Checking*. Springer, 2018.
6. M. J. P. Pereira, *Tools and Techniques for the Verification of Modular Stateful Code. (Outils et techniques pour la vérification de programmes impératives modulaires)*. PhD thesis, University of Paris-Saclay, France, 2018.
7. C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969.
8. E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, no. 8, pp. 453–457, 1975.
9. J. Filliâtre and A. Paskevich, "Why3 - where programs meet provers," in *Proceedings of the 22nd European Symposium on Programming - Programming Languages and Systems* (M. Felleisen and P. Gardner, eds.), vol. 7792 of *Lecture Notes in Computer Science*, pp. 125–128, Springer, 2013.
10. 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.
11. M. Pereira and A. Ravara, "Cameleer: a deductive verification tool for ocaml," *32 ème Journées Francophones des Langages Applicatifs*, p. 21.
12. G. Melquiond and R. Rieu-Helft, "WhyMP, a formally verified arbitrary-precision integer library," in *Proceedings of the 45th International Symposium on Symbolic and Algebraic Computation*, pp. 352–359, 2020.
13. J. K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "Hacl*: A verified modern cryptographic library," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pp. 1789–1806, ACM, 2017.
14. A. Charguéraud, J.-C. Filliâtre, M. Pereira, and F. Pottier, "VOCAL – A Verified OCaml Library." ML Family Workshop, September 2017.

A Deph first search full code

```
module DFS = struct

    module type G = sig
        type t
        type elt
        val iterAll : t -> (elt  -> unit) -> unit
        val iterAdj : t -> elt  -> (elt -> unit) -> unit
        val v : t -> int
    end

    module Make( Graph : G ) = struct

        let dfs g f =
            let marked = Hashtbl.create (Graph.v g) in
            let st = Stack.create () in
            let mark e = Hashtbl.add marked e e in
            let addStack e = Stack.push e st in
            let aFunc e =
                addStack e ;
                while not (Stack.is_empty st) do
                let curr = Stack.pop st in
                if not (Hashtbl.mem marked curr) then begin
                    Graph.iterAdj g curr addStack ; f curr ; mark curr
                end done in
            Graph.iterAll g aFunc;;
    end
end
```

B Cameleer specification example full code

```
module type COMPARABLE = sig
  type t

  val equal : t -> t -> bool
  (*@ b = equal x y
        ensures b <-> x = y *)

end

(* Module Concrete implements:
   - a persistent graph
   - a directed graph (digraph)
   - an unlabeled graph *)
module Concrete (V: COMPARABLE) = struct

  type vertex = V.t

  type t = {
    succ : vertex -> vertex list;
    dom  : vertex list;
  } (*@ invariant forall x. List.mem x dom -> forall y. List.mem y (succ x) ->
         List.mem y dom *)

  let empty =
    { succ = (fun _ -> []); dom = [] }

  let update g x v =
    (fun y -> if V.equal x y then v else g.succ y)


  let rec mem_list v = function
    | [] -> false
    | x :: r -> V.equal v x || mem_list v r
   (*@ b = mem_list l
         variant l
         ensures b <-> List.mem v l *)


  let[@logic] mem_vertex g v = mem_list v g.dom
  (*@ b = mem_vertex g v
        ensures b <-> List.mem v g.dom *)

  let[@logic] mem_edge g x y =
    mem_vertex g x && mem_list y (g.succ x)
```

```
45    (*@ b = mem_edge g x y
46          ensures b <-> mem_vertex g x && List.mem y (g.succ x) *)
47
48    let add_vertex g v =
49      if mem_vertex g v then g
50      else { dom = v :: g.dom; succ = update g v [] }
51    (*@ r = add_vertex g v
52          ensures mem_vertex r v
53          ensures forall x. List.mem x g.dom -> List.mem x r.dom *)
54
55    let add_edge g v1 v2 =
56      if mem_edge g v1 v2 then g
57      else
58        let g = add_vertex g v1 in
59        let g = add_vertex g v2 in
60        { g with succ = update g v1 (v2 :: g.succ v1) }
61    (*@ r = add_edge g v1 v2
62          ensures  mem_edge r v1 v2
63          ensures  mem_vertex r v1 && mem_vertex r v2 *)
64
65  end
```