Verificación de algoritmos y estructuras de datos en Dafny

Verifying Algorithms and Data Structures in Dafny

Rubén Rafael Rubio Cuéllar

Doble Grado en Ingeniería Informática y Matemáticas
Departamento de Sistemas Informáticos y Computación
Facultad de Informática
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo de fin de grado

Madrid, 17 de junio de 2016

Directores:

Narciso Martí Oliet
Isabel Pita Andreu
Alberto Verdejo López

«When Apollo was pursuing the virgin Daphne, daughter of the river Peneus, she begged for protection from Earth, who received her, and changed her into a laurel tree. Apollo broke a branch from it and placed it on his head.»

— Fabulæ, Caius Julius Hyginus (64 a.C.–17)

The additional material, including the source code of the Dafny programs that have been developed, is available in the attached CD and in `https://github.com/ningit/vaed`.

TÍTULO del TFG: Verificación de algoritmos y estructuras de datos en Dafny

Curso académico: 2015 / 2016

Nombre del alumno:
Rubén Rafael Rubio Cuéllar

Tutores del TFG y departamento al que pertenece:
Narciso Martí Oliet
Isabel Pita Andreu
Alberto Verdejo López
(Departamento de Sistemas Informáticas y Computación)

Firma del alumno

Firma de los tutores

Contents

List of Figures

Resumen

La verificación formal de un programa es la demostración de que este funciona de acuerdo a una descripción del comportamiento esperado en toda posible ejecución. La especificación de lo deseado puede utilizar técnicas diversas y entrar en mayor o menor detalle, pero para ganarse el título de *formal* esta ha de ser matemáticamente rigurosa.

El estudio y ejercicio manual de alguna de esas técnicas forma parte del currículo común a los estudios de grado de la Facultad de Informática y del itinerario de Ciencias de la Computación de la Facultad de Ciencias Matemáticas de la Universidad Complutense de Madrid, como es el caso de la verificación con pre- y postcondiciones o lógica de Hoare.

En el presente trabajo se explora la automatización de estos métodos mediante el lenguaje y verificador Dafny, con el que se especifican y verifican algoritmos y estructuras de datos de diversa complejidad.

Dafny es un lenguaje de programación diseñado para integrar la especificación y permitir la verificación automática de sus programas, con la ayuda del programador y de un demostrador de teoremas en la sombra. Dafny es un proyecto en desarrollo activo aunque suficientemente maduro, que genera programas ejecutables.

Palabras claves: algoritmos, estructuras de datos, especificación, verificación automática, lógica de Hoare, Dafny

Abstract

The formal verification of a program is the proof that it works according to a description of its expected behaviour in any possible execution. The specification of what is desired can use different techniques and go into more or less detail, but to win the *formal* title it must be mathematically rigorous.

The study and manual exercise of some of those techniques is part of the common curriculum of the degree studies at the School of Computer Science and of the Computer Science itinerary at the School of Mathematics at the Universidad Complutense de Madrid, such as verification with pre- and postconditions or Hoare logic.

In the current work, the automation of those methods is explored through the language and verifier Dafny, with has been used to specify and verify some algorithms and data structures of diverse complexity.

Dafny is a programming language designed to integrate specification and allow automatic verification of its programs, with the help of the programmer and a theorem prover in the shade. Dafny is in active development but mature enough and it generates executable programs.

Keywords: algorithms, data structures, specification, automatic verification, Hoare logic, Dafny

Chapter 1

Introduction

It is not incredible to conceive that computer programmers or algorithm designers want their creations to operate as desired. From the ancient Greeks to the modern computer programmers, all of them are supposed to wish that their algorithms or programs do what they want without errors.

The usual way of taking care of this praiseworthy aim is to *be careful*. From the informal knowledge on mathematical facts, language constructs, library functions, ... the programmers hoard and with the help of diverse documentation, previous works and examples, they can design and build programs in the hope that these meet their purposes.

When a feature is advanced enough, some tests can be carried out. If the program works properly, the programmers will be invested of a (maybe misleading) confidence to go on. Otherwise, there is a bug they should find and correct, carefully reviewing the code or more likely debugging.

This program genesis is generally more than acceptable and requires a reasonable effort. It may convince us of its correct behaviour, but would other people trust? When the program's task is sensitive or it cannot be tested in action or when dealing with common reusable components, it is worthwhile to take the supplementary effort formal verification methods require.

During this double degree studies, some of them have been introduced in different subjects. In the School of Mathematics (*Facultad de Ciencias Matemáticas*) we have attended *Programming Theory* (*Teoría de la programación*) lessons, where language semantics and static program analysis were taught. In the School of Computer Science (*Facultad de Informática*), we have heard about formal techniques on *Concurrent Programming* (*Programación concurrente*), and in *Data Structures and Algorithms* (*Estructuras de datos y algoritmos*) we have been taught pre- and postcondition specification and verification.

This work will explore the automation of the last formalism above at the hand of the Dafny language and verifier, in which we will specify and verify some interesting algorithms and data structures.

1.1 The foundations

In 1967, Robert W. Floyd (1936–2001) published *"Assigning Meaning to Programs"* [Flo67] and stated that all imperative programs could be represented in flowcharts and their constructs could be viewed as predicate transformers.

Two years later, C.A.R. Hoare (1934–) published *"An Axiomatic Basis for Computer Programming"* [Hoa69] where he claims that "computer programming is an exact science" and proposes a formal system for reasoning about programs, which we know nowadays as *Hoare logic*.

The axiomatization attempt of computer arithmetic, which starts the article, is not as famous as Hoare triples

$$\{P\}\ Q\ \{R\}$$

which mean "if the assertion P is true before initiation of a program Q, then the assertion R will be true on its completion". P was called precondition and R postcondition and a set of rules in this form was

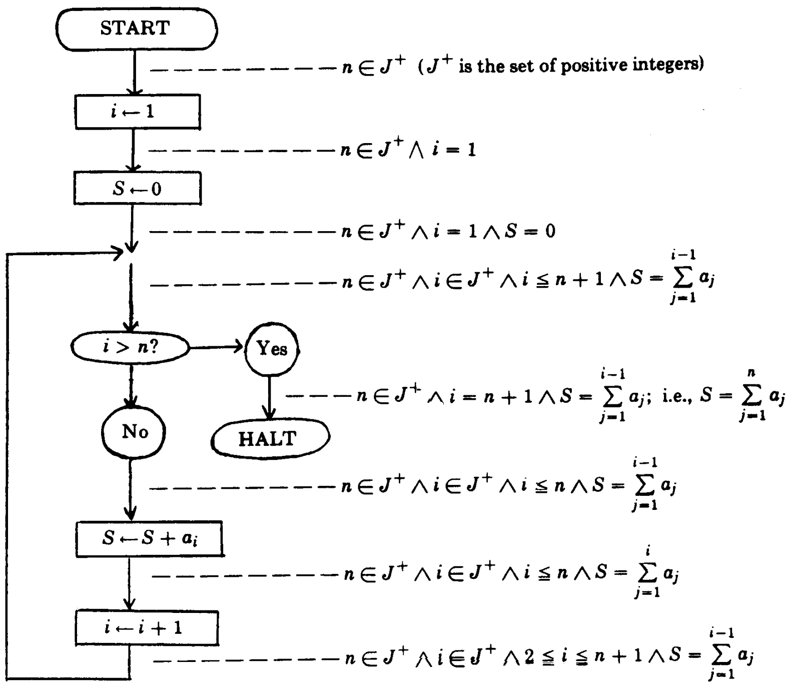

START
$n \in J^+$ (J^+ is the set of positive integers)
$i \leftarrow 1$
$n \in J^+ \wedge i = 1$
$S \leftarrow 0$
$n \in J^+ \wedge i = 1 \wedge S = 0$
$n \in J^+ \wedge i \in J^+ \wedge i \leq n+1 \wedge S = \sum_{j=1}^{i-1} a_j$
$i > n?$
Yes
$n \in J^+ \wedge i = n+1 \wedge S = \sum_{j=1}^{i-1} a_j$; i.e., $S = \sum_{j=1}^{n} a_j$
No
HALT
$n \in J^+ \wedge i \in J^+ \wedge i \leq n \wedge S = \sum_{j=1}^{i-1} a_j$
$S \leftarrow S + a_i$
$n \in J^+ \wedge i \in J^+ \wedge i \leq n \wedge S = \sum_{j=1}^{i} a_j$
$i \leftarrow i+1$
$n \in J^+ \wedge i \in J^+ \wedge 2 \leq i \leq n+1 \wedge S = \sum_{j=1}^{i-1} a_j$

Figure 1.1: "Flowchart of program to compute $S = \sum_{j=1}^{n} a_j$ $(n \geq 0)$" from [Flo67]

established for the basic constructs of a simple imperative language with assignments and loops. To the previous phrase "providing that the program successfully terminates" was appended later in the article because termination is not ensured by this method (*partial correctness*).

Following the abstraction and decomposition principles, this formalism let us reason about complex programs in a convenient and affordable way. It also allows choosing to which extent we want to talk about the program, which properties we want to consider.

Nevertheless, Hoare pointed out some limitations in the article: apart from the termination, real arithmetic, arrays, records, files, input/output, declarations, subroutines, recursion, parallel execution were not considered. He thought that dealing with these issues was not a difficult task and would be solved soon, and they have, usually with his contribution. He looked more pessimistic with respect to pointers and name parameters.

In further works like *"Proof of Correctness of Data Representations"* [Hoa72], Hoare set the bases for using abstract datatypes in his theory. Data can be represented as a mathematical object along with an abstract specification of its operations in those terms. Programs using data structures are proved correct according to their abstract specifications and the correctness of the concrete representations with respect to their specification is checked apart.

Hoare defines a representation function $\mathcal{A}$ to "map the concrete variables into the abstract object which they represent" and an invariant condition I to "place a constraint on the possible combinations of values the constituent concrete variables may take".

These concepts will be called *abstraction function* and *representation invariant* respectively from now on. We adopt this terminology and procedure in our work on data structures and algorithms which use them. Its adaptation to Dafny and our purposes, is described in Section 4.2.

1.2 Automatic program verification

As it was mentioned before, we will try the automation of the previously studied methods. We have just exercised them by hand in *ad-hoc* examples and they can and have been used manually in complex algorithms.

From the time Ada Lovelace designed and demonstrated her algorithm to calculate Bernoulli numbers for the hypothetical Charles Babbage's Analytical Engine [Men42], formal and informal correctness proofs have been written by hand. However, in the line traced by Leibniz, Hilbert, Ackermann, Church, and Turing, who wanted mathematical proofs to be checked by a machine for a greater trust, automatic tools were and are still being developed for program verification.

Like this, the correctness proof can be checked automatically and shipped as a guarantee jointly with the program, giving rise to what is called PCC (*proof-carrying code*). A wide range of tools are available with different goals and degrees of automation.

The hardware industry was one of the first to assimilate formal verification methods, which have been used for more than 20 years. The ACL2 system [KM16] was applied to the verification of critical hardware systems, which were modelled in the ACL2 language and then proved correct with the help of a theorem prover. In 1995 the correctness of the AMD K5 floating point division was demonstrated. Another tool called HOL Light [Har15] was used by Intel to formalize floating point arithmetic and verify several algorithms from their processors. On a slightly higher level, Microsoft uses symbolic execution in their *Static Driver Verifier* [Mic16a] to detect defects and design issues in drivers for the Windows operating system.

Another example is the *Astrée* analyser [Cou + 16] that was developed to prove the absence of runtime errors in C-written programs for real-time embedded software. It has been used in Airbus airplanes as well as in vehicles from the *European Space Agency*. The B Method [Cle16] was used to design and prove correct some parts of the automatic driver system for the line 14 of the Parisian underground, for instance.

The multiple techniques that can be ascribed to formal verification have their advantages and drawbacks. Searching potential null pointer exceptions is less ambitious than proving that there are not. And this is less than demonstrating that the program always terminates providing the correct result. However, the required effort increases from one to another.

Being humble, formal verification is not to rely on blindly. Just like the executable program code, the specification could be wrong due to errors caused by the human specifier. It also relies on the correctness of the verifier (human or machine), the compiler, the abstract machine if any and the real machine in any situation, which may include traversing the sky within a rocket. The convenient and inevitable abstraction also left some hollows that can give rise to errors in practice, albeit they may be treated and prevented apart.

In this project, we are interested in *functional correctness* and we will use *deductive verification* methods. A program is functionally correct if it really does what we have designed it for. Deductive verification is based on the application of proof rules like in the Hoare logic we have just described.

These techniques are quite powerful and expressive but they require a non-negligible amount of work. According to [BH14] it "is elusive for almost all application scenarios" and statistics show that annotations require five times more lines that the program code itself.

The availability of efficient satisfiability modulo theories (SMT) solvers has propitiated a rise in verification system performance and automation. However, when working with first-order logic, full automation could not be expected as a consequence of the negative answer to the *Entscheidungsproblem* (i.e. first-order logic is undecidable).

Verification systems like HOL [CCT16], Isabelle [CM16], Why3 [Bob + 16] or Dafny [Res16] make use of SMT solvers to operate. The differences between HOL/Isabelle and Why3/Dafny is in the degree of automation and so in the user role. Both Why3 and Dafny work as *Verification condition generators* (VCG): from the code they produce verification conditions for an automated theorem prover, which then tries to prove that they are satisfied.

The most similar to Dafny is the Why3 system. It provides a specification and programming language called WhyML which can be used to write code directly or as an intermediate language to translate code from C, Java or Ada. It is able to generate verification conditions for several SMT solvers. Why3 is a reimplementation of a former system called Why and it is developed since 2011 by the Toccata team at the Inria Saclay-Île-de-France research centre.

In the case of Dafny, the role of WhyML is played by Boogie2 and Dafny code is translated to this intermediate language. Then Boogie generates verification conditions for the Microsoft's prover Z3. Dafny will be described more in depth in Chapter 2.

1.3 Objectives

This project objective is to explore functional verification of programs within the Dafny language and verifier, and specify and verify some interesting algorithms and data structures using it.

The experience and the conclusions drawn during this process should serve to elaborate a convenient methodology to face specification and verification within Dafny.

1.4 Work plan

This project has been carried out between late July 2015 and early June 2016. The work has been arranged in two stages.

1. Foremost, some tutorials and articles on the verification tool were read in order to learn how to express, specify and verify algorithms with it. This learning was simultaneously put into practice through some classroom-like exercises from [MSV12], not without interest for themselves.

   Those exercises, up to 38 with alternative solutions, play with interesting properties of numbers and vectors. From greatest common divisor or sine approximation calculations to binary search, a variety of topics have been covered. The description and conclusion of the process are written in Chapter 3, along with the description of the most interesting examples.

2. In the second phase, we have specified and verified some well-known algorithms and data structures of a greater complexity.

   The implemented and verified data structures are stacks, lists and binary heaps. Two graph-related algorithms have been considered: Floyd-Warshall algorithm and Dijkstra's algorithm, as representatives respectively of dynamic programming and greedy techniques. Chapters 4 and 5 describe their implementation, specification and verification.

The first experiments with data structures began in late January and specification of renowned algorithms started in March with the Floyd-Warhsall algorithm. This report was started in mid-April.

Some auxiliary tools have been developed with the aim of making some tasks more comfortable. A brief description of those is in Section 2.5 and they can be found as part of the attached material as described in Appendix B.

Chapter 2

The Dafny system

Dafny [Res16] is a programming language for verification as well as its compiler and verifier. Created and maintained by members of the *Research in Software Engineering* (RiSE) group at Microsoft Research and coordinated by K. Rustan M. Leino, it includes built-in specification constructs and comes with a static verifier to validate the functional correctness of programs.

In its early days, Dafny was developed as part of another RiSE project called Boogie, an intermediate verification language, which is still Dafny's backend. In turn, Boogie was originally developed and distributed together with Spec#, a verification extension for C#. Their sources have been available since 2009 in a Microsoft code hosting website. In 2012 Dafny moved to a separate repository and the first standalone release was offered.

Dafny is in active development. From the beginning of our project, almost 420 commits have been pushed to the repository, more than 70 issues have been reported, 30 discussions have been opened in the repository forum and two new versions have been released. The code is licensed under the Microsoft Public Licence, considered a free software licence by the Free Software Foundation, although incompatible with GNU GPL.

According to its project page [Res16], Dafny is being used in teaching at some universities around the world and in the proof of complex algorithms' properties like the stability and correctness of natural mergesort [LL15]. It has also been used in some program verification competitions and benchmark challenges, like COST Verification Competition (Formal Verification of Object-Oriented Software) or VerifyThis, being the most used system in some of their editions.

As a language, Dafny is mainly imperative, sequential, strongly and statically typed with minimal type inference, generic, modular and somehow object-oriented. Specification is based on pre- and post-conditions, frame specifications and termination metrics. To support further specification the language provides recursive functions and suitable types like sets and sequences. Specification material is consumed in verification time, so that it is omitted from executable code.

Its official description admits being influenced by Euclid, Eiffel (built-in contracts), CLU (iterator and out parameter syntax), Java, C# and Scala (classes, traits and functions), ML (module system, functions and inductive datatypes), Coq and VeriFast (coinductive datatypes and inductive and coinductive proofs).

The axiomatic semantics principle harmoniously rules both the effective programming context and pure specification one, which in fact could be mixed at user discretion. Dafny programs are composed of two main top level constructs: methods and functions.

- Methods are imperative procedures with named parameters (passed by value) and named return values. There are a list of statements to produce an effect or calculation or to provide a proof. When we are only in the last case, they are called lemmas.
- Functions are expressions which admit recursive and mutually recursive calls. Actually they are functional programs, most likely used for specification.

Methods and functions which are not written to generate executable code and used only for specification or verification are named *ghost*. Functions are ghost by default. Methods, variables and also method or function parameters can be declared to be `ghost` with this keyword.

Other top level declarations include classes, iterators, subtype or type synonym definitions and the experimental async-task types. Its type system distinguishes value and reference types (as in C#) where reference types are pointers, whose referent is dynamically heap-allocated.

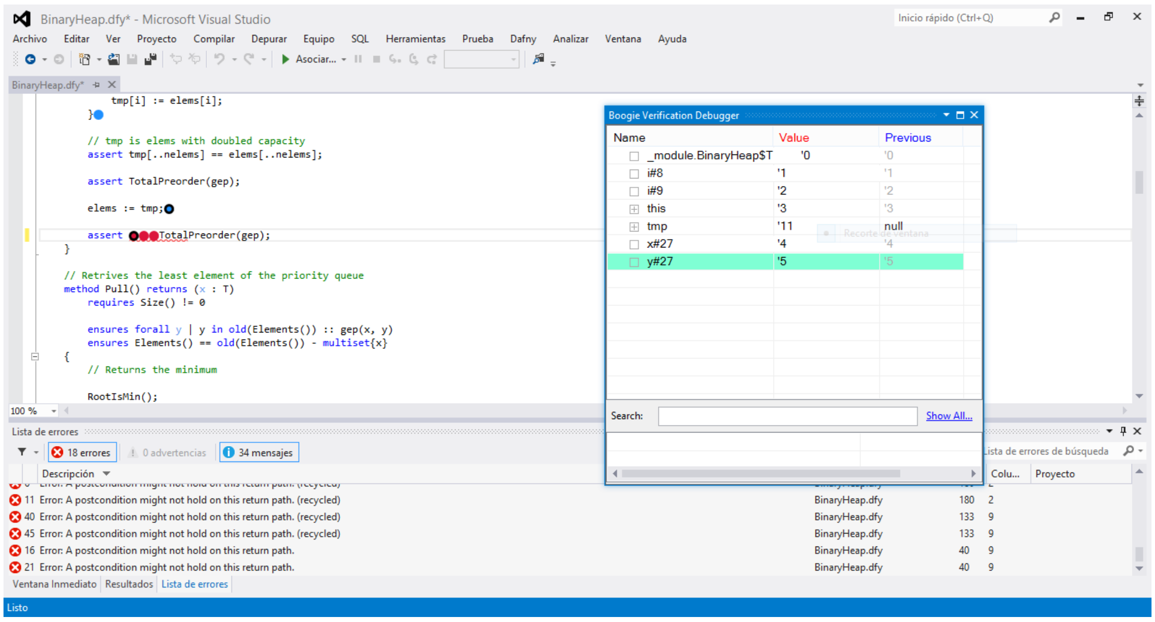
Built-in value types comprise integers (implemented as arbitrary-precision integers), natural numbers, characters, Booleans, and reals (implemented as quotients of arbitrary-precision integers). In addition, some other immutable types like sequences, sets, multisets, maps (both finite and infinite) are provided mainly to be used in specification context.

Reference types include type-parametric arrays of no matter what number of dimensions, and objects. Some object-oriented programming support is available. Classes don't allow inheritance or subclassing but there are *traits* for that purpose, borrowed from Self or Scala. This part of the language is still under construction and there are missing features like support for generic traits. Dafny has a particular way of dealing with dynamic memory locations and their influence in programs, called *dynamic frames*.

User interaction is not really supported by Dafny, apart from the `print` statement. Still and all, as Dafny generates .NET assemblies, the resulting programs, types and classes can be used from C# or any other language supporting the *Common Language Infrastructure*, but not necessarily in a clean and comfortable fashion.

Dafny can be used in several ways. The easiest one is the online interface at http://rise4fun.com/Dafny/ for which no installation is required. *rise4fun* hosts different software engineering tools from the RiSE group, including many other verification languages.

The recommended interface is the Dafny extension for Visual Studio, which provides syntax highlighting and live verification in the background as you write, powered by a caching system which restrains reverification to what is affected by the last changes. Errors are shown in place linked to their related locations. Tooltips appear when you hover over certain items, including information on automatically selected termination functions, triggers... It also integrates the Boogie Verification Debugger which can be sometimes used to see counterexamples.



```
            tmp[i] := elems[i];
        }

        // tmp is elems with doubled capacity
        assert tmp[..nelems] == elems[..nelems];

        assert TotalPreorder(gep);

        elems := tmp;

        assert TotalPreorder(gep);
    }

    // Retrives the least element of the priority queue
    method Pull() returns (x : T)
        requires Size() != 0

        ensures forall y | y in old(Elements()) :: gep(x, y)
        ensures Elements() == old(Elements()) - multiset{x}
    {
        // Returns the minimum

        RootIsMin();
```

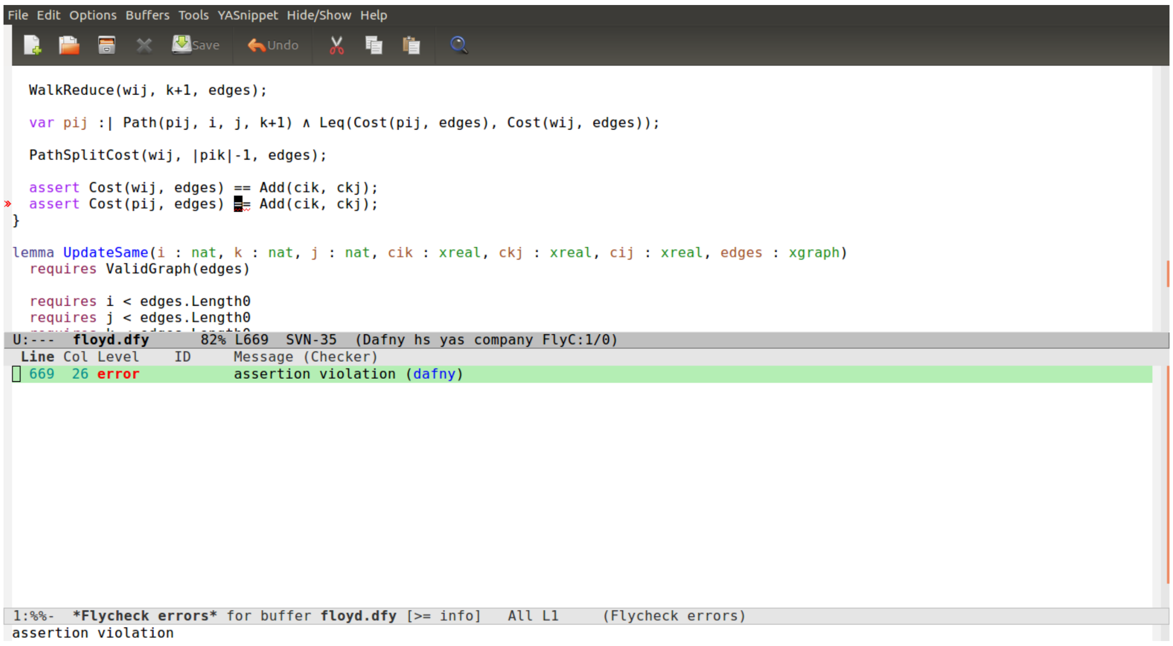
Name	Value	Previous
_module.BinaryHeap$T	'0	'0
i#8	'1	'1
i#9	'2	'2
this	'3	'3
tmp	'11	null
x#27	'4	'4
y#27	'5	'5

Descripción			Colu...	Proyecto
11 Error: A postcondition might not hold on this return path. (recycled)	BinaryHeap.dfy	180	2	
40 Error: A postcondition might not hold on this return path. (recycled)	BinaryHeap.dfy	133	9	
45 Error: A postcondition might not hold on this return path. (recycled)	BinaryHeap.dfy	133	9	
16 Error: A postcondition might not hold on this return path.	BinaryHeap.dfy	40	9	
21 Error: A postcondition might not hold on this return path.	BinaryHeap.dfy	40	9	

Figure 2.1: Visual Studio Dafny extension and Boogie Verification Debugger

Dafny is also available from the command line as a typical compiler. If the program gets past the validation phase, a .NET assembly will be generated: a dynamic library (`.dll`) or an executable (`.exe`) whenever the program includes a method called or designated as `Main`. Dafny personalization, by means of different parameters, is easier here.

Extensions for other editors like Sublime Text, Vim and Emacs exist. The Emacs' Dafny mode is officially proposed in Dafny's website too. Besides syntax highlighting, completion, code snippets... it offers on-the-fly verification. The last is possible thanks to the Dafny server, a tool which let external programs benefit from the verification result caching available in Visual Studio.



```
  WalkReduce(wij, k+1, edges);

  var pij :| Path(pij, i, j, k+1) ∧ Leq(Cost(pij, edges), Cost(wij, edges));

  PathSplitCost(wij, |pik|-1, edges);

  assert Cost(wij, edges) == Add(cik, ckj);
» assert Cost(pij, edges) == Add(cik, ckj);
}

lemma UpdateSame(i : nat, k : nat, j : nat, cik : xreal, ckj : xreal, cij : xreal, edges : xgraph)
  requires ValidGraph(edges)

  requires i < edges.Length0
  requires j < edges.Length0
U:---  floyd.dfy      82% L669  SVN-35  (Dafny hs yas company FlyC:1/0)
 Line Col Level     ID      Message (Checker)
 669  26 error              assertion violation (dafny)

1:%%-  *Flycheck errors* for buffer floyd.dfy [>= info]   All L1     (Flycheck errors)
assertion violation
```

Figure 2.2: Emacs Dafny mode

Dafny installation is quite easy in recent versions. To install the extension in Visual Studio, a simple click in the `.vsix` file that comes in the release package should be enough. Non Windows users need to install Mono and download the appropriate package from the Dafny project download page. Debian and Ubuntu packages will be available from the official repositories in the near future.

Dafny and Boogie are based on the .NET Framework and both are cross-platform. However, they depend on the Z3 SMT solver which is compiled to native code.

2.1 Brief description of the verification architecture

Dafny architecture is based on previous Microsoft Research's tools, namely Boogie, an intermediate verification language, and Z3, an SMT solver, although this dependency and collaboration is transparent to the user.

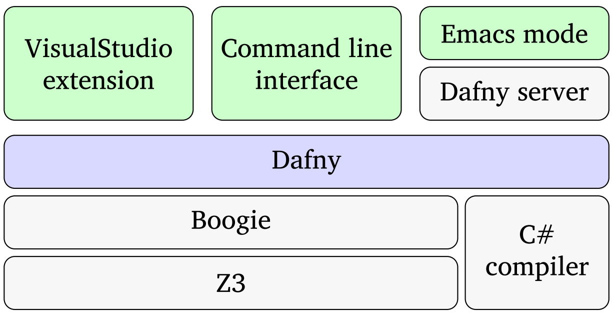


Figure 2.3: Dafny system architecture

First, Dafny performs the typical compiler work, it does parsing and checks types and identifiers. Errors are reported when found. Otherwise, Dafny continues to verification, by translating Dafny constructs into a Boogie program, including the necessary asserts and assumptions and Dafny's infrastructure from the `DafnyPrelude.bpl` file. The translation details can be read in [Lei08].

Then Boogie takes the responsibility to verify its code. Boogie generates several first-order verification conditions that the Z3 theorem prover will then check.

When all is done, Dafny generates C# code including the implementation of built-in data types and functions available in `DafnyRuntime.cs`. Later on, this program is compiled in a .NET assembly, a dynamic library or an executable depending on the presence of a method named or designated as `Main`.

During normal operation this process is not visible to the user. Some internal errors, which we have seldom suffered, like Boogie failures or errors in the generated C# code, may make lower layers arise.

2.2 Language syntax and semantics overview

The Dafny language is quite similar to the one used in the *Data Structures and Algorithms* lessons, as in [MSV12] and [Peñ06]. The procedures are equivalent to those we used by hand.

Dafny statements are those we can find in any usual imperative language, those which Hoare considered in his article [Hoa69]. Dafny has **`while`** statements and conditions (**`if`**, **`else if`** and **`else`**). Here we show simplified partial correctness rules for those constructs. They are simplified, as Dafny statements like **`break`** and **`return`** are not being contemplated. Actually, Dafny considers different *return paths*. In this rules, we also mix syntax and semantics but, as Dafny's Boolean expressions language for effective programming is a subset of the specification language, it is not serious.

$$\frac{\{B \wedge P\}\ \mathtt{S}\ \{P\}}{\{P\}\ \mathtt{while}\ \mathtt{B}\ \mathtt{invariant}\ \mathtt{P}\ \{\ \mathtt{S}\ \}\ \{P \wedge \neg B\}}$$

$$\frac{\{P\}\ \mathtt{S}_1\ \{Q\} \qquad \{Q\}\ \mathtt{S}_2\ \{R\}}{\{P\}\ \mathtt{S}_1\ ;\ \mathtt{S}_2\ \{R\}}$$

$$\frac{\{B \wedge P\}\ \mathtt{S}_1\ \{Q\} \qquad \{\neg B \wedge P\}\ \mathtt{S}_2\ \{Q\}}{\{P\}\ \mathtt{if}\ \mathtt{B}\ \{\ \mathtt{S}_1\ \}\ \mathtt{else}\ \{\ \mathtt{S}_2\ \}\ \{Q\}}$$

Dafny (or Boogie) communicates the errors to the user in a relatively pleasant way. For example, in the case of loops, the types of errors Dafny produces reminds me about the steps we are taught to follow to prove the correctness of an iterative algorithm; that is:

1. The invariant holds at the start of a loop.
2. The invariant holds after each execution of the loop body providing it and the loop condition hold before.
3. The invariant and the negation of the loop condition imply the desired property at the end of the loop.
4. The bound function is non-negative whenever the invariant and the loop condition hold.
5. The bound function decreases at every execution of the loop body whenever the invariant and the loop condition hold.

In fact, every loop, method or function to be called recursively needs a bound function in Dafny. It is usually able to provide one[1] but we may choose one explicitly using the **`decreases`** clause.

Assignments require special attention as they allow different variables to be assigned at the same time and in parallel. Moreover, a special form of assignment let the right-hand side expression be a single method call (with potential side effects) when the left-hand side should have as many variables as the method output parameter length.

[1]The default bound expression for (recursive) functions is the lexicographic tuple of its parameters. Formulas like **`if`** `n - m` $\geq$ `0` **`then`** `n - m` **`else`** `n - m` are used for loops whose condition looks like `n` $\neq$ `m`.

In any function or method call, we must ensure that its preconditions hold and that the callee is allowed to read or write what the function or method declares to read or write. Thereafter we can assume its postconditions.

`assert` and `assume` are other important statements. Both take a Boolean expression and cause the verifier to assume it afterwards. The `assert` statement also tries to prove it, showing an error if this is not possible. Assert and assume statements as well as lemma calls can appear in expressions, as a prefix separated with a semicolon. For example `var y := assume x > 0; 1 / x;` is a valid Dafny term.

2.3 Documentation and reference resources

Dafny's main references are its web page at Microsoft Research [Res16] and the project page at the Microsoft's code hosting website CodePlex `http://dafny.codeplex.com`.

My first reading on Dafny was a 3-page tutorial-like article called *"Developing Verified Programs with Dafny"* [Lei13]. Despite its small size it explains most of the Dafny we have dealt with. Other tutorials –more or less updated– can be found at the pages mentioned in the previous paragraph as well as in the bibliography. Leino's publications web page includes links to many articles, from tutorials to feature devoted monographs. A guide is also available inside the *rise4fun* website and also video tutorials are conducted by Leino in the *Verification Corner* YouTube channel.

To solve particular doubts about the language, there are two interesting and easy-reading references in *Dafny Quick Reference* [Mic16b] and *Types in Dafny* [Lei15]. A detailed language (draft) reference [FL16], hidden deep in the repository, will solve any further doubt.

Another valuable source of information is the *Formal methods for software development* (*Métodos formales de desarrollo de software*) course page [Luc16], taught by Paqui Lucio at the University of the Basque Country (Universidad del País Vasco/Euskal Herriko Unibertsitatea).

In CodePlex, apart from the Mercurial repository that contains the Dafny code, the project site offers a discussion forum and an issue tracking system open for contribution to the public. The repository forum has been useful to answer some questions we have come across. Here is the list of threads we created:

1. *Function type attributes* (654197)
2. *Express relations and their properties* (653715)
3. *Generic traits* (651246)
4. *Instability of verifications* (648164)

A Dafny tag exists in Stack Overflow too.

Some errors have been reported to the Dafny authors through the issue tracking system, in the hope that they were solved.

1. *Wrong z3.exe in .vsix* (105)
2. *Verification result depends on declaration order* (152)
3. *Bug in BigRational implementation* (148)
4. *Something wrong in release notes* (163)

Only the third issue has been closed and the problem was solved. None of the others has been considered for the moment (perhaps partially the first).

Comments on some reported issues

Let's make a parenthesis to comment two of those issues, not in order to rub in the failures but to emphasize and illustrate how the correctness of the final program relies not only in the user code itself, but in the validity of the verification tool, the code generator, the compiler, etc.

The third issue, *Bug in BigRational implementation*, was a bug in the implementation of the comparison method for the `BigRational` class, which is the base for the `real` datatype. As a consequence generated programs with real number comparisons generate bad results, against the specification. That was the case of the Floyd-Warshall algorithm described in Section 5.1. Investigating the causes, we arrived at the Dafny runtime sources where the bug was found.

The first issue *Wrong z3.exe in .vsix* is also related with this topic: the Z3 executable included within the Visual Studio extension installer (`.vsix`) differs from that distributed with the command line version, despite being shipped in the same compressed packet. So Dafny produces different verification results whether using Visual Studio or the command line interface (there might be other causes yet).

The concern which made us look carefully at the Z3 version was a curious story. The first versions of Dafny until 1.9.6 are offered as a downloadable packet which contains the Dafny binaries and the Z3 binary for Windows only. As Z3, required by the Dafny system and written in C++, is a native executable, we ourselves had to provide it in order to use Dafny in Linux. We downloaded Z3 from the Debian official repositories and we worked with it for months. Someday, Dafny arrived to prove something false. The reason was that the Z3 version was too old or buggy. We downloaded Z3 from its original repository at GitHub and the issue was solved.

2.4 Dafny configuration and options used in this project

The Dafny tool has a bunch of options which influence its verification results. Since we obtained different outcome for the same file whether we were using Visual Studio or the command line interface, we realised the need to find a common criterion to say whether the verification of a program has been successfully achieved.

As Visual Studio options are more difficult to change[2] and while they seem reasonable, we will adapt to them. At the present time, only a timeout of 10 seconds should explicitly be fixed at the command line to follow Visual Studio convention. Its extension uses another option to fix the number of prover threads to one less the total number in the machine, but we have omitted it as it seems to be a source of instability (see Section 2.3, thread *Instability of verifications*).

To sum up, the general criterion to admit that a program verification has been successful is that Dafny 1.9.7 shows no errors for the file at the command line when called with options

```
/compile:0 /timeLimit:10 /noCheating:1
```

where `/compile:0` omits the compilation phase (it has no effect on verification) and `/noCheating:1` treats `assume` as `assert`.

Most of the time, we have used Dafny from the command line and edited `.dfy` files within a text editor in a GNU/Linux environment. Visual Studio extension, Emacs Dafny mode and *rise4fun* have been used sometimes.

2.5 Auxiliary tools we have developed

Some programs have been built as auxiliary tools in the hope that they would be helpful for the project development.

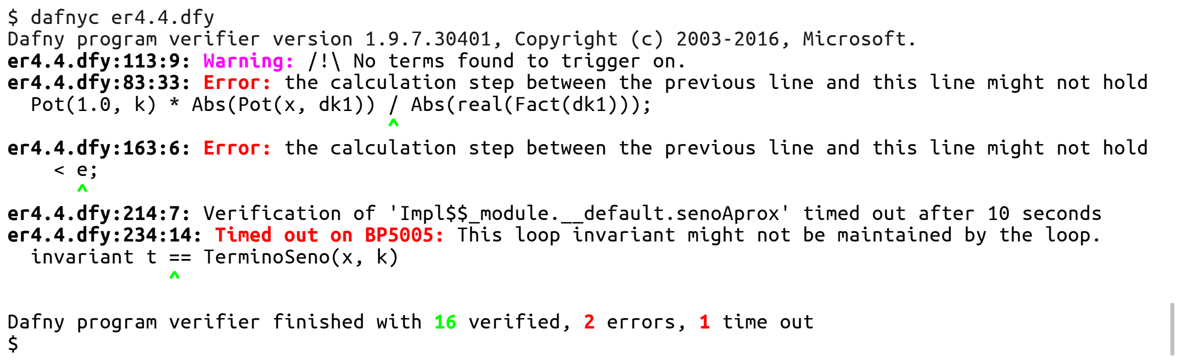
Vaed *Vaed* is a program to batch verify and keep track of the set of all Dafny files from the project. It keeps a register of the previous verification results and only reverifies those files which have changed since the last execution, like a *make* utility. The tool is able to generate a `.pdf` listing all Dafny programs along with their verification outcome. These can also be reviewed through a web interface, from where Dafny programs can be tested as a modest *rise4fun*.

The program can serve as a quick check for a third person to ensure that all programs written are syntactically correct and verified, and it has revealed to be very useful on every Dafny update, which usually make some programs stop working.

[2]There is a `DafnyOptions.txt` file in the extension folder where parameters can be set as in the command line.

This program was first written in Python 3 and rewritten in C# to reduce dependencies.

dafny.pl As the Dafny command line verifier output is not as comfortable as desired, a Perl script called `dafnyc.pl` was written to reformat and colourize its output, as well as to include the cited code line in it, following the style of modern GCC compilers up to a point, as shown in Figure 2.4.



```
$ dafnyc er4.4.dfy
Dafny program verifier version 1.9.7.30401, Copyright (c) 2003-2016, Microsoft.
er4.4.dfy:113:9: Warning: /!\ No terms found to trigger on.
er4.4.dfy:83:33: Error: the calculation step between the previous line and this line might not hold
  Pot(1.0, k) * Abs(Pot(x, dk1)) / Abs(real(Fact(dk1)));
                                 ^

er4.4.dfy:163:6: Error: the calculation step between the previous line and this line might not hold
    < e;
      ^

er4.4.dfy:214:7: Verification of 'Impl$$_module.__default.senoAprox' timed out after 10 seconds
er4.4.dfy:234:14: Timed out on BP5005: This loop invariant might not be maintained by the loop.
  invariant t == TerminoSeno(x, k)
              ^


Dafny program verifier finished with 16 verified, 2 errors, 1 time out
$
```

Figure 2.4: Dafny console beautifier

partes.pl `partes.pl` is another Perl script that was programmed to separately verify all the methods and functions of a Dafny file, using the `/proc:` option. In many situations, we have found programs that get verified in this way while they do not taking the file as a whole.

Other auxiliary works like syntax highlighting specifications and Debian packages were created to facilitate the use of the tool in a GNU/Linux environment.

All of them are included in the attached CD as well as in `https://github.com/ningit/vaed`. Their contents are described in Appendix B.

Chapter 3

Verifying iterative and recursive programs

In the first phase of this project and to get familiar with Dafny, up to 38 exercises on algorithm verification and derivation from the textbook "*Algoritmos correctos y eficientes*" (Correct and Efficient Algorithms) [MSV12], coauthored by two of my supervisors, have been written and proved in Dafny. The book contains mainly solved exercises along with some proposed unsolved ones. They come from three book's chapters

(2) Algorithm verification

(4) Iterative programs derivation

(5) Recursive programs derivation

Most exercises from these chapters have been considered except those which are too similar to the previously selected ones. Only three unsolved exercises have been included. Alternative solutions and Dafny features experiments have been carried out with the solved ones.

Algorithms deal with numbers or arrays. They calculate integer square roots, greatest common divisors, powers, logarithms... or they work with arrays to compute maxima, sums, dot products, binary search...

Seven exercises have been proved without any help, not a single assertion, invariant, lemma or termination bound specification; and eight more only need their `invariant`s or `decreases` clauses. Apart from that, the rest of the exercises require some help to the prover, with a varied degree of difficulty. If we have to put a star on the most complicated exercises, we would place it in Exercises 2.11, 4.4, 4.15, 4.27 and 5.18.

3.1 Initial difficulties

Since the Dafny language is quite similar to the pseudocode used in [MSV12] and that of *Data Structures and Algorithm*'s lessons at the School of Computer Science, code translation is almost direct. There are `int` for integer numbers, `nat` for non-negative numbers, `real` for real numbers and `bool` for Booleans. There are also `array`s but here we found more differences.

Dafny's arrays are type parametric, their indexes start from 0 to the length of the array (as in C), which can be retrieved by the `.Length` property and it is fixed from the time they are created. Be aware that `array` type is a reference type, i.e. a pointer to a heap allocated region on which the array elements are stored (like in Java or C#), and so array variables need to be initialized (with `new T[size]` syntax) and can be `null`. Consequently, we have the responsibility to require that any array parameter given to a method is properly allocated, unless a special meaning is reserved to the null array. There are plenty of exercises dealing with arrays in this collection, where `v` $\neq$ `null` is a boilerplate precondition.

These mundane particularities of arrays justifies the use of alternatives in the specification context. To emphasize array contents and elude the gory details, we have the more abstract and handy sequences, a parametrized type called `seq`, which stands for the mathematical object A^* where A is the set of all values of a given type. Sequences are a value type (not a pointer) and immutable. The Dafny documentation recommends its use as the specification arrays counterpart, and so it provides some language constructs to convert arrays or portions of arrays to sequences. They are called slices (and old invention that exists from Algol 68 and persists nowadays in Python) and written `[l..u]` where `l` is the lower index to be included in the sequence and `u` is the upper bound, which is not included. Both ends can be omitted. Slices can be applied to arrays producing sequences or to sequences themselves. They allow a convenient sequence manipulation.

We have not found many differences for now between Dafny and our previous knowledge regarding specification and verification. Their syntax and principles should be familiar to us, assuming we have studied manual pre/post specification or axiomatic semantics. Nevertheless, some differences appear soon concerning arrays and some quantifiers related to them.

To illustrate what we have just discussed, let us see Exercise 2.7 as an example. We are asked to verify an algorithm which sums all elements in an integer array. We are given the code and the following loop invariant $INV \equiv 0 \leq n \leq N \wedge x = (\sum i : n \leq i < N : V[i])$:

```
{ $N \geq 0$ }
func sum(V[0..N) of int) returns x : int
        var n : nat := N
        x := 0
        while n != 0 do
                x := x + V[n-1]
                n := n - 1
        endwhile
endfunc
{ $x = \sum i : 0 \leq i < N : V[i]$ }
```

Code translation is direct, taking care of what has been said before. But the postcondition cannot be easily expressed in Dafny. The language does not have any *sum quantifier* available, so we have to define it ourselves.

er2.7.dfy

```
function Sum(v : seq<int>) : int {
        if v = [] then 0 else v[0] + Sum(v[1..])
}

method sum(v : array<int>) returns (x : int)
        requires v ≠ null
        ensures x = Sum(v[..])
{
        var n := v.Length;

        x := 0;

        while n ≠ 0
                invariant 0 ≤ n ≤ v.Length
                invariant x = Sum(v[n..])
        {
                x, n := x + v[n-1], n - 1;
        }
}
```

We have defined sum by means of a recursive function that assigns zero, the identity element for the sum, to the empty list and otherwise decomposes the sum in the first element plus the rest. There are several other definitions we might have chosen. For instance, we can start the sum from right to left.

At this point, some philosophical question may arise. We want to say that the algorithm calculates "the sum of all elements in the sequence", but we should express it formally. The only way to specify it is with a function like that, which is a completely executable functional program. So let's use Haskell or any other functional language and forget about verification, because its programs will be correct by definition.

```
sum :: [Integer] -> Integer
sum []         = 0
sum (v:vs)     = v + sum vs
```

To make this definition more abstract we can derive properties from it. In fact, we will be obliged to do so.

Back to verification, note that the given recursive definition of sum works similarly to the iterative loop. Both of them refer to suffixes of the sequence, `v[n..]`. The previous program verifies without any further help. But, what happens if we change the loop index direction to sum elements from lower to higher indices? That is what we are required to do in Exercise 2.8, and Dafny is not able to prove the result without our help.

```
method sum'(v : array<int>) returns (x : int)
        requires v ≠ null
        ensures x = Sum(v[..])
{
        var n := 0;

        x := 0;

        while n ≠ v.Length
                invariant 0 ≤ n ≤ v.Length
                invariant x = Sum(v[..n])
        {
                x, n := x + v[n], n + 1;
        }
}
```

Dafny is not able to demonstrate that `Sum(v[..n]) + v[n] = Sum(v[..n+1])`. The human should provide the proof. This was our first lemma. `lemma` is a synonym for `ghost method`, i.e. a method only for verification.

er2.8.dfy

```
lemma LeftSumLemma(s : seq<int>, m : int)
        requires 0 ≤ m < |s|

        ensures Sum(s[..m]) + s[m] = Sum(s[..m+1])
{
        if m > 0 {
                LeftSumLemma(s[1..], m-1);

                // Induction hypothesis (only for clarity)
                assert Sum(s[1..][..m-1]) + s[1..][m-1] = Sum(s[1..][..m]);

                // Unavoidable assert
                assert s[1..][..m-1] = s[1..m];
                assert s[1..][..m] = s[1..m+1];

                // s[1..][m-1] = s[m] is deduced without help

                // We finally arrive  (can be omitted)
                assert Sum(s[1..m]) + s[m] = Sum(s[1..m+1]);
                // And the Sum definition does the rest
        }
}

method sum'(v : array<int>) returns (x : int)
        requires v ≠ null
        ensures x = Sum(v[..])
{
        var n := 0;

        x := 0;

        while n ≠ v.Length
                invariant 0 ≤ n ≤ v.Length
                invariant x = Sum(v[..n])
        {
                x, n := x + v[n], n + 1;

                LeftSumLemma(v[..], n-1);
        }

        // Dafny also needs help with that
        assert v[..v.Length] = v[..];
}
```

We can also derive other properties for the sum like the following which ensures that it is independent on the order of the elements in the array. It gets verified without human help, even if it looks harder than `LeftSumLemma`.

```
lemma SumElems(s : seq<int>, r : seq<int>)
        requires multiset{s} = multiset{r}
        ensures Sum(s) = Sum(r)
{
}
```

Another feature we miss in Dafny is the ability to give names to parameters which may vary, sometimes known as *logical* variables in the literature. For example take Exercise 2.14 where we are asked to verify an algorithm to *positivize* a vector, i.e. to change its negative values into zero. It is presented as:

```
$\{ N \geq 0 \land v = V \}$
**proc** positivize(v[0, N) **of** *int*)
    **var** j : *nat* := 0;
    **while** j < N **do**
        **if** v[ j ] < 0 **do**
            v[ j ] := 0
        **endif**
        j := j + 1
    **endwhile**
**endproc**
$\{ \forall i : 0 \leq i < N : (V[i] < 0 \Rightarrow v[i] = 0) \land (V[i] \geq 0 \Rightarrow v[i] = V[i]) \}$
```

In the precondition, $v = V$ means that the fresh variable V refers to the current value of v at that time (for this pseudocode, arrays are not pointers, the array value includes all of its elements). Then we can use V to compare and describe the changes the algorithm has made to v.

In Dafny, we can store an intermediate value of any variable simply by defining a ghost variable for it, like in `ghost var x0 := x;`. For parameters we ought to use a language primitive called `old`.

From [FL16] we learn that “an *old* expression is used in postconditions. `old(e)` evaluates to the value expression `e` had on entry to the current method”.

Then we can write the `positivize` algorithm in Dafny:

er2.14.dfy

```
method positivize(v : array<int>)
        requires v ≠ null
        ensures Positivized(v[..], old(v[..]))

        modifies v
{
        var j := 0;

        while j < v.Length
                invariant 0 ≤ j ≤ v.Length

                invariant Positivized(v[..j], old(v[..j]))
                invariant v[j..] = old(v[j..])
        {
                if v[j] < 0 { v[j] := 0; }

                j := j + 1;
        }
}
```

Note that we have included a `modifies` clause to allow the method to modify the heap-allocated array `v`. Also note that we have used `old` not only in the postcondition but also in the loop invariants to ensure that we have a prefix of the vector positivized while we maintain the rest unchanged. The postcondition is `Positivized(v[..], old(v[..]))` (you can read the attached file to see its definition), i.e. the sequence of elements at the end is the positivized version of the sequence of elements of the original array.

Here error risk is important. We may have written (and I have) `old(v)[..]` instead of `old(v[..])` (changing accordingly the loop invariants) and Dafny will give us its blessing too. But we are not saying what we pretend to say. Arrays are reference types, pointers, so `old(v) = v` because the program has never changed `v` itself which still points to the same memory address (in fact, method’s parameters are read-only in Dafny). As a result we are ensuring tautologies while vectors contents may be unleashed.

To end with this section some words should be said on existential and universal quantifiers. Dafny (or its children) are intelligent enough to use or to prove quantified properties in the common case when they are required. Another issue is how to prove yourself a quantifiers formula when Dafny alone is not enough.

Any method or lemma is implicitly a forall sentence. Its parameters are universally quantified under the constraint of the relations established by the preconditions; and for all of them the postconditions must hold. Dafny has a special block syntax to prove forall assertions, an almost undocumented feature, the `forall` statement, which has been used in Solved Exercises 2.11, 4.15 and Proposed 5.9.

Existential quantifiers can be proved in different ways: in a constructive manner, showing a variable (or collection of variables) who make its thesis true; negating them and so converting it to a universal quantifier. An interesting existential proof is described in depth in Section 3.4.

3.2 Other features worth mentioning

3.2.1 Calculations

In Exercise 5.7 we are told to write different recursive algorithms to calculate the dot product of two vectors. As usual, it is necessary to define the dot product as a function to be used in the specification.

er5.7.dfy

```
function DotProduct(v : seq<int>, w : seq<int>) : int
        requires |v| = |w|
{
        if v = []
        then 0
        else v[0] * w[0] + DotProduct(v[1..], w[1..])
}
```

As in many other cases, this definition proceeds recursively from left to right but one of the proposed algorithms does not follow this direction. So it is necessary to prove that we can start multiplying from the right end, for which we use a lemma, mainly identical to that which has been used to solve the same problem for the sum of a vector of integers.

The lemma was done (copy-pasted and adapted) and Dafny confirmed its validity. However small and innocent changes to the code or environmental conditions where the Dafny verifier execution takes place changed the verification result to a timeout. Then we decided to ask in Dafny forum why that happens and how this simple lemma could be written to take less time. A new thread called *Instability of verifications* (648164) was created.

As an anecdote, the initial code which gets verified in some cases with that version of Dafny, does not get verified at all with the current version. But, without timeout, the Spanish version gets verified in 14.596 seconds and when we translate identifiers to English it takes 40.612 seconds.

As a solution, we were proposed to use calculations (i.e. the `calc` syntax), a special language construct to establish a relation between two expressions through a stepwise transformation where each step can be justified separately. This style is more similar to that of mathematical proofs and makes verifiers' life easier as it restricts the effect of each step proof to the step itself, reducing the number of unneeded hypothesis the verifier may get confused with.

This syntax has been used widely in the following work, because despite of being more verbose it is clearer in case of doubt and reduces significantly the verification time. It is described in detail in *"Verified Calculations"* [LP13].

The final dot product lemma looks like this:

```
lemma DotProductLemma(v : seq<int>, w : seq<int>)
        requires |v| = |w|
        requires v ≠ []

        ensures DotProduct(v, w) = DotProduct(v[..|v|-1], w[..|v|-1])
                                        + v[|v|-1] * w[|v|-1]
{
        if (|v| = 1) {
                // Base case
        }
        else {
                // Previous in the order
                var vr, wr := v[1..], w[1..];

                calc = {
                        DotProduct(vr, wr);

                        // Induction hypothesis
                        { DotProductLemma(vr, wr); }
                        DotProduct(vr[..|vr|-1], wr[..|wr|-1])
                                        + vr[|vr|-1] * wr[|wr|-1];

                        // Subsequence simplification
                        { assert vr[..|vr|-1] = v[1..|v|-1];
                          assert wr[..|wr|-1] = w[1..|v|-1]; }
                        DotProduct(v[1..|v|-1], w[1..|v|-1])
                                        + vr[|vr|-1] * wr[|wr|-1];
```

```
                                // Subsequence elements simplification
                                { assert vr[|vr|-1] == v[|v|-1];
                                  assert wr[|vr|-1] == w[|v|-1]; }
                                DotProduct(v[1..|v|-1], w[1..|v|-1])
                                                + v[|v|-1] * w[|v|-1];
                        }

                        // Only remains to add the first coordinate in each term
                }
        }
```

3.2.2 Reductio ad absurdum

Reductio ad absurdum is available in Dafny. If we want to prove a Boolean expression P, we can translate "suppose that P does not hold then we arrived to contradiction" by

```
if ¬ P
{
        // [...]

        assert false;
}
```

At the end of the conditional, P will be true for sure if we are able to prove that denying it leads to a contradiction, i.e. if we can derive a contradiction inside the conditional branch. Then `assert false;` is redundant.

Proof by *reductio ad absurdum* is usually elegant and convenient for the way Dafny (or its auxiliary programs) verifies. This technique is used in Exercise 4.15 and 4.27 in the context of other proofs or methods.

3.2.3 Incremental program and proof building

Dafny allows the user to construct proofs and programs leaving blanks to be filled later. Apart from the `assume` statement, there are other mechanisms to postpone the proof (or the programming) of certain fact (or effect) but take advantage of its results meanwhile. Function, lemma, method, and loop bodies can be omitted. They will not serve to generate compiled programs, obviously, but this can be useful while a program or proof is still being written.

These will let us take an incremental approach to verification, thanks to dynamic frames and the black box model in which these components (methods, lemmas, loops, ...) are considered in the axiomatic semantics. However, reality is harder and there are interdependencies that make this method frustrating: what is established once does not remain stable when new methods are added or any independent body is changed, as experience has shown.

There are other means to focus Dafny efforts to a particular method we are working on. In Visual Studio or Emacs IDEs, and thanks to result caching, changes will only trigger the reverification of what is affected by them. But when this effect is desired between sessions or from command line, some active alternatives are at our disposal:

- Placing the attribute `{:verify false}` on each top level construct we don't want to verify for the moment.
- Putting the lemma or lemmas we are working on in a separate file and `include` the original file. Inclusion brings top level declarations available in the separate file without verifying them.
- Using command line options, passing the argument `/proc:'*<name>'` where `<name>` should be replaced by the name of the method (or any regular expression).

3.2.4 Matching triggers

In the most complex exercises from this part and in the following work, we have experienced what [LP16] names as *butterfly effect*. When using SMT-solvers often "a minor modification in one part of the program source causes changes in the outcome of the verification in other, unchanged and unrelated parts of the program", causing "verification instability" and "user frustration".

There are different reasons behind and one of them is the way the solver tackles quantifier instantiations.

At any time during the search proof, the SMT solver has a set of quantified formulas at its disposal, to which it may resort to complete the proof. In order to make use of the universally quantified formulas[1] the solver instantiates them to concrete values. This instantiation should be appropriate, avoiding instantiations that, far from making the proof easier, hinder or delay it. To that purpose *matching patterns*, also known as *matching triggers* or simply *triggers*, are used.

When the SMT solver finds a term in its state that matches any of those patterns associated with a quantified formula, it triggers its appropriate instantiation.

Oversimplifying, suppose that our prover state is $\{\ \forall x \quad f(x) = g(x) + 1, \quad g(3) = 0\ \}$. It may select $\{f(x)\}$ and $\{g(x)\}$ as triggers for the first formula. If we want to prove that $f(3) = 1$, our SMT solver will recognise $f(x)$ with $x = 3$ in the new term and will instantiate the quantified formula to give $f(3) = g(3) + 1$. Finally, it will conclude that $f(3) = 1$.

However, this technique has some inconveniences. Sometimes a formula or a set of them gets instantiated in a *matching loop* forever, and sometimes the selected triggers are based on lower level functions added during the translation phase which do not work properly.

In the recent versions of Dafny, the trigger selection has been moved from Z3 to Dafny, where some heuristics are carried out to avoid or mitigate the previously described issues. The change also allows the user to inspect the selected triggers. This new feature called *autotriggers* is the more noticeable change we have experimented during this project development. It can be disabled with `/autoTriggers:0` and custom triggers can be selected with an attribute like `{:trigger a[i]}` after the bound variable enumeration of the quantifier, but in general this is not necessary.

For instance, the *ordered* predicate for a sequence appears in many exercises from the book. We have tried different alternatives:

```
predicate Ord(s : int)
{
        (1) ∀ i | 0 ≤ i < |s|-1  •  v[i] ≤ v[i+1]
        (2) ∀ i | 0 < i < |s|  •  v[i-1] ≤ v[i]
        (3) ∀ i, j | 0 ≤ i ≤ j < |s|  •  v[i] ≤ v[j]
}
```

The first definition was proposed in the book, while the second is quite similar. Dafny complains about these because it finds that they may provoke matching loops since the trigger { `v[i]` } matches both with `v[i]` and `v[i+1]`. The third definition is the recommended one.

Any user should find learning something about triggers and bearing them in mind appropriate, because a correct trigger choice can make proofs easier, more efficient and less sensitive to changes in the rest of the code or to Dafny updates.

After having problems with complex expressions in quantifiers, we have drawn the lesson that defining predicates to confine those, instead of placing them inline in the quantifiers' statement, is quite useful in many situations. In particular, arithmetic expressions do not usually work well, as no triggers will be generated for them or for terms involving them.

[1]The existentially quantified formulas are skolemized, so converted to universally quantified.

3.2.5 Functional types and generics

Generics have been used in Exercises 4.7, 4.26, 5.14 and 5.17 (whose solutions differ from those of the book). Functional types have been used in conjunction with them in Exercise 4.7. The exercise statement asks to count the number of even elements in a given array. There is no need to use functional types but a great opportunity to experiment with them as the condition of the elements to be counted, that they are even, can be replaced by any other predicate on elements.

We can write a generic higher-order function like the following for specification and a generic parametrized algorithm for execution, and then call them with the desired predicate on duty; for example with a `predicate lemma IsEven(n : int)` whose body may be `n % 2 == 0` or an equivalent lambda expression like `n => n % 2 == 0`.

er4.7.dfy

```
function Count<T>(s : seq<T>, p : T -> bool) : nat
        // Count is allowed to read any object p can read
        reads p.reads

        // Predicate p's precondition hold for every element
        // in the sequence
        requires ∀ i | 0 ≤ i < |s| • p.requires(s[i])
{
        if s == [] then 0 else ( if p(s[|s|-1]) then 1 else 0 )
                + Count(s[..|s|-1], p)
}
```

The previous code shows that there are some things to take in consideration, which are reasonable according to Dafny rules. Up to now, there is no official documentation on the use of functional parameters but some examples can be found in the repository tests. In summary, what we need to do is to ensure that `p` invocations are licit: that it is called on parameters for which the precondition holds. And we need also to ensure that `Count` is able to read any object `p` can read.

There are also examples of use of generic types in Exercise 5.14, where we are asked to program an algorithm to decide whether an array is a palindrome. The exercise is solved generically providing the assumption that the type is equally comparable. This can be expressed in Dafny with a (==) following the type parameter name. This approach is also used in Exercise 5.17 where a matrix is tested for symmetry and in 4.26 where we are asked to count the occurrences of certain element in an array.

In Exercise 4.26 generics was dropped and substituted by an equivalent function with concrete types, because the proof seemed to be hard or impossible with the generic definition but immediate with the concrete one. This problem does not persist in Dafny's current version and the generic function has been restored.

3.2.6 Automated induction

In Exercise 2.10 we arrive to many problems regarding some lemmas where Dafny gets stuck evermore or comes to hang the computer, when used without time limit. What happens in `LemaJusto` from 2.10 and other methods is due to automated induction in presence of the simple recursive proof description provided for the lemma.

er2.10.dfy

```
lemma {:induction false} LemaJusto(b : int, e : nat)
        requires e % 2 == 0
        ensures pot(b * b, e / 2) == pot(b, e)

        decreases e
{
        if e > 0 { LemaJusto(b, e - 2); }
}
```

However, other lemma's bodies could be written to turn automated induction helpful, by removing the need for an explicit call to the induction hypothesis. Then automated induction is an advantage instead of a problem, as it was supposed to be.

```
lemma LemaJusto(b : int, e : nat)
        requires e % 2 = 0
        ensures pot(b * b, e / 2) = pot(b, e)

        decreases e
{
        if e > 0 { assert pot(b * b, e / 2 - 1) = pot(b, e - 2); }
}
```

Automated induction is described in *"Automating Induction with an SMT Solver"* [Lei12]. In summary, by the (complete) induction principle, $\forall n \quad P(n)$ holds if $\forall n \quad (\forall k \quad k \prec n \implies P(k)) \implies P(n))$ does, where $\prec$ stands for a well-founded order relation on a type or product of types. Article's Section 2.2. describes the *induction heuristic* used to discriminate whether a lemma or quantifier is suitable for automated induction, taking recursive functions and their parameters into account.

Automated induction can be prevented with the attribute `{:induction false}` which can also be used to specify the particular variables on which we want to apply induction. This has been useful to verify some simple lemmas like the following from Exercise 5.12:

er5.12.dfy

```
lemma {:induction n} LemaPotBase(x : int, y : int, n : nat)
        ensures Pot(x, n) * Pot(y, n) = Pot(x * y, n)
{
}
```

3.3 A detailed example: Euclidean algorithm

er2.11.dfy aritmnl.dfy

In Exercise 2.11 we have to prove the correctness of the following algorithm for greatest common divisor calculation. The statement does not say it explicitly but it is an iterative implementation of Euclidean algorithm.

```
{ $x = X \wedge y = Y \wedge x > y \geq 0$ }
while $y \neq 0$ do
    $\langle x, y\rangle := \langle y, x \mod y\rangle$
endwhile
{ $x = \mathrm{gcd}(X, Y)$ }
```

The following predicate $x > y \ \wedge \ \mathrm{gcd}(X, Y) = \mathrm{gcd}(x, y)$ is proposed as invariant. As expected, the gcd function is not available as a primitive in Dafny. So, to be true to the letter of the problem's formulation, we ourselves should define it when translating this code into Dafny.

```
method gcd(x0 : nat, y0 : nat) returns (x : nat)
        requires x0 > y0 ≥ 0
        ensures x = Gcd(x0, y0)
{
        var y;

        x, y := x0, y0;
        while y ≠ 0
                invariant 0 ≤ y < x
                invariant Gcd(x, y) = Gcd(x0, y0)
        {
                x, y := y, x % y;
        }
}
```

In order to define the `Gcd` function we may appeal to the gcd definition, which is inside its name: *greatest common divisor*

$$m = \gcd(x, y) \iff m \mid x \,\wedge\, m \mid y \,\wedge\, (\,\forall z \in \mathbb{Z} \quad z \mid x \,\wedge\, z \mid y \implies z \mid m\,)$$

where $a \mid b$ means "a divides b". We know that for every $x, y \in \mathbb{N}$ there is a unique $\gcd(x, y)$ in $\mathbb{N}$, so the function is well defined.

To define this function ourselves we have to give a more or less effective functional algorithm to compute it. We know one: Euclidean algorithm.

```
function Gcd(x : nat, y : nat) : nat
        requires x ≠ 0 ∨ y ≠ 0
{
        if x = 0          then    y
        else if y = 0     then    x
        else if x > y     then    Gcd(y, x % y)
                          else    Gcd(x, y % x)
}
```

As expected, verification goes hunky-dory with this definition as the imperative and declarative algorithms are essentially the same. But we have proven that this particular iterative implementation of Euclidean algorithm produces the same result as the more abstract Euclidean algorithm. We can stop here or go on to prove that Euclidean algorithm calculates the greatest common divisor according to its definition.

This proof is easy given the fact that $\gcd(y, x \bmod y) = \gcd(x, y)$ if $x > y$ with follows easily from the fact that $\forall d \quad d > 0 \,\wedge\, d \mid y \;\Rightarrow\; x \bmod d = (x \bmod y) \bmod d$ for which we dedicate a lemma called `LemaDivision`.

But `LemaDivision` is not as easy as it may look. For humans it is almost direct; we all know and Dafny does that $x = x \operatorname{div} y \cdot y + x \bmod y$

$$x \bmod d = (x \operatorname{div} y \cdot y + x \bmod y) \bmod d = ((x \operatorname{div} y \cdot y) \bmod d + x \bmod y) \bmod d = (x \bmod y) \bmod d$$

because $d \mid y$ and $y \mid (x \operatorname{div} y \cdot y)$. But even the latter is not automatically true for Dafny. Some evident results on nonlinear algebra are obscure for it and we have to demonstrate them. These essential propositions have been proved in a separate module called `AritmNL`:

(1) $m \in \mathbb{N}, k \in \mathbb{N}, m \neq 0 \implies (km) \operatorname{div} m = k \,\wedge\, (km) \bmod m = 0$

(2) $n \in \mathbb{Z}, m \in \mathbb{N}, k \in \mathbb{Z}, m \neq 0 \implies (n + km) \bmod m = n \bmod m$

The proofs for the corresponding lemmas are interesting and I recommend their reading in the attached file `aritmnl.dfy`. (2) uses (1) but both are based on the fact that for any $x, y \in \mathbb{Z}$ then $x = x \operatorname{div} y \cdot y + x \bmod y$. This is applied to $m \cdot k$ in order to prove that $d := (m \cdot k) \operatorname{div} m$ is k with a *sandwich* strategy. We are able to prove that $k \geq d$ and that $k - 1 < d$ so we conclude $k = d$.

3.4 An example in depth: limits

er4.4.dfy er4.4aux.dfy

Exercise 4.4 is interesting and singular because of its mathematical content, perhaps infrequent in the formal verifications we have seen up to now, since most code is devoted to the proof of the convergence of a sequence of real numbers. It is also unique for another reason: unlike most exercises developed in this work, the hard part is to prove that the algorithm terminates.

The statement asks to specify and derive an iterative algorithm that given a real number x and a $\varepsilon > 0$ is able to find an approximation of the sine $\sin x$ with an error less that ε, using the Taylor expansion of the function $\sin$ at 0, that is,

$$\sin x = \sum_{n=0}^{\infty} \underbrace{(-1)^n \frac{x^{2n+1}}{(2n+1)!}}_{a_n}$$

In the solution proposed by the authors, the sum of the k first terms of the Taylor expansion (not counting the null terms, as in the previous formula) is returned as the sine approximation along with the k itself for which the condition $|\, a_k \,| < \varepsilon$ is met. In Dafny that will look like:

```
method sinAprox(x : real, e : real) returns (k : nat, s : real)
        requires e > 0.0
        ensures s = SineTaylor(x, k)
        ensures Abs(SineTerm(x, k)) < e
```

where `SineTerm(x, k)` corresponds to a_k and `SineTaylor(x, k)` to $\sum_{n=0}^{k-1} a_n$, that can be easily expressed recursively. See attached file `er4.4.dfy` for details.

Indeed, assuming the preconditions, by Taylor's theorem with remainder in Lagrange form, there exists a ξ between 0 and x such that

$$|\sin x - s\,| = \frac{|\cos \xi\,| \cdot |x|^{2n+1}}{(2n+1)!} \overset{|\cos| \leq 1}{\leq} \frac{|x|^{2n+1}}{(2n+1)!} \leq \varepsilon$$

Therefore, ε is truly an upper bound of the sine's approximation error, if we assume that numerical operations are accurate in the ideal world of Dafny. In fact, the concrete representation of the `real` type in the code generated by Dafny are big-integer quotients, so no additional error will be commited when x is rational.

It was advanced at the beginning of the example and the astute reader may have guessed: the difficulty does not lie in the partial correction of the algorithm, which gets solved by a simple loop and its natural invariants. The difficulty lies in termination.

The loop needs an accumulator variable, the `s` itself which acts as a result, where to add a_n until it becomes small enough. In view of the cost of the independent calculation of every a_n and because this relation $a_{n+1} = x^2/(2n+3)(2n+1)\, a_n$[2] holds, another variable `t` should be defined to carry the current term of the sum. Almost complete, we arrive at

```
s, t, k := 0, x, 0;

while Abs(t) ≥ e
        invariant t = SineTerm(x, k)
        invariant s = SineTaylor(x, k)
{
        s := s + t;

        t :=    ((-1.0) * t * x * x) /
```

[2]Surprisingly, the proof of this fact has also had its complications, with Z3 errors (already solved) and unused but unavoidable local variables included. We should admit there are difficulties apart from the termination.

```
                real((2 * k + 3) * (2 * k  + 2));

        k := k + 1;
}
```

The algorithm terminates if some time `Abs(t) < e` holds or, in other words, $|a_k| < \varepsilon$. That is what the remaining almost 400 lines of the exercise are devoted to. The final result of that code is the following lemma:

```
lemma ExistsK0(x : real, e : real)
        requires e > 0.0
        ensures ∃ k : nat • Abs(SineTerm(x, k)) < e
```

which does not state but almost, because it is not necessary although it is true, that a_n converges to 0

$$\lim_{n\to\infty} a_n = \lim_{n\to\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!} = 0$$

that can be written like this

```
lemma Limit0(x : real, e : real)
        requires e > 0.0
        ensures ∃ k0 : nat • ∀ k | k ≥ k0 • Abs(SineTerm(x, k)) < e
```

To sum up, the loop would look like this, using k_0, whose existence states the lemma, as a bound for the number of iterations the loop can take.

```
ExistsK0(x, e);

ghost var k0 : nat :| Abs(SineTerm(x, k0)) < e;

while Abs(t) ≥ e
        invariant t = SineTerm(x, k)
        invariant s = SineTaylor(x, k)

        invariant 0 ≤ k ≤ k0
        decreases k0 - k
{
        s := s + t;

        t :=    ((-1.0) * t * x * x) /
                real((2 * k + 3) * (2 * k  + 2));

        k := k + 1;
}
```

The previous fragment includes a Dafny statement we have not presented yet. It is the *such-that* declaration which looks like any other variable declaration but it includes a condition after a `:|` symbol where the typical initialization occurs. The verifier must be able to prove that there exists a value which meets the condition and then the variable will stand for any value that fulfils it. The variable is declared `ghost` because it is only used for specification[3].

Now, we will briefly discuss the proof idea for the `ExistsK0` lemma. Although there are certainly easier ways to do this proof, especially considering that it was written in the first phase of this work, the proof process is not without interest.

The proof is split in two files `er4.4aux.dfy` and `er4.4.dfy` which includes the previous one through the `include` directive and contains the executable iterative algorithm. In the first one, something

[3]Any way, only some *such-that* declarations are compilable. The verifier may be able to prove that a value exists but not to construct it.

similar to `ExistsK0` is proved but with function

$$\mathrm{Pf}(x,k) = \frac{x^k}{k!}$$

and in the last file it is adapted to give place to the final `ExistsK0`.

This separation in files is not a purely organizational question. On the contrary it is essential for the correct verification of the algorithm. If `include "er4.4aux.dfy"` is replaced by the content of this file (as the C preprocessor would do) the verification result will be negative, while the separate verification of both files will be satisfactory. This is not the only case where this serious error occurs, against the desirable and expectable principle that a method, lemma or function proof only depends on the *public part* (signature, preconditions, postconditions, ...) of the declarations involved in it.

The following diagram outlines the proof's procedure, which composes simple results to achieve the more complicated ones. Let $x \in \mathbb{R}$ and $\varepsilon > 0$

$$\begin{tikzcd}
 & \forall x \in (0,1/2], m \in \mathbb{N} \quad \exists n \in \mathbb{N} \bullet x^n < 1/m \arrow[d, Rightarrow] & \texttt{[AcotaPotAux]} \\
 & \forall x \in (0,1/2], c > 0 \quad \exists n \in \mathbb{N} \bullet x^n < c \arrow[d, Rightarrow] & \texttt{[AcotaPot]} \\
 & \forall x > 0 \quad \exists k \in \mathbb{N} \bullet \mathrm{Pf}(x,k) < \varepsilon \arrow[d, Rightarrow] & \texttt{[FactVsPot]} \\
 & \forall x > 0 \quad \exists k \in \mathbb{N} \bullet \mathrm{Pf}(x,2k+1) < \varepsilon \arrow[d, Rightarrow] & \texttt{[FactVsPotAdapt]} \\
 & \exists k_0 \in \mathbb{N} \bullet |a_k| < \varepsilon & \texttt{[ExisteK0]} \\
 & |a_k| = \mathrm{Pf}(|x|,2k+1) \arrow[u, Rightarrow] & \texttt{[PfTermino]} \\
\forall n \in \mathbb{N} \bullet 1^n = 1 \quad \texttt{[PotUno]} \arrow[ur, Rightarrow] & & \forall x \in \mathbb{R}, n \in \mathbb{N} \bullet |x^n| = |x|^n \quad \texttt{[PotAbs]} \arrow[ul, Rightarrow]
\end{tikzcd}$$

Some of these lemmas seem to be too evident (like $1^n = 1$) but they are not for Dafny and for the power definition we use. The power and factorial functions have been defined like this:

```
function Pot(x : real, n : nat) : real
        ensures x > 0.0 ⟹ Pot(x, n) > 0.0
{
        if n = 0 then 1.0 else x * Pot(x, n - 1)
}

function Fact(n : nat) : nat
        ensures Fact(n) > 0
{
        if n = 0 then 1 else n * Fact(n-1)
}
```

It is worth commenting briefly some lemmas. For example, `AcotaPot` appears not to have an easy direct proof. Induction? There is nothing ordered. That is why the problem is reduced to the particular case in which c is a natural number inverse. By the Archimedean property, those can be lower than any given positive real number.

Constructively, given $c > 0$ we take $n = \lfloor 1/c \rfloor + 1$ (for which $1/n < c$) and the stated lemma as in `AcotaPotAux` can be addressed by induction, that is what we do.

The hardest lemma is `FactVsPot` and it is solved by an *iterative proof*. First, an $n_0 \in \mathbb{N}$ is found such that $x \,/\, n_0 < 1/2$ (just take $n_0 = 2(\lfloor x \rfloor + 1)$. The `AcotaPot` lemma is used with $\varepsilon \,/\, \mathrm{Pf}(x, n_0)$ from which we deduce $\exists\,\texttt{itmax} \in \mathbb{N} \quad \mathrm{Pf}(x, n_0)(\frac{x}{n_0})^{\texttt{itmax}} < \varepsilon$. This name explains what this value is, an upper bound to the number of iterations needed to find the k whose existence the lemma advocates.

The rest of the proof is a loop in increasing k in which the obvious lemma $x/k < x/n_0$ is maintained along with

$$\mathrm{Pf}(x, k) < \mathrm{Pf}(x, n_0)\left(\frac{x}{n_0}\right)^k$$

so before k arrives to `itmax`, $\mathrm{Pf}(x, k)$ will be less that ε.

The adaptations in `er4.4.dfy` may have been simplified in different ways, modifying the lemmas we just described. Proving that $|a_k| = \mathrm{Pf}(|x|, 2k + 1)$ is relatively easy: we have to introduce the absolute value in the power and manipulate the expressions. The second argument force us to find a value of Pf less than ε with an even argument, and this is not directly deducible from the previous.

It could have been easily proved if $x/k < 1/2$ had been included as a `FactVsPot`'s precondition, or we had asserted that both $\mathrm{Pf}(x, n)$ and $\mathrm{Pf}(x, n + 1)$ are less than ε or we had put the complete definition of limit. At the time it was decided not to modify the lemmas and to provide an independent proof, which is done through some case distinction and playing with the formal ε with which the lemmas are called.

Chapter 4

Data structures

Different data structures have been implemented and verified in this project. We have tried different approaches but a general idea or pretended systematic methodology has been sketched. Some difficulties, limitations or simply inconveniences have molded the strategy when doing things.

4.1 Introduction

In this first section, some Dafny features, that have not yet been explained, are presented and its convenience to represent data structures is discussed.

4.1.1 Class, traits and generics

Dafny classes' declaration has the following shape. The class body can contain method, function and field definitions. No nested classes are allowed and fields are defined like local variables..

```
class C<T> extends B1, …, Bn
{
        // members

        var field1 : type;
}
```

Constructors are a special type of methods only available in classes. They can be defined without name `constructor()` or with a name `constructor Ctor()` and they can receive either parameters or not.

The `<T>` introduces a type parameter. We have already introduced generics in Section 3.2.5. Every method, function, class, datatype, ... is allowed to receive type parameters, denoted by a comma-separated list of formal type names between angles.

Generics are interesting and useful when programming collections for which the concrete nature of its elements is not relevant and that are supposed to be used with many types.

The `extends` clause introduces a list of *traits* the class will extend. Traits syntax is similar to that of classes, except that they cannot extend other traits. Its meaning is also close. Traits are comparable to Java interfaces except that those can contain fields and definitions[1] which are allowed to use the other undefined trait methods.

Traits initially seemed to be an interesting feature to elaborate the specification of abstract data types (ADT) and then allow concrete implementations to inherit from it and share a common specification. In languages that also support traits like Scala and Rust, those can be used to specify the minimum

[1]In fact, Java 5 interfaces are allowed to contain method definitions called *default methods*.

interface type parameters must provide. This allows elegant programming of generic collections and algorithms that require certain operations to be defined on the parameter type, like being comparable for example. Dafny traits do not include this possibility.

Dafny traits have not been used in this project. One of the reasons is that they do not admit type parameters because this feature is not yet implemented. The second is that the trait specification needs to be copied in the extending classes, making futile the aim for clarification pursued with its use.

4.1.2 Dynamic frames

Dafny counts with a mechanism called *dynamic frames* to control access and modification to dynamically allocated memory, to the heap.

In order to reason about programs modularly it is convenient to control what its components depend on and what they are allowed to modify. This leads us to the *frame problem* first introduced by John McCarthy and Patrick J. Hayes in the artificial intelligence field. Regarding the description of a changing situation, it signals the difficulty to express in logic the dynamics of a situation without explicitly specifying all that has not been affected by those changes.

Dynamic frames were invented by I.T. Kassios [Kas11] to "deal with the frame problem in the presence of encapsulation and pointers" and it is used in Dafny with certain particularities. Dafny's frames have object granularity, in other words, frame specifications are sets of objects a function or method is allowed to read or write.

A *frame* is a set of memory positions, and a *dynamic frame* is an expression that designates a *frame*. Methods have `modifies` clauses in which they declare the memory positions they are allowed to modify. Functions use `reads` clauses to indicate which objects they can read.

Another element of the theory is the *swinging pivots requirement*, which means that a frame does not increase in any way other than the allocation of new memory. Leino admits that this principle is extremely strong but it is enforced in the language conventions. A language primitive function called `fresh` exists to say that the object or a collection of objects provided as argument have just been allocated in the current method.

Methods are allowed to read anything but to modify only what has been declared in their `modifies` clauses. Functions cannot modify anything and they must declare the heap variables they can read.

Any knowledge about a predicate or function value is invalidated whenever a method whose modify frame has non-empty intersection with its read frame is called.

```
class Example
{
        // Valid depends on this
        predicate Valid()
                reads this

        method Main()
                requires Valid()

                modifies this
        {
                // this changes
                value := 8;

                // Example is no longer valid
                assert Valid();
        }

        var value : int;
}
```

4.1.3 Inductive datatypes

Inductive or algebraic datatypes are available in Dafny. They are like Haskell's `data` and they are defined with a list of alternative constructors:

```
datatype Maybe = Nothing | Just(int)
```

For every constructor `C`, a *discriminator* member `C?` is defined. When Dafny is able to prove from which alternative an inductive datatype variable is constructed, its components can be safely retrieved accessing with a member if the component was given a name, like in `Just(value : int)`. Otherwise and in any case, `match` expressions and statements can be used to *destruct* an inductive datatype value:

```
match e {
        case Nothing    ⇒ Stmts/Expr
        case Just(v)    ⇒ Stmts/Expr
}
```

Walking through these types is easy. A rank value is defined in such a way that a constructed value has a strictly higher rank than that of any of its components. Like this, termination of recursive functions that decompose an inductive datatype parameter is direct.

Inductive datatypes are used in one of the stack implementations and in the Floyd-Warshall algorithm covered in Section 5.1.

4.1.4 Modules and refinement

Modules are not more than namespaces to arrange related declarations and definitions. Any top level definition is allowed here, even nested modules.

```
abstract module MyModule refines OtherModule {
        // decls
}
```

Modules can refine other modules, that is, provide definitions for declarations which have been left without body or concrete opaque types defined with `type T`. Modules can be declared abstract if no executable code is intended to be generated for them.

Module refinement is available to support a stepwise elaboration of programs, increasing the level of details in each step. Here we will only use them in the stack example to maintain different implementations of that abstract data type (each one in a separate refining module) and a common specification (in the base module).

When a function or method is defined its complete declaration can be elided with `...`. Further information can be found in *rise4fun*'s tutorial and [KL15].

```
abstract module A
{
        type T

        predicate Leq(x : T, y  : T)
                ensures Leq(x, y) ∧ Leq(y, x)  ⟹  x = y
}
module B refines A
{
        type T = int

        predicate Leq...
        {
                x ≤ y
        }
}
```

4.2 Data structures representation

In the specification and verification of data structures we follow the strategy suggested by Hoare in [Hoa72] and described in Section 1.1 just as in [Peñ06]. It is essentially the same approach Dafny's authors propose in [Lei13].

We consider *abstract datatypes* (ADT) composed of a set of types and operations called *signature* or *interface*. In our (class-based) object-oriented approach, we collect all those operations in a class definition. The class type will probably be involved in all those operations along with other types.

To the signature we add the *specification*, in our case, in the form of pre- and postconditions. Those conditions are written in terms of an abstract representation of the ADT based on basic Dafny value types like `set`, `seq`, `map`, ... which have a direct translation to elements of discrete mathematics (like sets, sequences and functions).

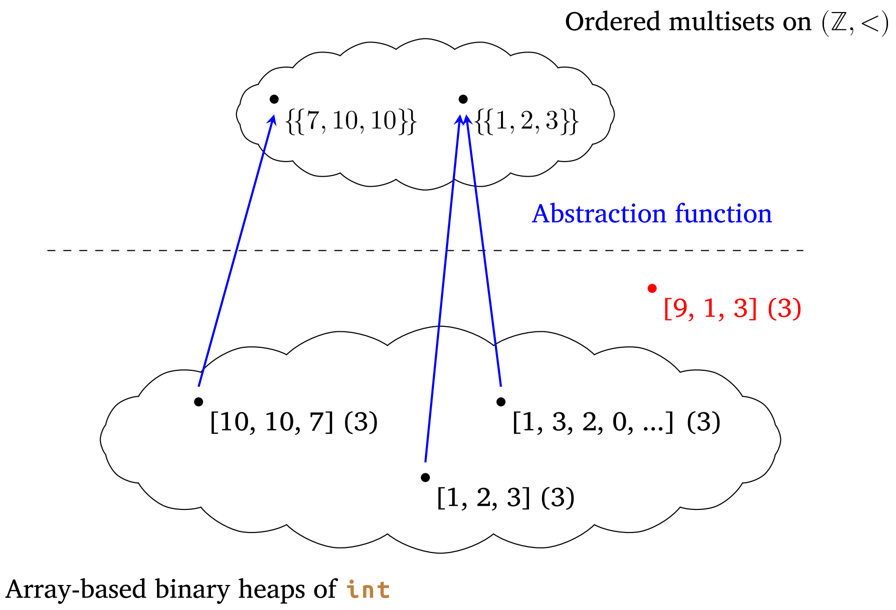


Figure 4.1: Abstraction function and representation invariant, for ADT described in Section 4.5

Each ADT can potentially have different concrete implementations which share the same specification inherited from their base and available for the ADT users. The abstract value is calculated from the concrete representation by means of an *abstraction function*. As not any concrete variable values may be admissible or correct we define a validity condition, the *representation invariant*, that is:

- Precondition of the abstraction function, as well as any other function in the class with the exception of those to be used internally and the representation invariant itself.
- Precondition and postcondition of every class method.
- Postcondition of every constructor.

Let us put into Dafny terms all we have already described. Here the representation invariant is stated using a predicate defined into the class, we conventionally name `Valid()`. As described above, it must be included in the precise `requires` and `ensures` clauses. The abstraction function is defined in another class function, whose name is `Elements()` in all exercises we have done. This function is used to specify the ADT in the pre- and postconditions of the class functions and methods. In addition, postconditions could be added to the abstraction function in order to declare desired properties of the abstract value for the ADT user.

Here there is a difference with the strategy proposed by Dafny's authors. They recommend maintaining the abstract value in a ghost field instead of calculating them with a function. In that case, the field should be updated properly whenever the abstract value changes and the class invariant should guarantee that the ghost field value is accurate, that is to say, that the ghost field contains the value our function would calculate.

Even though verifications work better with the author's alternative according to them, we have chosen to keep using the abstraction function as previously defined.

4.2.1 Allocated memory and dynamic frames

Data structures may use dynamically allocated memory. Dafny's *dynamic frames* convention uses a ghost field `Repr` of type `set<object>` to store all objects the structure is responsible for and is allowed to modify. The `Repr` set must contain `this` and any array or class instance that is part of the structure representation.

If a method has to be able to modify the structure representation a `modifies Repr` clause must be added to its declaration. The *swinging pivots requirement*, i.e. that `Repr` can only grow by means of new allocated memory, should be enforced with postconditions like `fresh(Repr - old(Repr))` for methods or `fresh(Repr - {this})` for constructors.

Not every method has to include a `modifies` clause, because they can implement *observer* operations. However, `function method`s are preferred for observer operations because they are more flexible and can appear in expressions. Sometimes an imperative body is required (or advisable for efficiency) in order to retrieve the requested value. Some observer operations might even need to modify the concrete representation, usually in the case of lazy or *amortized* structures like Fibonacci heaps.

4.2.2 Collections and generics

Data structures like lists, stacks, queues, ... must be generic because they treat data regardless of their nature. As a consequence, even if their element type is a reference type, elements should not be included in `Repr` or accounted in any way. Even if we wanted, elements cannot be included in `Repr` because they are not created by the data structure (*swinging pivots*) and they are not under its responsibility.

Some data structures like priority queues, sorted lists, search trees... require somehow to inspect the data, to compare elements against an order relation, for instance. Again, the structure's elements cannot be included in `Repr` but these objects should be accounted in order to be included in the read frame of the operations (comparison operators for example) which need access to them. The whole data structure validity is then conditioned to the invariability of this data: suppose we have an ordered list of objects and we changed one of them so that the comparison with the other elements varies; then the list would not be in a valid state.

When working with generics we cannot indicate that the elements belong to a frame because the type may be a value type. The plot thickens, so we find tolerable to assume that type parameters will only be value types in Section 4.5.

4.2.3 Autocontracts

There is an experimental attribute `{:autocontracts}` that makes much of what we have described here automatically. A predicate, compulsorily named `Valid`, must be defined for the representation invariant, and Dafny will automatically produce the dull writing of pre- and postconditions for methods and constructors, as well as the `Repr` updates and some other changes. Dafny will complete the representation invariant with `this in Repr` $\wedge$ `null` $\notin$ `Repr` and other `Repr` membership statements depending on the class fields.

Autocontracts is still an experimental feature and it has some drawbacks. Methods are always declared to modify `Repr`, which may be inconvenient. But the greatest difficulty lies in the way errors from automatically added code are reported to the user both in Visual Studio and from the command line. The errors in the pre- and postconditions added by this feature cannot be distinguished from one another. Moreover, the extra code added by *autocontracts* in the methods body to update `Repr` hinders the understanding of those errors causes. The absence of the `Valid` predicate produces an exception and the program termination.

So as to obtain the result of the *autocontracts* application to a class, include the command line option `/rprint:<filename>`[2]. The resulting file will contain internal Dafny definitions and identifiers, so it could not be compilable as is.

4.3 Stacks

stacks

Before going to the task of specifying and programming data structures for their own interest, some experiments have been carried out to help define the strategy to use. The stack example has taken multiple forms and we will succinctly describe its final state to illustrate the ideal procedure and note certain difficulties of the *dynamic frames* encoding.

In this example modules and refinement have been used along with classes. This is probably the easiest example we have done but also the most organized.

The stack signature and specification are written in an abstract module called `Stack` defined in `Stack.dfy`. The module consists of an opaque type declaration `type T` and a class `Stack`. The opaque type definition can be substituted by a generic parameter to the class. Stack operations are `Push`, `Pop`, `Empty`, `Size`, and a constructor to generate the empty stack. Observers have been written as function methods, and `Push` and `Pop` are methods (because they modify the structure).

The specification value is an element of `seq<T>` or T^* and the abstraction function was called `Elements`. Two declarations are included here as example:

```
// Pushes an element on the top of the stack
method Push(e : T)
        modifies this, Repr
        requires Valid()
        ensures Valid()
        ensures fresh(Repr - old(Repr))
        ensures Elements() == [e] + old(Elements())

// Number of elements
function method Size() : nat
        reads this, Repr
        requires Valid()
        ensures Size() == |Elements()|
```

Three different implementations have been written using different Dafny types as support. Each goes in a separate module which refines the `Stack` module. All those modules are abstract because the opaque type `T` remains still opaque; for them to be used `T` should be given a definition. In `Main.dfy` there is an example where `type T = int`.

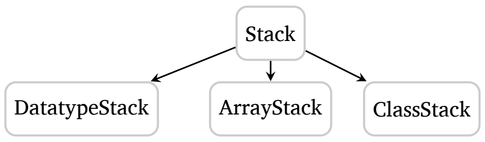

$$\begin{tikzcd}
 & \text{Stack} \arrow[dl] \arrow[d] \arrow[dr] & \\
\text{DatatypeStack} & \text{ArrayStack} & \text{ClassStack}
\end{tikzcd}$$

Figure 4.2: Stack examples

1. `DatatypeStack` uses an inductive datatype

```
datatype StackIR = EmptyStack | Cons(x : T, xs : StackIR)
```

[2]In general, it shows the resulting file after the resolution phase.

Every instance is valid so the representation invariant is `true`.

The abstraction function recursively decomposes the inductive datatype and collects its values, its xs, in a sequence.

2. `ArrayStack` uses a growing array. Its fields are the `array<T> elems` (where elements are stored from left to right) and the number of elements in the stack `nelems` (that can be less than the array capacity).

   Then the representation invariant is not trivial, we have to ensure that `array` $\neq$ `null` and that it has enough capacity `nelems` $\leq$ `elems.Length`[3].

   The abstraction function is not as direct as `elems[..nelems]`. The specification has been written with the top element to the left but the array is holding its values from left to right. So the previous sequence need to be reverted, for which some interesting functions and lemmas were written.

3. `ClassStack` uses linked nodes: values are stored in dynamically allocated *objects*[4] with links to the following value's node until the last element of the list, where the pointer is `null`.

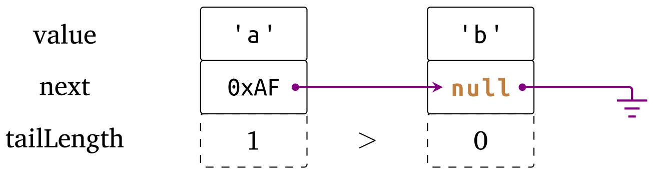


Figure 4.3: `ClassStack` nodes illustration

`ClassStack` is more complex than the previous implementations. We should be aware of the risks linked nodes imply: they could be linked in a loop. Thus, we have to ensure using the representation invariant that the list is properly connected. So we have added an extra ghost attribute `tailLength` to each node, an upper bound of the number of steps we have to take to arrive at the null pointer, i.e. to the end of the list.

Like this, we will guarantee the absence of loops as long as the same node cannot appear twice in a walk in strictly decreasing the `tailLength` values. Walks across the list are now safe. A recursive predicate is suitable to express that property, say:

```
predicate ValidLink(node : Node)
        reads node, node.next

        requires node ≠ null
{
        (node.tailLength = 0  ⟹  node.next = null)
        ∧
        (node.next ≠ null  ⟹  node.next.tailLength < node.tailLength)
}

function ValidChain(node : Node)
        decreases if node = null then 0 else node.tailLength + 1

{
        node = null ∨ ValidLink(node) ∧ ValidChain(node.next)
}
```

Even though termination is solved, `ValidChain` is not well defined yet. Any function is only able to read the objects in its read frame. How can we say that it should be able to read the whole list? Obviously we cannot gather the nodes walking with a function through the list, we will be in the same case. These objects must be available beforehand.

[3]Another statement `elems.Length > 0` is included but its justification comes from irrelevant implementation details.
[4]Despite being instances of a class, nodes do not have any behaviour and are rather plain registers as C-like `struct`s.

Then `Repr` will be a reasonable frame for the function, provided we have added the newly creates nodes to it and we have reflected that ultimately in the representation invariant. Nonetheless, it is not convenient to set the function's frame to `Repr`. Known facts are discarded or they need to be revaluated when their frames have suffered a change. So the smaller the frame the better. With this idea in mind, the validity condition has been written like that:

```
predicate ValidChain(node : Node, nodes : set<object>)
        reads nodes
        requires node = null ∨ node in nodes
{
        node = null
        ∨
        (node.next ≠ null  ⟹  node.next in nodes ∧ ValidLink(node)
                ∧ ValidChain(node.next, nodes - {node}))
}

predicate Valid()
{
        (head ≠ null  ⟹  head in Repr) ∧ ValidChain(head, Repr - {this})
}
```

Note we have added `node.next in nodes` to `ValidChain`. It is in the function itself where we say that the frame contains all the list's nodes. The initial call takes `Repr - {this}` as the `node`. `this` has been removed from the set because there is no need to include it, and in the hope that changes to `this` do not invalidate the known facts about the list chain.

The previous wordy bound function has been omitted since the `node` set is a parameter which decreases in size. However, `.tailLength` is still essential to ensure the absence of loops.

Following the same principles, the representation invariant is written like that:

```
function Elements...
{
        ValueChain(head, Repr - {this})
}

function ValueChain(node : Node, nodes : set<object>) : seq<T>
        reads nodes

        requires node = null ∨ node in nodes
        requires ValidChain(node, nodes)
{
        if node = null

        then    []
        else    [node.value] + ValueChain(node.next, nodes - {node})
}
```

4.4 Lists

list/LinkedList.dfy

A list represents an ordered sequence of values and offers operations to view and modify it. This fundamental datatype is very often implemented as a linked list, and this is our case.

That implementation template was already used in the previous section in the ClassStack example. List elements are stored in dynamically allocated nodes along with a pointer to the node for the next element in the list. The chain is closed by a null pointer and the first element's node is reached through an initial separate pointer.

We have followed the specification and verification principles described in Section 4.2 in an unorthodox manner, as we will see. The chosen operations allow users to view, modify, remove and insert elements to the list using a positional index, in linear time. Other operations exist to get the list length and add elements to the left.

The code fragment below is an extract of the LinkedList class code. We omit the implementation bodies as well as the representation invariant calls and dynamic frames boilerplate in each method or function. *Autocontracts* has not been used in this program and those have been written by hand.

```
class LinkedList<T>
{
  /// Abstract value

  function Elements() : seq<T>

  predicate Valid()

  /// Observers

  // Tests whether the list is empty
  predicate method Empty()
    ensures Empty() ⟺ Elements() = []

  // Gets the list size
  function method Size() : nat
    ensures Size() = |Elements()|

  // Gets the front of the list
  function method Front() : T
    requires ¬Empty()
    ensures Front() = Elements()[0]

  // Gets the n-th element of the list
  method Get(n : nat) returns (x : T)
    requires 0 ≤ n < Size()
    ensures x = Elements()[n]

  /// Modifiers

  // Sets the n-th element of the list
  method Set(n : nat, x : T)
    requires 0 ≤ n < Size()

    ensures Elements() =
      old(Elements())[n := x]

  // Removes the n-th element of the list
  method Remove(n : nat)
    requires 0 ≤ n < Size()
    ensures Elements() =
      old(Elements())[..n]
        + old(Elements())[n+1..]

  // Insert at n-th position of the list
  method Insert(n : nat, x : T)
    requires 0 ≤ n ≤ Size()

    ensures Elements() =
      old(Elements())[..n]
        + [x] +  old(Elements())[n..]

  /// Constructors

  constructor()
    ensures Elements() = []

  method Cons(x : T)
    ensures Elements() =
      [x] + old(Elements())


  // Data representation
  var head : Node<T>;

  // Sequence of nodes
  ghost var Nodes : seq<Node<T> >;
}

// Single linked nodes
class Node<T> {
  var next : Node<T>;

  var value : T;
}
```

Note that the memory representation of the class consists of a set of Node objects in the heap and a pointer which provides access to them from the LinkedList class. The class also counts with a specification-only field Nodes of type seq<T>, which is a key element of the correctness proof of this implementation.

Nodes is the sequence of all nodes (or pointers to nodes) in the list, placed in strictly list order. Then the ith element's next is the $(i+1)$th, the last element of Nodes's next is null and the first is the head.

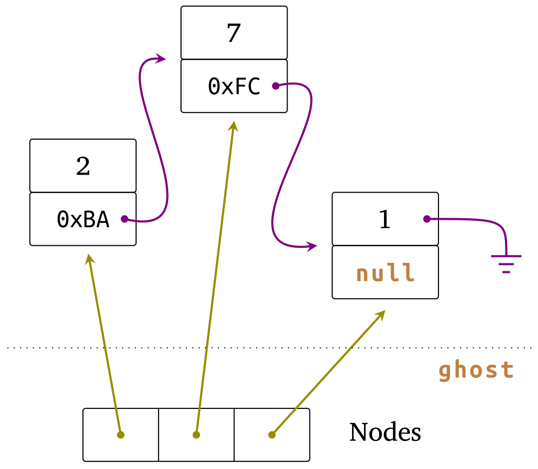


Figure 4.4: `List` illustration

This artifice helps to prove that no loops are produced and that the walks through the references chain are well defined.

Those good properties of the `Nodes` sequence (in fact and most importantly, of the chain of linked nodes) are established by means of the representation invariant.

```
predicate Valid()
        reads Repr()
{
        // The sequence is filled with valid nodes
        (∀ i | 0 ≤ i < |Nodes| • Nodes[i] ≠ null)
        ∧
        // Nodes are linked in a sequence
        (∀ i | 0 ≤ i < |Nodes|-1 • Nodes[i].next = Nodes[i+1])
        ∧
        // If head is empty, there are not nodes
        (Nodes = [] ⟺ head = null)
        ∧
        // Otherwise, the first one is the head and the last one closes
        (head ≠ null ⟹ Nodes[0] = head ∧ Nodes[|Nodes|-1].next = null)
}
```

The `Nodes` field convenience can be discussed but proofs with it seem to be more comfortable than those which would have resulted by using the strategy in Section 4.3. There are no extra requirements as the linear order of the nodes (across their links, not physically in memory) is an inherent characteristic of the list. No execution overhead is produced because `Nodes` existence is restricted to verification time.

The ghost file used to act as the dynamic frames `Repr`, but the code was modified to be closer to the standards and a `Repr` function was written as

```
{this} + set n : object | n in Nodes
```

and used in frame specifications. In this expression, intensional set syntax and the `in` operator for sequences have been used.

The type's abstract value can be obtained recursively with the help of `Nodes`, and it is ultimately a map of the `.value` attributes to `Nodes`.

```
function CollectElements(ns : seq<Node<T> >) : seq<T>
        reads (set n | n in ns)

        requires ∀ i | 0 ≤ i < |ns| • ns[i] ≠ null
        ensures |CollectElements(ns)| = |ns|
        ensures ∀ i | 0 ≤ i < |ns| • CollectElements(ns)[i] = ns[i].value
{
        if ns = [] then [] else

        [ns[0].value] + CollectElements(ns[1..])
}
```

Hereafter we will comment the implementation of a pair of simple operations, which suggest how the rest of the `LinkedList` code was written.

`Cons` is a method to add elements to the left of the list in constant time. The operation is simple: a new node is created to host the new element and its next pointer is set to the previous head, which it replaces.

```
method Cons(x : T)
        modifies this

        requires Valid()
        ensures Valid()
        ensures fresh(Repr() - old(Repr()))

        ensures Elements() = [x] + old(Elements())
{
        var head0 := head;

        head := new Node;

        head.value      := x;
        head.next       := head0;

        Nodes := [head] + Nodes;
}
```

We also need to update the `Nodes` field to accommodate the new node. Dafny is astute enough to demonstrate that all remains correct.

The `Set` method is a bit more complicated. It modifies the value at the nth position of the list. In order to modify the nth node value, access to it is needed. To this end, we iteratively walk the list, keeping the invariant that the current node after m iterations is the mth in `Nodes`. Finally, we can assign the new value `cnod.value := x` in the knowledge that `cnod = Nodes[m]` and we are then modifying `Elements()[m]`.

```
method Set(n : nat, x : T)
        modifies (set n | n in Nodes)

        requires Valid()
        requires 0 ≤ n < Size()

        ensures Valid()
        ensures fresh(Repr() - old(Repr()))

        ensures Elements() = old(Elements())[n := x]
{
        var m := 0;

        // Current node
        var cnod := head;

        while m < n
```

```
                invariant n < |Nodes|
                invariant 0 ≤ m ≤ n
                invariant cnod = Nodes[m]
        {
                cnod := cnod.next;

                assert cnod = Nodes[m+1];

                m := m + 1;
        }

        cnod.value := x;

        DistinctNodes();

        // This is unnecessary but decreases verification time
        ∀ i | 0 ≤ i < |Nodes|
                ensures i ≠ m ⟹ Nodes[i].value =
                        old(Nodes[i].value)
                ensures i = m ⟹ Nodes[i].value = x
        {
                assert Nodes[m] = cnod;
        }
}
```

Dafny is able to complete the proof of the operation description and the representation invariant with a little help, the DistintNodes lemma, which states that all Nodes components are distinct. Then, only the nth position of Nodes, and so only the nth element of the abstract list, has been modified. That is made explicit in a dispensable *forall* statement.

The DistintNodes proof is not difficult. It is done by induction, growing the sequence from right to left. In every step a reduction ad absurdum argument is invoked: if Node[i] were equal to Node[j] then Node[i].next and Node[j].next would. From the properties of Nodes, Nodes[i+1] and Nodes[j+1] will coincide, against the induction hypothesis.

There is another lemma with an easier proof, NullIsLast, which establishes that any node whose next is **null** is the last in Nodes. The proof by reduction ad absurdum is included below:

```
lemma NullIsLast(m : nat)
        requires Valid()
        requires 0 ≤ m < |Nodes|
        requires Nodes[m].next = null

        ensures m = |Nodes| - 1
{
        // Suppose not: next is a valid node which is not null
        if (m ≠ |Nodes| - 1) {
                assert Nodes[m].next = Nodes[m+1];
        }
}
```

Something could be done to reduce the cost of the list operations. Iterators let us work more efficiently with lists. Dafny has a special syntax for them and we have included one in this list implementation, but we have not written a specification for it. In the dynamic frames context, such a specification may be possible, assuming iterators only remain valid as long as no change is done in the structure. We include the iterator code here as a suggestion:

```
iterator Iter<T>(nod : Node<T>) yields (x : T) {
        var curr := nod;

        while curr ≠ null {
```

```
                    yield curr.value;

                    curr := curr.next;
            }
    }
```

4.5 Binary heaps

prioqueue

A priority queue is a collection of elements that can be retrieved according to certain order or priority. This abstract data type is usually implemented in the form of a heap, an arborescent structure where each vertex has higher priority than its children, so the highest priority value is in the root of the tree and every branch is in order.

There are different alternatives to implement a heap. Here we will use a binary heap based on an array. A binary heap is an almost complete binary tree, i.e. all the tree levels are complete except perhaps the deeper one, which only has empty positions to the right. These properties allow a handy implementation using an array where the vertex at position n[5] has its children in $2n+1$ and $2n+2$, whenever they exist.

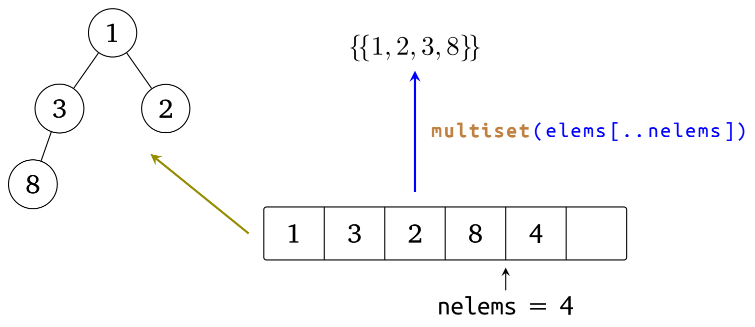


Figure 4.5: Binary heap illustration

In our case, there are three different implementations, almost identical. `IntBinaryHeap.dfy` was written first. It is a binary heap of integers where lower value means higher priority (the order relation for the priority). Then it was generalized in `BinaryHeapMod.dfy` to use a generic element type and a comparison function `gep` (greatest or equal priority) encoded as a function method member in the class. The function must meet the properties of a total preorder relation.

```
predicate method gep(a : T, b : T)
        // Reflexive
        ensures a = b ⟹ gep(a, b)
        // Transitive
        ensures ∀ c | gep(a, c) ∧ gep(c, b) •
                gep(a, b)
        // Total
        ensures gep(a, b) ∨ gep(b, a)
```

However, the previous does not work when the element type is a reference type. The problem (as we anticipate in Section 4.2) is related to *dynamic frames*. The comparator frame must include its parameters but that cannot be expressed with a `reads` frame because the generic type is not known to be a reference type. Note also that the heap may become invalid if an element in the heap changes, because the comparator results may differ.

The initial idea was to let the users set the priority function in the structure constructor, using a lambda or a function defined elsewhere. This has been tried in `BinaryHeap.dfy` without success. We

[5]The array numbering starts at 0.

might have rewritten the structure using `object` (the super type of all reference types) as the element type instead of generics, but we would have left value types out.

The two main operations of the ADT are `Insert` and `Pull` to retrieve the highest priority element. The abstract value is the multiset of all the heap elements. Here is the class code, after removing the bodies and the boilerplate code[6]:

```
class {:autocontracts} BinaryHeap<T(=)>
{
  // Greatest or equal priority comparator
  predicate method gep(a : T, b : T)
    // Reflexive
    ensures a = b ⟹ gep(a, b)
    // Transitive
    ensures ∀ c | gep(a, c) ∧ gep(c, b) •
      gep(a, b)
    // Total
    ensures gep(a, b) ∨ gep(b, a)

  function Elements() : multiset<T>

  // Number of elements in the heap
  function Size() : nat

  // Increase the size of the full up
  // internal array
  method Expand()
    ensures nelems < elems.Length
    ensures Elements() = old(Elements())

  // Retrives the least element of the
  // priority queue
  method Pull() returns (x : T)
    requires Size() ≠ 0
    ensures ∀ y | y in old(Elements()) •
      gep(x, y)
    ensures Elements() =
      old(Elements()) - multiset{x}

  // Inserts an element in the queue
  method Insert(x : T)
    ensures Elements() =
      old(Elements()) + multiset{x}

  constructor()
    ensures Elements() = multiset{}

  predicate Valid()

  // Index of the parent node
  function Parent(n : nat) : nat
    requires n ≠ 0

  // Elements
  var elems : array<T>;

  // Elements count
  var nelems  : nat;
}
```

The concrete representation consists of an array `elems` of type `array<T>`, whose capacity grows as needed using the `Expand` method, and the element count `nelems`. The abstract value is calculated as `multiset(elems[..nelems])`. The representation invariant states that the array is allocated and has enough capacity, and the fundamental property of a heap: that parents have higher priority than their children.

```
predicate Valid()
{
        // There is an array with enough capacity
        elems ≠ null ∧ nelems ≤ elems.Length
        ∧
        // Parents have higher priority
        ∀ i | 0 < i < nelems  •  gep(elems[Parent(i)], elems[i])
}

// Index of the parent node
function Parent(n : nat) : nat
        requires n ≠ 0
{
        (n - 1) / 2
}
```

That has been illustrated in Figure 4.1. Concrete values are represented as the combination of an array `[...]` (`elems`) and a number in parenthesis (`nelems`). The red instance out of the cloud does not meet the invariant because 9 is higher than their children. The other two rightmost elements are equivalent from the abstract point of view, because both contain the same elements in the portion of the array than is considered according to `nelems`.

[6] *Autotriggers* was used here but functions (function methods too) are not managed by it, so some routine code is needed anyway.

Let’s see the code of `Insert` as an example.

```
method Insert(x : T)
        ensures Elements() = old(Elements()) + multiset{x}
{
        if nelems = elems.Length
        {
                Expand();
        }

        ghost var elems0, Repr0 := elems, Repr;

        elems[nelems] := x;

        nelems := nelems + 1;

        // Floats the number

        var pos : nat := nelems - 1;

        ghost var Elements0 := multiset(elems[..nelems]);

        while pos > 0 ∧ ¬gep(elems[(pos - 1) / 2], elems[pos])
                invariant 0 ≤ pos < nelems

                // Keeps validity
                invariant elems = elems0
                invariant Repr = Repr0
                invariant elems ≠ null ∧ nelems ≤ elems.Length

                // All children are lower than their parents except pos

                invariant ∀ i | 0 < i < nelems ∧ i ≠ pos  •
                        gep(elems[Parent(i)], elems[i])

                // But pos's children are lower than their grandparent
                invariant 0 < pos  ⟹  ∀ chd : nat | 0 < chd < nelems  •
                        Parent(chd) = pos  ⟹  gep(elems[Parent(pos)], elems[chd])

                // Elements are only changed by permutations
                invariant Elements0 = multiset(elems[..nelems])
        {
                var parent := (pos - 1) / 2;

                elems[parent], elems[pos] := elems[pos], elems[parent];

                pos := parent;
        }
}
```

If the array capacity is exhausted, a call to `Expand` provides the necessary space for the new element (the array is doubled in size). Then the element is written in the first empty position of the array and elements counter `nelems` is updated. At that moment, the heap may be in an invalid state. To fix this, the inserted element is floated (i.e. permuted with its ancestors) until its parent has higher priority than it or it arrives to the root.

Most invariants in the floating loop say that certain variables do not change. The most interesting invariants are the last which says that the array is only changed by permutations and the two above it, which say that every node except for the floating one is lower priority than its parent and that the floating one’s children are lower priority than their grandparent. So the new element is the only one out of order.

When the loop ends, either the new element arrives to the root or to a position where its parent has higher priority. The structure has recovered its valid state.

The `Pull` implementation is similar. `elems[0]` is returned as the minimum and it is removed and replaced by `elems[nelems-1]`, which is then sunken to its correct position. The element count is reduced by one. These procedures are standard and can be viewed in Section 6.1 of [Cor+09] for example.

The fact that `elems[0]` is the highest priority is not evident for Dafny. The `RootIsMin` lemma demonstrates it. The iterative proof uses reduction ad absurdum: suppose there is an element with strictly higher priority in the heap; then by climbing the tree to the root we conclude `elems[0] < elems[0]`, a contradiction. The key is the fundamental property of the heap, that the parent has higher priority than its child, so we have climbed in increasing priority values.

```
lemma RootIsMin()
        ensures ∀ y | y in Elements() • gep(elems[0], y)
```

Chapter 5

Algorithms

5.1 Floyd-Warshall algorithm

dynamic/floyd.dfy

The Floyd-Warshall algorithm (published by Robert Floyd in 1962 and similar to one of Bernard Roy in 1959 and Stephen Warshall in 1962) is a dynamic programming algorithm to calculate the distance between every two nodes of a directed weighted graph without any restriction except for the absence of negative cost loops. This is not a restrictive requirement since their presence implies that there is no optimal path between nodes of the graph, as iterating such a loop will decrease the cost forever.

The algorithm code has been adapted from the pseudocode of its article in English Wikipedia[1]. It is also available in Section 25.2 of [Cor+09].

The absence of negative loops restriction is assumed and declared as a method's precondition. However, the algorithm is profitable even if the condition does not hold. It terminates and allows us to know precisely if the precondition is satisfied in the graph; because when it is not, at least the calculated cost from a node to itself must be negative.

5.1.1 Specification outline

In order to treat uniformly and elegantly the absence of path or direct connection between two vertices, we have decided to extend real numbers (in which costs are expressed) with an infinity value. Hereafter we will assume that there is an edge between every two vertices, with an infinite cost if it does not really exist. To this end, we define an inductive `datatype` and the operations to work with it, as it is shown thereupon:

```
// Extended real numbers with (positive) infinity
datatype xreal = Real(value : real) | Infty

// Less or equal on xreals
predicate method Leq(x : xreal, y : xreal)
{
        match y
        {
                case Infty ⇒ true

                case Real(yv) ⇒ match x
                        {
                                case Infty ⇒ false

                                case Real(xv) ⇒ xv ≤ yv
                        }
```

[1] Floyd-Warshall algorithm (https://en.wikipedia.org/wiki/Floyd-Warshall_algorithm?oldid=710345047).

```
        }
}

// Addition in xreals
function method Add(x : xreal, y : xreal) : xreal
{
        if x.Infty? ∨ y.Infty?
        then Infty
        else Real(x.value + y.value)
}
```

The graph is given as a square matrix of `xreals` where each cell (i, j) stores the cost of the edge between i and j. The matrix may be symmetric or not. We have defined a type synonym for it `type xgraph = array2<xreal>`.

Once the data representation has been solved, the algorithm specification gives rise to some doubts. How to define the distance between two nodes?

For that, we first define the concept of *path*: a sequence of at least two vertices where only the first and the last are allowed to coincide. The distance between two nodes is the minimum cost of a path between them, where the cost of a path is the sum of the edges costs between every two consecutive vertices.

In Dafny, we represent paths as sequences of node indices of type `seq<int>`. Different predicates are used to state that a sequence is a path. We have defined less restrictive entities that are helpful in verification:

```
predicate Walk(w : seq<int>, bound : nat)
{
        // At least two vertices (an edge)
        |w| ≥ 2
        ∧
        // Intermediate nodes are valid and not higher than bound
        ∀ i | 0 < i < |w|-1 • 0 ≤ w[i] < bound
}

predicate AnyPath(s : seq<int>, bound : nat)
{
        // It is a walk
        Walk(s, bound)
        ∧
        // And it doesn't contain inner loops
        ∀ i, j | 0 ≤ i < |s| ∧ 0 < j < |s|-1 •
                i ≠ j ⟹ s[i] ≠ s[j]
}

predicate Path(s : seq<int>, from : nat, to : nat, bound : nat)
{
        AnyPath(s, bound) ∧ Start(s) = from ∧ End(s) = to
}
```

As we can see above, a *walk* is a path which allows vertices repetitions. Note that all predicates take an extra parameter called bound which restricts the vertices that can appear as intermediate nodes in the walk or path to those whose index is lower than the bound. The reason is in the algorithm strategy.

It stepwise calculates the optimal distances between nodes with that extra restriction. When bound is zero, only direct connections (edges) are considered (this is the base case). In each step the algorithm calculates the optimal path cost with a higher bound. When the bound arrives to the number of nodes in the graph, the algorithm has calculated the distances without restrictions.

In each step, different cases are possible. Suppose we are considering the distance between A and B with intermediate nodes below $C + 1$, we write $d_{C+1}(A, B)$. The distance must be given by an optimal path from A to B:

- If the path does not go through C the minimum distance is $d_C(A, B)$, that we have already calculated.

- If the path passes by C, the minimum distance must be the sum of $d_C(A, C)$ and $d_C(C, B)$. This is a consequence of the Bellman optimality principle.

So the algorithm only has to update the distance when the sum $d_C(A, C) + d_C(C, B)$ is less than $d_C(A, B)$ and those three values have been calculated in the previous step.

Note also that the definitions are lax. In general we do not need to ensure that the indices are in bounds, i.e. that the indices are lower than the number of vertices in the array. When talking about costs, that we read from a matrix of fixed size, this information is essential and predicates and preconditions are used to ensure that the vertex indices are correct.

```
// The cost of travel through a path
// (it does not require that the path is actually a path)
function Cost(s : seq<int>, edges : xgraph) : xreal
        reads edges
        requires edges ≠ null
        requires edges.Length0 = edges.Length1

        requires ∀ i | 0 ≤ i < |s| • 0 ≤ s[i] < Size(edges)
{
        if |s| < 2 then Real(0.0) else Add(edges[s[0], s[1]],
                Cost(s[1..], edges))
}
```

At this point we are able to express the distance definition as the minimum cost of a path between them using a Dafny predicate. Given a cost and a pair of node indices in a graph, the predicate says that this cost is the distance between them if there is a path with this cost and any other path cost is greater or equal, always taking care of the bound restriction.

```
predicate MinCost(from : nat, to : nat, graph : xgraph,
                bound : nat, cost : xreal)

        reads graph
        requires ValidGraph(graph)

        requires bound ≤ Size(graph)
        requires from < Size(graph) ∧ to < Size(graph)
{
        (∀ path | Path(path, from, to, bound) •
                Leq(cost, Cost(path, graph)))
        ∧
        ∃ path | Path(path, from, to, bound) • cost = Cost(path, graph)
}
```

In this code we mention ValidGraph. It is the predicate used to ensure that the graph is valid, i.e. that it is a square matrix whose diagonal is filled with zeros and there are no negative cycles in it.

```
/*
 * Graphs are represented as square matrices of costs (xreal
 * numbers) with no negative cycles.
 */
predicate ValidGraph(edges : xgraph)
        reads edges
{
        edges ≠ null
        ∧
        edges.Length0 = edges.Length1
```

```
          ∧
          (∀ node | 0 ≤ node < Size(edges) •
                  edges[node, node] = Real(0.0))
          ∧
          // No negative cycles
          (∀ path, bound : nat |
                  AnyPath(path, bound)
                  ∧
                  bound ≤ Size(edges)
                  ∧
                  Start(path) = End(path)
                  ∧
                  0 ≤ Start(path) < Size(edges)
          •
                  Leq(Real(0.0), Cost(path, edges))
          )
  }
```

The algorithm signature is shown thereupon. edges is the graph and the out parameter `dist` will contain the distances between every two nodes. The algorithm initializes it with a copy of edges and then iteratively improves it while increasing bound values. The algorithm cost is $O(|V|^3)$ where V is the set of vertices.

```
  method Floyd(edges : xgraph) returns (dist : xgraph)
          requires ValidGraph(edges)

          requires Size(edges) > 0

          ensures dist ≠ null
          ensures dist.Length0 = Size(edges)
          ensures dist.Length1 = Size(edges)

          ensures ∀ i, j | 0 ≤ i < edges.Length0 ∧ 0 ≤ j < edges.Length0 •
                  MinCost(i, j, edges, edges.Length0, dist[i, j])
```

The source code file `floyd.dfy` is available to read the algorithm implementation, specification and verification in detail. We hope its comments would let readers easily understand the specification choices and the verification ideas.

The verification of this algorithm is a little rough, due to the so-called *butterfly effect* which sometimes makes established lemmas to stop working when another lemma's body is modified. It has also been affected by Dafny updates; a previous version worked in Dafny 1.9.6 but it stopped working after the update. The current version verifies with the repository version of Dafny at the moment we write these lines[2], but it will not probably do so with older versions.

5.1.2 Verification outline

As an iterative algorithm, `Floyd` method verification relies on the loop invariants. As a dynamic programming algorithm, we have a matrix, whose contents we use in each iteration to calculate other values. Apart from the invariants and some asserts in the main method, the proof takes place in different separate lemmas, which are called from the main method. Some of them are devoted to the proof that the invariants hold on entry or that they are maintained in different circumstances after each iteration. Others provide more general properties about paths and walks: paths can be split in two paths whose costs sum is equal to the original cost, and walks with an inner loop can be split in loop and cutoff (the rest), whose sum produces the same cost.

`WalkReduce` is another interesting and important lemma which states that for any walk in the graph there is a path of equal or lower cost sharing origin and destination vertices.

[2]2090:8a0ad1c50f24

5.1.3 Using the algorithm from C#

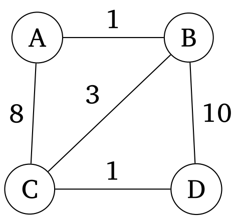
graphs/graph.cs

Dafny is said to generate executable code. The Dafny repository includes in its `Test` folder different programs using a `Main` method to produce an executable. We have already done it for stacks. Another possibility we can learn in `Test` is the use of external methods and classes within Dafny. Here we have tried the opposite: to use Dafny methods and types in external code.

With the aim to see our algorithm running and to test the ability of Dafny to generate usable code for real programs, we have written a simple C# program. It deals with what is beyond the Dafny scope, the interaction with the environment and the user. It reads a graph from a file, fills a matrix with the edge costs and calls the `Floyd` method, printing the results.

We use the `Floyd.dll` assembly generated by Dafny after successful verification, which we link to the final program. Dafny methods are called and Dafny types are used from the C# code.

```
: Minus creates a two-way edge
:
A       B       1       -
A       C       8       -
B       C       3       -
B       D       10      -
C       D       1       -
```



```
Output data:
0       1       4       5
1       0       3       4
4       3       0       1
5       4       1       0
```

Figure 5.1: Example graph with its result (`map2.txt`)

Different graph examples have been included as examples. In one of them we provide a graph which violates the Floyd algorithm preconditions. Note that no verification information is included in the executable code and preconditions are not checked at runtime. However, as we have informally said at the beginning of this section, the algorithm still terminates and provides useful information.

There is virtually no documentation about that. However, .NET assemblies can be inspected to find out how methods are organized and how Dafny types have been translated. In Mono, *monodoc* can be used for that purpose.

In brief, user methods are gathered into a class called `__default`. Its output parameters have been encoded as output parameters in C# and arrays are still arrays, but datatypes are more difficult to manage.

5.2 Dijkstra's algorithm

greedy/dijkstra.dfy

Dijkstra's algorithm, first described by Edsger Dijkstra in 1959, is a greedy algorithm to calculate the shortest paths from a fixed node to the rest of nodes in a graph whose edge distance or cost is non negative.

In each greedy step, the algorithm determines the distance and the minimum path to a new node, the nearest one from the origin for which the distance has not been calculated yet. There are three classes of nodes at any given time:

- *Closed* nodes, for which the final distance is already calculated.
- *Frontier* nodes, which are reachable by an edge from closed nodes. A provisional distance has been assigned to them, but it may be improved when new nodes are closed.
- *Outside* nodes, those which are not directly connected to any closed node. When the algorithm finishes, the remaining outside nodes are unreachable.

The algorithm uses a priority queue to retrieve where the neighbours of each closed node are inserted, taking as priority the distance to the closed node plus the edge cost. If the node was already present the

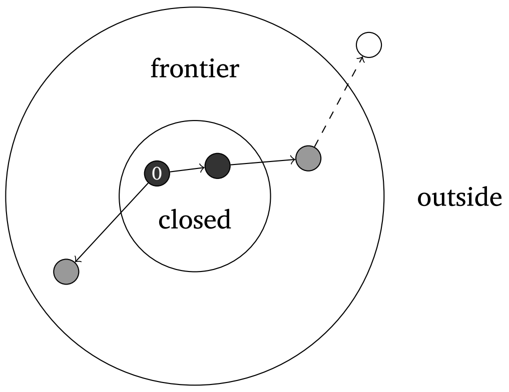


Figure 5.2: Dijkstra's nodes illustration

priority queue is updated only if it is improved. Like this, the `Pull` operation of the priority heap offers the nearest node from the origin.

The original Dijkstra's implementation ran in $O(|V|^2)$ without priority queue, where V is the set of vertices or nodes of the graph. Using a Fibonnaci heap as the queue (see [Cor+09], Section 24.3) the algorithm runs in $O(|E| + |V| \log |V|)$ where E is the set of edges. In our case, we have used an abstract queue called `PriorityQueue` with a complete specification but unimplemented.

There are many differences in the specification and data representation from the Floyd-Warshall algorithm. Both agree that the vertices are identified with a number from 0 but the graph representations differ. An abstract trait has been used to represent the graph, which can be implemented using adjacency lists. An `Adjacents` function is provided; given a node index, it returns a `map<int, real>` where the keys are the neighbour nodes indices and the value the cost of the edge towards them.

```
trait Graph
{
        function method Count() : nat

        function method Adjacents(n : nat) : map<int, real>
                reads this
                requires 0 ≤ n < Count()

                ensures n ∉ Adjacents(n)

                ensures ∀ node | node in Adjacents(n) •
                        0 ≤ node < Count()
                        ∧
                        Adjacents(n)[node] ≥ 0.0
}
```

Edge costs are reals but no infinite cost is explicitly defined: if there is not a direct link between two nodes the corresponding key should not be present in the map. To represent the absence of path to a node a negative value is used, as no path can have negative cost.

The distance, in this case from the origin to any node, is defined in a similar way: the minimum cost of a *walk* between the origin and the given node. We do not use *paths* but *walks*, i.e. we do not ban loops in the path as it would only contribute to the walk cost for worst. The algorithm result is an array `dist` of distances to nodes, each in its corresponding array position. A twin array `prev` can be used to reconstruct the path from the origin to the given node, going backwards to the origin following `prev` values.

We have proved many interesting properties of the algorithm but we have not arrived to prove it completely in time. The file `dijkstra.dfy` is available and commented to read and understand the specification we have used and the verification tactic we have put in place. Succinctly we can comment two lemmas we have proved:

- `IsTheDistance` guarantees that the distance to the node Dijkstra's algorithm extracts from the priority queue is definitive, which its the key point of the greedy character of the program.
- `Unreachables` proves that those vertices with are not closed at the end of the algorithm are unreachable from the origin, i.e. there is not a path from the origin to them. It is demostrated by reductio ad absurdum.

Chapter 6

Conclusions

This final chapter is a reflection on the experience using Dafny and on the work done. Some conclusions are drawn in the hope that they would contribute to a better use and design of these tools in the future.

The experience with Dafny has been bittersweet. The facility to try and install the system and the clarity and simplicity of the language make adaptation to it an easy and enjoyable task. It has been pleasant to recognize the methods learned during the studies, to observe the harmony in mixing the executable code with the specification, to see how mathematical results like the convergence of a numerical limit can be demonstrated in virtually the same terms with which an algorithm to sum a vector is written. Regarding the verifier, its ability to prove the correctness of many algorithms without help, not all of them trivial, has been striking.

Given the experimental nature of the tool and the difficulty its development must imply, the less pleasant aspects should also be pointed out. Error reporting, although it is sufficient for simple programs, is not of much help in more complex programs. However, the most serious problem is what one of the articles we have read called *butterfly effect*: "a minor modification in one part of the program source causes changes in the outcome of the verification in other, unchanged and unrelated parts of the program". Along this project, we have experienced distinct verification results after identifier renaming, permutations of the order of the declarations, changes into the body of a method far from the error origin or after the addition of a new declaration to a file, unrelated to the previous ones.

As for the work done and the results obtained, we can say we are satisfied. However, the easiness in solving the exercises gave us hope to go further in verifying data structures and algorithms. The difficulties appeared as the complexity and size of the programs increase. Some of these difficulties are originated in slip-ups, extreme cases of certain properties that were not considered, or the resistance to the adaptation to the machine reasoning... many have their roots in the problems described in the preceding paragraph.

At this point, we consider that some earlier reading about the theorem prover tactics and about triggers could have been useful. Dynamic frames should have been considered carefully before, because its knowledge could have helped us to better organize ideas, attempts and plans concerning data structures.

For better or for worse, it seems necessary to know the intricacies of Dafny to make an efficient use of it. That is not comparable to using a compiler, for which knowledge of the syntax and semantics of the language must be sufficient to operate with it, regardless of how the source is parsed or the machine code generated.

In the introductory chapter, we include a cite from [BH14] which states that functional verification "is elusive for almost all application scenarios" and pointed to the large ratio between specification and useful code. In the Floyd algorithm, the ratio is seven to one. Nevertheless, if we consider the number of times the verified program would be used or the errors we can prevent with it, this effort could be worthwhile.

We finally conclude that a verification system as Dafny which does not want to frustrate the user must respect the principle of decomposition: each method should be a black box which only depends on the other methods by their preconditions, postconditions or other public specifications they declare. What happens between the brackets, between the brackets must remain. This principle theoretically holds in Dafny, but in practise it is not met. Avoiding those *butterfly effects* should be a priority for a better user experience, although that supposes a loose of performance or a reduction in the capacity of autonomous reasoning of the system.

Capítulo 7

Conclusiones

En esta sección se hace una reflexión final sobre la experiencia con el sistema Dafny y sobre el trabajo realizado. De todo ello se han extraído algunas conclusiones que esperamos puedan contribuir a un mejor uso y diseño de estas herramientas en el futuro.

La experiencia con Dafny ha sido agridulce. La facilidad para probar e instalar el sistema y la claridad y sencillez del lenguaje hacen de su adaptación a él una tarea fácil y amena. Ha sido grato reconocer los métodos aprendidos durante los estudios, observar la armonía con que se mezclan el código ejecutable y la especificación, ver cómo resultados matemáticos como la convergencia de un límite numérico se pueden demostrar en prácticamente los mismos términos en que se escribe un bucle para sumar un vector. Llama también la atención la capacidad del verificador para demostrar sin ayuda la corrección de no pocos algoritmos, no siempre evidentes.

Habida cuenta de la naturaleza experimental de la herramienta y la dificultad que desarrollarla ha de suponer, es preciso señalar también los aspectos menos agradables. La notificación de errores, si bien suficiente para programas simples, no es de tanta ayuda en programas de mayor complejidad. Pero el problema más grave es aquel que uno de los artículos consultados denomina *efecto mariposa*: «una pequeña modificación en una parte del código del programa provoca cambios en el resultado de la verificación en otras partes del mismo, que no han sido modificadas ni están relacionadas». En el trascurso de este trabajo, se han experimentado resultados de verificación dispares ante un renombramiento de identificadores o una permutación del orden de las declaraciones. El cambio en el cuerpo de un método o la incorporación una nueva declaración ha producido en ocasiones cambios en otros elementos del programa sin relación alguna con ellos.

Valorando el trabajo realizado y los resultados obtenidos, se puede decir que se está satisfecho y no se considera que hayan sido escasos. Sin embargo, la facilidad para escribir y verificar los primeros problemas dio lugar a la esperanza de poder llegar más lejos en la verificación de estructuras de datos y algoritmos. Las dificultades fueron apareciendo al aumentar la complejidad y el tamaño de los programas. Algunas de esas dificultades tienen su origen en despistes, casos extremos de ciertas propiedades que no se habían considerado o la no adecuada adaptación al razonamiento máquina. Otras tienen su origen en los problemas descritos en el párrafo anterior.

Echando la vista atrás, una lectura más temprana sobre el funcionamiento del demostrador y sobre disparadores podría haber resultado de utilidad. Un aprendizaje más cuidadoso sobre los marcos dinámicos hubiese ayudado a organizar mejor las ideas, los intentos y los planes respecto a las estructuras de datos.

Por suerte o por desgracia, parece necesario conocer las interioridades de un sistema de verificación como Dafny para hacer un uso eficiente de él. No es comparable al uso de un compilador, para el cual el conocimiento de la sintaxis y semántica del lenguaje ha de ser suficiente para operar con él, independiente cómo se analice el código o qué representación interna se utilice.

En el capítulo introductorio, se incluye una cita de [BH14] que afirma que la verificación funcional «es inasequible para casi todas las aplicaciones» y señala la gran desproporción entre código de especificación y verificación y el código útil. En el algoritmo de Floyd, la proporción es de siete a uno. Aún así, si consideramos el número de ejecuciones del programa verificado o los errores que se pueden evitar con ello, el esfuerzo puede merecer la pena.

Concluimos finalmente que un sistema de verificación como Dafny que no quiera frustrar al usuario debe respetar el principio de descomposición: cada método debe ser verdaderamente una caja negra que solo dependa de otros métodos por sus precondiciones, postcondiciones y otras especificaciones públicas que declare. Lo que ocurra entre las llaves entre las llaves ha de quedar.

En teoría Dafny sigue este principio, pero en la práctica no se cumple. Evitar estos efectos mariposa debe ser una prioridad para una mejor experiencia de usuario, aunque ello suponga un deterioro del rendimiento o una merma de la capacidad de razonamiento autónomo de la herramienta.

Appendix A

Dafny program examples

A.1 An iterative and recursive binary search: er5.12.dfy

```
/*
 * En la versión anterior de este ejercicio Ord se definía como s[i] ≤ s[i+1]
 * y así sólo funcionaba con la opción /autoTriggers:0.
 *
 * Hay un ejemplo oficial en:
 * http://dafny.codeplex.com/SourceControl/latest#Test/triggers/
 * some-proofs-only-work-without-autoTriggers.dfy
 */

// ¿Está la secuencia ordenada?
predicate Ord(s : seq<int>) {
    ∀ i, j | 0 ≤ i ≤ j < |s| - 1 • s[i] ≤ s[j]
}


// Versión recursiva

method busq_binaria(a : array<int>, x : int, c : nat, f : nat) returns
    (b : bool, p : nat)

    requires a ≠ null
    requires Ord(a[..])
    requires 0 ≤ c ≤ f ≤ a.Length-1

    ensures b ⟹ c ≤ p < f ∧ a[p] = x
    ensures ¬b ⟹ c ≤ p ≤ f
        ∧ (∀ j | c ≤ j < p • a[j] < x)
        ∧ (∀ j | p ≤ j < f • a[j] > x)

    decreases f - c
{
    var m : nat;

    if c = f {
        b, p := false, c;
    }
    else /* c < f */ {
        m := (c + f) / 2;

        if x < a[m] {
            b, p := busq_binaria(a, x, c, m);
        }
        else if x = a[m] {
            b, p := true, m;
        }
        else /* x > a[m] */ {
            b, p := busq_binaria(a, x, m+1, f);
        }
    }
}


// Versión iterativa

method busq_binaria_it(a : array<int>, x : int, c : nat, f : nat) returns
    (b : bool, p : nat)

    requires a ≠ null
    requires Ord(a[..])
    requires 0 ≤ c ≤ f ≤ a.Length-1

    ensures b  ⟹  c ≤ p < f ∧ a[p] = x
    ensures ¬b  ⟹  c ≤ p ≤ f
        ∧ (∀ y | y in a[c..p]  •  y < x)
        ∧ (∀ y | y in a[p..f]  •  y > x)
{
    var m : nat := (c + f) / 2;

    var c', f' := c, f;

    while c' < f' ∧ x ≠ a[m]
        invariant c ≤ c' ≤ f' ≤ f
        invariant m = (c' + f') / 2;

        // Resto de invariantes
        invariant ∀ y | y in a[c..c']  •  y < x
        invariant ∀ y | y in a[f'..f]  •  y > x
    {
        if x < a[m] {
            f' := m;
        }
        else /* x > a[m] */ {
            c' := m+1;
        }

        m := (c' + f') / 2;
    }

    if c' = f' {
        b, p := false, c';
    }
    else /* x = a[m] */ {
        b, p := true, m;
    }
}
```

A.2 Euclidean algorithm

A.2.1 er2.11.dfy

```
/**
 * En este ejercicio se prueba la corrección un algoritmo iterativo para el
 * cálculo del máximo común divisor por medio del algoritmo de Euclides.
 *
 * Por un lado se demuestra que el algoritmo iterativo da como resultado el
 * de la función Mcd, que es una versión funcional del algoritmo de Euclides.
 * Por otro, se prueba que Mcd es el máximo común divisor según su definición,
 * es decir, el mayor de todos los divisores comunes.
 *
 * Se ha utilizado (en LemaDivision) una notación específica para demostrar
 * para-todos. Se ha encontrado por casualidad en la siguiente página:
 * http://www.lexicalscope.com/blog/2014/07/31/dafny-proving-∀-x-px-qx/
 */

include "aritmnl.dfy"

// x divide a y
```

```
predicate Divide(x : nat, y : nat) {
    x ≠ 0 ∧ y % x = 0
}

// Mcd por el algoritmo de Euclides
function Mcd(x : nat, y : nat) : nat
    requires x ≠ 0 ∨ y ≠ 0
{
    if x = 0    then     y
    else if y = 0   then     x
    else if x > y   then     Mcd(y, x % y)
            else     Mcd(x, y % x)
}

// m es el máximo común divisor de x e y
predicate EsMcd(x : nat, y : nat, m : nat)
    requires x ≠ 0 ∨ y ≠ 0
{
    Divide(m, x) ∧ Divide(m, y)
    ∧ ∀ d : nat | Divide(d, x) ∧ Divide(d, y)  •  Divide(d, m)
}

// Lema útil para demostrar McdEsMcd
lemma LemaDivision(x : nat, y : nat)
    requires y > 0
    ensures ∀ d : nat | Divide(d, y)  •  x % d = (x % y) % d
{
    // Descomposición por división entera
    assert x = (x/y) * y + x % y;

    // Se utiliza la sintaxis específica para demostrar paratodos
    // Aquí d un natural cualquiera que divide a y
    ∀ d : nat | Divide(d, y)
        ensures x % d = (x % y) % d
    {
        // Eso permite la siguiente descomposición
        assert y = (y/d) * d;

        // Y se puede reescribir la descomposición
        var f := (x/y) * (y/d);

        assert x = f * d + x % y;

        AritmNL.ModMasMultiplo(x % y, d, f);

        // De la igualdad primera y el resultado del lema
        assert x % d = (x % y) % d;
    }
}

lemma McdEsMcd(x : nat, y : nat)
    requires x ≠ 0 ∨ y ≠ 0
    ensures  EsMcd(x, y, Mcd(x, y))

    decreases y, x
{
    // Dafny sabe que Mcd y EsMcd son conmutativos en las
    // dos primeras entradas

    // Supone sin pérdida de generalidad que x ≥ y
    if x < y {
        McdEsMcd(y, x);
    }
    else {
        if y = 0 {
            // Caso fácil

            // Curiosamente son necesarios
            assert Mcd(x, 0) = x;
            assert Divide(x, 0);
        }
        else {
            // Inducción
            McdEsMcd(y, x % y);

            // Por hipótesis y Mcd(x, y) = Mcd(y, x % y)
            assert EsMcd(y, x % y, Mcd(x, y));

            // Aplicando el lema de la división se obtiene que
            // d es divisor de x y x % y  ⟺  es divisor de x e y
            LemaDivision(x, y);

            // Como conclusión, Mcd(x, y) | x

            // Y además todo divisor de x e y divide a Mcd(x, y)
        }
    }
}

method mcd(x0 : nat, y0 : nat) returns (x : nat)
    requires x0 > y0 ≥ 0
    ensures x = Mcd(x0, y0)
{
    var y;

    x, y := x0, y0;

    while y ≠ 0
        invariant 0 ≤ y < x
        invariant Mcd(x, y) = Mcd(x0, y0)
    {
        x, y := y, x % y;
    }
}
```

A.2.2 aritmnl.dfy

```
/**
 * Este módulo incluye algunos resultados bastante elementales sobre la
 * división entera y el módulo que el demostrador no sabe por defecto.
 */

/**
 * La demostración de estos dos lemas resultó más difícil de lo esperado.
 * Muchos asertos son prescindibles pero se incluyen para seguir la idea
 * de la demostración. Tal vez haya demostraciones más directas sabiendo
 * cómo trata el demostrador la aritmética no lineal (hay una opción
 * «/noNLarith» para disminuir su conocimiento sobre el particular).
 *
 * Las demostraciones aquí utilizan que el módulo es positivo y menor que
 * el divisor y con ello hacen acotaciones en forma de sándwich.
 *
 * Ha sido necesario definir variables en las demostraciones. Sin ellas el
 * demostrador no era capaz de demostrar lo que está demostrado. Parece que
 * aunque dos términos compuestos sean sintácticamente iguales no los
 * considera el mismo y no puede aplicarles algo como
 * ∀ x • P(x) ⟹ Q(x).
 */

module AritmNL {

lemma DivPorFactor(m : nat, k : int)
    requires m ≠ 0
    ensures (k*m) / m = k
    ensures (k*m) % m = 0
{
    // Al divisor le llamo d
    var d := (k*m)/m;

    // Se toma lo siguiente como base
    assert k*m = d * m + (k*m) % m;
    assert 0 ≤ (k*m) % m < m;

    // Se substituye de acuerdo a la acotación
    assert k*m ≥ d * m;
    assert k*m < d * m + m;

    // Se cancelan factores comunes
    assert k ≥ d;

    // La otra desigualdad resulta más compleja
    assert k*m - d*m - m < 0;
    assert (k - d - 1) * m < 0;

    var dd := k - d - 1;

    assert dd ≥ 0 ⟹ dd * m ≥ 0;

    assert dd < 0;
    assert k - d - 1 < 0;

    // Con esta acotación sólo puede ser k
    assert k - 1 < d ≤ k;
}

lemma ModMasMultiplo(n : int, m : nat, k : int)
    requires m ≠ 0
    ensures (n + k * m) % m = n % m
{
    // Partimos de los dos siguientes asertos
    assert n+k*m = (n+k*m)/m * m + (n+k*m)%m;
    assert n = n/m*m + n%m;

    // Substituyendo el segundo en el primero
    assert n/m*m + n%m + k*m = (n+k*m)/m * m + (n+k*m)%m;

    // Se pasan a la izda los múltiplos de m
    assert n/m*m + k*m - (n+k*m)/m * m = (n+k*m)%m - n%m;

    // Se saca factor común
    assert (n/m + k - (n+k*m)/m) * m = (n+k*m)%m - n%m;

    // Por la acotación de los módulos se sabe que la parte dcha
    // está entre -m y m exclusive.
    // De la parte izda se deduce que es múltiplo de m y el único
    // múltiplo de m en ese intervalo es 0.
    assert -m < (n+k*m)%m - n%m < m;

    // Hay que ayudarle para llegar a esa conclusión
    // t es la parte izquierda sin m
    var t := n/m + k - (n+k*m)/m;

    assert -m < t * m < m;

    // Aplicamos el lema demostrado anteriormente
    DivPorFactor(m, t);

    assert -1 < t < 1;

    // Luego t = 0 y eso implica lo buscado
}

lemma SumaMod0(a : int, b : int, m : nat)
    requires m ≠ 0
    requires a % m = 0
    requires b % m = 0

    ensures (a + b) % m = 0
    ensures (a - b) % m = 0
{
        // Descompone en divisor * cociente + resto
        assert a = (a / m) * m;
        assert b == (b / m) * m;

        // Para luego sacar factor común
        assert a + b == (a/m + b/m) * m;
        assert a - b == (a/m - b/m) * m;

        DivPorFactor(m, a/m + b/m);
        DivPorFactor(m, a/m - b/m);
}

lemma SumaMod(a : int, b : int, m : nat)
    requires m != 0

    ensures (a + b) % m == (a % m + b % m) % m
    ensures (a - b) % m == (a % m - b % m) % m
{
        // Descompone en divisor * cociente + resto
    //  assert a == (a / m) * m + a % m;
    //  assert b == (b / m) * m + b % m;

        // Para luego sacar factor común
    //  assert a + b == (a/m + b/m) * m + (a % m + b % m);
    //  assert a - b == (a/m - b/m) * m + (a % m - b % m);

        ModMasMultiplo(a % m + b % m, m, a/m + b/m);
        ModMasMultiplo(a % m - b % m, m, a/m - b/m);
}

}
```

A.3 Sine calculation

A.3.1 er4.4.dfy

```
/**
 * Cálculo aproximado del seno por Taylor.
 */

/*
 * Lo que aparece en los includes no se demuestra.
 */
include "er4.4aux.dfy"

// Valor absoluto
// (es function-method para poder usarla en la implementación)
function method Abs(x : real) : real
    ensures Abs(x) >= 0.0
{
    if x < 0.0 then -x else x
}

// «El valor absoluto del cociente es el cociente de los valores absolutos»
lemma AbsCociente(x : real, y : real)
    requires y != 0.0
    ensures Abs(x / y) == Abs(x) / Abs(y)
{
}

// Término de orden 2k+1 de la serie de Taylor del seno en x
function TerminoSeno(x : real, k : nat) : real {
    Pot(-1.0, k) * Pot(x, 2*k + 1) / real(Fact(2 * k + 1))
}

// Polinomio de Taylor del seno de orden 2n-1 evaluado en x
function TaylorSeno(x : real, n : nat) : real {
    if n == 0
    then 0.0
    else TerminoSeno(x, n-1) + TaylorSeno(x, n-1)
}
```

```
// Prueba que ciertos productos convierten un coeficiente en su siguiente
lemma AvanceTerm(x : real, k : nat)
    ensures ((-1.0) * TerminoSeno(x, k) * x * x)
        / real((2 * k + 3) * (2 * k + 2))
        == TerminoSeno(x, k+1)
{
    assert (-1.0) * Pot(-1.0, k) == Pot(-1.0, k+1);

    // Al hacer estos dos pasos juntos se daba un error de Z3
    // con versiones antiguas de Dafny
    assert Pot(x, 2*k + 1) * x * x == Pot(x, 2*k + 3);
    assert Pot(x, 2*(k+1) + 1) == Pot(x, 2*k + 3);

    assert real((2 * k + 3) * (2 * k + 2)) * real(Fact(2 * k + 1))
        == real(Fact(2 * k + 3));
}


/**
 * Demostración de la terminación.
 */

lemma PotAbs(x : real, k : nat)
    ensures Abs(Pot(x, k)) == Pot(Abs(x), k)
{
}

lemma PotUno(k : nat)
    ensures Pot(1.0, k) == 1.0
{
}

lemma PfTermino(x : real, k : nat)
    ensures Abs(TerminoSeno(x, k)) == Pf(Abs(x), 2 * k + 1)
{
    var dk1 := 2 * k + 1;

    calc == {
        Abs(TerminoSeno(x, k));
        // Definición de TerminoSeno (parece necesario explicitarla)
        { assert TerminoSeno(x, k) == Pot(-1.0, k) * Pot(x, dk1)
            / real(Fact(dk1)); }
        Abs(Pot(-1.0, k) * Pot(x, dk1) / real(Fact(dk1)));
        // Pasa el valor absoluto a los factores
        Abs(Pot(-1.0, k)) * Abs(Pot(x, dk1)) / Abs(real(Fact(dk1)));
        // Mete el valor absoluto dentro de la potencia (fase 1)
        { PotAbs(-1.0, k); }
        Pot(1.0, k) * Abs(Pot(x, dk1)) / Abs(real(Fact(dk1)));
        // Quita la potencia de 1
        { PotUno(k); }
        Abs(Pot(x, dk1)) / Abs(real(Fact(dk1)));
        // Quita el valor absoluto del denominador (es positivo)
        { assert Fact(dk1) > 0; }
        Abs(Pot(x, dk1)) / real(Fact(dk1));
        // Vuelve a conmutar Abs y Pot (fase 2)
        { PotAbs(x, dk1); }
        Pot(Abs(x), dk1) / real(Fact(dk1));
        // Definición de Pf
        Pf(Abs(x), dk1);
    }
}

/*
 * Adaptación de PotVsFact para tomar impar el n0
 *
 * Otras posibles alternativas:
 * - Cambiar la postcondición de FactVsPot a un existe-para todos
 *   los mayores (como la definición de límite).
 * - Añadir a la postcondición de FactVsPot que el x / k es menor
 *   que 1.
 *
 * Ambas opciones no parecen difíciles sobre el papel.
 */
lemma FactVsPotAdapt(x : real, e : real)
    requires x > 0.0
    requires e > 0.0

    ensures ∃ k : nat • Pf(x, 2*k+1) < e
{
    var k : nat;

    if x ≤ 1.0 {
        FactVsPot(x, e);

        var dk : nat :| Pf(x, dk) < e;

        // En todos los casos vale la misma
        k := dk / 2;

        // Ya está, sólo hay que despejar k
        if dk % 2 = 1 {
            calc = {
                Pf(x, 2 * k + 1);
                { assert dk = 2 * k + 1; }
                Pf(x, dk); < e;
            }
        }
        // Como x < 1.0 el Pf siguiente es también menor que e
        else {
            calc = {
                Pf(x, 2*k+1);
                { assert 2 * k = dk; }
                Pf(x, dk+1);
                // Definición de Pf
                Pf(x, dk) * (x / real(dk + 1));
                ≤
                { assert x / real(dk + 1) ≤ 1.0; }
                Pf(x, dk); < e;
            }
        }
    }

    else {
        // Se coge como épsilon e / x < e
        FactVsPot(x, e / x);

        var dk : nat :| Pf(x, dk) < e / x;

        // Valor común
        k := dk / 2;

        if dk % 2 = 1 {
            calc = {
                Pf(x, 2 * k + 1);
                { assert dk = 2 * k + 1; }
                Pf(x, dk);
                < e / x;
                < e;
            }
        }
        else {
            calc = {
                Pf(x, 2 * k + 1);
                { assert dk = 2 * k; }
                Pf(x, dk + 1);
                // Definicion de Pf
                Pf(x, dk) * x / real(dk + 1);
                ≤
                { assert real(dk + 1) ≥ 1.0; }
                Pf(x, dk) * x;
                // Pf(x, dk) < e / x;
                < (e / x) * x;
                e;
            }
        }
    }

}


/**
 * Cuidado se pide que exista uno menor que no que todos lo sean.
 */
lemma ExisteK0(x : real, e : real)
    requires e > 0.0
    ensures ∃ k : nat • Abs(TerminoSeno(x, k)) < e
{
    if x = 0.0 {
        calc = {
            TerminoSeno(x, 0);
            // Definición
            -1.0 * Pot(x, 1) / real(Fact(1));
            // Simplificando
            -1.0 * x;
            0.0;
        }
    }
    else {
        // Utilizamos lo demostrado en el archivo auxiliar
        FactVsPotAdapt(Abs(x), e);

        var k : nat :| Pf(Abs(x), 2*k+1) < e;

        // Hay que hacer una pequeña adaptación
        PfTermino(x, k);
    }
}

method senoAprox(x : real, e : real) returns (k : nat, s : real)
    requires e > 0.0
    ensures s = TaylorSeno(x, k)
    ensures Abs(TerminoSeno(x, k)) < e
{
    // El término actual de la serie
    var t := x;

    k, s := 0, 0.0;

    // Esto ayuda a probar que TerminoSeno(x, 0) = x
    assert Pot(x, 1) = x;

    // Toma el k0 que para e asegura la existencia del límite
    ExisteK0(x, e);

    // Si se quita la especificación de tipo hay fallos
    ghost var k0 : nat :| Abs(TerminoSeno(x, k0)) < e;

    while Abs(t) ≥ e
        invariant t = TerminoSeno(x, k)
        invariant s = TaylorSeno(x, k)

        invariant 0 ≤ k ≤ k0
        decreases k0 - k
    {
        // Guarda el valor inicial de t
        ghost var t0 := t;

        s := s + t;

        t :=    ((-1.0) * t * x * x) /
            real((2 * k + 3) * (2 * k  + 2));
        // Demuestra el avance correcto de la variable t
        AvanceTerm(x, k);

        k := k + 1;
    }
}
```

A.3.2 er4.4aux.dfy

```
// Potencia (de números reales)
function Pot(x : real, n : nat) : real
    ensures x > 0.0 ⟹ Pot(x, n) > 0.0
{
    if n = 0 then 1.0 else x * Pot(x, n - 1)
}

// Factorial
function Fact(n : nat) : nat
    ensures Fact(n) > 0
{
    if n = 0 then 1 else n * Fact(n-1)
}


/*
 * Lemas auxiliares relativos al orden de los números
 * reales.
 */

// Orden inverso de los inversos
lemma OrdenInversos(x : real, y : real)
    requires 0.0 < x < y
    ensures 1.0 / x > 1.0 / y
{

}

// Dividir respeta el orden
lemma AcotaFrac(x : real, y : real, c : real)
    requires c > 0.0
    requires 0.0 ≤ x ≤ y

    ensures x / c ≤ y / c
{
}

// Lema para acotar productos (orden no estricto)
lemma AcotacionProducto(x : real, y : real, s : real, t : real)
    requires 0.0 ≤ x ≤ y
    requires 0.0 ≤ s ≤ t

    ensures x * s ≤ y * t
{
    // ¿Por qué este lema se verifica solo en archivo aparte
    // y aquí no?

    calc ≤ { x * s; y * s; y * t; }
}

// Lema para acotar productos (orden estricto)
lemma AcotaProdEst(x : real, y : real, s : real, t : real)
    requires 0.0 < x < y
    requires 0.0 < s < t

    ensures 0.0 < x * s < y * t
{
    // ¿Por qué este lema se verifica solo en archivo aparte
    // y aquí no?

    calc < { x * s; y * s; y * t; }
}

// Las potencias de base en (0, 1) tienden a 0 (lema auxiliar)
lemma AcotaPotenciaAux(x : real, m : int)
    requires 0.0 < x ≤ 1.0 / 2.0
    requires m > 0

    ensures ∃ n : nat • Pot(x, n) < 1.0 / real(m)
{
    // Demostración por inducción

    if m = 1 {
        assert Pot(x, 1) < 1.0 / real(1);
    }
    else {
        AcotaPotenciaAux(x, m-1);

        var n : nat :| Pot(x, n) < 1.0 / real(m-1);

        // n+1 vale como aquello que existe para m

        calc {
            Pot(x, n+1);
            = // Definición de Pot
            x * Pot(x, n);
            < // Hipótesis
            x / real(m-1);
            // Parece evidente pero funciona aleatoriamente
            ≤ { AcotaFrac(x, 1.0/2.0, real(m-1)); }
            (1.0 / 2.0) / real(m-1);
            = // Pasar al denominador
            1.0 / real(2 * (m-1));
            ≤ { assert m > 1; }
            1.0 / real(m);
        }
    }
}


// Las potencias de base en (0, 1) tienden a 0
lemma AcotaPotencia(x : real, c : real)
    requires 0.0 < x ≤ 1.0 / 2.0
    requires c > 0.0

    ensures ∃ n : nat • Pot(x, n) < c
{
    var inv := 1.0 / c;

    // Un entero tal que 1/m < c
    var m : nat := inv.Trunc + 1;

    assert real(m) > inv;

    OrdenInversos(inv, real(m));

    assert 1.0 / real(m) < c;

    AcotaPotenciaAux(x, m);
}


// Da un nombre al factor x^n / n¬
function Pf(x : real, n : nat) : real {
    Pot(x, n) / real(Fact(n))
}


// Prueba que el avance de Pf que se usa en FactVsPot es correcto
lemma AvancePf(x : real, k : nat, f : real)
    requires k > 0
    requires f = x / real(k)
    ensures Pf(x, k) == Pf(x, k-1) * f
{
    assert Pot(x, k) == Pot(x, k-1) * x;

    assert Fact(k) == k * Fact(k-1);
}

// PF converge a 0
lemma FactVsPot(x : real, e : real)
    requires x > 0.0
    requires e > 0.0

    ensures ∃ k : nat • Pf(x, k) < e
{
    // Un n0 tal que a partir de entonces P/F
    // decrezca (divida entre dos) al aumentar n
    var n0 : nat := 2 * (x.Trunc + 1);

    var factor := x / real(n0);
    var origen := Pf(x, n0);

    // Demuestra que el n0 está bien escogido
    calc {
        factor;
        == // Definición de factor
        x / real(n0);
        == // Definición de n0
        x / (2.0 * real(x.Trunc + 1));
        < // x es menor que su parte entera más uno
        {
            assert real(x.Trunc + 1) > x;

            /* Descomentar el siguiente assert impide que se
               demuestre este paso y provoca un error en el
               calc de más abajo en ciertas versiones de Dafny.
               En la última versión estable es imprescindible.
             */
        //  assert 2.0 * real(x.Trunc + 1) > 2.0 * x;
        }
        x / (2.0 * x);
        == // Cancelan las x
        0.5;
    }

    // Acota el número máximo de iteraciones que hay que hacer
    // suponiendo que P/F decrece por el «factor» cuando en
    // realidad decrece más rápidamente
    AcotaPotencia(factor, e / origen);

    var itmax : nat :| origen * Pot(factor, itmax) < e;


    // Variables del bucle
    var pfreal, pfcota, k := origen, origen, n0;

    while pfreal ≥ e
        decreases itmax + n0 - k
        invariant n0 ≤ k ≤ n0 + itmax
        invariant k == n0 + itmax ⟹ pfreal < e

        invariant pfreal == Pf(x, k)
        invariant pfcota == origen * Pot(factor, k - n0)

        invariant pfreal ≤ pfcota

        invariant factor == x / real(n0)
    {
        // Aumenta el índice de Pf (el exponente de la cota)
        k := k + 1;
        // Guarda pfreal y pfcota para uso posterior
        var pfreal0, pfcota0 := pfreal, pfcota;

        // Factor real por el que cambia Pf
        var facreal := x / real(k);


        // Actualización de pfreal y pfcota y demostración de que
        // siguen cumpliendo sus invariantes definitorios

        // ** pfreal
        pfreal := pfreal * facreal;

        AvancePf(x, k, facreal);

        // ** pfcota
        pfcota := pfcota * factor;

        calc == {
            pfcota;
            // Asignación anterior
            pfcota0 * factor;
            // Invariante sobre pfcota0
            { assert pfcota == origen * Pot(factor, k - n0); }
            origen * Pot(factor, k - 1 - n0) * factor;
            // Suma de exponentes
            origen * Pot(factor, k - n0);
        }

        // Prueba que pfcota acota a pfreal
        //FactorDecreciente(x, k, n0);

        assert k > n0;

        AcotacionProducto(pfreal0, pfcota0, facreal, factor);


        assert pfreal ≤ pfcota;

        // Demostración del primer y segundo invariante
        // (llegado al límite no se dan más vueltas)
        if k == n0 + itmax {
            calc {
                pfreal;
                // Acota por pfcota
                ≤ pfcota;
                // Invariante sobre pdfcota
                // Valor de k == n0 + itmax
                origen * Pot(factor, itmax);
                // Definición de itmax
                < e;
            }
        }
    }
}
```

A.4 Floyd-Warshall algorithm

```
/**
 * Floyd-Warshall algorithm (1959)
 *
 * Minimum path cost in a negative cycle free directed graph.
 *
 * The absence of negative cycles is set as a precondition.
 * However it can be modified to detect the presence of
 * negative cycles.
 *
 * Taken from:
 * https://en.wikipedia.org/wiki/-FloydWarshall_algorithm&oldid=710345047
 */
```

```
/**
 * A special cost value (infinity) is used to point out that there is no
 * connection between two nodes. Dafny doesn't support "extended real numbers"
 * or something similar so an algebraic datatype is used.
 *
 * Floyd-Warshall algorithm let edge costs be negative, then using negative
 * numbers as infinity is not a choice.
 *
 * A real implementation could use floating point arithmetic according to the
 * IEEE 754 standard which provides special values for infinities.
 */

// Extended real numbers with (positive) infinity
datatype xreal = Real(value : real) | Infty

// Type synonym for a graph (a nxn matrix of xreals)
type xgraph = array2<xreal>

// Less or equal on xreals
predicate method Leq(x : xreal, y : xreal)
{
    match y
    {
        case Infty ⇒ true

        case Real(yv) ⇒ match x
            {
                case Infty ⇒ false

                case Real(xv) ⇒ xv ≤ yv
            }
    }
}

function method Add(x : xreal, y : xreal) : xreal
{
    if x.Infty? ∨ y.Infty?
    then Infty
    else Real(x.value + y.value)
}


/**
 * Path formalization.
 *
 *
 * The key definition is "path". Paths are represented as sequences of vertex
 * indices (non negative integers) and intermediate nodes cannot appear twice
 * (while ends may agree).
 *
 * Due to way the Floyd-Warshall algorithm works, a upper bound for
 * intermediate vertices has has been added to the definition.
 */

predicate Walk(w : seq<int>, bound : nat)
{
    // At least two vertices (an edge)
    |w| ≥ 2
    ∧
    // Intermediate nodes are valid and not higher than bound
    ∀ i | 0 < i < |w|-1 • 0 ≤ w[i] < bound
}

predicate WalkIn(w : seq<int>, bound : nat, edges : xgraph)
    reads edges
    requires ValidGraph(edges)
{
    bound ≤ Size(edges)
    ∧
    Walk(w, bound)
    ∧
    0 ≤ Start(w) < Size(edges)
    ∧
    0 ≤ End(w) < Size(edges)
}


predicate AnyPath(s : seq<int>, bound : nat)
{
    // It is a walk
    Walk(s, bound)
    ∧
    // And it doesn't contain inner loops
    ∀ i, j | 0 ≤ i < |s| ∧ 0 < j < |s|-1 •
        i ≠ j ⟹ s[i] ≠ s[j]
}

function Start(w : seq<int>) : int
    requires w ≠ []
{
    w[0]
}

function End(w : seq<int>) : int
    requires w ≠ []
{
    w[|w|-1]
}

predicate Path(s : seq<int>, from : nat, to : nat, bound : nat)
{
    AnyPath(s, bound) ∧ Start(s) = from ∧ End(s) = to
}

function Size(graph : xgraph) : nat
    reads graph
    requires graph ≠ null
{
    graph.Length0
}


// The cost of travel thought a path
// (it does not require that the path is actually a path)
function Cost(s : seq<int>, edges : xgraph) : xreal
    reads edges
    requires edges ≠ null
    requires edges.Length0 = edges.Length1

    requires ∀ i | 0 ≤ i < |s|  •  0 ≤ s[i] < Size(edges)
{
    if |s| < 2 then Real(0.0) else Add(edges[s[0], s[1]],
        Cost(s[1..], edges))
}

/*
 * Graph are represented as square matrices of costs (xreal
 * numbers) with no negative cycles.
 */
predicate ValidGraph(edges : xgraph)
    reads edges
{
    edges ≠ null
    ∧
    edges.Length0 = edges.Length1
    ∧
    (∀ node | 0 ≤ node < Size(edges) •
        edges[node, node] = Real(0.0))
    ∧
    // No negative cycles
    (∀ path, bound : nat |
        AnyPath(path, bound)
        ∧
        bound ≤ Size(edges)
        ∧
        Start(path) = End(path)
        ∧
        0 ≤ Start(path) < Size(edges)
      •
        Leq(Real(0.0), Cost(path, edges))
    )
}

/*
 * 'cost' is the minimum cost of any path in 'graph' from 'from' to 'to'
 * using nodes whose index is lower than 'bound'
 */
predicate MinCost(from : nat, to : nat, graph : xgraph,
        bound : nat, cost : xreal)

    reads graph
    requires ValidGraph(graph)

    requires bound ≤ Size(graph)
    requires from < Size(graph) ∧ to < Size(graph)
{
    (∀ path | Path(path, from, to, bound) •
        Leq(cost, Cost(path, graph)))
    ∧
    ∃ path | Path(path, from, to, bound) • cost = Cost(path, graph)
}


method Floyd(edges : xgraph) returns (dist : xgraph)
    requires ValidGraph(edges)

    requires Size(edges) > 0

    ensures dist ≠ null
    ensures dist.Length0 = Size(edges)
    ensures dist.Length1 = Size(edges)

    ensures ∀ i, j | 0 ≤ i < edges.Length0 ∧ 0 ≤ j < edges.Length0 •
        MinCost(i, j, edges, edges.Length0, dist[i, j])
{

    // Initialization
    var n := edges.Length0;

    dist := new xreal[n, n];

    var i : nat, j : nat, k : nat;

    j := 0;

    while j < n
        invariant 0 ≤ j ≤ n

        invariant ∀ r, s | 0 ≤ r < n ∧ 0 ≤ s < j •
            dist[r, s] = edges[r, s]
    {
        i := 0;

        while i < n
            invariant 0 ≤ i ≤ n
            invariant 0 ≤ j ≤ n

            invariant ∀ r, s | 0 ≤ r < n ∧ 0 ≤ s < j •
                dist[r, s] = edges[r, s]

            invariant ∀ r | 0 ≤ r < i •
```

```
                dist[r, j] = edges[r, j]
        {
            dist[i, j] := edges[i, j];

            i := i + 1;
        }

        j := j + 1;
    }

    // The algorithm itself: iterative refinement of upper bounds

    k := 0;

    // Lemma to prove that initial values are correct
    Initialization(edges);

    while k < n
        invariant 0 ≤ k ≤ n

        invariant ∀ r, s | 0 ≤ r < n ∧ 0 ≤ s < n  •
            MinCost(r, s, edges, k, dist[r, s])
    {
        j := 0;

        while j < n
            invariant 0 ≤ j ≤ n

            // Preserve
            invariant ∀ r, s | 0 ≤ r < n ∧ j ≤ s < n  •
                MinCost(r, s, edges, k, dist[r, s])

            // Advance
            invariant ∀ r, s | 0 ≤ r < n ∧ 0 ≤ s < j  •
                MinCost(r, s, edges, k+1, dist[r, s])
        {
            i := 0;

            while i < n
                invariant 0 ≤ j < n
                invariant 0 ≤ i ≤ n

                // Preserve
                invariant ∀ r, s | 0 ≤ r < n ∧ j+1 ≤ s < n  •
                    MinCost(r, s, edges, k, dist[r, s])

                invariant ∀ r | i ≤ r < n  •
                    MinCost(r, j, edges, k, dist[r, j])

                invariant ∀ r, s | 0 ≤ r < n ∧ 0 ≤ s < j  •
                    MinCost(r, s, edges, k+1, dist[r, s])

                // Advance
                invariant ∀ r | 0 ≤ r < i  •
                    MinCost(r, j, edges, k+1, dist[r, j])

            {
                /*
                 * dist contains at the same time minimum costs
                 * with bound k and k+1. Still and all, where k
                 * is an end those minimums coincide.
                 */

                MinPathBound(i, k, j, edges);

                assert MinCost(i, k, edges, k, dist[i, k]);
                assert MinCost(k, j, edges, k, dist[k, j]);
                assert MinCost(i, j, edges, k, dist[i, j]);


                var through_k := Add(dist[i, k], dist[k, j]);
                // It worth taking the path through k
                if ¬Leq(dist[i, j], through_k)
                {
                    UpdateBetter(i, k, j, dist[i, k], dist[k, j],
                        dist[i, j], edges);

                    dist[i, j] := through_k;
                }
                else {
                    UpdateSame(i, k, j, dist[i, k], dist[k, j],
                        dist[i, j], edges);
                }

                assert MinCost(i, j, edges, k+1, dist[i, j]);

                // The following seems to be unavoidable. Why?

                assert ∀ r | r = i  •  dist[i, j] = dist[r, j];

                i := i + 1;
            }

            j := j + 1;
        }

        k := k + 1;
    }
}



/*
 * Auxiliary lemmas
 */

lemma Initialization(edges : xgraph)
    requires ValidGraph(edges)

    ensures ∀ i, j | 0 ≤ i < edges.Length0 ∧ 0 ≤ j < edges.Length0  •
        MinCost(i, j, edges, 0, edges[i, j])
{
    // In this case is not even necessary to invoke the absence
    // of negative cycles

    ∀ i, j | 0 ≤ i < edges.Length0 ∧ 0 ≤ j < edges.Length0
        ensures MinCost(i, j, edges, 0, edges[i, j])
    {
        // There is a path whose cost is edges[i, j]

        var thePath := [i, j];

        assert Path(thePath, i, j, 0);
        assert Cost(thePath, edges) = edges[i, j];


        // And is the only one

        ∀ path | Path(path, i, j, 0)
            ensures path = thePath
        {
            if |path| > 2 {
                var med := path[1];

                assert 0 ≤ path[1] < 0;
            }
        }
    }
}
```

```
/*
 * A stretch of a path is a path itself.
 */
lemma PathSplitArePath(path : seq<int>, start : nat, end : nat,
    i : nat, bound : nat)

    requires Path(path, start, end, bound)

    requires 0 < i < |path|-1

    ensures Path(path[..i+1], start, path[i], bound)
    ensures Path(path[i..], path[i], end, bound)
{

}

/*
 * The total cost of a path can be summed by sections.
 *
 * We don't really require that 'path' is a path, taking
 * advantage of the lack of this restriction in PathCost.
 */
lemma PathSplitCost(path : seq<int>, i : nat, edges : xgraph)
    requires ValidGraph(edges)
    requires 0 ≤ i < |path|

    requires ∀ k | 0 ≤ k < |path| • 0 ≤ path[k] < edges.Length0

    ensures Add(Cost(path[..i+1], edges), Cost(path[i..], edges))
        = Cost(path, edges)

    decreases i
{
    if i = 0 {
        // Base case
        return;
    }
    else if i = |path|-1
    {
        assert path[..i+1] = path;
        assert Cost(path[i..], edges) = Real(0.0);

        return;
    }

    calc {
        Cost(path, edges);

        Add(edges[path[0], path[1]], Cost(path[1..], edges));
        { PathSplitCost(path[1..], i-1, edges); }

        Add(edges[path[0], path[1]], Add(Cost(path[1..][..i], edges),
            Cost(path[1..][i-1..], edges)));
        {
            assert path[1..][i-1..] = path[i..];
            assert path[1..][..i] = path[1..i+1];
        }
        Add(edges[path[0], path[1]], Add(Cost(path[1..i+1], edges),
            Cost(path[i..], edges)));

        Add(Add(edges[path[0], path[1]], Cost(path[1..i+1], edges)),
            Cost(path[i..], edges));

        Add(Add(edges[path[0], path[1]], Cost(path[..i+1][1..], edges)),
            Cost(path[i..], edges));

        Add(Cost(path[..i+1], edges), Cost(path[i..], edges));
    }

}
```

```
/*
 * The sum of the two previous lemmas.
 */
lemma PathSplit(path : seq<int>, start : nat, end : nat,
    i : nat, bound : nat, edges : xgraph)

    requires ValidGraph(edges)
    requires Path(path, start, end, bound)

    requires 0 < i < |path|-1
    requires bound ≤ edges.Length0
    requires start < edges.Length0
    requires end < edges.Length0

    ensures Path(path[..i+1], start, path[i], bound)
    ensures Path(path[i..], path[i], end, bound)

    ensures Add(Cost(path[..i+1], edges), Cost(path[i..], edges))
        = Cost(path, edges)
{
    PathSplitArePath(path, start, end, i, bound);
    PathSplitCost(path, i, edges);
}

/*
 * A concatenation of two path is a path under certain conditions:
 *  1. Ends where they join match.
 *  2. Both are optimal paths.
 *  3. The optimal cost between the start and end (avoiding the contact
 *     vertex) exceeds the sum of the costs of the two paths.
 *
 * A easy handwritten proof can be done by reductio ad absurdum, finding
 * a path which, avoiding k, has a cost lower than cij, against (3).
 * However, a "easier" proof has been applied instead.
 */

lemma WalkLoop(walk : seq<int>, edges : xgraph, k : nat, r : nat, s : nat,
    loop : seq<int>, cutoff : seq<int>)

    requires ValidGraph(edges)
    requires WalkIn(walk, k, edges)

    // There are two stops in the walk that
    // share the same vertex
    requires s < |walk|
    requires r < s
    requires ¬(r = 0 ∧ s = |walk|-1)

    requires walk[r] = walk[s]

    requires loop = walk[r..s+1]
    requires cutoff = walk[..r] + walk[s..]

    ensures Cost(walk, edges) = Add(Cost(loop, edges), Cost(cutoff, edges))
{
    /*
     * A tedious calculation.
     * Dafny alone can deal with the two first postconditions.
     */
    calc {
        Cost(walk, edges);

        // First split
        { PathSplitCost(walk, r, edges); }
        Add(Cost(walk[..r+1], edges), Cost(walk[r..], edges));

        // Second split, we have 3 sections
        { PathSplitCost(walk[r..], s-r, edges); }
        Add(Cost(walk[..r+1], edges), Add(Cost(walk[r..][..s-r+1], edges),
            Cost(walk[r..][s-r..], edges)));
        // Slice simplification and loop = walk[r..s+1]
        {
            assert walk[r..][..s-r+1] = walk[r..s+1];
            assert walk[r..][s-r..] = walk[s..];
        }
        Add(Cost(walk[..r+1], edges), Add(Cost(loop, edges),
            Cost(walk[s..], edges)));

        // Add is associative and commutative
        Add(Cost(loop, edges), Add(Cost(walk[..r+1], edges),
            Cost(walk[s..], edges)));

        // That's cutoff
        {
            // Reduces verification time a lot (~4s)
            assert walk[r] = walk[s];

            assert cutoff[..r+1] = walk[..r+1];
            assert cutoff[r..] = walk[s..];

            PathSplitCost(cutoff, r, edges);
        }
        Add(Cost(loop, edges), Cost(cutoff, edges));
    }

}

/*
 * Given a walk between two vertices there is a path between those vertices
 * whose cost is lower than that of the walk.
 */
lemma WalkReduce(walk : seq<int>, k : nat, edges : xgraph)
    requires ValidGraph(edges)

    requires WalkIn(walk, k, edges)

    ensures ∃ path • Path(path, Start(walk), End(walk), k)
        ∧ Leq(Cost(path, edges), Cost(walk, edges))

    decreases |walk|
{
    // If walk is a path there is nothing to do
    if AnyPath(walk, k) {
        assert Path(walk, Start(walk), End(walk), k);

        return;
    }

    // When walk is not a path...

    var r, s :| 0 ≤ r < |walk| ∧ 0 < s < |walk|-1 ∧ r ≠ s ∧
        walk[r] = walk[s];

    // Without loss of generality, suppose r < s
    if s < r { r, s := s, r; }

    // There is at least a loop. Lets remove it.

    // At least one of {r, s} is an inner index
    assert 0 < s < |walk|-1 ∨ 0 < r < |walk|-1;

    // Breaks the walk
    var loop := walk[r..s+1];
    var cutoff := walk[..r] + walk[s..];

    assert Start(loop) = End(loop);
    assert WalkIn(loop, k, edges);

    assert Start(walk) = Start(cutoff);

    var cutoff_cost := Cost(cutoff, edges);
    //var loop_cost := Cost(loop, edges);
    var walk_cost := Cost(walk, edges);

    WalkLoop(walk, edges, k, r, s, loop, cutoff);

    // Loop has non negative cost so Cost(cutoff) ≤ Cost(walk)
    LoopCosts(loop, k, edges);

    assert Leq(cutoff_cost, walk_cost);

    // Induction (as cutoff is strictly smaller)
    WalkReduce(cutoff, k, edges);

    var path :| Path(path, Start(cutoff), End(cutoff), k)
        ∧ Leq(Cost(path, edges), cutoff_cost);

    assert End(walk) = End(cutoff);
    assert Start(walk) = Start(cutoff);
}


/*
 * The minimum cost of a path which starts or ends in k using 0..k vertices
 * coincides with that of a path whose vertices are vertices in 0..k-1.
 * (because k cannot appear as an intermediate vertex).
 */
lemma MinPathBound(i : nat, k : nat, j : nat, edges : xgraph)
    requires ValidGraph(edges)

    requires i < edges.Length0
    requires j < edges.Length0
    requires k < edges.Length0

    ensures ∀ cik • MinCost(i, k, edges, k+1, cik) ⟹
        MinCost(i, k, edges, k, cik)
    ensures ∀ ckj • MinCost(k, j, edges, k+1, ckj) ⟹
        MinCost(k, j, edges, k, ckj)
{
}


/*
 * Two lemmas to help proving that the update (or the lack of it) in the
 * innermost loop is correct.
 */

lemma UpdateBetter(i : nat, k : nat, j : nat, cik : xreal,
    ckj : xreal, cij : xreal, edges : xgraph)

    requires ValidGraph(edges)

    requires i < edges.Length0
    requires j < edges.Length0
    requires k < edges.Length0

    requires MinCost(i, k, edges, k, cik)
    requires MinCost(k, j, edges, k, ckj)
    requires MinCost(i, j, edges, k, cij)

    requires ¬Leq(cij, Add(cik, ckj))

    ensures MinCost(i, j, edges, k+1, Add(cik, ckj))
{
    // The cost of any path is higher or equal to cik + ckj

    ∀ path | Path(path, i, j, k+1)
        ensures Leq(Add(cik, ckj), Cost(path, edges))
    {
        MinPathBound(i, k, j, edges);

        if Path(path, i, j, k)
```

```
            {
                // Definition of cij and cik + ckj < cij

                assert Leq(cij, Cost(path, edges));
            }
            else {
                // Due to MinPathBound (otherwise Path(path, i, j, k))
                assert i ≠ k ∧ j ≠ k;

                var ind : nat :| 0 < ind < |path|-1 ∧ path[ind] = k;

                /*
                 * The split subpaths costs are higher than cik and ckj
                 * respectively and the total cost is the sum.
                 *
                 * So all paths bounded by k+1 have a cost bounded by
                 * cik + ckj.
                 */
                PathSplit(path, i, j, ind, k+1, edges);
            }
        }

        // There is a path whose cost is cik + ckj

        var pik : seq<int> :| Path(pik, i, k, k) ∧ Cost(pik, edges) = cik;
        var pkj : seq<int> :| Path(pkj, k, j, k) ∧ Cost(pkj, edges) = ckj;

        var wij := pik + pkj[1..];

        WalkReduce(wij, k+1, edges);

        var pij :| Path(pij, i, j, k+1) ∧ Leq(Cost(pij, edges), Cost(wij, edges));

        PathSplitCost(wij, |pik|-1, edges);

    //  assert Cost(wij, edges) = Add(cik, ckj);
    //  assert Cost(pij, edges) = Add(cik, ckj);
    }

    lemma UpdateSame(i : nat, k : nat, j : nat, cik : xreal,
        ckj : xreal, cij : xreal, edges : xgraph)

        requires ValidGraph(edges)

        requires i < edges.Length0
        requires j < edges.Length0
        requires k < edges.Length0

        requires MinCost(i, k, edges, k, cik)
        requires MinCost(k, j, edges, k, ckj)
        requires MinCost(i, j, edges, k, cij)

        requires Leq(cij, Add(cik, ckj))

        ensures MinCost(i, j, edges, k+1, cij)
    {
        // The cost of any path is higher or equal to cij

        ∀ path | Path(path, i, j, k+1)
            ensures Leq(cij, Cost(path, edges))
        {
            MinPathBound(i, k, j, edges);

            if Path(path, i, j, k)
            {
                // Definition of 'cij'
                assert Leq(cij, Cost(path, edges));
            }
            else {
                // Due to MinPathBound (otherwise Path(path, i, j, k))
                assert i ≠ k ∧ j ≠ k;
                var ind : nat :| 0 < ind < |path|-1 ∧ path[ind] = k;

                /*
                 * The splitted subpathes costs are higher than cik and ckj
                 * respectively and the total cost is the sum, in the other
                 * hand cij ≤ cik + ckj.
                 *
                 * So all paths bounded by k+1 have a cost bounded by cij.
                 */
                PathSplit(path, i, j, ind, k+1, edges);
            }
        }
    }

    // The minimum path cost is also the minimum walk cost
    // between two given nodes
    lemma MinWalkCost(walk : seq<int>, bound : nat, cost : xreal, edges : xgraph)
        requires ValidGraph(edges)
        requires WalkIn(walk, bound, edges)
        requires MinCost(Start(walk), End(walk), edges, bound, cost)

        ensures Leq(cost, Cost(walk, edges))

        decreases |walk|
    {
        if AnyPath(walk, bound)
        {
            // Base case (true by definition of MinCost)
        }
        else
        {
            // If it's not a path, there are repeated nodes
            assert ∃ r, s | 0 ≤ r < |walk| ∧ 0 < s < |walk|-1 •
                r ≠ s ∧ walk[r] = walk[s];

            var r, s :| 0 ≤ r < |walk| ∧ 0 < s < |walk|-1 ∧ r ≠ s ∧ walk[r] = walk[s];

            // Suppose r ≤ s without loss of generality
            if s < r
            {
                r, s := s, r;
            }

            // Any of them is an inner index
            assert 0 < r < |walk|-1 ∨ 0 < s < |walk|-1;

            // Split the path in a loop and the rest
            var loop := walk[r..s+1];
            var cutoff := walk[..r] + walk[s..];

            WalkLoop(walk, edges, bound, r, s, loop, cutoff);

            assert Walk(loop, bound);

            // Induction using min cost for a loop is 0
            StaticCost(Start(loop), bound, edges);

            MinWalkCost(loop, bound, Real(0.0), edges);

            // Then Cost(walk) = Cost(loop) + Cost(cutoff)
            // ≥ Cost(cutoff)

            // Induction again cost ≤ Cost(cutoff) ≤ Cost(walk)

            assert Walk(cutoff, bound);
            assert Start(cutoff) = Start(walk);
            assert End(cutoff) = End(walk);

            MinWalkCost(cutoff, bound, cost, edges);
        }
```

```
}

// The minimum cost from a vertex to itself is 0
lemma StaticCost(i : nat, bound : nat, edges : xgraph)
    requires ValidGraph(edges)

    requires bound ≤ Size(edges)

    requires i < Size(edges)

    ensures MinCost(i, i, edges, bound, Real(0.0))
{
    var zeropath := [i, i];

    assert Cost(zeropath, edges) = Real(0.0);
}

// Loops have non negative cost even if they aren't paths
lemma LoopCosts(loop : seq<int>, bound : nat, edges : xgraph)
    requires ValidGraph(edges)
    requires WalkIn(loop, bound, edges)
    requires Start(loop) = End(loop)

    ensures Leq(Real(0.0), Cost(loop, edges))
{
    StaticCost(Start(loop), bound, edges);

    MinWalkCost(loop, bound, Real(0.0), edges);
}
```

A.5 Dijkstra's algorithm

```
/**
 * Dijkstra algorithm (Edsger Dijkstra, 1959)
 */

/*
 * Abstract graph to allow easy implementation as an adjacency list.
 *
 * Count() provides the number of vertices and the Adjacents(nat)
 * function returns a map whose key are the neighbour vertices and
 * its value is the cost of the edge which connect them.
 */
trait Graph
{
    function method Count() : nat

    function method Adjacents(n : nat) : map<int, real>
        reads this
        requires 0 ≤ n < Count()

        ensures n ∉ Adjacents(n)

        ensures ∀ node | node in Adjacents(n) •
            0 ≤ node < Count()
            ∧
            Adjacents(n)[node] ≥ 0.0
}

/*
 * Improved priority queue.
 *
 * - Priorities can be updated.
 * - There are a key and a value.
 * - A map is used as its representation.
 */
class PrioQueue
{
    // Creates the empty queue
    constructor()
```

```
        ensures Elems = map[]
        modifies this

    // Is the queue empty?
    predicate method Empty()
        reads this
        ensures Empty()  ⟺  Elems = map[]

    // Sets or update a queue value
    method Set(key : nat, value : real)
        ensures Elems = old(Elems)[key := value]
        modifies this

    // Gets the minimum entry in the queue
    method Pop() returns (key : nat, value : real)
        requires ¬Empty()

        ensures key in old(Elems)
        ensures value = old(Elems)[key]

        ensures ∀ k | k in old(Elems) •  old(Elems)[k] ≥ value

        // Removes the the element from the table of elements
        ensures Elems = map k | k in old(Elems) ∧ k ≠ key  •  old(Elems)[k]
        modifies this

    // Abstract representation
    ghost var Elems : map<int, real>
}

predicate ValidGraph(graph : Graph)
    reads graph
{
    graph ≠ null
    ∧
    graph.Count() > 0
}

/*
 * A walk is a sequence of connected vertices represented by their indices.
 *
 * They must be non-empty. Singleton walks are allowed.
 */
predicate Walk(w : seq<int>, graph : Graph)
    reads graph
    requires ValidGraph(graph)
{
    // At least one element
    w ≠ []
    ∧
    // Walk vertices are in bounds
    (∀ i | 0 ≤ i < |w|  •  0 ≤ w[i] < graph.Count())
    ∧
    // The nodes are connected
    (∀ i | 0 ≤ i < |w|-1  •  w[i+1] in graph.Adjacents(w[i]))
}

// It says: is w a walk from 0 to node within graph?
predicate WalkTo(node : nat, w : seq<int>, graph : Graph)
    reads graph

    requires ValidGraph(graph)
    requires 0 ≤ node < graph.Count()
{
    // A walk from 0 to node
    Walk(w, graph)
    ∧
    w[0] = 0
    ∧
    w[|w|-1] = node
```

```
}

// The cost of a walk is the sum of the cost of its composing edges
function WalkCost(w : seq<int>, graph : Graph) : real
    reads graph

    requires ValidGraph(graph)
    requires Walk(w, graph)
{
    if |w| = 1 then 0.0 else
        graph.Adjacents(w[|w|-2])[w[|w|-1]] + WalkCost(w[..|w|-1], graph)
}


/*
 * About distances.
 */

// It says: dist is the distance to node in graph.
predicate DistanceTo(dist : real, node : nat, graph : Graph)
    reads graph

    requires ValidGraph(graph)
    requires 0 ≤ node < graph.Count()
{
    // Its a lower bound of the cost of any path
    (∀ walk | WalkTo(node, walk, graph) •
        dist ≤ WalkCost(walk, graph))
    ∧
    // And there is a path with this cost
    (∃ walk • WalkTo(node, walk, graph)
        ∧ WalkCost(walk, graph) = dist)
}


/*
 * Definitions to ensure the correction of the list of links to previous
 * nodes in the calculated path from the origin.
 *
 * The predicate is enunciated in two steps and it states that there is a
 * prev cell for each node, whose value could be -1 (if the node is not
 * connected with the origin, or not yet) or the index of the previous nodes
 * in an optimal path from the origin.
 *
 * Every non negative cell let return back to the origin and so reconstruct
 * a path from there.
 */

predicate ValidPrev(prev : seq<int>, graph : Graph)
    reads graph
    requires ValidGraph(graph)
{
    BasicValidPrev(prev, graph)
    ∧
    ∀ i | 0 ≤ i < |prev| • prev[i] ≥ 0 ⟹
        BackToZero(i, prev, graph, graph.Count())
}

predicate BasicValidPrev(prev : seq<int>, graph : Graph)
    reads graph
    requires ValidGraph(graph)
{
    |prev| = graph.Count()
    ∧
    prev[0] < 0
    ∧
    (∀ i | 1 ≤ i < |prev| • prev[i] < 0 ∨ (0 ≤ prev[i] < graph.Count()
        ∧ i in graph.Adjacents(prev[i])))
}

// It is possible to go back to zero in a bounded number of steps
```

```
predicate BackToZero(i : nat, prev : seq<int>, graph : Graph, steps : nat)
    reads graph

    requires ValidGraph(graph)
    requires BasicValidPrev(prev, graph)
    requires 0 ≤ i < graph.Count()

    decreases steps
{
    steps > 0 ∧ (i = 0 ∨ (prev[i] ≥ 0 ∧
        BackToZero(prev[i], prev, graph, steps - 1)))
}

/*
 * Function methods to get the path from the origin based on prev.
 *
 * The auxiliary function includes an extra parameter
 * to prove termination.
 */

function method WalkFromPrev(i : nat, prev : seq<int>, graph : Graph) : seq<int>
    reads graph
    requires ValidGraph(graph)
    requires ValidPrev(prev, graph)
    requires 0 ≤ i < graph.Count()

    requires BackToZero(i, prev, graph, graph.Count())

    ensures WalkTo(i, WalkFromPrev(i, prev, graph), graph)
{
    WalkFromPrevAux(i, prev, graph, graph.Count())
}

function method WalkFromPrevAux(i : nat, prev : seq<int>,
    graph : Graph, ghost bound : nat) : seq<int>

    reads graph
    requires ValidGraph(graph)
    requires ValidPrev(prev, graph)
    requires 0 ≤ i < graph.Count()

    requires BackToZero(i, prev, graph, bound)

    ensures WalkTo(i, WalkFromPrevAux(i, prev, graph, bound), graph)

    decreases bound
{
    (if prev[i] ≥ 0
    then WalkFromPrevAux(prev[i], prev, graph, bound-1)
    else []) + [i]
}

/**
 * Computes the minimum path from the origin (numbered as 0) to the rest of
 * vertices.
 *
 * dist and prev will contain the distance from the origin and the previous
 * node in an optimal path from it for each node in the graph. When a node
 * is unconnected to the origin -1 will be written in both cells.
 */
method Dijkstra(graph : Graph) returns (dist : array<real>, prev : array<int>)
    requires ValidGraph(graph)

    ensures dist ≠ null
    ensures dist.Length = graph.Count()

    ensures prev ≠ null
    ensures ValidPrev(prev[..], graph)

    ensures ∀ id | 0 ≤ id < graph.Count() • dist[id] ≥ 0.0
        ⟹ DistanceTo(dist[id], id, graph)
    ensures ∀ id | 0 ≤ id < graph.Count() • dist[id] ≥ 0.0
        ⟹ dist[id] =
           WalkCost(WalkFromPrev(id, prev[..], graph), graph)

    ensures ∀ id | 0 < id < graph.Count() • dist[id] < 0.0
        ⟹ ¬(∃ walk • WalkTo(id, walk, graph))
{
    var queue := new PrioQueue();
    var closed  := new bool[graph.Count()];

    dist := new real[graph.Count()];
    prev := new int[graph.Count()];

    // Initialization

    var i := 0;

    while i < graph.Count()
        invariant 0 ≤ i ≤ graph.Count()

        invariant ∀ j | 0 ≤ j < i • dist[j] = -1.0
        invariant ∀ j | 0 ≤ j < i • prev[j] = -1
        invariant ∀ j | 0 ≤ j < i • ¬closed[j]

        invariant queue.Elems = map[]
    {
        dist[i] := -1.0;
        prev[i] := -1;
        closed[i] := false;

        i := i + 1;
    }

    queue.Set(0, 0.0);

    dist[0] := 0.0;

    ghost var npend := closed.Length;

    AllFalse(closed[..]);

    Initialization(graph);

    while ¬queue.Empty()

        invariant npend = CountFalses(closed[..])

        // About the final results

        // The distance is negative iff the node has not been visited yet
        invariant ∀ id | 0 < id < graph.Count() •
            dist[id] < 0.0  ⟺  IsOutside(id, graph, closed[..])

        // Nodes which have been visited but are not yet closed, have the
        // nearest distance from the closed nodes assigned
        invariant ∀ id | 0 < id < graph.Count() • dist[id] ≥ 0.0
             ⟺  BestApproach(id, graph, closed[..], dist[..], prev[..])

        // For the closed nodes, the final distance is already calculated
        invariant ∀ id | 0 ≤ id < graph.Count() • closed[id]
            ⟹ DistanceTo(dist[id], id, graph)

        invariant 0 ∉ queue.Elems  ⟹  closed[0]

        // Prev is valid
        invariant ValidPrev(prev[..], graph)

        // About the queue and its values

        invariant ∀ id | id in queue.Elems • 0 ≤ id < graph.Count()
        invariant ∀ id | 0 ≤ id < graph.Count() •
            dist[id] ≥ 0.0 ∧ ¬closed[id]  ⟺  id in queue.Elems

        invariant ∀ id | id in queue.Elems  •  dist[id] = queue.Elems[id]

        invariant ∀ id, c | id in queue.Elems ∧ 0 ≤ c < graph.Count()
            ∧ closed[c]  •  dist[c] ≤ queue.Elems[id]

        decreases npend
    {
        // Keeps a copy of the old close
        ghost var closed0 := closed[..];

        // Gets the nearer node from the queue
        var nearer, nearcost := queue.Pop();

        // Closes the node
        closed[nearer] := true;

        npend := npend - 1;

        // Some proofs for the initialization of invariants

        CancelOne(closed0, nearer);

        var adjacents := set node | node in graph.Adjacents(nearer);

        StillOutside(nearer, graph , closed0, dist[..], adjacents);

        assert ∀ id | 0 < id < graph.Count() ∧ id ∉ adjacents  •
            dist[id] < 0.0  ⟺  IsOutside(id, graph, closed[..]);

        assert nearer ∉ queue.Elems;

        // Checks every (not closed) adjacent node for a lower
        // distance to it

        while |adjacents| > 0
            invariant npend = CountFalses(closed[..]);

            // About the final results

            invariant ∀ id | 0 < id < graph.Count() ∧ id ∉ adjacents  •
                dist[id] < 0.0  ⟺  IsOutside(id, graph, closed[..])

            invariant ∀ id | 0 < id < graph.Count() ∧ id in adjacents  •
                dist[id] < 0.0  ⟺  IsOutside(id, graph, closed0)

            invariant ∀ id | 0 < id < graph.Count() ∧ id ∉ adjacents  •
                dist[id] ≥ 0.0   ⟺
                BestApproach(id, graph, closed[..], dist[..], prev[..])

            invariant ∀ id | 0 < id < graph.Count() ∧ id in adjacents  •
                dist[id] ≥ 0.0  ⟺
                BestApproach(id, graph, closed0, dist[..], prev[..])

            invariant ∀ id | 0 ≤ id < graph.Count()  •  closed[id]
                 ⟹  DistanceTo(dist[id], id, graph)

            invariant 0 ∉ queue.Elems  ⟹  closed[0]

            invariant ValidPrev(prev[..], graph)

            // About the queue and its values

            invariant ∀ id | id in queue.Elems  •  0 ≤ id < graph.Count()

            invariant ∀ id | 0 ≤ id < graph.Count()  •
                dist[id] ≥ 0.0 ∧ ¬closed[id]  ⟺  id in queue.Elems
            invariant ∀ id | id in queue.Elems  •  dist[id] = queue.Elems[id]

            invariant ∀ id, c | id in queue.Elems ∧ 0 ≤ c < graph.Count() ∧
                closed[c]  •  dist[c] ≤ queue.Elems[id]

            decreases |adjacents|
        {
            var next :| next in adjacents;

            var cost := graph.Adjacents(nearer)[next];

            if ¬closed[next] ∧ dist[next] > nearcost + cost
            {
                // Updates the distance and antecedent

                dist[next] := nearcost + cost;
                prev[next] := nearer;

                queue.Set(next, dist[next]);
            }

            i := i + 1;

            adjacents := adjacents - {next};
        }
    }

    Unreachables(graph, closed[..], dist[..]);
}

/*
 * Functions for readability
 */

// The vertex 'id' is outside, i.e. it is not reachable from a
// closed node
predicate IsOutside(id : nat, graph : Graph, closed : seq<bool>)
    reads graph

    requires ValidGraph(graph)
    requires 0 < id < graph.Count()

    requires |closed| = graph.Count()
{
    ∀ c | 0 ≤ c < graph.Count() ∧ closed[c]  •
        id ∉ graph.Adjacents(c)
}

/*
 * The distance we have assigned to a frontier node is the best possible from
 * the current closed nodes. This is also valid for closed nodes (except for 0).
 */
predicate BestApproach(id : nat, graph : Graph, closed : seq<bool>,
    dist : seq<real>, prev : seq<int>)

    reads graph

    requires ValidGraph(graph)
    requires 0 < id < graph.Count()

    requires |closed| = graph.Count()
    requires |dist| = graph.Count()
    requires |prev| = graph.Count()

    requires ValidPrev(prev, graph)
{
    // The previous ∃ and is closed
    prev[id] ≥ 0
    ∧
    closed[prev[id]]
    ∧
    // The distance is the best from the closed set
    dist[id] == dist[prev[id]] + graph.Adjacents(prev[id])[id]
    ∧
    ∀ c | 0 ≤ c < graph.Count() ∧ closed[c] ∧ id in graph.Adjacents(c)  •
        dist[id] ≤ dist[c] + graph.Adjacents(c)[id]
}

/*
 * Auxiliary definitions to prove termination.
 */

function CountFalses(s : seq<bool>) : nat
{
    if s == [] then 0 else (if s[|s|-1] then 0 else 1) + CountFalses(s[..|s|-1])
}

// When all elements are false, the count gives the array length
lemma AllFalse(s : seq<bool>)
    requires ∀ i | 0 ≤ i < |s|  •  ¬s[i]

    ensures CountFalses(s) == |s|
{
}

lemma CancelOne(s : seq<bool>, k : nat)
    requires 0 ≤ k < |s|
    requires ¬s[k]

    ensures CountFalses(s[k := true]) == CountFalses(s) - 1
{
}

/*
 * Lemmas
 */

// Any walk has have positive costs
lemma AllWalkPositive(walk : seq<int>, graph : Graph)
    requires ValidGraph(graph)
    requires Walk(walk, graph)

    ensures WalkCost(walk, graph) ≥ 0.0
{
}

// Lemma to prove the invariants at the start of the loop
lemma Initialization(graph : Graph)
    requires ValidGraph(graph)

    ensures DistanceTo(0.0, 0, graph)
{
    var zeropath := [0];

    assert WalkTo(0, zeropath, graph);

    assert WalkCost(zeropath, graph) == 0.0;

    ∀ walk | WalkTo(0, walk, graph)
        ensures 0.0 ≤ WalkCost(walk, graph)
    {
        AllWalkPositive(walk, graph);
    }
}

/*
 * If we close a vertex, the outside vertices which are not adjacent to it
 * remains outside.
 */
lemma StillOutside(newcl : nat, graph : Graph, closed : seq<bool>,
    dist : seq<real>, adjacents : set<int>)
    // The preconditions of routine
    requires ValidGraph(graph)
    requires |closed| = graph.Count()
    requires |dist| = graph.Count()
    requires 0 ≤ newcl < graph.Count()

    // Adjacents to the recently closed nodes
    requires adjacents = set node | node in graph.Adjacents(newcl)

    // The invariant of the main method loop we wanted to preserve
    requires ∀ id | 0 < id < graph.Count() •
        dist[id] < 0.0  ⟺  IsOutside(id, graph, closed)

    ensures ∀ id | 0 < id < graph.Count() ∧ id ∉ adjacents •
        dist[id] < 0.0  ⟺  IsOutside(id, graph, closed[newcl := true])
{

}

/*
 * When we get the a vertex from the queue, its distance is definitive.
 */
lemma IsTheDistance(id : nat, graph : Graph, closed : seq<bool>,
    dist : seq<real>, prev : seq<int>)

    // The preconditions of routine
    requires ValidGraph(graph)
    requires |closed| = graph.Count()
    requires |dist| = graph.Count()
    requires 0 < id < graph.Count()

    // Prev is valid and id is in the frontier
    requires ValidPrev(prev, graph)

    requires ∀ id | 0 < id < graph.Count() •
        (∃ c | 0 ≤ c < graph.Count() •  id in graph.Adjacents(c))
        ⟹ dist[id] ≥ 0.0

    requires ∀ i | 0 < i < graph.Count() ∧ dist[i] ≥ 0.0 •
        BestApproach(i, graph, closed, dist, prev)

    requires BestApproach(id, graph, closed, dist, prev)

    requires closed[0]

    // For the closed nodes, the final distance is already calculated
    requires ∀ id | 0 ≤ id < graph.Count() •  closed[id]
        ⟹ DistanceTo(dist[id], id, graph)

    // The property that comes from the min-heap
    // requires ∀ c | 0 ≤ c < graph.Count() ∧ closed[c]  •
    //  dist[c] ≤ dist[id]

    // Consequence of the min-heap extraction
    requires ∀ jd | 0 ≤ jd < graph.Count() ∧ ¬closed[jd] ∧
        dist[jd] ≥ 0.0 •  dist[id] ≤ dist[jd]

    ensures DistanceTo(dist[id], id, graph)
{
    // The previous node is closed
    assert closed[prev[id]];

    // So its distance is final
    assert DistanceTo(dist[prev[id]], prev[id], graph);

    // (1) We try to prove that there is a walk with dist[id] cost
    // using the optimum walk to prev[id]

    assert ∃ walk  •  WalkTo(prev[id], walk, graph)
        ∧ WalkCost(walk, graph) = dist[prev[id]];
    var walk :| WalkTo(prev[id], walk, graph)
        ∧ WalkCost(walk, graph) = dist[prev[id]];

    var walk' := walk + [id];

    assert WalkTo(id, walk', graph);

    // (2) We try to prove that every other walk has higher cost

    ∀ walk | WalkTo(id, walk, graph)
        ensures dist[id] ≤ WalkCost(walk, graph)
    {
        // As id ≠ 0, |walk| ≥ 2
        assert |walk| ≥ 2;

        // Let's consider some cases depending on the antecedent in the walk
        var pr := walk[|walk|-2];

        /*
         * The idea is that a path cost is higher than dist[j] for sure
         * only if j is closed. There is at least a closed node (0) in
         * the path.
         */

        if closed[pr]
        {
            /*
             * The key elements are:
             * - dist[id] = dist[pr] + edge(pr, id)
             * - dist[pr] ≤ WalkCost(walk[..|walk|-1], graph)
             */

            assert dist[id] ≤ WalkCost(walk, graph);
        }
        else if |walk| > 2
        {
            var i := |walk| - 2;

            // We find the last closed node in the walk
            while ¬closed[walk[i]]
                invariant 0 ≤ i < |walk|
                invariant ∀ j | i+1 ≤ j < |walk|-1  •  ¬closed[walk[j]]

                decreases i
            {
                i := i - 1;
            }

            assert closed[walk[i]];
            assert ¬closed[walk[i+1]];

            // As walk[i] is closed
            assert DistanceTo(dist[walk[i]], walk[i], graph);

            assert dist[walk[i]] ≤ WalkCost(walk[..i+1], graph);

            assert dist[walk[i+1]] ≥ 0.0;

            assert BestApproach(walk[i+1], graph, closed, dist, prev);

            calc ≤ {
                dist[id];
                dist[walk[i+1]];
                WalkCost(walk[..i+2], graph);
                { SubWalkCost(walk, graph, i+2); }
                WalkCost(walk, graph);
            }
        }
    }
}
```

```
lemma SubWalkCost(walk : seq<int>, graph : Graph, i : nat)
    requires ValidGraph(graph)
    requires Walk(walk, graph)
    requires 0 < i < |walk|

    ensures WalkCost(walk[..i], graph) ≤ WalkCost(walk, graph)

    decreases |walk| - i
{
    if i < |walk|-1
    {
        calc ≤ {
            WalkCost(walk[..i], graph);
            // We add a positive number
            WalkCost(walk[..i], graph) + graph.Adjacents(walk[i-1])[walk[i]];
            { assert walk[..i+1][..i] = walk[..i]; }
            WalkCost(walk[..i+1][..i], graph)
                + graph.Adjacents(walk[..i+1][i-1])[walk[..i+1][i]];
            WalkCost(walk[..i+1], graph);
            // Induction hypothesis
            { SubWalkCost(walk, graph, i+1); }
            WalkCost(walk, graph);
        }
    }
}

/*
 * At the end, those nodes which have not been visited by the algorithm
 * are unreachable.
 */
lemma Unreachables(graph : Graph, closed : seq<bool>, dist : seq<real>)
    // The preconditions of routine
    requires ValidGraph(graph)
    requires |closed| = graph.Count()
    requires |dist| = graph.Count()

    requires ∀ id | 0 < id < graph.Count() •
        dist[id] < 0.0  ⟺  IsOutside(id, graph, closed[..])

    requires ∀ id | 0 < id < graph.Count() •
        dist[id] < 0.0  ⟺  ¬closed[id]

    requires closed[0]

    ensures ∀ id | 0 < id < graph.Count() ∧ dist[id] < 0.0 •
        ¬(∃ walk • WalkTo(id, walk, graph))
{
    ∀ id | 0 < id < graph.Count() ∧ dist[id] < 0.0
        ensures ¬(∃ walk • WalkTo(id, walk, graph))
    {
        /*
         * Reductio ad absurdum.
         *
         * Suppose there is a walk to id. As id is outside, no closed
         * node connect to it. So its antecedent in the walk is not
         * closed. If it is not closed, it is outside. And we iterate.
         *
         * Finally, we arrive to the conclusion that the origin is not
         * closed, against the hypothesis.
         */

        if ∃ walk • WalkTo(id, walk, graph)
        {
            var walk :| WalkTo(id, walk, graph);

            if |walk| > 2
            {
                var i := |walk| - 1;

                while i > 1
                    invariant 1 ≤ i < |walk|
                    invariant ¬closed[walk[i]]
                {
                    assert ¬closed[walk[i-1]];

                    i := i - 1;
                }

                assert walk[1] in graph.Adjacents(0);
            }
        }
    }
}
```

Appendix B

Attached material

Both attached CD and the repository at `https://github.com/ningit/vaed` include the Dafny programs written in this project. They are organized in directories we will describe below. Source code for auxiliary programs is also included there.

/

`vaed.exe`	Batch verifier binary
`memoria.pdf`	A .pdf version of this document
`exercises`	Exercises from [MSV12], see Chapter 3
`structures`	Implemented data structures, see Chapter 4
`algorithms`	Implemented algorithms, see Chapter 5
`util`	Auxiliary programs' code, see Section 2.5

The root directory contains a copy of this document and the binary for the *Vaed* program. It can be used to automatically verify all the included `.dfy` files, although it may take some time. This program uses Dafny which is not included here. If it is available in the binary path as `dafny`, *Vaed* will be able to find it. Otherwise, its path must be set with the `--dafnyexe` option.

Exercises

```
exercises
├─ er*.dfy
├─ er*.dfy
└─ aritmln.dfy
```

The `exercises` directory contains all exercises from the first phase of the project, including those which have been described in Chapter 3. They are written in Spanish.

The exercises can be found easily looking at their file name. Solved exercises names start with `er` while proposed exercises begin with `ep`. The number of the chapter and the exercise in [MSV12] follow.

Some exercises have alternative versions, marked with an `v2` prefix. `aritmnl.dfy` includes general properties about non linear arithmetic which are used in other exercises.

Data structures

The data structures code is in `structures` directory. Each datatype goes in a separate folder.

Alternative implementations exist in different files for both priority queues and stacks, as described in Chapter 4. A simple test program called `Main.dfy` is available in `stacks`.

```
structures
├─ list
│  └─ LinkedList.dfy
├─ prioqueue
│  └─ *BinaryHeap.dfy
└─ stack
   ├─ *Stack.dfy
   └─ Main.dfy
```

Algorithms

```
algorithms
├─ dynamic
│  ├─ floyd.dfy
│  └─ graph
│     ├─ graph.cs
│     ├─ graph.exe
│     ├─ floyd.dll
│     └─ map*.dfy
└─ greedy
   └─ dijkstra.dfy
```

The `algorithms` directory includes the `.dfy` files for the Floyd-Warhsall algorithm and Dijkstra's algorithm.

The Floyd algorithm also includes a C# program `graph.cs` in order to apply the algorithm to graphs described in a file. Four map examples and the binaries are included.

`floyd.dll` has been generated by Dafny from `floyd.dfy`.

Utils

The `util` directory contains auxiliary programs and resources, described in Section 2.5.

Both `dafnyc.pl` and `partes.pl` are Perl scripts. The first one is a wrapper for Dafny. It calls the verifier with the given parameters, arranges and colours the verifiers' output and includes the line from where the reported errors come from.

`partes.pl` receives a `.dfy` file as argument and verifies each function or method separately.

```
util
├─ vaed2
│  ├─ vaed.sln
│  └─ ...
├─ dafnyc.pl
├─ partes.pl
└─ dafny.lang
```

`dafny.lang` is a *GtkSourceView*'s language description for syntax highlighting. If included in the appropriate directory (like `~/.local/share/gtksourceview-3.0/language-specs`) it can be used from Gedit and other applications.

`vaed2` contains the batch verifier tool, written in C# and in the form of a Visual Studio (or MonoDevelop) solution. The tool can also be built from the command line using `msbuild` in Visual Studio or `xbuild` in Mono.

Bibliography

*«El que lee mucho y anda mucho,
ve mucho y sabe mucho.»*

— Don Quijote de la Mancha, Segunda parte,
Miguel de Cervantes Saavedra

The bibliography includes articles we have read or consulted both for this document elaboration and the development of the Dafny programs.

Dafny manuals

[HLQ11] Luke Herbert, K. Rustan M. Leino, and Jose Quaresma. "Using Dafny, an Automatic Program Verifier". In: *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*. 2011, pp. 156–181. DOI: 10.1007/978-3-642-35746-6_6. URL: http://research.microsoft.com/en-us/um/people/leino/papers/krml221.pdf.

[Lei13] K. Rustan M. Leino. "Developing Verified Programs with Dafny". In: *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. 2013, pp. 1488–1490. DOI: 10.1109/ICSE.2013.6606754. URL: http://research.microsoft.com/en-us/um/people/leino/papers/krml233.pdf.

Articles on Dafny features

[ALN15] Reza Ahmadi, K. Rustan M. Leino, and Jyrki Nummenmaa. "Automatic verification of Dafny programs with traits". In: *Proceedings of the 17th Workshop on Formal Techniques for Java-like Programs, FTfJP 2015, Prague, Czech Republic, July 7, 2015*. 2015, 4:1–4:5. DOI: 10.1145/2786536.2786542. URL: http://research.microsoft.com/en-us/um/people/leino/papers/krml247.pdf.

[KL15] Jason Koenig and K. Rustan M. Leino. "Programming Language Features for Refinement". 2015. URL: http://research.microsoft.com/en-us/um/people/leino/papers/krml248.pdf.

[Lei12] K. Rustan M. Leino. "Automating Induction with an SMT Solver". In: *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*. 2012, pp. 315–331. DOI: 10.1007/978-3-642-27940-9_21. URL: http://research.microsoft.com/en-us/um/people/leino/papers/krml218.pdf.

[LP13] K. Rustan M. Leino and Nadia Polikarpova. "Verified Calculations". In: *Verified Software: Theories, Tools, Experiments - 5th International Conference, VSTTE 2013, Menlo Park, CA, USA, May 17-19, 2013, Revised Selected Papers*. 2013, pp. 170–190. DOI: 10.1007/978-3-642-54108-7_9. URL: http://people.csail.mit.edu/polikarn/publications/vstte13.pdf.

[LP16] K. Rustan M. Leino and Clément Pit-Claudel. “Trigger Selection Strategies to Stabilize Program Verifiers”. 2016. URL: http://research.microsoft.com/en-us/um/people/leino/papers/krml253.pdf.

Dafny basic references

[FL16] Richard L. Ford and K. Rustan M. Leino. *Draft Dafny Reference Manual*. 2016. URL: https://hg.codeplex.com/dafny/raw-file/tip/Docs/DafnyRef/out/DafnyRef.pdf.

[Lei15] K. Rustan M. Leino. *Types in Dafny*. 2015. URL: http://research.microsoft.com/en-us/um/people/leino/papers/krml243.html.

[Mic16b] Microsoft Research. *Dafny Quick Reference*. 2016. URL: https://research.microsoft.com/en-us/projects/dafny/reference.aspx.

Other Dafny-related references

[Kas11] I. T. Kassios. “The Dynamic Frames Theory”. In: *Formal Aspects of Computing* 23.3 (2011), pp. 267–289. URL: http://pm.inf.ethz.ch/publications/getpdf.php?bibname=Own&id=Kassios10.pdf.

[Lei08] K. Rustan M. Leino. “Specification and verification of object-oriented software”. In: *Engineering Methods and Tools for Software Safety and Security* (2008). URL: http://research.microsoft.com/en-us/um/people/leino/papers/krml190.pdf.

[LL15] K. Rustan M. Leino and Paqui Lucio. “An Assertional Proof of the Stability and Correctness of Natural Mergesort”. In: *ACM Trans. Comput. Logic* 17.1 (Nov. 2015), 6:1–6:22. ISSN: 1529-3785. DOI: 10.1145/2814571. URL: http://research.microsoft.com/en-us/um/people/leino/papers/krml241.pdf.

[Luc16] Paqui Lucio. *Métodos formales de desarrollo de software*. UPV-EHU. 2016. URL: http://www.sc.ehu.es/jiwlucap/MFDS.html.

[Res16] Microsoft Research. *Dafny: a language and program verifier for functional correctness*. 2016. URL: http://research.microsoft.com/en-us/projects/dafny/.

Introduction

[Cou90] Patrick Cousot. “Methods and Logics for Proving Programs”. In: *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*. 1990, pp. 841–994. URL: http://www.di.ens.fr/~cousot/publications.www/Cousot-HTCS-vB-FMS-c15-p843--993-1990.pdf.gz.

[Flo67] Robert W. Floyd. “Assigning meanings to programs”. In: *Mathematical aspects of computer science* 19.19-32 (1967), p. 1.

[Hoa69] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming”. In: *Communications of the ACM* 12.10 (1969), pp. 576–580. DOI: 10.1145/363235.363259.

[Hoa72] C. A. R. Hoare. “Proof of Correctness of Data Representations”. In: *Acta Informatica* 1 (1972), pp. 271–281. DOI: 10.1007/BF00289507.

[Men42] L. F. Menabrea. *Sketch of The Analytical Engine Invented by Charles Babbage*. Trans. by Ada Augusta Lovelace. Oct. 1842. URL: https://www.fourmilab.ch/babbage/sketch.html.

Introduction: verification systems

[Bob+16] François Bobot et al. *Why3*. Inria Saclay-Île-de-France / LRI Univ Paris-Sud 11 / CNRS. 2016. URL: http://why3.lri.fr/.

[CCT16] University of Cambridge, Csiro, and Univerisity of Technology Chalmers. *HOL Interactive Theorem Prover*. 2016. URL: https://hol-theorem-prover.org/.

[Cle16] Clearsy. *Méthode B, méthode formelle de développement logiciel*. French. 2016. URL: http://www.methode-b.com/.

[CM16] University of Cambridge and Technische Universität München. *Isabelle*. 2016. URL: https://isabelle.in.tum.de/.

[Cou+16] Patrick Cousot et al. *The Astrée Static Analyzer*. CNRS/École normale supérieure/Inria. 2016. URL: http://www.astree.ens.fr/.

[Har15] John Harrison. *HOL Light*. University of Cambridge. Oct. 2015. URL: http://www.cl.cam.ac.uk/~jrh13/hol-light/.

[KM16] Matt Kauffman and J. Strother Moore. *ACL2*. University of Texas at Austin. Jan. 2016. URL: http://www.cs.utexas.edu/users/moore/acl2/.

[Mic16a] Microsoft. *Static Driver Verifier*. 2016. URL: https://msdn.microsoft.com/en-us/library/windows/hardware/ff552808(v=vs.85).aspx.

Other references

[BH14] Bernhard Beckert and Reiner Hähnle. “Reasoning and Verification: State of the Art and Current Trends”. In: *IEEE Intelligent Systems* 29.1 (2014), pp. 20–29. DOI: 10.1109/MIS.2014.3.

[Cor+09] Thomas H. Cormen et al. *Introduction to Algorithms*. 3rd ed. The MIT Press, 2009. ISBN: 978-0-262-03384-8.

[MSV12] Narciso Martí, Clara Segura, and Alberto Verdejo. *Algoritmos correctos y eficientes. Diseño razonado ilustrado con ejercicios*. 2012. ISBN: 978-84-1545-232-4.

[Peñ06] Ricardo Peña Marí. *Diseño de programas: formalismo y abstracción*. 3rd ed. Prentice Hall, 2006.

Index