Learn Verification With Cameleer

Verified Algorithms and Data Structures

Draft Version: November 30, 2025

© Pedro Gasparinho[†]
Mário Pereira[†]

[†] Nova School of Science and Technology, Portugal

Acknowledgements

In this textbook, we have adapted several algorithms from third-party sources, namely *Apprendre à programmer avec OCaml* [5], *Algorithms* [18], and proofs from the Why3's public gallery[1]. We thank the authors of these works for their contributions, which have greatly inspired and made this work possible, as least to this extent.

We also thank Ion Chirica for his helpful comments during the development of this textbook, and other investigation activities associated with this work.

[1]Why3's public gallery: https://toccata.gitlabpages.inria.fr/toccata/gallery/why3.en.html

Preface

This textbook is ongoing work and has been temporarily released in PDF format. The work here presented was developed in the context of a Master's dissertation. It is currently available for evaluation purposes, both as an artifact for the corresponding dissertation defence, and for a paper submission at the 27th International Symposium on Trends in Functional Programming. Thank you for your interest and time spent reading the textbook! Questions and/or suggestions are welcome, and can be directed to the main author of this work at: p.gasparinho@campus.fct.unl.pt.

Contents

Chapter 1

Introduction

In this chapter we will go through the installation processes of our tools, as well as the required background for subsequent chapters. This includes reviewing basic Logic and OCaml concepts, in addition to introducing Hoare Logic, GOSPEL, Cameleer and Why3.

We recommend to our readers the use of a Unix operating system, since the tools used in this book have better support in these types of systems.

1.1 *OCaml & Opam*

The first step in this process is to install the *OCaml* programming language, or rather, it's package manager, *opam*, which comes with *OCaml's* compiler, the basic packages and auxiliary tools.

This can be done either through the operating system's package manager (not to be confused with a programming language package manager, such as *opam*), or using the provided script.

For a simple guide regarding *opam*'s installation process visit:

https://ocaml.org/install

For a more complete guide visit:

https://opam.ocaml.org/doc/Install.html

1.1.1 Via Script

Open the terminal an execute this command:

```
bash -c "sh <(curl -fsSL https://opam.ocaml.org/install.sh)"
```

1.1.2 Via OS's Package Manager

For debian, ubuntu, mint or similar users, one may use execute the following command in the terminal:

```
apt install opam
```

Opam is available in Homebrew and MacPorts for macOS users:

```
brew install opam # Homebrew
port install opam # MacPorts
```

For other operating systems check the complete guide mentioned above.

1.1.3 Setting up *opam*

After installing *opam*, one needs to initialize it using:

```
opam init
```

At the end of the last operation, a prompt appears to set the environment variables, it is recommended to answer **yes**. Otherwise, one must run the following command when accessing *opam*'s installation:

```
eval ‘opam config env‘
```

1.2 Installing *Why3* and provers

Why3 [8] is a platform for deductive program verification with use cases in both academia and industry. If features a rich specification library, along many other important features, however its main drawback, from an industrial standpoint, is that it only accepts directly programs written in its own programming and specification language, *WhyMl*. This language is not widely used as general-purpose programming language in the industry, when compared to *OCaml*. Despite this, *WhyMl* has been used as an intermediary language in various projects to prove more mainstream languages, such as *C*, *Java*, *Ada* or *OCaml*.

More information about installing *Why3* and the automated provers can be found at:

https://marche.gitlabpages.inria.fr/lecture-deductive-verif/install.html

1.2.1 Installing *Why3*

Why3 and its *IDE* can be installed through *opam*:

```
opam install why3 why3-ide
```

Why3 relies on third-party provers, which allows combining multiple provers in the same proof, and each may have its own strengths and limitations. To start with we recommend installing *Alt-Ergo* [4], *cvc5* [1], *Z3* [13], and *Eprover* [17].

1.2.2 Installing *Alt-Ergo*

Similarly, it can be installed through *opam*:

```
opam install alt-ergo
```

1.2.3 Installing *cvc5*

Unlike *Alt-Ergo*, *cvc5* and *Z3* are not available in *opam*, as of the release date of this version of the book. Instead, one must download the provers from the corresponding *github* pages.

The latest versions of *cvc5* include numerous options for each OS and architecture, which may be hard to distinguish which is the best suited for the user, one may experiment or go back to version 1.1.0, which features simpler variations.

To install version 1.1.0 for Linux, via terminal, one can use the following sequence of commands:

```
wget https://github.com/cvc5/cvc5/releases/download/cvc5-1.1.0/cvc5-Linux
sudo cp cvc5-Linux /usr/local/bin/cvc5
sudo chmod +x /usr/local/bin/cvc5
cvc5 --version
```

This sequence, by step, is equivalent to:

1. Downloading the desired file from *github*
2. Moving it from the working directory to */usr/local/bin*
3. Granting the file permission to execute
4. Checking the version (and if it is actually installed)

1.2.4 Installing *Z3*

Similarly to *cvc5*, to install the latest version (4.15.0) of *Z3*, via terminal, one may use the following sequence of commands:

```
wget https://github.com/Z3Prover/z3/releases/download/z3-4.12.2/z3-4.15.0-x64-glibc-2.39.zip
unzip z3-4.15.0-x64-glibc-2.39.zip
sudo cp z3-4.15.0-x64-glibc-2.39/bin/z3 /usr/local/bin
sudo chmod +x /usr/local/bin/z3
z3 --version
```

1.2.5 Installing *Eprover*

Why3, currently (version 1.8.1), only supports up to version 2.0 Turzum of Eprover.

```
wget http://wwwlehre.dhbw-stuttgart.de/~sschulz/WORK/E_DOWNLOAD/V_2.0/E.tgz
```

```
unzip E.zip
cd E
make
cd PROVER
sudo cp PROVER/eproveer /usr/bin
sudo chmod +x /usr/bin/eprover
eprover --version
```

1.3 Installing Cameleer

For the reasons mentioned above, *Cameleer* [14] emerged. This piece of software is used to verify OCaml programs with GOSPEL [2] (Generic OCaml SPEcification Language) annotations. *Cameleer*, internally, translates these programs into *WhyMl*, which can then be verified in the *Why3* platform. For more information check:

https://github.com/ocaml-gospel/cameleer

Cameleer can be installed via terminal with the following sequence of commands:

```
git clone https://github.com/ocaml-gospel/cameleer
opam pin add path/to/cameleer
cameleer --version
```

1.4 Boolean Algebra

The field of Formal Verification is closely related to Mathematics, with one of the most influential subfields being Logic. As such, it is important to start by disclosing it, through Boolean Algebra. This branch of algebra studies truth variables, and how these interact with each other. A truth variable may assume one of two values, true or false [3], and its negation can be obtained by preceding it with the ¬ symbol. To calculate the result of an expression, one may use a truth table, which has a column for each variable, one for the result, and potentially other auxiliary columns with intermediate results. In terms of rows, it usually has $2^{\#\text{variables}}$ to allow all possible combination between the values of the variable [20]. For instance, the truth table of the negation operation is:

p	$\neg p$
T	F
F	T

Given the limited set of values in Boolean Algebra, this is the only unary operation, *i..e* only receives one argument. Most other operations are binary (2 arguments), and some of the most common include:

- Conjunction ($\wedge$), which is true when both variables are true.
- Disjunction ($\vee$), which is true when at least one variable is true.
- Consequence ($\implies$), which is true when the first variable is false or both are true.
- Equivalence ($\iff$), which is true when both variables have the same value.

p	q	$p \wedge q$	$p \vee q$	$p \implies q$	$p \iff q$
T	T	T	T	T	T
T	F	F	T	F	F
F	T	F	T	T	F
F	F	F	F	T	T

Note that when p is false, $p \implies q$ is true. This can be explained from the fact that we are in the context of false premises. Consider the following example:

p – Going to the park

q – Feeding the ducks

$p \implies q$ – If I go the park, then I will feed the ducks

The only scenario where we would be "lying" is if we went to the park, but did not feed the ducks. In the event of not going to the park, it does not matter if we actually got to feed ducks (in any other place), since we do not state what we would be doing instead. Hence, when p is false, $p \implies q$ is true, independently of q.

1.5 First-Order Logic

Unlike Prepositional Logic, which exclusively uses Boolean Algebra, First-Order Logic expands upon the aforementioned concepts with the introduction of quantifiers and predicates. This family of logic systems include three basic building blocks [19]:

- **Constants** - Which denote the objects in the logic system. A constant may only represent a single object.
- **Functions** - Which receive one or more objects to produce another. Functions are pure, *i.e.* they always produce the same output for a given input (there is no notion of state).
- **Predicates** - Which are properties or relations between objects that may be true or false. In first-order logic systems, predicates cannot be arguments in other predicates.

One common example of a First-Order Logic system is elementary algebra:

- Constants - $\mathbb{N}_0$ *i.e.* $\{0, 1, 2, ...\}$
- Unary Functions - $\{-, +, !, ...\}$ (1 argument)
- Binary Functions - $\{+, -, *, /, ^\wedge, \%, ...\}$ (2 arguments)
- Predicates - $\{=, \neq, >, \geq, <, \leq\}$

In addition to the basic building blocks, there are also quantifiers, which apply the same property to each element in a given set (or subset) of objects. There are two quantifiers:

- For all ($\forall$) - If every single element in that set complies with the desired property, then the quantified expression is true.
- Exists ($\exists$) - If at least one element in that set complies with the desired property, then the quantified expression is true.

Examples:

$$\begin{aligned}
&\forall x \in \mathbb{N}_0 : x > -1 \text{ (true)}\\
&\forall y \in \{1, 2\} : 2y = 4 \text{ (false)}\\
&\exists z \in \mathbb{N}_0 : 3z - 5 = 4 \text{ (true)}\\
&\exists a, b \in \mathbb{N}_0 : a + b = 7 \text{ (true)}\\
&\forall c \in \mathbb{N}_0 : \exists d \in \mathbb{N}_0 : 2c = d \text{ (true)}\\
&\forall c \in \mathbb{N}_0 : \exists d \in \mathbb{N}_0 : c \neq d \wedge 2c = d \text{ (false)}
\end{aligned}$$

Exercise: What does each of the previous expressions mean?

1.5.1 Other important definitions

Other important concepts commonly found in formal logic systems, include:

Definition 1.1 (Axiom) *An axiom or a postulate is a statement that is assumed to be true without proof. These are the basic building blocks in a formal reasoning system [16].*

Definition 1.2 (Lemma) *A lemma is a minor result that can be used to help prove other more significant results [16]. Lemmas must also be proved.*

1.5.2 Hoare Logic

Hoare-Floyd Logic [10, 9], commonly referred to as Hoare Logic, is the basis of deductive verification of software. It originated from the observation that before the execution of a program fragment the memory state must respect a set of logical conditions. Moreover, after the computation, the memory state may change and will respect a new set of logical conditions. These can be expressed as a Hoare Triple:

$$\{P\}\ S\ \{Q\}$$

Where P is the set of pre-conditions, S is a program (or a fragment) and Q is the set of post-conditions. A Hoare Triple can be understood as: "If P holds immediately before the execution of S, then after it terminates it produces a state where Q holds". This is partial correctness, since the termination is not assured by the triple. Total correctness is achieved when termination is also guaranteed. This is only problematic when dealing with recursion or `while` loops, since `for` loops have a finite number of iterations. Hoare's original work features a number of rules, based on language construct, which have been extended over the years. Of these rules, it is essential to know the while rule:

$$\frac{\{\mathit{Inv} \land \mathit{Cond}\}\ \mathit{Exp}\ \{\mathit{Inv}\}}{\{\mathit{Inv}\}\ \text{while}\ \mathit{Cond}\ \text{do}\ \mathit{Exp}\ \{\mathit{Inv} \land \neg \mathit{Cond}\}}$$

When we are dealing with the repetition of instructions, we stop thinking about pre- and post-conditions, to think about invariants. An invariant is a property that holds immediately before the start and immediately after the end of an iteration. In the case of a `while` loop, the loop condition is also respected when entering a new iteration, while its negation is respected when exiting the loop.

1.5.3 OCaml

OCaml is a functional-first programming language that also supports the imperative and object-oriented paradigms (although the latter will not be our focus). One of its main characteristics is type-safety, meaning that any compiled program will not produce type errors. This is particularly important in a formal verification setting, since it is a strong safety guarantee that we do not have to check ourselves. As a functional language, OCaml considers functions as primitive types, this allows for many possibilities, including passing functions as parameters or returning them. Consider the following function:

OCaml

```ocaml
let seconds_to_hours s =
  let h = s / 3600 in
  let r = s mod 3600 in
  let m = r / 60 in
  let s' = r mod 60 in
  (h, m, s')
```

As we can observe, the `let` keyword is used to name the expression after the equal sign. A function is distinguished by having other identifiers before `=`, in this case `s`. To define auxiliary values (including other functions) inside the body of a function, one must follow the `let` expression with the `in` keyword, this is due to scoping and avoiding ambiguities. Note that it is possible to chain `let` expressions, as seen above, with each identifier being available in the subsequent expressions. The readers might have noticed already that this function converts a single value in seconds (`s`) to a triple containing the equivalent amount of time in hours (`h`), minutes (`m`) and seconds (`s'`). This can be achieved by using the integer division (/), to obtain the highest whole value, and the remainder operation (`mod`), to spread across the subunits. For instance, by dividing the original amount by 3600 (number of seconds in an hour), we obtain the maximum number of complete hours:

$$\left\lfloor \frac{3600}{3600} \right\rfloor = 1 \qquad \left\lfloor \frac{3661}{3600} \right\rfloor = 1 \qquad \left\lfloor \frac{7200}{3600} \right\rfloor = 2$$

Another important feature in OCaml is pattern matching, which allows deconstructing complex data types in a case by case analysis using the different patterns, generally from the various constructors that combined make the data type. Consider a function that calculates the number of instances of a given element in a list:

OCaml
```ocaml
let rec count x l =
  match l with
  | [] -> 0
  | h::t -> (if x = h then 1 else 0) + count x t
```

The two most common patterns used when deconstructing a list is the empty list ([]) and the list with at least one element (`h::t`), where `h` is the first element (head) and `t` is the sub-list containing the remaining elements (tail). Note that `h` and `t` are just naming conventions, other identifiers could have been used. The `count` function recursively traverses the list element by element. In OCaml recursive functions must contain the `rec` keyword after the corresponding `let`. Also note that `if-blocks` can be used as expressions in many functional languages, in contrast to imperative languages, where these are treated as statements. Another language detail is the following syntactic sugar:

OCaml
```ocaml
let rec count x = function
  | [] -> 0
  | h::t -> (if x = h then 1 else 0) + count x t
```

If we have a parameter that is immediately deconstructed in pattern matching by itself, then we may omit both of these with the `function` keyword.

Immutability is one of the main characteristics of the functional paradigm. In OCaml, however, the programmer is not limited by immutability, since the language also features numerous imperative constructs. This includes mutable variables, through references, arrays, and loops. Observe the imperative version of the previously presented `count` function:

OCaml
```ocaml
let count_imp x a =
  let r = ref 0 in
  for i = 0 to Array.length a - 1 do
    if a.(i) = x then r := !r + 1
  done;
  !r
```

Unlike other languages, OCaml's `for` loop is quite different, it does not directly support early stopping (can still be achieved by using exceptions) nor defining a different step. The `to` keyword is used to increment the loop variable by one in each iteration. Alternatively, the `downto` keyword is used for decremental iterations, with these two keywords being the only available options for the loop's step. Moreover, the terminal value (`i.e.` the one that comes after `to` or `downto`) is inclusive, that is why we must subtract one from the array's length. Another particularity in OCaml is its syntax concerning array operations. To access an index of an array one must use `a.(i)` instead of the usual `a[i]`. Mutable variables, `i.e.` references, are declared using the `ref` keyword, which is placed immediately before the initial value. Operations to manipulate mutable variables include dereferencing (`!`), to read its current value, and updating (`:=`).

1.5.4 GOSPEL and Cameleer

The Generic OCaml SPEcification Language (GOSPEL), as the name states, is a specification language for OCaml programs that is not tied to a single tool or purpose, as such, other developers may use it as they see fit. Besides Cameleer, which is the focus of this work, other tools that use GOSPEL include `Ortac`[1] [7] and `Why3gospel`[2]. Cameleer is a tool that enables the deductive verification of OCaml programs with GOSPEL annotations within the Why3 platform. Internally, these programs are translated into an WhyML equivalent, which is Why3's specification and programming language. Some advantages of this platform include the use of multiple third-party theorem provers, either interactive or automatic, and treating each proof goal independently. As we have mentioned before, the base of Deductive Verification is Hoare Logic. We can equip an OCaml function with pre- and post-conditions, similar to a Hoare triple, with GOSPEL as seen below:

GOSPEL + OCaml

```
let seconds_to_hours s =
  let h = s / 3600 in
  let r = s mod 3600 in
  let m = r / 60 in
  let s' = r mod 60 in
  (h, m, s')
(*@ h, m, s' = seconds_to_hours s
    requires s >= 0
    ensures 3600*h + 60*m + s' = s *)
```

[1]`Ortac`'s repository: https://github.com/ocaml-gospel/ortac

[2]`Why3gospel`'s repository https://github.com/ocaml-gospel/why3gospel

GOSPEL annotations are written in the form of special comments, with the intent of OCaml's compilers ignoring these annotations and compiling successfully. To distinguish from regular comments, a special syntax with the at sign (@) is used: `(*@ ... *)`. Pre-conditions start with the `requires` keyword. In this case, the input parameter `s` (the number of seconds to be converted) must be a non-negative integer. On the other hand, post-conditions are represented with the `ensures` keyword, and in this context we must guarantee that the result is effectively correct, by re-converting each component to seconds, using the respective unit conversion rate, and summing the three to obtain the original value. In GOSPEL, by default, the parameters retain their original identifier, and if the function returns a value, then it can be accessed in the annotation with the identifier `result`. However, one may re-define these identifiers, as shown in the first line in the comment of the example above. In this case, that line also holds a special purpose, it explicitly deconstructs the tuple into single elements.

The previous function is automatically proven to terminate by the SMT solvers effortlessly, since it does not perform any kind of recursion or iteration. However, that is not always the case. For instance, consider the `count` function previously presented:

GOSPEL + OCaml

```
let rec count x l =
  match l with
  | [] -> 0
  | h::t -> (if x = h then 1 else 0) + count x t
(*@ r = count x l
    variant l
    ensures 0 <= r <= List.length l
    ensures r > 0 -> List.mem x l
    ensures r = 0 -> not List.mem x l *)
```

When dealing with recursive functions or `while` loops (not `for` loops since those have a guaranteed finite number of iterations), termination must be proven. This can be achieved with a monotonically decreasing expression that remains non-negative during iteration, in other words, the value of the expression in a given iteration must be strictly smaller than in the previous iteration, and both must be non-negative. In the context of the `count` function, using the length of the list suffices to prove its termination, since each recursive call deals with the tail of the current list, therefore each successive call receives a list with one less element than the previous, and the length of a list is undeniably non-negative. Alternatively, one may use the list itself, instead of `List.length l`, which checks the structure of the list, rather than just its length, however it works similarly.

If readers notice closely, this function returns the number of elements in a list that comply with a given property, this being the equality to a given element. However, since some elements may not comply with this property, it is not possible to quantify this exact amount with either the existential or universal quantifiers. To do so, we would need to introduce more complex

GOSPEL constructs, which is not the objective of this chapter, but it will be done so in the next chapters of this textbook. So, for now, let us focus on partial correctness guarantees. In this field, it is important to be ambitious and aim to provide the strongest known safety guarantees, however that may not always be possible, for instance due to tool limitations, as such, knowing when to relax a problem is also a great quality. Whenever it is not possible to check for the exact result, then a more relaxed alternative is to restrict it to an interesting interval. In this case, the result ranges from 0 and the length of the list (both inclusive), since `x` (the value to be counted) may not be present in the list, or, at most, all elements of the list have the value of `x`. This leads to the second and third post-conditions, if the result is positive, then `x` must belong to the list, inversely, if the result is 0, `x` does not belong in the list.

Moving on to the imperative version of this function. It is important to take into account that `for` and `while` loops are also meant to be annotated, which leads to multiple GOSPEL annotations in the same function. In this case, `counter_imp` contains a single `for` loop, so, in total, it must contain two annotations: one applied to the function body, as a whole, and the other, internally, to the `for` loop. Starting with the function body:

GOSPEL + OCaml
```
let count_imp x a = (* ... *)
(*@ r = count_imp x l
    ensures 0 <= r <= Array.length l
    ensures r > 0 -> Array.mem x l
    ensures r = 0 -> not Array.mem x l *)
```

For the most part, the specification above is quite similar to its functional counterpart. The only real difference consists in the use of the operations provided by the array library, which should not be surprising, since we are dealing with arrays rather than lists. A possible approach to specify the `for` loop is as follows:

GOSPEL + OCaml
```
for i = 0 to Array.length a - 1 do
(*@ invariant 0 <= !r <= i
    invariant !r > 0 -> Array.mem x l
    invariant !r = 0 -> forall k. 0 <= k < i -> a.(k) <> x *)
  if a.(i) = x then r := !r + 1
done;
```

When specifying loops, it is important to remember that we are no longer able to use pre- and post-condition. Instead, invariants should be used, and these are logical properties that must hold both at the start and end of an iteration, including at the time of entering and exiting the loop. Finding the correct invariants is an arduous task, a general guideline is that they are related to function's post-conditions, although other invariants may be necessary. Also note that given the iterative nature of loops, at any given point only a subset of the arrays has been covered, hence, these properties may need to be adapted, based around the iteration variable (in this case i), instead of the array as a whole. This can be seen in the first invariant, where the upper bound becomes

i, instead of the length of the array. This can be explained from the fact that when exiting a given iteration, the visited indexes range from 0 to i (both inclusive), and we can state for certain that, as most, r may hold the value of i. That property holds when entering the loop, since the upper bound is inclusive ($0 \leq$ `!r` ≤ 0). If we were to set a higher upper bound, namely the length of the list, that would logically mean that, when entering an intermediate iteration, `r` could already hold the length of the list, and at the time of exiting the loop, `r` could be incremented by one, which would break the loop invariant. The second loop invariant remains unchanged, since we are only confirming than a positive result leads to belonging in the list, once again this is a more relaxed condition, to preserve the simplicity of the example. By contrast, that is not the case when `r` holds 0, since occurrences of `x` may be present in the remainder of the list, which must be taken into account. The solution is to define the interval of indexes that have already been traversed, namely from 0, inclusive, up to i, exclusive. Note that the upper bound is exclusive ($<$), otherwise when entering a given iteration we would already be including a value that has not yet been processed, which would lead to an incorrect proof.

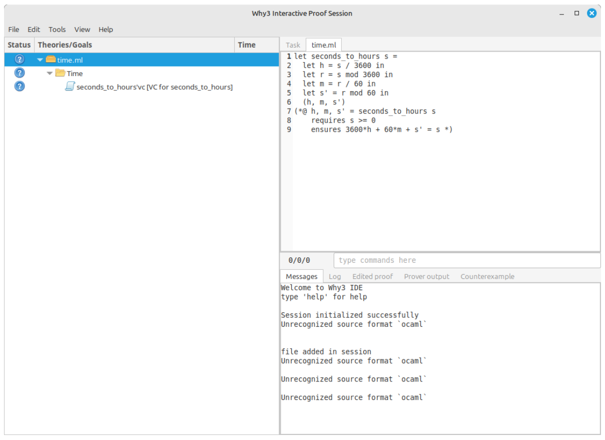
1.5.5 Cameleer and Why3

As previously mentioned, Cameleer uses the Why3 platform in order to verify OCaml programs with possibly several SMT-solvers allowed by that platform. So, this may raise the question: how to use Cameleer? Assuming that we have an OCaml program in our file system, we can open a terminal in that directory and type:

```
cameleer [filename].ml
```

This will launch the Why3 IDE, which we will explain briefly in this subsection with various images and how to conduct a concrete proof, this being the `seconds_to_hours` function from before. To start, let us present the interface that is displayed immediately after the command above:

On the left side we can see the generated proof goals. In this case we can see three nested proof goals, the top-most being the file as a whole, and the innermost being the `seconds_to_hours` function. The goal in the middle is an implicit module that wraps all functions in the file. The more (external) functions we have in our file, the more proof goals will be generated. If we were using OCaml's module system (which we will cover in the last chapters) this would also generate more proof goals. To dispatch the SMT-solvers we can select any proof goal and click one of the following keys: 0, 1, 2 and 3. Each key has a different time limit imposed by Why3 due to the halting problem, which is the famous limitation of being impossible to create a program that can determine if another program will terminate execution or run forever. So, Why3 can not determine if the SMT-solvers will ever terminate checking a proof goal. This leads us to discussing the aforementioned keys and respective time limits:


Why3 Interactive Proof Session
File Edit Tools View Help
Status Theories/Goals Time
time.ml
Time
seconds_to_hours'vc [VC for seconds_to_hours]
Task time.ml
0/0/0
type commands here
Messages Log Edited proof Prover output Counterexample

```
let seconds_to_hours s =
  let h = s / 3600 in
  let r = s mod 3600 in
  let m = r / 60 in
  let s' = r mod 60 in
  (h, m, s')
(*@ h, m, s' = seconds_to_hours s
    requires s >= 0
    ensures 3600*h + 60*m + s' = s *)
```

```
Welcome to Why3 IDE
type 'help' for help

Session initialized successfully
Unrecognized source format `ocaml`


file added in session
Unrecognized source format `ocaml`

Unrecognized source format `ocaml`

Unrecognized source format `ocaml`
```

Figure 1.1: Why3 IDE

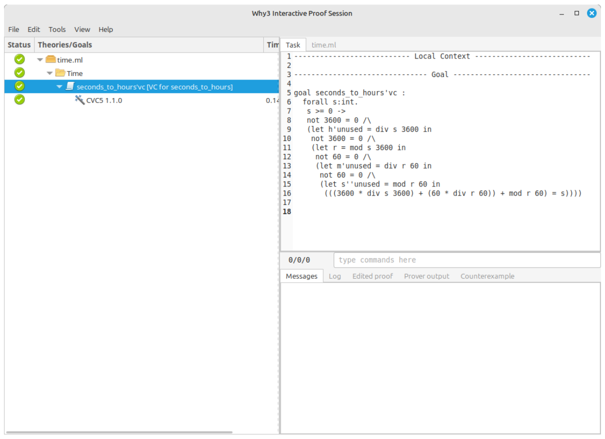
- **0** – Time limit of 1s.
- **1** – Time limit of 5s.
- **2** – Starts with a time limit of 1s. If it fails, increases time limit to 10s.
- **3** – Starts with a time limit of 1s. If it fails, increases time limit to 5s. Finally, increases time limit to 30s, and may perform special proof goal splits.

When pressing one of these keys, it recursively dispatches the SMT-solvers to any children proof goals. Alternatively, if every child proof goal is successfully verified, then the parent is automatically updated. Since this is a simple case study it suffices to press 0 on any of the proof goals to successfully verify it:

Taking a step back, the second most import window in Why3 is the `Task` window on the top right side. This window displays our current goal and the context behind it:

Despite the simplicity of the function, the goal looks complex. Using the **s** key will split the selected proof goal into smaller goals, and, if pressed repeatedly, also simplifies its descendants. Although, splitting may not always be possible.

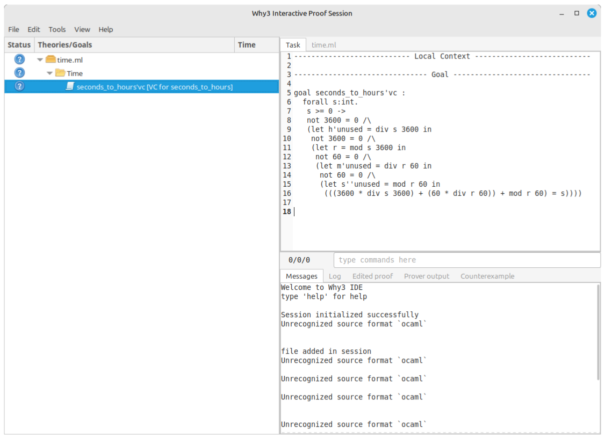
By selecting one of the new sub-goals, we can observe how the logical context changes. For instance, proof goal 4 has a simpler expression, this being 3600 ∗


Why3 Interactive Proof Session
File Edit Tools View Help
Status Theories/Goals Tim
time.ml
Time
seconds_to_hours'vc [VC for seconds_to_hours]
CVC5 1.1.0
0.14
Task time.ml
0/0/0
type commands here
Messages Log Edited proof Prover output Counterexample

```
--------------------------- Local Context ---------------------------

------------------------------- Goal -------------------------------

goal seconds_to_hours'vc :
  forall s:int.
   s >= 0 ->
   not 3600 = 0 /\
   (let h'unused = div s 3600 in
    not 3600 = 0 /\
    (let r = mod s 3600 in
     not 60 = 0 /\
     (let m'unused = div r 60 in
      not 60 = 0 /\
      (let s''unused = mod r 60 in
       (((3600 * div s 3600) + (60 * div r 60)) + mod r 60) = s))))
```

Figure 1.2: Successful proof

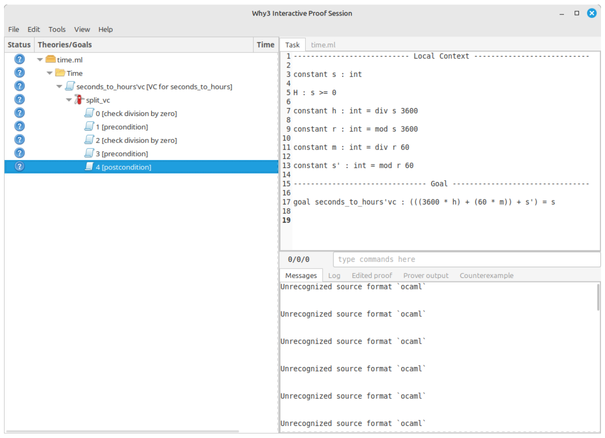
$h + 60 * m + s' = s$. Moreover, the other sub-goals are used as hypotheses, despite not being proven yet. Treating proof goals independently of each other is one of Why3's advantages. The proof is, also, successful after splitting:


Why3 Interactive Proof Session
File Edit Tools View Help
Status Theories/Goals Time
time.ml
Time
seconds_to_hours'vc [VC for seconds_to_hours]
Task time.ml
0/0/0
type commands here
Messages Log Edited proof Prover output Counterexample

```
--------------------------- Local Context ---------------------------

------------------------------ Goal --------------------------------

goal seconds_to_hours'vc :
  forall s:int.
   s >= 0 ->
   not 3600 = 0 /\
   (let h'unused = div s 3600 in
    not 3600 = 0 /\
    (let r = mod s 3600 in
     not 60 = 0 /\
     (let m'unused = div r 60 in
      not 60 = 0 /\
      (let s''unused = mod r 60 in
       (((3600 * div s 3600) + (60 * div r 60)) + mod r 60) = s))))
```

```
Welcome to Why3 IDE
type 'help' for help

Session initialized successfully
Unrecognized source format `ocaml`


file added in session
Unrecognized source format `ocaml`

Unrecognized source format `ocaml`

Unrecognized source format `ocaml`


Unrecognized source format `ocaml`
```

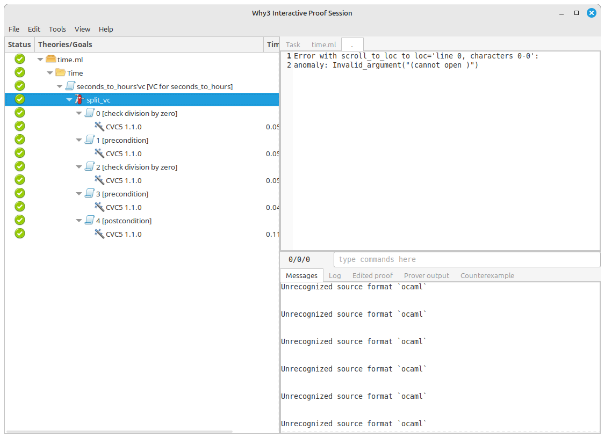
Figure 1.3: Before splitting


Why3 Interactive Proof Session
File Edit Tools View Help
Status Theories/Goals Time
time.ml
Time
seconds_to_hours'vc [VC for seconds_to_hours]
split_vc
0 [check division by zero]
1 [precondition]
2 [check division by zero]
3 [precondition]
4 [postcondition]
Task time.ml
0/0/0
type commands here
Messages Log Edited proof Prover output Counterexample

```
--------------------------- Local Context ---------------------------

constant s : int

H : s >= 0

constant h : int = div s 3600

constant r : int = mod s 3600

constant m : int = div r 60

constant s' : int = mod r 60

------------------------------ Goal --------------------------------

goal seconds_to_hours'vc : (((3600 * h) + (60 * m)) + s') = s
```

```
Unrecognized source format `ocaml`


Unrecognized source format `ocaml`


Unrecognized source format `ocaml`


Unrecognized source format `ocaml`


Unrecognized source format `ocaml`


Unrecognized source format `ocaml`
```

Figure 1.4: After splitting


Why3 Interactive Proof Session
File Edit Tools View Help
Status Theories/Goals Tim
Task time.ml .
0/0/0
type commands here
Messages Log Edited proof Prover output Counterexample

```
time.ml
  Time
    seconds_to_hours'vc [VC for seconds_to_hours]
      split_vc
        0 [check division by zero]
          CVC5 1.1.0                0.05
        1 [precondition]
          CVC5 1.1.0                0.05
        2 [check division by zero]
          CVC5 1.1.0                0.05
        3 [precondition]
          CVC5 1.1.0                0.04
        4 [postcondition]
          CVC5 1.1.0                0.11
```

```
Error with scroll_to_loc to loc='line 0, characters 0-0':
anomaly: Invalid_argument("(cannot open )")
```

```
Unrecognized source format `ocaml`

Unrecognized source format `ocaml`

Unrecognized source format `ocaml`

Unrecognized source format `ocaml`

Unrecognized source format `ocaml`

Unrecognized source format `ocaml`
```

Figure 1.5: Successful proof with simpler goals

Chapter 2

Arithmetic

2.1 Extended Euclidean Algorithm

We begin our algorithmic journey with Euclid's classical method to calculate the greatest common divisor (gcd) of two numbers, for instance:

$$\gcd(72, 48) = 24$$

$$\begin{aligned}\text{Divisors of } 72 &= \{1, 2, 3, 4, 6, 8, 9, 12, 18, 24, 36, 72\}\\ \text{Divisors of } 48 &= \{1, 2, 3, 4, 6, 8, 12, 16, 24, 48\}\end{aligned}$$

But, how does Euclid's algorithm work? This is the most important question to answer from both the programming and verification perspectives. This classical algorithm has been studied for centuries, with the original idea being based on the subtraction operation, while a more recent and optimized method uses the remainder operation. For the time being, we will concentrate on the latter:

Algorithm 1 (Optimized Euclidean Algorithm) *Given an integer, x, that differs from 0, and an integer, y, find their greatest common divisor.*

```
1. If $y = 0$ then return $n$;
2. Calculate the remainder: $r = x \% y$;
3. Set $x \leftarrow y$, $y \leftarrow r$, go back to step (1).
```

Not only that, but we will also consider the extended version of the algorithm. This version also calculates the Bézout's coefficients, two additional numbers, that comply with Bézout's Identity.

Lemma 1 (Bézout's Identity) *Let x and y be integers with d as their greatest common divisor. Then, there exists integers a and b such that x*a + y*b = d.*

This lemma is quite powerful, since it combines the two input numbers and the respective greatest common divisor in a single equation. A purely functional implementation of the extended Euclidean algorithm can be seen below:

```ocaml
let rec extended_gcd x y =
  if y = 0 then (1, 0, x)
  else
    let q = x / y in
    let (a, b, d) = extended_gcd y (x - q * y) in
    (b, a - q * b, d)
```

Before analysing the code, we must first consider the input and output of the `extended_gcd`, it receives two parameters x and y, as expected, and recursively calculates the two Bézout's coefficients and the greatest common divisor of x and y, respectively, as an ordered triple. With this in mind, let's have a look into two examples:

$$\begin{aligned} \text{egcd}(72,48) &= (1,-1,24) \\ \text{egcd}(48,24) &= (0,1,24) \\ \text{egcd}(24,0) &= (1,0,24) \end{aligned} \qquad \begin{aligned} \text{egcd}(1071,462) &= (-3,7,21) \\ \text{egcd}(462,147) &= (1,-3,21) \\ \text{egcd}(147,21) &= (0,1,21) \\ \text{egcd}(21,0) &= (1,0,21) \end{aligned}$$

The base case is simple to explain, the last call to the `extended_gcd` (`egcd` for short) function contains the result in parameter x and the value 0 in parameter y. As such, the Bézout's coefficients should be 1 and 0, respectively, to maintain Bézout's Identity:

$$\begin{aligned} 24 &= 24 * 1 + 0 * 0, \ \text{when egcd}(24,0) \text{ in egcd}(72,48) \\ 21 &= 21 * 1 + 0 * 0, \ \text{when egcd}(21,0) \text{ in egcd}(1071,462) \end{aligned}$$

One way to calculate the remainder is to use the Euclidean division lemma:

Lemma 2 (Euclidean Division) *Given two integers, a and b, with $b \neq 0$, then, there exists two unique integers, q and r, such that: $a = b * q + r$, with $0 \leq r < |b|$*

By applying this lemma to our context, we obtain:

$$\begin{aligned} x &= y * q + r \\ r &= x - y * q \text{ (q is the integer division between } x \text{ and } y) \\ r &= x - y * \lfloor x/y \rfloor \end{aligned}$$

Since each intermediate result respects Bézout's Identity, the recursive step can be obtained by comparing two consecutive calls:

$$d = x_n * a_n + y_n * b_n \text{ (Step } n\text{)}$$
$$d = x_{n+1} * a_{n+1} + y_{n+1} * b_{n+1} \text{ (Step } n+1\text{)}$$
$$\text{With } x_n = y_{n+1} \text{ and } y_n = x_{n+1} - q * y_{n+1}$$

$$d = x_n * a_n + y_n * b_n \equiv$$
$$d = y_{n+1} * a_n + (x_{n+1} + q * y_{n+1}) * b_n \equiv$$
$$d = y_{n+1} * a_n + x_{n+1} * b_n + q * y_{n+1} * b_n \equiv$$
$$d = x_{n+1} * b_n + y_{n+1} * (a_n + q * b_n)$$

$$\therefore a_{n+1} = b_n \text{ and } b_{n+1} = a_n - q * b_n$$

This result is exactly what is stated in `let (a, b, d) = extended_gcd y (x - q * y) in (b, a - q * b, d)`. The triple `(a, b, d)` corresponds to the n-th step, while `(b, a - q * b, d)` is the next. Also, note that `d`, the greatest common divisor, does not change between steps.

This previous discussion, although displayed somewhat informally, is a proof by induction, in which, one must present the base case, usually constant values that comply with a given formula, and the inductive step, where one has to reach the formula for step $n + 1$ from the previous step. From a machine-checked proof perspective, proofs by induction are quite powerful. Moreover, the fact we calculate the Bézout's coefficients for every function call is a very effective way guarantee that the Bézout's Identity is maintained every time. Alternatively, it would be possible to use the existential quantifier, however, these logical constructs are notoriously difficult to be checked by machines, due to potentially large value spaces. With the mathematical background out of the way, we present specified `OCaml` code:

GOSPEL + OCaml

```
let rec extended_gcd x y =
  if y = 0 then (1, 0, x)
  else
    let q = x / y in
    let (a, b, d) = extended_gcd y (x - q * y) in
    (b, a - q * b, d)
(*@ (a, b, d) = extended_gcd x y
    requires x <> 0
    variant abs y
    ensures d = a*x + b*y *)
```

The optimized version of the algorithm is prepared to deal with negative numbers, since it uses the remainder (although indirectly in our case), therefore the only restriction on the inputs is that x must be different from 0. This is due to x holding the result in the final functional call, and a divisor, by definition, is a value different from 0. Furthermore, y is the stopping condition, in the form of the value 0, as such, there are no additional requirements. The `ensures` clause is simply the Bézout's Identity formula, which, once more, is a very strong property that guarantees that `d` is effectively the greatest common divisor of x and y. Finally, the `variant` clause serves to prove the termination of `while` loops and recursive functions. This is obligatory due to the halting problem, a famous computational limitation which precludes the creation of a program that can always determinate if another program terminates. As such, we have to provide a non-negative expression that strictly decreases in every iteration. In this case, `abs y`, the absolute value is essential for dealing with initial values of x or y that are negative, since the subsequent values of y may bounce between positive and negative values, hence not making a clear monotonically decreasing expression.

2.2 McCarthy 91 function

An historical example in Formal Verification is the McCarthy 91 function [12]. It was developed with the purpose of testing verification tools. In the past, it was viewed as a difficult problem due to its nested recursion, in particular, for automated tools. At the time of writing this book, it does not pose as much of a challenge any more due to significant advancements in automated tools. Mathematically, the 91 function is defined as:

$$M(n) = \begin{cases} M(M(n+11)), & \text{if } n \leq 100 \\ n - 10, & \text{if } n > 100 \end{cases}$$

The particularity of this function is that it converges to 91 for any integer value of $n \leq 100$. For $n = 101$ it also results in 91, however, from there it increases by 1 in comparison to the previous, for instance, $M(102) = 92$, $M(103) = 93$, and so forth. While this particularity is hard to visualize at first glance, it is clear when analysing the recursive calls for a given n, for example:

$$
\begin{aligned}
M(88) &= M(M(99)) \\
&= M(M(M(110))) \\
&= M(M(100)) \\
&= M(M(M(111))) \\
&= M(M(101)) \\
&= M(91) \\
&= M(M(102)) \\
&= M(92) \\
&= M(M(103)) \\
&= \ldots && \text{(The pattern from M(91) onwards repeats)} \\
&= M(100) \\
&= M(M(111)) \\
&= M(101) \\
&= 91
\end{aligned}
$$

Based on this example, it is possible to see that the recursive calls easily form patterns, such as the one from $M(M(99))$ to $M(M(101))$ or from $M(91)$ to $M(100)$.

Representing a recursive mathematical function in OCaml is quite straightforward, by using an `if-expression` to model the two branches:

GOSPEL + OCaml
```
let rec f91 n =
  if n <= 100 then f91(f91 (n + 11))
  else n - 10
(*@ r = f91 n
    variant 101 - n
    ensures n <= 100 -> r = 91
    ensures n > 100 -> r = n - 10 *)
```

Due to the simplicity of the function, we have also decided to present the specification. In terms of pre-conditions, there are no restrictions to the value of n, since this function also allows negative numbers, as such, we can omit the `requires` clause. The correctness of this function can be expressed conditionally, based on the two possible outcomes, with logical implications: one for $n \leq 100$ where the result is always 91, and another for $n > 100$, where the result is $n - 10$. Alternatively, in GOSPEL, one may use an `if-expression` (or even pattern matching) to express conditional properties:

GOSPEL
```
(*@ r = aux n
    variant 101 - n
    ensures if n <= 100 then r = 91
            else r = n - 10 *)
```

To prove termination we have to find a non-negative monotonically decreasing expression. In this case, the only variable at our disposal is n, as such, we have to include it somehow. If we analyse the outermost function calls in the $M(88)$ execution example, it is possible to observe that the argument is increasing, so to obtain a decreasing expression we must use $-n$ as a term. Another possible observation, is that n converges to 101, exactly one of the boundary values of function M, and the one that fits inside the non-recursive case. This leads to the expression $101 - n$ that can prove the termination.

2.3 Euclidean Division

Until now, we have covered two examples with purely functional implementations. So, in this section we will present an imperative version of the Euclidean Division algorithm. This algorithm is known to perform divisions through repeated subtractions, and is based on the previously mentioned lemma 2. The lemma states that for integers x and y the equation $x = y * q + r$ with $b \neq 0$ and $0 \leq r < |b|$ is unique. The second condition ($0 \leq r < |b|$) is exactly what restricts the equation to a single solution. So, if we, momentarily, disregard it, then it is possible to find multiple solutions, for instance:

$$\begin{aligned}
&\text{Let } x = 13 \text{ and } y = 6, \text{ then} \\
&13 = 6 * 0 + 13 \quad (q = 0; r = 13) \\
&13 = 6 * 1 + 7 \quad\ \ (q = 1; r = 7) \\
&13 = 6 * 2 + 1 \quad\ \ (q = 2; r = 1)
\end{aligned}$$

If we look closely at the previous example, then it is possible to find a pattern, if we increase q by unit then we must subtract y from r. This is exactly the process found in the Euclidean Division algorithm, and can easily be proven mathematically:

$$\begin{aligned}
&x = y * (q + 1) + (r - y) \equiv \\
&x = y * q + y + r - y \equiv \\
&x = y * q + r
\end{aligned}$$

Algorithm 2 (Euclidean Division) *Given a non-negative integer, x, and a positive integer, y, find the quotient, q, and remainder, r, that comply with Euclid's lemma.*

1. *Set $r \leftarrow x$ and $q \leftarrow 0$;*
2. *While $r \geq y$ do: set $r \leftarrow r - y$ and $q \leftarrow q + 1$*
3. *Return q and r.*

The first step effectively amounts to saying that x is equal to itself, since $q = 0$. Not only is this the safest choice possible, but it is also the only correct one, because when dealing with a smaller divided, x, in comparison to the divisor, y, it respects the following result:

$$\begin{aligned}
&\text{Let } y > x, \text{ then let } q = 0 \text{ and } r = x \\
&x = y * 0 + x \equiv \\
&x = x \\
&\text{Note that } 0 \leq r < |y|, \text{ since } 0 \leq x < y \\
&\therefore \text{ Euclid's lemma is valid in these conditions.}
\end{aligned}$$

On the other hand, if we are dealing with a larger divided, eventually the remainder will be in the Euclid's lemma conditions, because the loop's exit condition is $r < y$, and given that in this implementation y is strictly positive, then it is equivalent to $r < |y|$, in this context. This algorithm can be implemented in OCaml as:

OCaml

```ocaml
let euclidean_div x y =
  let r = ref x in
  let q = ref 0 in
  while !r >= y do
    r := !r - y;
    q := !q + 1;
  done;
  (!q, !r)
```

Syntactically, a mutable variable is declared in OCaml with the `ref` keyword preceding the initial value, which can later be changed using the attribution operator (`:=`) or accessed using the `!` operator. Unlike other languages, such as C or Java, the `;` in OCaml does not represent the termination of a command, instead, it is used to sequence multiple operations. It is commonly used in conjunction with imperative features, since the result of a sequence operation is the result of its last expression, meaning that the previous results are discarded, and are, generally, expected to produce side effects, such as printing or altering the memory state.

The previous code can be proven with the following specification:

GOSPEL + OCaml

```ocaml
let euclidean_div x y = (* ... *)
(*@ (q, r) = euclidean_div x y
    requires x >= 0
    requires y > 0
    ensures x = y * q + r
    ensures 0 <= r < y *)
```

For presentation purposes we omitted the body of the function. However, it is worth noting that it contains a `while` loop that must also be specified, which can be found in the listing below. As previously mentioned, this implementation of the Euclidean Division algorithm is only capable of handling non-negative dividends ($x \geq 0$) and positive divisors ($y > 0$), since the division by 0 is not defined. The first `ensures` clause correspond to the Euclidean lemma equation, while second clause concerns the condition on the remainder to make it and the quotient unique values.

GOSPEL + OCaml

```
while !r >= y do
(*@ invariant x = y * !q + !r
    invariant 0 <= !r
    variant !r *)
        (* ... *)
done;
```

Inside the `while` loop the $0 \geq r < |y|$ is not valid, since $r > y$, however, we must guarantee that r remains positive, so that the condition is valid at the end of the loop. However, the equation is always respected as we have proven mathematically before. Given that the value of r is strictly positive and decreases in every iteration, it suffices to prove termination in the `variant` clause.

2.4 Fibonacci sequence

The Fibonacci sequence is well-known for its primary characteristic: each element is the sum of the two previous values (with the exception of the first elements). Its first twelve elements are:

$$\text{Fibonacci sequence: } 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...$$

In this textbook we consider 0 to be the first element, although there is some discourse on whether the Fibonacci sequence starts with 0 or 1. Given our choice, the corresponding mathematical definition is:

$$\text{Fib}(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ \text{Fib}(n-1) + \text{Fib}(n-2), & \text{if } n \geq 2 \end{cases}$$

In contrast to the previously presented examples, the Fibonacci sequence is not described by a linear equation or simple conditional case study, but rather by the mathematical function above. As such, we present one more feature available in GOSPEL, logical functions:

GOSPEL

```
(*@ function rec fib (n: int) : int =
    if n <= 1 then n
    else fib (n-1) + fib (n-2) *)
(*@ requires n >= 0
    variant n *)
```

These functions can be used to express complex behaviours from a logical perspective. However, when multiple versions are available (for instance consider the recursive, memoized or the iterative versions of the Fibonacci sequence) it is recommended to pursue simplicity instead of performance, as the proof may become more demanding. For this reason, we have selected the recursive version of the Fibonacci sequence for the logical definition. Usually, when one defines a logical function, it is because there is no other way to express the corresponding behaviour, so, we have to manually ensure that it is correct. Although, it is possible to annotate the function with more general properties, such as only receiving non-negative numbers as input (`requires n >= 0`).

By using the recursive Fibonacci variant as a logical function, we are able to verify any Fibonacci implementation, including itself:

GOSPEL + OCaml

```
let rec recursive_fib n =
  if n <= 1 then n
  else recursive_fib (n-1) + recursive_fib (n-2)
(*@ res = recursive_fib n
    requires n >= 0
    variant n
    ensures res = fib n *)
```

The biggest difference is that we are now able to guarantee the correctness of the result, in other words, ensure that it is in fact the n-th number in the Fibonacci sequence. However, beware that this is not a good practice. Ultimately, what we did was to copy the implementation of our desired algorithm and adapted it to a logical function, not only is this very uninteresting from a verification perspective, since it is the same code with a few syntactic differences, but it may induce subtle errors, given that a faulty logical function will negatively condition the proof. Ideally, we should aim to verify a different version of the algorithm, for instance, dynamic programming applied to Fibonacci:

GOSPEL + OCaml

```
let fib_iter n =
  let y = ref 0 in
  let x = ref 1 in
  for i = 0 to n - 1 do
  (*@ invariant !y = fib i
      invariant !x = fib (i+1) *)
    let sum = !y + !x in
    y := !x;
    x := sum
  done;
  !y
(*@ res = fib_iter n
    requires n >= 0
    ensures res = fib n *)
```

Recursion, despite being a beloved feature by the functional programming enthusiast, has a clear archenemy, if used naively: finite memory, a sufficiently large input may generate more function calls than memory available. Moreover, the lack of memory is also a common problem, this means that the same input may be recalculated multiple times. The recursive Fibonacci is commonly used to exemplify these limitations, due to the two recursive calls. To solve these problems there are numerous optimization techniques, with one of the best being a domain-specific solution that iteratively calculates the current value (x) and the next (y). From a verification perspective, the annotation of the main function is quite similar from the previous example, the input number n should be non-negative, and the result should indeed correspond to the n-th number in the Fibonacci sequence. Given that this solution uses a `for` loop, which has a finite number of steps, there is no need to prove termination, however, we must guarantee a few invariant properties, i.e. properties that are true both at the start and end of an iteration, but may break in the middle of the iteration. In particular, the y variable must contain the value in the sequence of the current index, while x contains the next element.

Previously, we have introduced logical functions in GOSPEL, while, alternatively, it is also possible to write these constructs directly in OCaml, using GOSPEL tags:

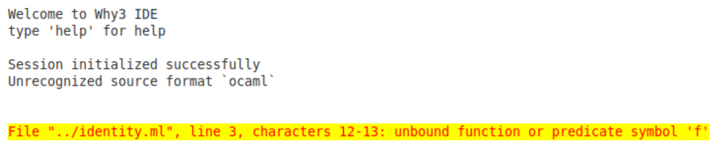
GOSPEL + OCaml

```
let[@ghost][@logic] rec fib n =
  if n <= 1 then n
  else fib (n - 1) + fib (n - 2)
(*@ r = fib n
    requires n >= 0
    variant n *)
```

One of the advantages of using this technique is to make use of OCaml features that are not present in GOSPEL, such as type inference. OCaml functions, by default, are not visible in the logical domain, even within the corresponding annotation, for instance, the following example with the identity function produces an error:

GOSPEL + OCaml

```
let f x = x
(*@ r = f y
    ensures f r = y *)
```



```
Welcome to Why3 IDE
type 'help' for help

Session initialized successfully
Unrecognized source format `ocaml`

File "../identity.ml", line 3, characters 12-13: unbound function or predicate symbol 'f'
```

Figure 2.1: Scoping error from OCaml to GOSPEL

While we are allowed to write f in the first line of the annotation, this line only serves the purpose of renaming variables, in this case, *result* (default and hidden name for the function's result) to r, and x to y. However, when specifying any kind of behaviour, calling f is prohibited, since it is not considered in scope. The `logical` tag, could be used for this purpose, however, only works for the functions below it, so a corrected (and expanded) version of the previous example would be:

GOSPEL + OCaml

```
let[@logic] f x = x
(*@ r = f y
    ensures r = y *)

let g x = f x
(*@ ensures result = f x *)
```

The ghost tag, will be discussed in more detail in the next chapter, but the basic notion is that it must not affect the result, either directly or indirectly. So, the combination of these two tags perfectly resembles the behaviour of a GOSPEL logical function, those cannot be called by other OCaml functions given their representation as OCaml comments, but are available in later GOSPEL annotations. Ghost code can be called by other OCaml functions, but when doing so, whatever code associated with it must also be ghost.

2.5 Revisiting Extended Euclidean Algorithm

From a practical standpoint, when one uses the Euclidean Algorithm it is most likely to calculate the greatest common divisor. The extended version does have realistic use cases, however, in our context it is mainly used for its logic properties to facilitate the verification process. As such, returning a triple with the Bézout's coefficients and the greatest common divisor, similar to section 2.1, contains too much information. In this section we will revisit the extended Euclidean algorithm with an imperative and iterative implementation that uses ghost code for the Bézout's coefficients.

Contrary to the recursive function, the process to calculate the coefficients becomes slightly more complex, since we are computing in the "opposite direction". In this version, we must store both the current pair of Bézout's coefficients candidates and the next. The current pair will be associated with the current x, while the next pair is associated with the current y, since the next x will be the current y:

For step n, we know that:
$x_n = a_n * x + b_n * y$
$y_n = a_{n+1} * x + b_{n+1} * y$

Where:
- x and y are the numbers from the original input
- x_k and y_k are the derived numbers from the input at step k
- a_k and b_k are the coefficients at step k

For step $n + 1$, and based on previous results, we know:

$$\begin{aligned} x_{n+1} &= y_n \\ &= a_{n+1} * x + b_{n+1} * y \\ y_{n+1} &= x_n \,\%\, y_n \\ &= x_x - y_n * q_n \end{aligned}$$

Where:
- % is the remainder operator
- q_k is the integer division of x_k by y_k

Then:

$$\begin{aligned} y_{n+1} &= x_x - y_n * q_n \\ &= (a_n * x + b_n * y) - (a_{n+1} * x + b_{n+1} * y) * q_n \\ &= (a_n * x) - (a_{n+1} * x * q_n) + (b_n * y) - (b_{n+1} * y * q_n) \\ &= (a_n - q_n * a_{n+1}) * x + (b_n - q_n * b_{n+1}) * y \\ &= a_{n+2} * x + b_{n+2} * y \end{aligned}$$

$$\therefore\ a_{n+2} = a_n - q_n * a_{n+1} \ \wedge\ b_{n+2} = b_n - q_n * b_{n+1}$$

The previous results demonstrate the formula to update the coefficients candidates, in particular, and using auxiliary variables:

$a_{aux} \leftarrow a_{current}$	$b_{aux} \leftarrow b_{current}$
$a_{current} \leftarrow a_{next}$	$b_{current} \leftarrow b_{next}$
$a_{next} \leftarrow a_{aux} - q * a_{next}$	$b_{next} \leftarrow b_{aux} - q * b_{next}$

The last remaining detail that needs to be discussed is what values should be used to initialize the coefficient candidates:

At step 0, $x_0 = x$ and $y_0 = y$, as expected.
So, using the generic formulas from before:
$x_0 = a_0 * x + b_0 * y \equiv \qquad (x_0 = x)$
$x = a_0 * x + b_0 * y$

$y_0 = a_1 * x + b_1 * y \equiv \qquad (y_0 = y)$
$y = a_1 * x + b_1 * y$
The most natural candidates are:
$a_0 = 1$, $b_0 = 0$, $a_1 = 0$ and $b_1 = 0$

With this in mind, we can now present the algorithm implemented in OCaml:

GOSPEL + OCaml

```
let gcd x y =
  let xs = ref x and ys = ref y in
  let [@ghost] a = ref 1 and [@ghost] a_next = ref 0 in
  let [@ghost] b = ref 0 and [@ghost] b_next = ref 1 in

  while !ys > 0 do
    let[@ghost] q = !xs / !ys in
    let r = !xs mod !ys in
    xs := !ys;
    ys := r;

    let a' = !a in
    a := !a_next;
    a_next := a' - q * !a_next;

    let b' = !b in
    b := !b_next;
    b_next := b' - q * !b_next;

  done;
  !xs
```

This code closely follows what we have discussed previously, we start by initializing six references, so that the value may be changed later. Reference xs will, eventually, contain the result and is derived from the argument x, similarly, ys is derived from y, although it is used to check the termination of the algorithm. Additionally, a and b are the current coefficient candidates, a is associated with x, while b is related to y. Since, this version requires to be one step ahead, a_next and b_next contain the values of the coefficient candidates in the next step, respectively. By using the `[@ghost]` tag these values are strictly used for logical purposes. Inside the `while` loop, the new value of xs will be the current value of ys, while ys is updated to the remainder of xs by ys (before updating the values). In terms of the coefficients, we use the mathematical formulas we have reached and demonstrated previously. The main function can be specified as:

GOSPEL + OCaml
```
let gcd (x:int) (y:int) = (* ... *)
(*@ r = gcd x y
    requires x >= 0
    requires y >= 0
    ensures exists a,b. r = a*x+b*y *)
```

This implementation is not ready to deal with negative numbers, so for it to function well, we must restrict the domain of the inputs to the non-negative numbers. As previously mentioned, `exists` statements are notably hard for machines to prove, since the value space might be extremely large and checking every single value is not a valid strategy. Instead, we, as programmers, must help the machine, and this is possible with the following specification on the `while` loop:

GOSPEL + OCaml
```
while !ys > 0 do
(*@ invariant !xs >= 0
    invariant !ys >= 0
    invariant !xs = !a * x + !b * y
    invariant !ys = !a_next * x + !b_next * y
    variant !ys *)
  (* ... *)
done;
```

Undoubtably, the two last invariants, these being:

- `invariant !xs = !a * x + !b * y`
- `invariant !ys = !a_next * x + !b_next * y`

are very important conditions, why? The former guarantees that, when exiting the loop, the post-condition `ensures exists a,b. r = a*x+b*y` is true, since `xs` contains the final result, `r`, and we provide one example of a concrete instantiation for the existential quantifier (through the ghost references `a` and `b`). On the other hand, the latter `invariant` serves to guarantee that on the next iteration `xs` also complies with the former `invariant`, since the reference `xs` is updated with the value of the reference `ys`, as well as `a` and `b` with `a_next` and `b_next`, respectively. The two other invariants conditions, `!xs >= 0` and `!ys >= 0` serve to ensure that the values are well-behaved and are within the expected interval, which is the set of non-negative integers. To prove termination, the value of the `ys` reference (`!ys`) suffices, since it is positive and decreases each iteration.

2.6 Fast Exponentiation

Fast Exponentiation, also known as Exponentiation by Squaring, is an efficient algorithm to computer the power of a number, and is particularly relevant when dealing with large exponents. It is mathematically defined as:

For $n > 0$, then:

$$x^n = \begin{cases} (x^2)^{\frac{n}{2}}, & \text{if } n \,\%\, 2 = 0 \\ x * (x^2)^{\frac{n-1}{2}}, & \text{if } n \,\%\, 2 = 1 \end{cases}$$

This is possible due to the following properties:
- $n^a * n^b = n^{a+b}$
- $(n^a)^b = n^{a*b}$

So:
- when n is even: $x^n = x^{2*\frac{n}{2}} = (x^2)^{\frac{n}{2}}$
- when n is odd: $x^n = x * x^{n-1} = x * x^{\frac{2*(n-1)}{2}} = x * (x^2)^{\frac{n-1}{2}}$

From a computational perspective, this method allows to progressively calculate smaller components, in the form of powers of two, rather than a single larger exponent at once, for instance:

$$\begin{aligned} 3^5 &= 3 * (3^2)^{\frac{5-1}{2}} && \text{(simplify)} \\ &= 3 * 9^2 && \text{(apply fast exp.)} \\ &= 3 * (9^2)^{\frac{2}{2}} && \text{(simplify)} \\ &= 3 * (81)^1 && \text{(apply fast exp.)} \\ &= 3 * 81 * (81^2)^{\frac{1-1}{2}} && \text{(simplify)} \\ &= 3 * 81 * 6561^0 && \text{(simplify)} \\ &= 3 * 81 * 1 && \text{(simplify)} \\ &= 243 \end{aligned}$$

Although applying the fast exponentiation method when $n = 2$ or $n = 1$ is mathematically redundant, it is still computationally advantageous. By following the same procedure uniformly, the final result is always stored in a single variable, which simplifies both the program and its proof. This algorithm can be represented in pseudocode as:

Algorithm 3 (Fast Exponentiation) *Given an integer, x, and a non-negative integer, n, find x^n using the exponentiation by squaring technique.*

```
1. Create $r \leftarrow 1$, $p \leftarrow x$ and $e \leftarrow n$;
2. While $e > 0$ do:
    (a) If $e$ is odd then set $r \leftarrow r * p$;
    (b) Set $p \leftarrow p * p$;
    (c) If $e$ is odd then set $e \leftarrow (e - 1)/2$ else set $e \leftarrow e/2$
3. Return $r$.
```

To calculate the final result we need three variables: r to store intermediate results, and eventually the correct value, p and e to keep the intermediate bases and exponents, respectively. The main idea behind this algorithm is that every step the equation $x^n = r * p^e$, until $e = 0$ and $x^n = r$. While the exponent is positive, the first operation of the loop body checks for its parity, if it is odd, then r is updated with the previous value times the current base p, this corresponds to the mathematical operation of separating x^n into $x * x^{n-1}$, with the exception that we do not update the exponent, right away, instead, it is done on the third operation. The second operation updates p by raising it to the power of two, or rather, by multiplying p with itself, this corresponds to the x^2 present in both cases. The previous example is calculated computationally as:

$$\begin{array}{lll}
\text{(Step 0)} & r = 1; p = 3; e = 5 & (\text{Is } e > 0\text{? Yes!}) \\
\text{(Step 1)} & r = 3; p = 9; e = 2 & (\text{Is } e > 0\text{? Yes!}) \\
\text{(Step 2)} & r = 3; p = 81; e = 1 & (\text{Is } e > 0\text{? Yes!}) \\
\text{(Step 3)} & r = 243; p = 6561; e = 0 & (\text{Is } e > 0\text{? No!})
\end{array}$$

Result: 243

This algorithm can be implemented in OCaml as:

OCaml

```ocaml
let fastexp x n =
  let r = ref 1 in
  let p = ref x in
  let e = ref n in
  while !e > 0 do
    if !e mod 2 = 1 then r := !r * !p;
    p := !p * !p;
    e := !e / 2;
  done;
  !r
```

The most notable difference to the algorithm description is the lack of an `if`-statement on the third instruction inside the `while` loop body. This simplification is possible since `/` operator in OCaml represents the integer division. When e is odd, `e/2` and `(e-1)/2` yield the same result, because `e/2` mathematically produces a rational number, therefore the decimal place is truncated to retain the representation as integer and avoid over-approximations. Another important detail to notice is that the first sequence operator (`;`) refers to the `if`-statement as a whole, rather that solely to the `then` branch.

With this context, we are ready to prove the fast exponentiation algorithm. In the first place, we have to define a logical power function:

GOSPEL

```
(*@ function rec power (x n: int) : int =
    if n = 0 then 1
    else x * power x (n-1) *)
(*@ requires n >= 0
    variant n *)
```

Once more, the recursive and inefficient definition suffices as simplicity matters. The computation of x to the power of n is simply multiplying x by itself n times. In the context of this problem, we restrict the domain of n to the non-negative integers. This logical definition can be used to annotate the `fastexp` function:

GOSPEL + OCaml

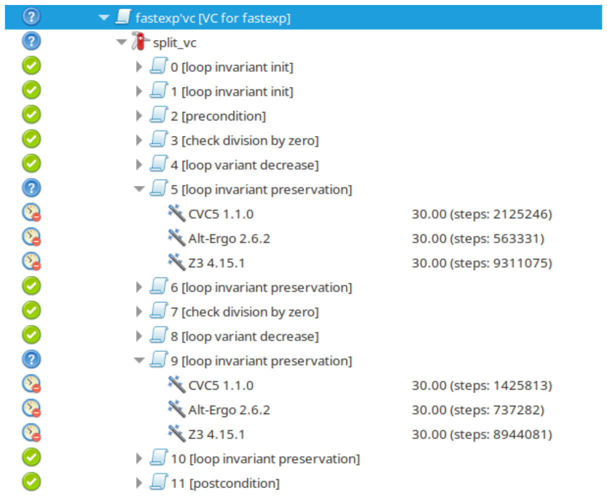
```
let fastexp x n = (* ... *)
(*@ r = fastexp x n
    requires n >= 0
    ensures r = power x n *)
```

As expected, the result r should effectively correspond to x raised to the power of n. Additionally, the restriction on n is also applied. The `while` loop is slightly more complicated:

GOSPEL + OCaml

```
while !e > 0 do
(*@ invariant !r * power !p !e = power x n
    invariant !e >= 0
    variant !e *)
  (* ... *)
end;
```

Starting with the termination proof, in this case it is as simple as the value of the reference e, since it decreases every iteration by approximately half, which also guarantees that it says positive given that the integer division between two positive numbers does not return a negative number, at most it is 0, which is the expected value in the last iteration. The second invariant clause serves to guarantee that the exponent does not become negative, due to what we have explained before. However, when launching the provers in the Why3 platform, even with the maximum time limit available, it is clear that the proof is incomplete, why is this?



```
? fastexp'vc [VC for fastexp]
?   split_vc
✓     0 [loop invariant init]
✓     1 [loop invariant init]
✓     2 [precondition]
✓     3 [check division by zero]
✓     4 [loop variant decrease]
?     5 [loop invariant preservation]
          CVC5 1.1.0          30.00 (steps: 2125246)
          Alt-Ergo 2.6.2      30.00 (steps: 563331)
          Z3 4.15.1           30.00 (steps: 9311075)
✓     6 [loop invariant preservation]
✓     7 [check division by zero]
✓     8 [loop variant decrease]
?     9 [loop invariant preservation]
          CVC5 1.1.0          30.00 (steps: 1425813)
          Alt-Ergo 2.6.2      30.00 (steps: 737282)
          Z3 4.15.1           30.00 (steps: 8944081)
✓     10 [loop invariant preservation]
✓     11 [postcondition]
```

Figure 2.2: Failed Verification Attempt

The previous algorithms could be verified very quickly, additionally, the presented code does not seem to be that much more complex than the rest. This statement is true, time itself is not the issue. Is the previous specification incorrect? Not exactly, it correctly models the logical properties we have discussed beforehand. Instead, there are a few properties concerning the power operation that the theorem provers are not aware. Computers, in general, not just theorem provers or proof assistants, are very good in the domains of mathematics and logic, however, their deduction capabilities are not so stellar, as they are not aware of many properties that may seem intuitive to us as humans. The research and development teams behind verification tools make serious efforts to provide their users with powerful verification libraries with properties such as lemmas, axioms, predicates or even logical functions, however, these libraries must not contain errors, otherwise the logic will be faulty, which is a considerable effort, and many properties will have to be user defined.

In Cameleer, currently, there is not much support regarding the power operation, as such, we will have to define our own lemmas and prove them. And the most natural question to follow-up is: what lemmas do we need? First, let's analyse the first failed goal (number 5) and the respective hypothesis. For goal

5 (using mathematical notation):

$$\begin{aligned}
H_0&\colon\ e_1 \,\%\, 2 = 1\\
H_1&\colon\ e_1 > 0\\
H_2&\colon\ e_1 >= 0\\
H_3&\colon\ r_1 * p_1^{e_1} = x^n\\
H_4&\colon\ n >= 0\\
\\
\text{Let:}\ & r = (r_1 * p_1)\\
& p = (p_1 * p_1)\\
& e = \lfloor e_1/2 \rfloor\\
\\
\text{Goal:}\ & r * p^e = x^n
\end{aligned}$$

Let's expand the right-hand side of the equation in hypothesis number three:

$$\begin{aligned}
r_1 * p_1^{e_1} &= r_1 * (p_1^2)^{e_1/2}\\
&= r_1 * p_1 * (p_1^2)^{\lfloor e_1/2 \rfloor}\\
&= (r_1 * p_1) * (p_1 * p_1)^{\lfloor e_1/2 \rfloor}\\
&= r * p^e
\end{aligned}$$

Despite the variable names, this process is the same as saying $x_n = x * (x^2)^{(n-1)/2}$, when n is odd. The last step is trivial to the provers, it simply replaces variables names with their define. The issue is between steps one and two (for presentation purposes is separated into two steps, but may be represented computationally as one). Ideally, we want a lemma that states:

$$\forall n, x \in \mathbb{Z} : n \geq 0 \wedge n \,\%\, 2 = 1 \rightarrow x^n = x * (x^2)^{\lfloor n/2 \rfloor}$$

This lemma can be implemented in GOSPEL as:

GOSPEL
```
(*@ lemma power_odd : forall x n: int.
    n >= 0 && mod n 2 = 1 ->
      power x n = x * (power (x*x) (div n 2)) *)
```

Unlike functions, lemmas do not receive parameters, any variable must be declared with a quantifier. In GOSPEL, a quantifier declaration ends with a dot (.), and type annotations are preceded by a colon (:) (sometimes can be omitted). One syntactical difference in GOSPEL, when compared to OCaml, is that `div` and `mod` are treated as functions, rather than operators, therefore these names precede the input numbers, which are the arguments. Once this lemma is placed in the program (before the function `fastexp`), the previously mentioned goal is now easily cleared. Furthermore, a proof goal for the lemma was also generated, some lemmas can be discharged automatically by the provers, in this case we have to prove it, but we will save it for later. For now, let's consider the second failed goal (number 9), using mathematical notation:

$$\begin{aligned}
H_0&: e_1 \,\%\, 2 \neq 1\\
H_1&: e_1 > 0\\
H_2&: e_1 >= 0\\
H_3&: r * p_1^{e_1} = x^n\\
H_4&: n >= 0\\
\\
\text{Let}&: p = (p_1 * p_1)\\
& e = e_1/2\\
\text{Goal}&: r * p^e = x^n
\end{aligned}$$

Not surprisingly, we can mathematically prove the previous goal with the following procedure:

$$\begin{aligned}
r * p_1^{e_1} &= r * (p_1^2)^{e_1/2}\\
&= r * (p_1 * p_1)^{e_1/2}\\
&= r * p^e
\end{aligned}$$

Similarly, the provers are not aware of the property used in the first step. So, we need to define a similar lemma, mathematically defined as:

$$\forall n, x \in \mathbb{Z} : n \geq 0 \land n \,\%\, 2 = 0 \rightarrow x^n = (x^2)^{\frac{n}{2}}$$

And written in GOSPEL as:

GOSPEL
```
(*@ lemma power_even : forall x n: int.
    n >= 0 && mod n 2 = 0 ->
      power x n = (power (x*x) (div n 2)) *)
```

Once more, this lemma is not automatically discharged by the provers yet. However, we can observe that the `fastexp` function is now fully verified. So, now we have to prove the lemmas. Why is this important? Well, because the Why3 platform uses the lemma as a hypothesis in the next proof goals, independently of being proven beforehand or not. On one hand, this design decision is very useful for prototyping lemmas and prove them afterwards. On the other hand, it puts more responsibility on the user, and may induce lesser experienced users in error, since the lemma itself might be logically inconsistent. So, how can we prove a lemma in GOSPEL? Simple lemmas may be automatically discharged by the provers, when that is not the case, we need to defined what is called as a lemma function. Instead of using the lemma construct in GOSPEL, we will define a function in OCaml with the lemma tag. The annotations on this function correspond to the lemma itself, while its body is the proof. One should start by expressing the specification first, then move to the proof itself. The previous lemmas can be considered obsolete, so we recommended adapting them to lemma functions, starting with the `power_even` lemma:

GOSPEL + OCaml
```
let[@lemma] power_even (x: int) (n: int) =
(* TODO *)
(*@ requires n >= 0
    requires mod n 2 = 0
    ensures power x n = (power (x * x) (div n 2)) *)
```

Ultimately, we are dealing with a regular OCaml function, so we can use the usual keywords, such as `requires`, `ensures` and `variant`. So, now the question is: how do we approach this proof? There are numerous methods that can be applied when proving, however, one important detail that is universal to all lemma functions is that the function itself must be of the unit type, effectively meaning that we cannot return any proper kind of value. Instead, we have to make clear for the provers that the post-condition is true under that context. Let's consider a concrete example, for instance 2 raised to the power of 16:

$$\begin{aligned} 2^{16} &= (2^2)^{16/2} = 4^8 && \text{(True)} \\ 4^8 &= (4^2)^{8/2} = 16^4 && \text{(True)} \\ 16^4 &= (16^2)^{4/2} = 256^2 && \text{(True)} \\ 256^2 &= (256^2)^{2/2} = 65536 && \text{(True)} \end{aligned}$$

Power	Result
2^{16}	65536
4^8	65536
16^4	65536
256^2	65536

As we can see in this example, however, even for a small base and a relatively small exponent, the resulting number is quite large, since it grows exponentially. Therefore, directly calculating any two inputs would be unfeasible. Instead, we must do a proof by induction, which can be achieved with recursion:

GOSPEL + OCaml
```
let[@lemma] rec power_even (x: int) (n: int) =
  if n > 1 then power_even x (n-2)
(*@ requires n >= 0
    requires mod n 2 = 0
    variant n
    ensures power x n = (power (x * x) (div n 2)) *)
```

Before explaining the body, it is important to notice the two differences with the previously presented `power_even` lemma function: it became a recursive definition with the `rec` keyword, and we added a `variant` clause with the value of n to prove termination, since n will decrease between every recursive call. Let's now focus on the body of the function, two questions may arise: "What is the base case?" and "What is the inductive step?". The answer to the latter, as one might suspect, is the `then` branch of the `if`-statement. The answer to the former, on the other hand, may be harder to notice: in OCaml, and other functional programming languages, an `if`-statement always produces a value, in contrast to the imperative languages, as such, omitting the `else` branch will produce the unit value when the tested condition fails, and as a consequence, the type checker enforces the `then` branch to evaluate to the unit value for type consistency. Based on the `requires` clauses, we know that the only number that fails the `if` check is 0. There is no need to use the `assert` construct, since the `ensures` clause applied to a given x and 0, $x^0 = (x^2)^0$, is trivial to prove given the logical definition of the `power` function, when the exponent is 0 it evaluates to 1 independently of the base value. The inductive step is trickier, the strategy behind it is to subtract two from the exponent:

$$\text{Let } n, x \in \mathbb{N} : n > 1, \text{ then:}$$

$$\begin{aligned} x^n &= x^2 * x^{n-2} \\ &= x^2 * (x^2)^{\frac{n-2}{2}} \\ &= x^2 * (x^2)^{\frac{n}{2}-1} \\ &= (x^2)^{\frac{n}{2}} \end{aligned}$$

So, why is this result, in particular, important? If we observe the third step closely, it is possible to see the resemblance between that mathematical expression and the `else` branch of the `power` logical function with arguments $x * x$ and `div n 2`, which is exactly the transition to the fourth step and that the provers can easily achieve. The simplification between steps two and three is also performed by the provers without further human assistance. The real difficulty lies in the first step, since the `power` logic function only subtracts one exponent unit at a time. Subtracting two at a time might seem obvious us humans, however, the computer is much more sceptical. That is why, we must help the provers by separating two exponent units at a time:

GOSPEL + OCaml

```
let[@lemma] power_odd (x: int) (n: int) =
  power_even x (n-1)
(*@ requires n >= 0
    requires mod n 2 = 1
    ensures power x n = x * (power (x * x) (div n 2)) *)
```

We can reuse `power_even` as proof for `power_odd`, by subtracting 1 from the exponent, for similar reasons: `power_even` states that $x^n = (x^2)^{n/2}$, so $x^{n+1} = x*(x^2)^{n/2}$ is easily verified based on the `power` logical function definition.

2.7 Exercises

Implement and specify the:

1. Euclidean division that supports negative numbers (Hint: study the 4 possible cases and develop a sign function)
2. Functional Euclidean division
3. Iterative factorial function
4. Tribonacci sequence (i.e. $0, 1, 1, 2, 4, 7, 13, 24, 44, 81, ...$)
5. Functional fast exponentiation

Chapter 3

Searching

Undoubtably, a common use for computers is to store data, especially as a group. There are numerous possible layouts to the structure containing the data, and, naturally, a problem that arises, no matter the organization, is how to search for a particular element. The various solutions to this problem are intrinsically tied to the structure of the collection. We will start with linear data structures, such as lists and arrays, and, eventually, move on to more complex representations, for instance, trees.

3.1 Linear Search

Within linear structures, the most natural searching strategy is to traverse the data element by element from one end to the other, this is known as the linear search algorithm. Let us consider the following data, assume we want to find the index of value 0:

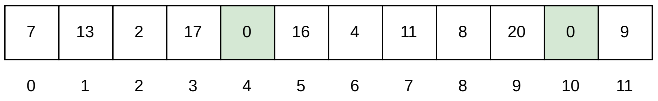


Figure 3.1: A sequence

There are two instances of the value 0, the first at index 4 and the second at index 10. In this particular case, it would actually be faster to search the data structure backwards, however, that is not always the case. Without any kind of prior information concerning the disposition of the elements, we have to pre-establish a direction for consistency and simplicity purposes. Latin-based languages, such as English, are written from left to right, as such, we will use this direction in this section's algorithm, as it is the most natural and intuitive in the context of this textbook.

Algorithm 4 (Linear Search) *Given a value, x, find its index, if it exists in a linear structure, s.*

```
1. Create variable $i \leftarrow 0$;
2. While $i < length(s)$ do:
    (a) Compare $x$ with $s[i]$:
        i. If $x = s[i]$ return $i$
        ii. If $x \neq s[i]$ set $i \leftarrow i + 1$
    (b) Go back to step (2)
3. Return error value (Not found)
```

Another implementation decision originates from the multiple strategies to deal with the error case, i.e. when the desired value is not inside the linear data structure. One possible choice is to encode a value specifically to the error case. This value must not cause conflict in any way possible, which leads to commonly using -1 (or any other negative number, although less common). Another viable option is to use the `Option` data type, which encapsulates both successful and error cases, with the `Some` and `None` values, respectively. The `Some` expects an argument, while `None` does not. Alternatively, it is also possible to use the exception mechanism in OCaml in order to raise an error. In this chapter, we will explore all three possibilities, in that order.

Based on the previous decisions, we arrive at out OCaml implementation of the linear search algorithm for lists:

OCaml

```ocaml
let linear_search a v =
  let exception Break of int in
  try
    for i = 0 to Array.length a - 1 do
      if a.(i) = v then raise (Break i)
    done;
    -1
  with Break i -> i
```

Despite using the exception mechanism, our strategy with this implementation is to encode -1 as the not found value. The `Break` exception serves to stop the `for` loop earlier if the value was found, which otherwise would not be possible in OCaml. With language details out of the way, this implementation simply amounts to traversing an array from its first index until either the first instance of the desired element is found or the end of the array if not found.

Unlike the previous chapter there is no mathematical equation that we must comply to in every iteration. However, that does not mean that we cannot logically express the excluded area, and by excluded we mean what has already been searched:

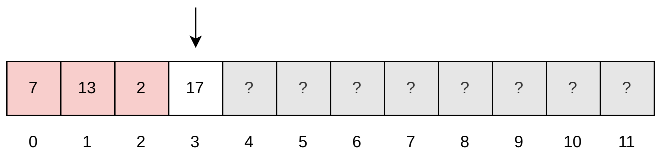


Figure 3.2: Linear Search Excluded Area

Let's consider the example above, where $i = 3$, then this means that for every integer between 0 (inclusive) and 3 (exclusive), we know that the corresponding value in the array is different from the desired value. This property can be expressed logically with the for all quantifier:

$$\text{Let } a \text{ be an array in}$$
$$\forall k \in \mathbb{Z} : 0 \leq k < i \leftarrow a[k] \neq v$$

When dealing with arrays it is generally important to express the parts of the array that are already concluded. In this case, modelling the property above is crucial to prove the annotations of the `linear_search` function, and can be done so:

GOSPEL + OCaml
```
for i = 0 to Array.length a - 1 do
(*@ invariant forall k. 0 <= k < i -> a.(k) <> v *)
  (* ... *)
done
```

For the most part the `invariant` is quite similar, with some syntactic differences, these being that the integer type could be omitted (may not always be the case), to access an array position OCaml uses the `a.(i)` notation, which is quite different from most programming languages, and the different symbol is represented as `<>`. This condition is essential to prove the `linear_search` function:

GOSPEL + OCaml
```
let linear_search a v = (* ... *)
(*@ r = linear_search a v
    ensures r >= 0 -> a.(r) = v
    ensures r = -1 -> forall k. 0 <= k < Array.length a ->
      a.(k) <> v *)
```

If the element was not found, which happens when $r = -1$, then we state that for every single index from 0 (inclusive) to `length`(a) (exclusive) the element within is different from the desired value. Otherwise, when $r >= 0$, we must guarantee that corresponding index effectively contains the desired value.

3.2 Binary Search

In the worst case, the desired element of a potentially very large sequence is its last, which means traversing every single element. Even on average, we are expected to iterate over a substantial part of the sequence. So, how can we optimize the search process? Sometimes we can use context to our advantage: let's say we have a sorted sequence:

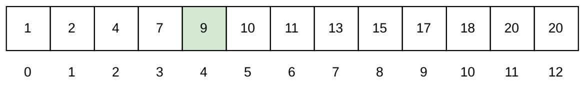


Figure 3.3: A sorted Sequence

In this case, we are looking for the number 9. We have two crucial pieces of information, the element we are looking for and that the sequence is sorted. In this situation, if we look at the content of a given index, we know for certain which direction to go next if the element is different:

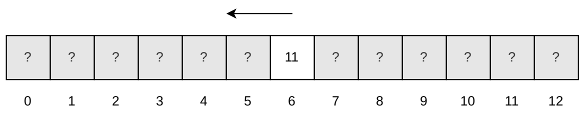


Figure 3.4: Which direction?

For instance, we are currently at index 6, then if we look at its element, 11, we know that the number we are looking for, 9, is to the left of our current position. Moreover, we can actually divide the sequence in half every iteration:

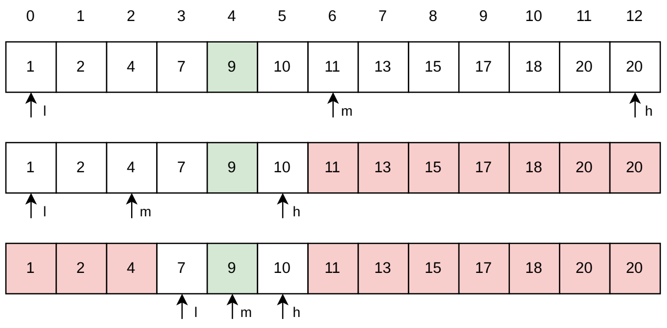


Figure 3.5: Binary Search example

By having two pointers, one for the lower end and the other for the higher end of our search area, we can calculate the middle point, which will serve to decide the direction to seek next, if the element was not the correct one. Additionally, we can change the pointers to exclude the opposite direction and the middle point itself, and restrict the search area by half. There are other cases worth discussing, the three pointers may reference the same index, this means that there is only one position left to check, or when the higher end pointer is smaller than the lower end pointer, this means that all positions of the sequence have been excluded, therefore the desired value is not an element. This algorithm can be described as:

Algorithm 5 (Binary Search) *Given a value, x, find its index, if it exists in a sorted linear structure, s.*

```
1. Create two variables: $l \leftarrow 0$ and $h \leftarrow$ length(s) - 1;
2. While $l \leq h$ do:
    (a) Calculate the middle point: $m \leftarrow \lfloor (l+h)/2 \rfloor$;
    (b) Compare $x$ with $s[m]$:
        i. If $x < s[m]$ set $h \leftarrow m-1$
        ii. If $x > s[m]$ set $l \leftarrow m+1$
        iii. If $x = s[m]$ return $m$
    (c) Go back to step (2)
3. Return error value (Not found)
```

Note that this algorithm only is beneficial in terms of speed if the cost of accessing a position is instantaneous. Linked lists and similar implementations are usually not adequate for this algorithm, unlike arrays. With this in mind, the binary search algorithm can be implemented in OCaml, with option values and arrays, as:

GOSPEL + OCaml

```
let binary_search a v =
  let l = ref 0 in
  let h = ref (Array.length a - 1) in
  let res = ref None in
  while !l <= !h do
    let m = (!h + !l)/ 2 in
    if a.(m) < v then l := m+1
    else if a.(m) > v then h := m-1
    else begin res := (Some m); l := m+1; h := m end
  done;
  !res
```

The option data type can be used to encapsulate both the successful and unsuccessful cases of an algorithm. It contains two values `None`, when there is no value to be returned, which can be used to express an error case, and `Some` x, which must always receive an argument with the value to return, in this case the index when the desired element was found. Another important implementation detail is how to break the loop when the element was found, to achieve this we set h to m and l to $m+1$,

Now to verify this implementation, we must remember that a pre-condition to this algorithm is that it must be sorted. How can we logically express that an array is sorted? Well, we could use a logical function, similar to when calculating the Fibonacci sequence, however, this is unusual when dealing with boolean results (unless the expression to calculate is harder to express logically rather than programmatically). Instead, we may use the following predicate defined logically as:

Let a be an array of integers, then:
$\forall i, j \in \mathbb{Z} : 0 \leq i \leq j < \text{length}(a) \rightarrow a[i] \leq a[j]$

Simply, this means that for every index i that precedes and index j (or when $i = j$), then the corresponding element of index i must be lesser or equal to the element in position j. Since this property ranges from 0 (inclusive) to `length(`a`)` (exclusive), then it effectively means that the array is sorted. In GOSPEL this predicate can be written as:

GOSPEL
```
(*@ predicate sorted (a: int array) =
    forall i j:int. 0 <= i <= j < Array.length a ->
      a.(i) <= a.(j) *)
```

Using the `sorted` predicate (or any other) in a subsequent GOSPEL annotation is quite straightforward, with the syntax being akin to calling a function with its arguments. In the context of `binary_search` function, one may write `requires sorted a`, where `a` is the array, as seen below:

GOSPEL + OCaml
```
let binary_search a v = (* ... *)
(*@ r = binary_search a v
    requires sorted a
    ensures match r with
      | None -> forall k. 0 <= k < Array.length a -> a.(k) <> v
      | Some i -> a.(i) = v *)
```

Since the result of this function is encoded as an option type, we may use pattern matching to deconstruct r. If a value was found, *i.e.* when r is a `Some` i value, then we must confirm that the element at index i is effectively the value we are searching for, this being v from the parameters. In case it is `None` value, then it means that every index ranging from 0 (inclusive) to `length(`a`)` (exclusive) stores an element that is different from v. And now for the specification of the `while` loop:

GOSPEL + OCaml
```
while !l <= !h do
(*@ invariant 0 <= !l
    invariant !h < Array.length a
    invariant match !res with
      | None -> forall k. 0 <= k < !l \/
        !h < k < Array.length a -> a.(k) <> v
      | Some i -> a.(i) = v
    variant !h - !l *)
  (* ... *)
done;
```

With the first two invariants and the loop condition we express that the lower and higher pointers are within the boundaries of the array during the search process. These conditions are separated for good reasons: (1) we cannot define a direct relation between l and h as an invariant, since it will break when exiting the loop; (2) the lower end only grows, so it is best to avoid defining an upper limit, since it might not hold when the lower limit surpasses it, in either case for termination (finding the desired element or excluding all elements); (3) similarly, we should not define a lower limit for h. Regarding the current result, in the reference `res`, we must also deconstruct it using pattern matching. If the value is found, the condition is identical to the previous annotation. The main difference is the `None` case, in which we have to exclude the elements to the left of l and to the right of h, within the array length of course. To achieve the union of these two disjoint sets of elements, we must use the logical `or` operator. Finally, to prove termination we can use the expression `!h - !l`, since the upper and lower limits are approaching each other.

3.3 Ternary Search

What if instead of diving an array in half every iteration, we could divide it into three similar parts? That is the idea behind the ternary search algorithm:

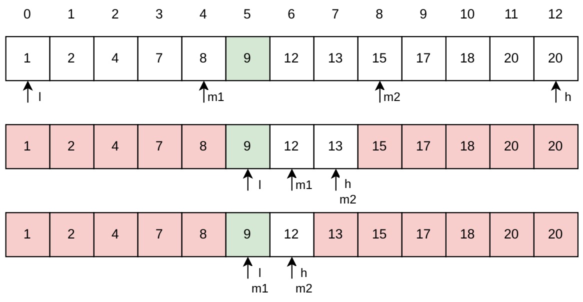


Figure 3.6: Binary Search example

At first may seem as an improvement over binary search, however that is not the case. Despite its noble intentions, ternary search does not actually achieve significant increases in performance due to the increased number of comparisons, and, at times, may actually be slower than binary search. Nonetheless, it is an interesting experiment from both the algorithmic and formal verification perspectives, with a very pertinent question being raised: "How does the specification change from Binary Search to Ternary Search". Conceptually, it still uses a lower and a higher limit, however, it calculates two middle points instead of one, as a way to divide the array in three similar parts:

Algorithm 6 (Ternary Search) *Given a value, x, find its index, if it exists in a sorted linear structure, s.*

```
1. Create two variables: $l \leftarrow 0$ and $h \leftarrow$ length(s) - 1;
2. While $l \leq h$ do:
    (a) Calculate the middle points:
        i. $m_1 \leftarrow l + \lfloor (h-l)/3 \rfloor$
        ii. $m_2 \leftarrow h - \lfloor (h-l)/3 \rfloor$
    (b) Compare $x$ with $s[m]$:
        i. If $x = s[m_1]$ return $m_1$
        ii. If $x = s[m_2]$ return $m_2$
        iii. If $x < s[m_1]$ set $h \leftarrow m_1 - 1$
        iv. If $x > s[m_2]$ set $l \leftarrow m_2 + 1$
        v. If $s[m_1] < x < s[m_2]$:
           set $l \leftarrow m_1 + 1$ and $h \leftarrow m_2 - 1$
    (c) Go back to step (2)
3. Return error value (Not found)
```

By having two middle points the array is divided into three larger parts plus the two middle points, meaning that there are, effectively, five different cases that must be taken into account. This is the main reason for the increase in the number of comparisons, which used to be three in binary search. The middle points are obtained by dividing the space between the lower and higher boundaries in three parts with approximately the same size, although due to the properties of the integer division operation slight differences in size may occur between the three parts. Ternary Search can be implemented in OCaml, using arrays and options, as follows:

OCaml
```
let ternarySearch a v =
  let l = ref 0 in
  let u = ref (Array.length a - 1) in
  let res = ref None in
  while !l <= !u do
    let m1 = !l + (!u - !l)/3 in
    let m2 = !u - (!u - !l)/3 in
    if (a.(m1) = v) then begin
      res := Some m1; l := m1+1; u := m1
    end
    else if (a.(m2) = v) then begin
      res := Some m2; l := m2+1; u := m2
    end
    else if (v < a.(m1)) then u := m1 - 1
    else if (v > a.(m2)) then l := m2 + 1
    else begin l := m1 + 1; u := m2 - 1 end
  done;
  !res
```

Note that the `begin ... end` blocks in the first two cases are optional and used for display purposes only, while the last is to avoid possible ambiguities stemming from the sequence operator. So, how can we verify this function?

GOSPEL + OCaml
```
let ternarySearch a v = (* ... *)
(*@ r = ternarySearch a v
    requires sorted a
    ensures match r with
      | None -> forall k. 0 <= k < Array.length a -> a.(k) <> v
      | Some i -> a.(i) = v *)
```

If we notice closely that is the same exact specification used in the binary search example, except for the function's name. Well, this is not surprising, since the same conditions are used in the context of the linear search algorithm, except for the pre-condition, which is not necessary in that context. What about the while loop?

GOSPEL + OCaml
```
while !l <= !u do
(*@ invariant 0 <= !l
    invariant !u < Array.length a
    invariant match !res with
      | None -> forall k. 0 <= k < !l \/
         !u < k < Array.length a -> a.(k) <> v
      | Some i -> a.(i) = v
    variant !u - !l *)
  (* ... *)
done;
```

To our surprise, the specification is the same. One may ask why, however, on closer inspection we realize the middle points do not impact the specification, in fact, it is not even possible to use them there, since the scope of an annotation on a loop does not include whatever is declared on its body. The important logical properties stem from the diminishing window comprised by the lower and upper bounds, which is exactly the same in this family of search algorithms: binary search, ternary search, quaternary search, or even nonary search and beyond.

Much like this algorithm can be seen as an experiment, so can this first part of this section. Our goal was to demonstrate that significant implementation details, such as the number of middle points, may not impact the specification. With this accomplished, we must return to our promise to demonstrate verify error values encoded as exceptions, which will use a slightly modified implementation of the ternary search:

OCaml

```
let ternarySearch a v =
  let exception Break of int in
  try
    let l = ref 0 in
    let u = ref (Array.length a - 1) in
    while !l <= !u do
      let m1 = !l + (!u - !l)/3 in
      let m2 = !u - (!u - !l)/3 in
      if (a.(m1) = v) then raise (Break m1)
      else if (a.(m2) = v) then raise (Break m2)
      else if (v < a.(m1)) then u := m1 - 1
      else if (v > a.(m2)) then l := m2 + 1
      else begin l := m1 + 1; u := m2 - 1 end
    done;
    raise Not_found
  with Break i -> i
```

Instead of using a reference to store the result (either as a number or an option), in this implementation we use exceptions to deal with the result. Similar to the linear search algorithm, we use the `Break` exception to escape the `while` loop and use a `try-with` block to return the corresponding argument, which allows omitting the instructions used in the previous implementations to break the loop. To deal with the unsuccessful case, we used the `Not_found` exception from the standard library. This exception is purposefully not caught as not to return a value. The `ternarySearch` function is adapted to:

GOSPEL + OCaml

```
let ternarySearch a v = (* ... *)
(*@ r = ternarySearch a v
    requires sorted a
    ensures a.(r) = v
    raises Not_found -> forall k. 0 <= k < Array.length a ->
      a.(k) <> v *)
```

The `raises` clause serves to denote the behaviour of the function when an uncaught exception is triggered. In this case, out of the two exceptions, only `Not_found` is uncaught, and in this scenario every element of the array must differ from the desired value. The condition itself does not change, only how to "unwrap" the result. Since an uncaught exception terminates a function abruptly without returning, then we can simply express the successful case as an `ensures` clause without any condition preceding it, unlike the previous implementations. The specification on the `while` loop also changes slightly:

GOSPEL + OCaml

```
while !l <= !u do
(*@ invariant 0 <= !l
    invariant !u < Array.length a
    invariant forall k. 0 <= k < !l \/ !u < k < Array.length a ->
      a.(k) <> v
    variant !u - !l *)
  (* ... *)
done;
```

Since we do not store the result any more, we can remove the `invariant` condition that expressed that it contained the correct index. We still, however, have to express the area that has been excluded. We need not worry about the `Not_found` exception since it is raised outside the loop.

3.4 Depth-first Search in Binary Trees

Moving on from linear data structures, a tree can either be empty or a group of nodes. Each node may be connected to multiple distinct nodes below it, known as its children, and to at most one node above it, called its parent. The only exception is the root node, which has no parent. Binary Trees are a special case of trees where each node may only have at most two children, and can easily be defined in OCaml using a variant:

OCaml

```
type 'a tree = E | N of 'a tree * 'a * 'a tree
```

The `E` constructor models an empty tree. While the `N` constructor represents a node comprised by the left subtree, an item and the right subtree. Note that a subtree is also of type `'a tree`. One of the most common searching algorithms for trees is depth-first search, which amounts to visiting one of child node and their descendants before the other nodes with the same parent:

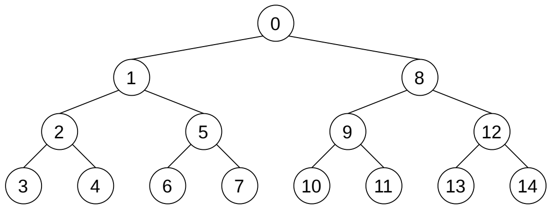


Figure 3.7: Depth-first search in a Binary Tree

This algorithm, assuming a left to right traversal, visits the nodes by the order of the identifiers shown in the image above, in other words, the first node that would be visited would be 0, followed by 1, then 2, and so on, until it would reach 14. Of course the algorithm should stop early if the desired value was found. A recursive depth-first search implementation in OCaml can be:

OCaml

```
let rec dfs v = function
  | E -> false
  | N (l, x, r) -> x = v || dfs v l || dfs v r
```

Using OCaml's shorthand for pattern matching, *i.e.* the `function` keyword it is possible to omit the parameter that contains the tree instance. In this version of the algorithm we simply wish to calculate if a value can be found in the tree, as such, if it is empty, then we simply evaluate to false. When traversing a node, then we must first compare the element in the node, if it has the same value as what we are looking for, then the `or` operators (——) are ignored. When the values differ, we call the `dfs` function with the left subtree first, which will fully evaluate (recursively) first before calling the right subtree, this will lead to the desired behaviour.

Since our objective with this algorithm is to find if an element belongs to the tree, then the specification should reflect exactly that. In order to avoid using the same algorithm as a logical function (trivializes the proof and may lead to subtle errors), one possible strategy would be to transform the tree into a list and use the `mem` operation. There are numerous ways to traverse a tree, one possibility is the prefix order, which is exactly the order we have described before: look at the current node, go left first, and explore the right side after:

GOSPEL

```
(*@ function rec prefix (t: 'a tree) : 'a list =
    match t with
    | E -> []
    | N l x r -> x::(prefix l) @ (prefix r) *)
(*@ variant t *)
```

This function can then be applied in the `dfs` function:

GOSPEL + OCaml

```
let rec dfs v = (* ... *)
(*@ r = dfs v t
    variant t
    ensures r <-> List.mem v (prefix t) *)
```

In both functions, `prefix` and `dfs`, we may use the structure of the tree to prove termination, similar to lists. The `ensures` clause is quite strong due to the use of the equivalent symbol, nonetheless, it is essential to guarantee that there are no false positives or false negatives, in other words, a value that does not belong to the tree but is marked as such or a value that is marked as negative despite belonging to the tree, respectively.

3.5 Exercises

Implement and specify the:

1. Backwards linear search on arrays with a while loop
2. Recursive binary search on arrays

Chapter 4

Sorting

Similarly to searching, one problem that may arise when dealing with data is how to sort it. The first step in this process is to find a suitable total order relation for the desired data type, in other words, any two elements must be comparable.

4.1 Integer Lists

The mathematical inequality operators ($<, \leq, >, \geq$), with the exception of the not equal operator ($\neq$), are total order relations for real numbers, and its subsets, including the integer numbers, since any two numbers (even if they are the same) can be compared. In this section, we will only consider integer numbers since Cameleer currently does not support the float data type. Moreover, for simplicity, duplicates are allowed, and the data will be sorted ascendingly, this means that we will use the $\leq$ operator in the following implementations.

4.1.1 Small verification library

Until now, we have dealt with either independent problems each with its own verification strategies (in chapter 2) or a problem class that is not logically intensive (in chapter 3). That is not the case with sorting problems. First, we need to define our concept of sorted sequence of elements, which is mathematically described as:

Let s be a sequence of n elements such that $s = s_0, ..., s_{n-1}$

Then s is sorted if $\forall i, j \in \{0, ..., n-1\} : i \leq j \rightarrow s_i \leq s_j$

This definition is the basis for the predicate used in the Binary Search algorithm (see section 3.2). However, due to the explicit indexing used, this definition is not as natural for OCaml lists, instead we may traverse the list and compare two adjacent elements at a time:

GOSPEL
```
(*@ predicate rec sorted (l: int list) =
    match l with
    | [] | _::[] -> true
    | x::(y::ls) -> x <= y && sorted (y::ls) *)
(*@ variant l *)
```

Since we compare two elements at a time, both empty lists and lists with a single element are considered sorted by default. Otherwise, we must compare the first values with our total order relation. For a list to be considered sorted this must be true for every single pair of adjacent elements, that is why we must use the logical and operator (&&) and in the recursive call we still "propagate" y.

There are many strategies when it comes to sorting data. However, any (serious) sorting algorithm should only (potentially) reorder the original elements, this means that it is prohibited to insert, remove or change any element by the end. Although this may not necessarily be true in the middle of execution, more so when dealing with arrays and memory. This discussion leads to the concept of permutation, two lists are permutations of one another if they have exactly the same elements independently of their order. One possible way to define this concept is through the number of occurrences for each element, which must be equal in both lists. Assuming the existence of a logical function `occ`, we may model a permutation as the following predicate:

GOSPEL
```
(*@ predicate permut (l1 l2: int list) =
    forall x. occ x l1 = occ x l2 *)
```

Using the previously discussed GOSPEL tags, the `occ` function may be defined as:

GOSPEL + OCaml
```
let[@logic][@ghost] rec occ v = function
  | [] -> 0
  | h::t -> (if h = v then 1 else 0) + occ v t
(*@ res = occ v l
    variant l
    ensures 0 <= res <= List.length l
    ensures res > 0 <-> List.mem v l *)
```

Simply traversing the list and comparing its elements suffices for a logical function. One good practice is to equip the specification of a logical function with important properties, as a means to avoid having to define more axioms and lemmas (which have to be proved). In this case, we state that the number of occurrences is non-negative, and may only be as high as the number of elements in the list, since it is nonsensical to have a negative number of occurrences or have more occurrences of a single element than the number of elements in the list itself. Furthermore, a positive number of occurrences must correspond to belonging in the list, and vice versa.

These predicates are our basic verification blocks, however, by themselves, are quite fragile and SMT solvers will have a hard time verifying that these properties are maintained after an iteration. As such, we have to define axioms and lemmas that will help up. However, the first question that comes to mind is where do we begin? Well, the truth is that it is hard to predict what will be needed, a more natural workflow would be to start by annotating the algorithm, finding the needed predicates, and then analysing, within the Why3 platform, the failed proof goals in order to reach the missing properties. Since most of these lemmas are required in the various sorting algorithm we will present next, for presentation purposes we are not able to show this process. Instead, we will present these lemmas as a verification library. Most of these lemmas stem from a similar reason, this being basic list operations, namely the `cons` operator (::) and concatenation (@). Let us say we have two sorted lists, is their concatenation also sorted? Not necessarily, for instance:

$$\text{Let } a = [0, 1, 2],\ b = [1, 2, 3], \text{ and } c = [2, 3, 4]$$
$$\text{Then, } a \mathbin{@} b = [0, 1, 2, 1, 2, 3], \text{ and } a \mathbin{@} c = [0, 1, 2, 2, 3, 4]$$

The concatenation operation creates a new list where the elements of the left operand are followed by those of the right operand, so for the concatenation to remain sorted then it means that each element of the left operand must be lesser or equal to every element of the right operand:

GOSPEL + OCaml
```
let[@lemma] s_concat (l1: int list) (l2: int list) =
  (* TODO *)
(*@ s_concat l1 l2
    requires sorted l1
    requires sorted l2
    requires forall x y. List.mem x l1 -> List.mem y l2 -> x <= y
    ensures sorted (l1 @ l2) *)
```

So, how do we prove this lemma? An astute solution would be to obtain the last element of `l1`, by reversing it, and comparing it to the first element of `l2`, however, SMT solvers have a hard time checking this solution, and may require extra assertions, that may complicate the final proof. Instead, we may opt for comparing each element of $l1$ to every element of $l2$:

GOSPEL + OCaml
```
let[@lemma] rec s_concat l1 l2 =
  match l1 with
  | [] -> ()
  | h1::t1 ->
    match l2 with
    | [] -> ()
    | h2::t2 -> assert (h1 <= h2); s_concat l1 t2; s_concat t1 l2
(*@ s_concat l1 l2
    requires sorted l1
    requires sorted l2
    requires forall x y. List.mem x l1 -> List.mem y l2 -> x <= y
    variant l1, l2
    ensures sorted (l1 @ l2) *)
```

Similarly, the `cons` operation does not always guarantee the result is sorted if the list is sorted, instead it only happens if the element to be added is lesser or equal than every element on the list:

GOSPEL
```
(*@ lemma sorted_cons :
    forall x:int, l: int list.
    (sorted l /\
    (forall e. List.mem e l -> x <= e)) <->
    sorted (x::l) *)
```

Unlike the previous lemma, however, we opted from a stronger statement by using the equivalent sign. The reasoning behind this choice is related to the nature of the two operations. Concatenation is mostly used as a constructive operation, especially in the context of sorting algorithms, i.e. usually we want to build a bigger sorted list from two smaller sorted ones, rather than building two smaller sorted lists from one bigger sorted list. The `cons` operation, may be used both in a constructive manner, to iteratively build a sorted list element by element, or in a destructive way, when using pattern matching. If we separate the equivalent sign into two implications, we obtain:

Let l be an integer list in:

$$(1)\ \forall x \in \mathbb{Z} : \text{sorted}(l) \land (\forall e \in l : x \leq e) \rightarrow \text{sorted}(x :: l)$$
$$(2)\ \forall x \in \mathbb{Z} : \text{sorted}(x :: l) \rightarrow \text{sorted}(l) \land (\forall e \in l : x \leq e)$$

The first logical consequence denotes the constructive use of the `cons` operator, where an element can be inserted at the head of the list if it is smaller or equal to every single element in the list. This proof goal is easily dispatched by SMT solvers. By contrast, the second logical consequence models its destructive use. If a list with at least one element is sorted, then the element at its head is smaller or equal to every element in the tail, which remains sorted. This proof goal is not as trivial, and we even need to define an additional lemma (which must be placed before this one):

GOSPEL + OCaml
```
let[@lemma] rec sorted_head (h: int) (t: int list) =
  match t with
  | [] -> ()
  | x::tt -> assert (h <= x); sorted_head h tt
(*@ requires sorted (h::t)
    variant t
    ensures forall e. List.mem e t -> h <= e *)
```

By having `sorted(h::t)` as a requirement can recursively compare the elements in the tail to the head, by assert that the head is effectively lesser or equal to each element, which translates logically to the last ensures clause (`forall e. List.mem e t -> h <= e`).

Besides these lemmas related to the `sorted` predicate, the `permut` predicate also requires some logical support. One thing that may seem trivial to us, but is quite complex for SMT solvers to prove is that two permutations have the same length. The problem lies in the universal quantifier from the `occ` function. So, how do we go about proving that if the number of occurrences for every single element is the same between two lists then their length is the same? When a property looks trivial, but the provers cannot solve it, then it generally means that it is impossible to prove (it is an axiom), or it has a complex proof. In this case it is the latter. Complex proofs can be tedious and lead to several lost hours, for comfort purposes, sometimes it is okay to "cheat", instead of proving the lemma, we may consider it as an axiom. However, this practice should not be taken lightly, as it can quickly lead to possibly faulty programs being proved under false assumptions and inconsistent logic, which will cause more harm than good in real world software. As such, this practice, of transforming hard lemmas into axioms, must be avoided at all costs. This does not have any implications on real axioms, since there is no way to prove them. So, to ensure that a permutation of a list has the same length we define:

GOSPEL
```
(*@ axiom permut_len : forall l1 l2: int list.
    permut l1 l2 -> List.length l1 = List.length l2 *)
```

Finally, we also have to define the behaviour on the number of occurrences when concatenating two lists. This amounts to summing the elements of the two lists, since no element is changed, removed or inserted:

GOSPEL + OCaml
```
let[@lemma] occ_append (x: int) (l1: int list)
(l2: int list) =
  (* TODO *)
(*@ occ_append x l1 l2
    ensures occ x (l1 @ l2) = occ x l1 + occ x l2 *)
```

For simplicity, we only consider one element at a time, in order to omit the universal quantifier, and streamline the post-condition. To prove this lemma, we do a proof by induction:

GOSPEL + OCaml
```
let[@lemma] rec occ_append (x: int) (l1: int list)
(l2: int list) =
  let p = occ x (l1 @ l2) = occ x l1 + occ x l2 in
  match l1 with
  | [] -> assert (p)
  | _::t -> occ_append x t l2; assert (p)
(*@ occ_append x l1 l2
    variant l1
    ensures occ x (l1 @ l2) = occ x l1 + occ x l2 *)
```

We define `p`, as the property we want to prove, i.e. the post-condition, and recursively iterate over one of the lists. In the base case, `l1 = []`, which implies that `occ x ([] @ l2) = occ x [] + occ x l2`, and since the empty list does not contain any element, then it is trivial to prove that `occ x l2 = occ x l2`. The inductive step implicitly states that if `occ x (t @ l2) = occ x t + x l2` then `occ x (h::t @ l2) = occ x (h::t) + x l2`, where `l1 =`

`h::t` (note that `h` is omitted in the implementation above). This can be proved by the SMT solvers since each side of the logical equality receives the same new element.

4.1.2 Selection Sort

One possible strategy to sort a sequence of elements is to find the minimum value and place it at the front of the sequence, then find the second smallest element and place it in the second position, and so on…

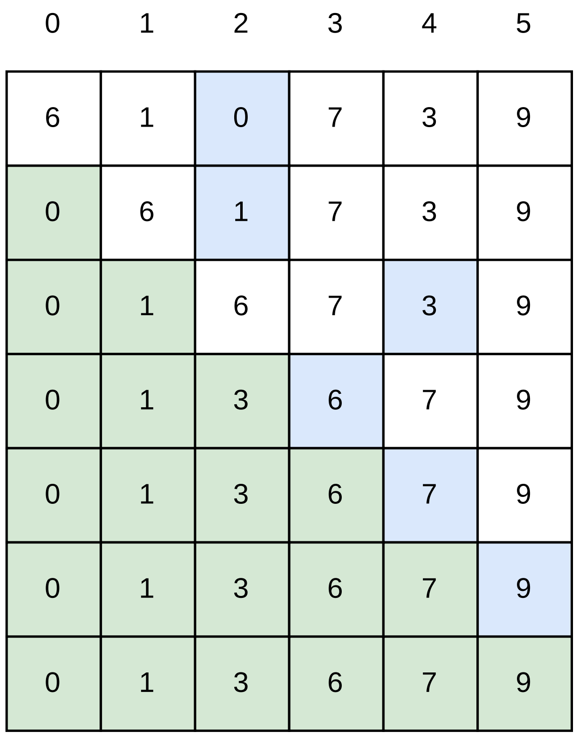


Figure 4.1: Selection sort example

However, shifting the sequence is unpractical and costly. In a functional setting, we can, instead, allow elements to be reorganized without any time complexity penalty. For instance, let us consider the iteration where 3 is the minimum value:

Minimum	Remaining	Processed
6	[7; 3; 9]	[ ]
6	[3; 9]	[7]
3	[9]	[7; 6]
3	[ ]	[7; 6; 9]

In this case 6 and 7 are swapped at the end of the iteration, this is because every element is placed at the end of processed list. At first one solution may seem to add the previous minimum candidate at the front of the list, however, for more complex examples this is interaction does not guarantee the original placement:

Minimum	Remaining	Processed
6	[8; 10; 12; 4; 5; 7; 3; 9]	[ ]
6	[10; 12; 4; 5; 7; 3; 9]	[8]
6	[12; 4; 5; 7; 3; 9]	[8; 10]
6	[4; 5; 7; 3; 9]	[8; 10; 12]
4	[5; 7; 3; 9]	[6; 8; 10; 12;]
4	[7; 3; 9]	[6; 8; 10; 12; 5;]
4	[3; 9]	[6; 8; 10; 12; 5; 7]
3	[9]	[4; 6; 8; 10; 12; 5; 7]
3	[ ]	[4; 6; 8; 10; 12; 5; 7; 9]

So, to simplify, it is best to place each processed element at the end of the list, which can be achieved using the `cons` operator (::), as we may see:

OCaml
```ocaml
let rec selection_aux min = function
  | [] -> (min, [])
  | x::r ->
    if min <= x then
      let (m, l) = selection_aux min r in m, x::l
    else
      let (m, l) = selection_aux x r in m, min::l
```

The `selection_aux` function has two parameters, the first is the minimum candidate, and the second is the list without that same element. By the end of the recursion, a pair with the minimum value and empty list is returned. When the list has at least one element we must compare its head to the current minimum candidate. If the minimum candidate is smaller or equal to the head element, then the minimum candidate for the next recursive call remains unchanged, and the head element is placed at the head of the resulting list. Otherwise, the minimum candidate is changed to the head element, and the previous minimum candidate is placed at the head of the resulting list.

The main function repeats this process if there are two or more elements in the received list or one of its sub-lists:

OCaml
```
let rec selection_sort l =
  match l with
  | [] -> []
  | [x] -> [x]
  | x::ls ->
    let m, r = selection_aux x ls in
    m::(selection_sort r)
```

By calling the `selection_aux` function we obtain the minimum value, `m`, and the sub-list with the remaining elements of `x::ls`. The result is obtained from inserting `m` of the list produced by the recursive call of `selection_sort` with `r` as its parameter. This corresponds to finding every element from lowest to highest and reconstructing it backwards to ensure that it is sorted in that order, since we are inserting at the head.

Verification-wise, the major logic is placed in the auxiliary function, which leads to a relatively simple annotation for the main function (similar to most sorting algorithms):

GOSPEL + OCaml
```
let rec selection_sort l = (* ... *)
(*@ r = selection_sort l
    ensures sorted r
    ensures permut r l
    variant List.length l *)
```

As expected, the result should be sorted and a permutation of the original list. Moreover, since this function is applied to every single element in the list, to prove termination, we may simply use the length of the list.

GOSPEL + OCaml
```
let rec selection_aux min = (* ... *)
(*@ m, r = selection_aux min l
    variant l
    ensures m <= min
    ensures forall x. List.mem x r -> m <= x
    ensures permut (m::r) (min::l) *)
```

Similar to the main function, this one can be proven to be terminal using the list itself, since the function basically amounts to traversing the list from one end to the other (despite doing more than just that). In terms of post conditions, it is necessary to ensure that `m`, the resulting minimum value, can be lesser or equal to the minimum candidate from the parameters (`min`). Furthermore, `m` should not be greater that any value on the resulting list, to guarantee that it is indeed the minimum value, or in other words, it is lesser or equal to every element in `r`. The final ensures clause serves to maintain the property that at every iteration we produce a permutation of the previous one, therefore, by the end, the result will be a permutation of the original list.

Tail recursion

Does tail recursion have an impact on a program proof? The answer is likely yes. In many scenarios an accumulator variable is introduced and along with it some logical properties may change slightly, since part of the original input is now placed on the accumulator, and new requirements are usually introduced to reflect the properties of the data stored in the accumulator.

To update the auxiliary function, `selection_aux`, to be tail recursive, we may do this:

```
let rec selection_aux min acc = function                          OCaml
  | [] -> (min, acc)
  | x::r ->
    if min <= x then
      selection_aux min (x::acc) r
    else
      selection_aux x (min::acc) r
```

In its original version, to build the pair of results when the list had at least one element (besides the minimum candidate) we had to first obtain the previous pair from the recursive call, which help build the new results on top of the previous ones. In the tail recursive version, the same behaviour is achieved by using an accumulator. When the current candidate is lesser or equal to the head of the list, assuming it has at least one element, the minimum candidate remains the same while the head element is placed at the head of the accumulator. On the other hand, if we have found a better minimum candidate, the first parameter is updated, and the old candidate is now placed at the head of the accumulator. By the end, the accumulator will contain the list of elements that are greater or equal to the minimum candidate.

```
let rec selection_aux min acc = (* ... *)                 GOSPEL + OCaml
(*@ m, r = selection_aux min acc l
    requires forall x. List.mem x acc -> min <= x
    variant l
    ensures m <= min
    ensures forall x. List.mem x r -> m <= x
    ensures permut (m::r) (min::(l @ acc)) *)
```

The most significant change is the addition of a `requires` clause. Not just that, it is quite similar to one of pre-existing post conditions. This is due to the underlying relation between `acc` and `r`, since whatever is placed in `acc` during the final iteration will become `r`. So, the `acc` variable will contain the processed elements that are guaranteed to not be the minimum value, either due to being greater or equal to the current candidate. A question may arise, why is it a `requires` rather than an `ensures` clause? In a purely functional setting, such as this, data is immutable, so any property that impacts intermediate variables and that holds at the end of the iteration also holds at the start, thereby eliminating the need for post conditions related to intermediate variables, since these are only updated with new function calls. Therefore, it is mandatory for properties on intermediate variables to hold at the start of a given iteration. A small detail that has also change is in the last `ensures` condition, in which the input data is now split into two variables `l`, which contains the unprocessed data, and `acc`, which contains the processed data.

Moving on to the main function, `selection_sort`:

OCaml
```
let rec selection_sort acc = function
  | [] -> acc
  | [x] -> acc @ [x]
  | x::ls ->
    let m, r = selection_aux x [] ls in
    selection_sort (acc @ [m]) r
```

In this case, the accumulator will hold each consecutive minimum value, and each new one will be placed at the end of the list to guarantee the ascending sorting. The specification of this function also changes:

GOSPEL + OCaml
```
let rec selection_sort acc = (* ... *)
(*@ r = selection_sort acc l
    requires forall x y. List.mem x acc -> List.mem y l -> x <= y
    requires sorted acc
    ensures sorted r
    ensures permut r (l @ acc)
    variant List.length l *)
```

As we have previously mentioned, one of the new requirements is that the accumulator is sorted at the start of every iteration (implicitly at the end too due to immutability). The other pre-condition, however, can be more unnoticeable, but every element in `acc` is guaranteed to be lesser of equal to every element of `l`, since we select the minimum value from `l` and placed it at the end of `acc` in every iteration. This property is very important to ensure that the result, as well as the `acc` at every iteration, are sorted. This specification is not enough however, SMT solvers have a hard time recognizing that `m` belongs to `x::ls`, since we separate a minimum candidate (`x`) right from the start, so `m` does not necessarily belong to `ls` from the perspective of the auxiliary function, so we need to add an assertion to do just that:

GOSPEL + OCaml
```
let m, r = selection_aux x [] ls in
assert (List.mem m (x::ls));
selection_sort (acc @ [m]) r
```

4.1.3 Merge Sort

Another approach to problem-solving is to repeatedly divide a larger problem into smaller subproblems until each becomes simple enough to be solved directly, these can then be combined, in a way that maintains the validity of each intermediate result, to form the solution to the original problem. This is known as the divide and conquer strategy, which is the approach used in our current case study: merge sort. This algorithm is divided into two phases, split and merge. In the split phase, the recursively divided in half, for instance:

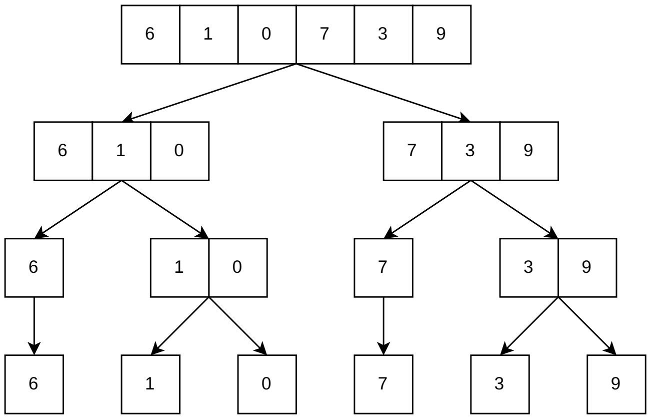


Figure 4.2: Merge sort: split phase example

At the end of the split phase, each sub-list will be sorted, since it only contains a single element. In the final stage, the sub-lists will be merged in a way that the intermediate results are also sorted, thereby guaranteeing that in the final iteration the original list will be sorted:

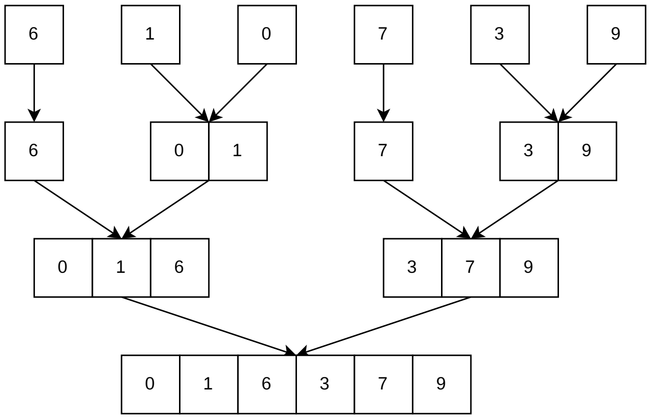


Figure 4.3: Merge sort: merge phase example

In a functional setting, such as the one in this case study, it is quite inconvenient to split a list exactly at its half point, since it would be necessary to know both the current index and the length of the original list. This translates into two unnecessary parameters. Note that the order of the elements before and after the split does not impact the performance of the merge sort algorithm, as long as the splitting strategy does not exceed linear time complexity. A more elegant splitting strategy is to send one element to the first sub-list and the next to the other sub-list, and repeating. This ensures that both lists have at most a difference of one element, in case the original list has an odd number of elements. This can be programmed as:

```ocaml
let rec split (l: int list) =
  match l with
  | [] -> ([], [])
  | x::ls -> let ll, rl = split ls in (rl, x::ll)
```

The result of `split` function is a pair of lists, let us call the first element of the pair `ll` (left list) and the second element `rl` (right list). The quirk of this function is that at the end of every iteration the sub-lists are switching positions inside the pair, which allows fixating one of the members of the pair to receive the new element. This pattern leads exactly to the behaviour we desired, since we are not placing the new element twice in succession in the same list.

The merge phase, from an implementation perspective, is not as tricky:

```ocaml
let rec merge l1 l2 =
  match l1, l2 with
  | z, [] | [], z -> z
  | x::ls, y::rs ->
    if x <= y then x::(merge ls (y::rs))
    else y::(merge (x::ls) rs)
```

As one might suspect from the previous figure, the `merge` function receives two (sorted) lists as parameters. Whenever one of these lists is empty, the result is simply the other list, since it has been sorted beforehand. If both lists have at least one element, then we must first compare their heads to finds which is the smallest, which is then appended to the head of the result. The recursive call also includes the other element, as not to miss any. This behaviour allows creating a sorted list from two smaller sorted lists via insertion.

Finally, the main function orchestrates the process between the phases:

```ocaml
let rec merge_sort l =
  match l with
  | [] -> []
  | [x] -> [x]
  | _ -> let (a, b) = split l in
    merge (merge_sort a) (merge_sort b)
```

If we have at least two elements, we split the list and then recursively call the `merge_sort` function on each of these sub-lists to possibly be split further, and then the `merge` function combines the various intermediate results.

From a verification standpoint, the annotations on main function of a sorting algorithm might not change significantly in relation to other algorithm, and this is the case:

GOSPEL + OCaml

```
let rec merge_sort l = (* ... *)
(*@ r = merge_sort l
    ensures sorted r
    ensures permut r l
    variant List.length l *)
```

This is due to the main properties of each of the solutions being equal, each algorithm should produce a sorted list that is a permutation of the original list. Moreover, in functional programming these algorithms traverse the original list in the main function, which leads to the same termination proof, which is exactly the original list itself (or its length, which are equivalent).

To verify the `split` function, we may do it in such manner:

GOSPEL + OCaml

```
let rec split (l: int list) = (* ... *)
(*@ (r1, r2) = split l
    ensures permut l (r1 @ r2)
    ensures List.length r2 = List.length r1 \/
      List.length r2 = List.length r1 + 1
    ensures List.length r2 + List.length r1 = List.length l
    variant List.length l *)
```

Since we are splitting the original list, `l`, in two halves, we are effectively creating a permutation of `l`, meaning that no elements are inserted, removed or have its value altered, through the concatenation of the two sub-lists. Moreover, the sum of lengths of the sub-lists should be equal to the length of the original list, and the sub-lists cannot exceed a difference of an element in their size, otherwise it would not be a split in half.

Finally, the `merge` function:

GOSPEL + OCaml

```
let rec merge l1 l2 = (* ... *)
(*@ r = merge l1 l2
    requires sorted l1
    requires sorted l2
    ensures sorted r
    ensures permut r (l1 @ l2)
    variant List.length l1 + List.length l2 *)
```

As previously mentioned, the `merge` function receives two sorted lists and produces another sorted list, which permutation of both. To prove termination we use the length of both input lists, since we are iterating over both, and only removing one element of one of the lists at a time, meaning that at most we will iterate the same number of times of the sum of their lengths.

4.1.4 Quick Sort

Merge sort is not the only divide and conquer sorting algorithm, Quick sort applies the same principle with a different approach when breaking down the problem. The first step in this algorithm is to choose a pivot element, which will serve as a reference value to split apart the sequence. The remaining elements that are lesser or equal to the pivot will be placed in a separate sub-list from those that are strictly greater than the pivot. There are several methods to choose the pivot, with the simplest being selecting either the first or last element. Since we are dealing with linked lists, the only feasible method, without altering the time complexity, is to select the head of the list as pivot.

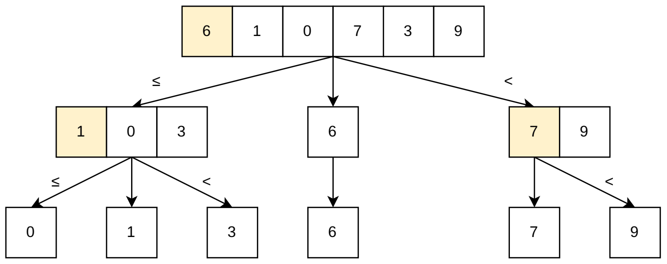


Figure 4.4: Quick sort example: breaking down the list

The pivots of each list before breaking down are highlighted in yellow. One other detail from the figure is that by the last iteration, once all elements are isolated, they are pretty much sorted. Note that this is just a visual representation, but the fact is that reconstructing sorted intermediate lists amounts to concatenating the sub-list with the smaller/equal values with the pivot and the sub-list of higher values.

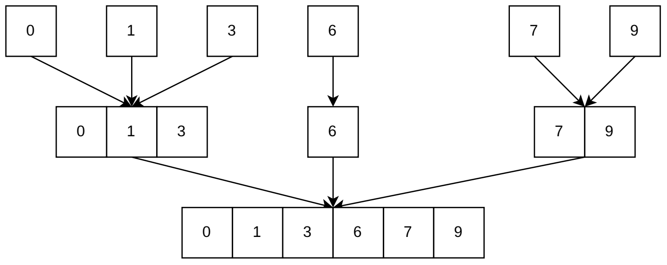


Figure 4.5: Quick sort example: breaking down the list

To define the behaviour of splitting the list, we will create an auxiliary function:

```
let rec quick_aux p = function                                        OCaml
  | [] -> ([], [])
  | x :: r ->
    let ll, lr = quick_aux p r in
    if x <= p then x :: ll, lr
    else ll, x :: lr
```

The main objective of this function is that given a pivot, `p`, and the remaining elements, we want to separate those that are lesser or equal to `p` from the ones that are higher, as such we will return a pair of lists. Assuming that the lesser or equal elements are placed in the first member of the pair, we start by performing the recursive call on the tail of the list, `r`, and we will obtain the pair (`ll`, `lr`), then insert at the head in the corresponding list, based on the comparison with the pivot. The main function, on the other hand, will contain the combination process:

```
let rec quick_sort l =                                                OCaml
  match l with
  | [] -> []
  | [x] -> [x]
  | p :: ls ->
    let (left, right) = (quick_aux p ls) in
    (quick_sort left) @  p :: quick_sort right
```

It is only sensible to perform a split if we have more than one element, since the first will be the pivot. After obtaining the `left` and `right` sub-lists, to combine the sub-lists it simply amounts to appending `left` to `p` and `right`, since both `left` and `right` were sorted beforehand and contain values based on the pivot, due to splitting. As per usual, the annotations on the main function, *i.e.* `quick_sort`, are as follows:

```
let rec quick_sort l = (* ... *)                                      OCaml
(*@ r = quick_sort l
    ensures sorted r
    ensures permut r l
    variant List.length l *)
```

The `quick_aux` function on the other hand:

```
let rec quick_aux p = (* ... *)                                       OCaml
(*@ ll, lr = quick_aux p l
    variant l
    ensures forall x. List.mem x ll -> x <= p
    ensures forall x. List.mem x lr -> x > p
    ensures permut l (ll @ lr) *)
```

Starting with the last `ensures` clause, since we are performing a split operation, we must guarantee that the result have the same elements as the original spread across the two sub-lists. Moreover, each list should respect the aforementioned properties, these being that every element of `ll` should be lesser or equal when compared to the pivot, while `lr` contains the values strictly greater than `p`.

Optimized tail recursion

One of the goals of tail recursion is to prevent stack overflows, however, when multiple recursive calls have to be made this is not guaranteed, and the calling order matters. Using quick sort as an example, if in the first call the sub-list contains more elements than the other sub-list, this means that we are making an inefficient use of stack space, since there will be a bigger number of consecutive function calls in memory, and by the time we do the second call, the tail call, the program might have stopped abruptly due to stack overflow or even if it doesn't the benefits of doing a tail call have diminished greatly. If we notice closely, the length of the smallest sub-list ranges from 0 to half of the number of elements in the original list (both inclusive), while the largest sub-list ranges from half of the number of elements and the total number of elements in the original list (both inclusive), this means that if we consistently call the largest sub-list first we will have expected linear stack space, however, if we call the smallest sub-list first we will have expected logarithmic stack space.

So, let's update out quick sort implementation to be tail recursive and optimized. Starting with the auxiliary function:

OCaml
```
let rec quick_aux p left right = function
  | [] -> (left, right)
  | x::ls ->
    if x <= p then quick_aux p (x::left) right ls
    else quick_aux p left (x::right) ls
```

For `quick_aux` to be tail recursive we may use two additional parameters to accumulate the results, since the result is a pair of lists with distinct meanings, therefore `left` will store the values that are lesser or equal than the pivot, and `right` will store those that are strictly greater than `p`. The real challenge stems from the main function:

OCaml
```
let rec quick_sort lacc racc = function
  | [] -> lacc @ racc
  | [x] -> lacc @ x::racc
  | p::ls ->
    let (left, right) = quick_aux p [] [] ls in
    if List.length left <= List.length right then begin
      let sorted_left = quick_sort lacc [] left in
      quick_sort (sorted_left @ [p]) racc right
    end else begin
      let sorted_right = quick_sort [] racc right in
      quick_sort lacc (p::sorted_right) left
    end
```

In a non-optimized tail call version of quick sort, we might have defined a precedence over one of the sub-lists, which would result in only one accumulator. However, that is not the case, since we are potentially alternating between going left or right first, then it means we are sorting the list on two fronts at the same time, and therefore we need two accumulators, one for each front. Moreover, the elements on `lacc` (left accumulator) must be lesser or equal than every element in the remainder of the list (omitted by the `function`) keyword, while the elements on `racc` (right accumulator) are strictly greater than those in the remainder of the list, and, consequently, greater than the elements on `lacc`. This means that if the remaining list to be process is empty, then we can simply concatenate `lacc` with `racc`. If the remaining list only contains one item, then it will be placed exactly in the middle of the two accumulators. If the remainder of the list has two or more elements we shall split its tail according to the pivot element, its head, and the `quick_aux` must be initialized with its accumulators as empty lists to avoid errors. The next step is to compare the lengths of the split sub-lists. If `left` is smaller or equal to `right`, then we must first perform the recursive call on `left`. This function call is initialized with `racc` as empty, otherwise it would be the same as placing the highest value elements in the original list in the middle of the list, which would be incorrect, in these intermediate calls we may only place the corresponding extreme of the list. Once all elements on the left side have been sorted, then we can proceed to sort the right side, and can be done so by another recursive call, now this time with the `racc` obtained previously and `sorted_left` sub-list with the pivot appended at the end since `p` is guaranteed to be greater or equal than every element of `left`, and consequently `sorted_left`, but also strictly smaller than every element in the `right` list, and consequently `racc` accumulator. Similarly, the first recursive call will have an empty `lacc`, since it would lead to incorrect values placed in the middle of the resulting list. Moreover, in the second call, since `p` is greater or equal than every element in the `left` list, and, consequently, the `lacc` accumulator, then it can be inserted at the head of the `sorted_right` (the new `racc`), since it is also strictly smaller than every element in right.

Verification-wise, the `quick_aux` function needs a strong set of post conditions to help verify the complex main function presented beforehand:

```
let rec quick_aux p left right = (* ... *)                    GOSPEL + OCaml
(*@ ll, lr = quick_aux p left right l
    requires forall x. List.mem x left -> x <= p
    requires forall x. List.mem x right -> x > p
    ensures forall x. List.mem x ll -> x <= p
    ensures forall x. List.mem x lr -> x > p
    ensures permut (l @ left @ right) (ll @ lr)
    variant l *)
```

As expected, we must control what is placed on the accumulators, `left` and `right`, which is adapted from what they represent by the end of the computations, `ll` and `lr`, respectively. The first four clauses, two `requires` (for the accumulators) and two `ensures` (for the final results) conditions, represent exactly this, since one of the sub-lists must contain the elements that are lesser or equal to the pivot, and the other must contain the elements that are strictly greater than `p`. The remaining condition from the original specification is concerned with the permutation of the results from the results, which must be adapted, since the input is now split across three parameters rather than one. The two accumulators and the remainder of the list to be process when concatenated must contain all the elements present in both the resulting sub-lists, meaning that we can not introduce, remove or change any element.

Moving on to the main function:

GOSPEL + OCaml

```
let rec quick_sort lacc racc = (* ... *)
(*@ r = quick_sort lacc racc l
    requires forall x y.
      List.mem x racc -> List.mem y l -> x >= y
    requires forall x y.
      List.mem x l -> List.mem y lacc -> x >= y
    requires forall x y.
      List.mem x racc -> List.mem y lacc -> x >= y
    requires sorted lacc
    requires sorted racc
    ensures sorted r
    ensures permut r (lacc @ l @ racc)
    variant List.length l *)
```

Once more, since the input is now spread over three different parameters, these being `lacc`, `racc`, and the omitted list to be processed, we must update the permutation condition from before. In terms of new conditions, we need to add a staggering total of five `requires` clauses. The last two pre-conditions state that the accumulators must be sorted, otherwise it would not be possible to prove that at the end of the computations the result would be sorted. The first three pre-conditions serve to establish the relative order between each of the input lists, in the sense that any element in `lacc` is lesser or equal to any element in `l` (omitted parameter), which in turn are lesser or equal to the elements in `racc`. However, due to the complexity of this function and algorithm, these specifications by themselves are not enough. In fact, proving that the result is a permutation of the inputs combined is a quite troublesome affair for the SMT solvers. This can be solved with two simple lemmas (which are automatically proved):

GOSPEL

```
(*@ lemma permut_append_mem: forall l1 l2 l3: int list.
    permut l1 (l2 @ l3) ->
      (forall x. List.mem x l2 -> List.mem x l1) &&
      (forall x. List.mem x l3 -> List.mem x l1) *)

(*@ lemma permut_elems: forall l1 l2 l3: int list.
    permut l1 (l2 @ l3) ->
      (forall x. List.mem x l1 <-> List.mem x (l2 @ l3)) *)
```

The first auxiliary lemma, `permut_append_mem` states that if any list `l1` is a permutation of a concatenation of two other lists, `l2` and `l3`, then every element that belongs in either of those lists also belongs in `l1`. The second lemma, `permut_elems`, goes a bit further, it states that if `l1` is a permutation of `l2 @ l3`, then every element that belongs in `l1` also belongs in `l2 @ l3`.

Even after all this trouble the proof is not complete, however we are very close. Only a few assertions are needed to terminate the proof:

GOSPEL + OCaml
```
(* ... *)
  | [x] -> assert (List.mem x l); lacc @ x::racc
(* ... *)
```

This assertion is essential to ensure that `x` is smaller than the elements in `racc` and greater or equal to the elements of `lacc`, thereby guaranteeing that the result is sorted.

GOSPEL + OCaml
```
(* ... *)
  | p::ls ->
    assert (List.mem p (p::ls));
    let (left, right) = quick_aux p [] [] ls in
    if List.length left <= List.length right then begin
      assert (permut l (left @ [p] @ right));
      let sorted_left = quick_sort lacc [] left in
      quick_sort (sorted_left @ [p]) racc right
    end else begin
      let sorted_right = quick_sort [] racc right in
      quick_sort lacc (p::sorted_right) left
    end
```

The first assertion might seem obvious (and it is in fact), however, in the context of `quick_aux` that is not the case, since `p` and `ls` are separate parameters. Within the auxiliary function the solvers know that `left @ right` is a permutation of `ls`, however `p` is not a part of it, that is that that condition is important. The case where we call the right side first is unexpectedly well-behaved, however, that is not when the left side is called first, and we have to assert that `l` is a permutation of `left @ [p] @ right`.

4.2 Polymorphic Lists

As we have previously discussed, to sort a sequence of elements with a given data type, then that data type must be equipped with a total order relation. In OCaml this can be achieved by using its module system:

OCaml
```
module type Cmp = sig
  type t
  val eq: t -> t -> bool
  val leq: t -> t -> bool
end
```

`Cmp` is a signature, which is a collection of definitions that must be present in any module that implements it. In this signature, we define our generic type, `t`, and provide two functions, one to test equality, `eq`, between two values of type `t`, and the other, `leq`, to test if the first value is considered lesser or equal than the second. By using a signature we are abstracting from concrete implementations, so that we can achieve a polymorphic type.

Signatures can also be annotated with GOSPEL, in fact this was one of its original uses, before being used for deductive verification with Cameleer. In particular, we need to ensure some basic properties regarding total order:

GOSPEL + OCaml

```
module type Cmp = sig
  type t

  val eq: t -> t -> bool [@@logic]
  (*@ b = eq x y
      ensures b <-> x = y *)

  (*@ function le: t -> t -> bool *)

  (*@ axiom reflexive : forall x. le x x *)
  (*@ axiom total : forall x y. le x y \/ le y x *)
  (*@ axiom transitive: forall x y z. le x y -> le y z -> le x z *)

  val leq: t -> t -> bool [@@logic]
  (*@ b = leq x y
      ensures b <-> le x y *)
end
```

When used, the `[@@logic]` tag enables OCaml definitions (in signatures) to be visible in subsequent annotations (not including in its own definition). The `eq` definition is used to avoid typing errors derived from WhyML, in which the equality operator (`=`) expected integer operands. The result of any implementation of `eq` must be equivalent to the logical equality between the same two elements. Similarly, the results of a concrete `leq` function must be equivalent to the logical `le` function. In GOSPEL it is also possible to provided logical definitions, such as `le`, that must be "implemented" logically. Furthermore, in this context we define the necessary axioms for the `le` definition to be considered a (non-strict) total order relation, these being: reflexivity ($x \leq x$ is true), totality (Let S be some set, then $\forall x, y \in S : x \leq y \lor y \leq x$), and transitivity (if $x \leq y$ and $y \leq z$, then $x \leq z$). Beware that any logical implementation of `le` must be carefully checked beforehand, since the axioms on top are considered true regardless of its correctness, and if it happens to be incorrect, then the proof will be flawed.

4.2.1 Revisiting the verification library

To make use of that signature, we now may make use of a functor, i.e. a module that is parameterized by other modules (or signatures):

OCaml
```
module SomeSort (E: Cmp) = struct
  type elt = E.t

  (* Omitted *)
end
```

We can use this formula for any sorting algorithm in order to allow polymorphism. The updated verification library should be inside this module for access to the `eq` and `leq` functions. So, starting with the function to calculate the number of occurrences:

GOSPEL + OCaml
```
let[@logic][@ghost] rec occ v = function
  | [] -> 0
  | h::t -> (if E.eq h v then 1 else 0) + occ v t
(*@ res = occ v l
    variant l
    ensures 0 <= res <= List.length l
    ensures res > 0 <-> List.mem v l *)
```

The only significant change is to update the if-statement condition by replacing the equality check, `h = v`, with `E.eq` function to `E.q h v`. The `permut` predicate remains unchanged, while the `sorted` predicate updated the `<=` operator to the `E.leq` function:

GOSPEL
```
(*@ predicate permut (l1 l2: elt list) =
    forall x. occ x l1 = occ x l2 *)

(*@ predicate rec sorted (l: elt list) =
    match l with
    | [] | _::[] -> true
    | x::(y::ls) -> E.leq x y && sorted (y::ls) *)
(*@ variant l *)
```

Lemma-wise, `occ_append` does not see any changes, while the remainder have the `<=` operator changed to the `E.leq` function:

GOSPEL + OCaml
```
let[@lemma] rec occ_append (x: elt) (l1: elt list) (l2: elt list) =
  let p = occ x (l1 @ l2) = occ x l1 + occ x l2 in
  match l1 with
  | [] -> assert (p)
  | _::t -> occ_append x t l2; assert (p)
(*@ occ_append x l1 l2
    variant l1
    ensures occ x (l1 @ l2) = occ x l1 + occ x l2 *)
```

```
let[@lemma] rec sorted_head (h: elt) (t: elt list) =
  match t with
  | [] -> ()
  | x::tt -> assert (E.leq h x); sorted_head h tt
(*@ requires sorted (h::t)
    variant t
    ensures forall e. List.mem e t -> E.leq h e *)

(*@ lemma sorted_cons :
    forall x: elt, l: elt list.
    (sorted l /\
    (forall e. List.mem e l -> E.leq x e)) <->
    sorted (x::l) *)

let[@lemma] rec sorted_append (l1: elt list) (l2: elt list) =
  match l1 with
  | [] -> ()
  | h1::t1 ->
    match l2 with
    | [] -> ()
    | h2::t2 -> assert (E.leq h1 h2);
      sorted_append l1 t2; sorted_append t1 l2
(*@ sorted_append l1 l2
    requires sorted l1
    requires sorted l2
    requires forall x y. List.mem x l1 -> List.mem y l2 -> E.leq x y
    ensures sorted (l1 @ l2) *)
```

4.2.2 Revisiting Selection Sort

Adapting from integer-specific code to a polymorphic implementation does not pose significant challenge, since it amounts to changing from concrete operators, such as $\leq$ or $=$, to the operations provided in the `Cmp` signature. In the case of the selection sort algorithm, there are no changes to the main function, since it does not use any of the aforementioned operators. Consequently, the specification of the `selection_sort` function also remains unchanged. By contrast, the auxiliary function has one slight change:

OCaml

```
let rec selection_aux min = function
  | [] -> (min, [])
  | x::r ->
    if E.leq min x then
      let (m, l) = selection_aux min r in m, x::l
    else
      let (m, l) = selection_aux x r in m, min::l
```

This change is in the if-statement condition, which is now `E.leq min x`, to convey that `min` is lesser or equal to `x`, independently of the criteria used to determine when a value is effectively lesser or equal than other for a given concrete data type. Similarly, the annotations also made of use of concrete operators, which need to be replaced:

GOSPEL + OCaml
```
let rec selection_aux min = (* ... *)
(*@ m, r = selection_aux min l
    variant l
    ensures E.leq m min
    ensures forall x. List.mem x r -> E.leq m x
    ensures List.length r = List.length l
    ensures permut (m::r) (min::l) *)
```

There are two slight changes in the specification, in particular in the first and second `ensures` clauses.

4.3 Integer Arrays

4.3.1 Re-revisiting the verification library

Besides syntactic differences in OCaml and GOSPEL, arrays and linked lists are fundamentally different in terms of accessing a given index. One one hand, arrays offer access in constant time complexity. While elements in linked lists, on the other hand, are access, on average, in linear time complexity. This performance "penalty" makes constructing an entirely new list more viable, while arrays may be sorted in-place, or at least with duplicate memory. This drastic change in approaches has repercussions in the verification library, since the sub-lists would be separate entities from the original list, and would be entirely sorted. Meanwhile, in the context of arrays, since we are using the same memory throughout the algorithm, only subsets of the array may be sorted at a given time, so this leads to the creation of a sorted interval predicate:

GOSPEL
```
(*@ predicate sorted_sub (a: int array) (l u: int) =
    forall i j: int. l <= i <= j < u -> a.(i) <= a.(j) *)

(*@ predicate sorted (a: int array) =
    sorted_sub a 0 (Array.length a) *)
```

To define a sorted interval within an array we need a lower and an upper bound. The lower bound is inclusive, while the upper bound is exclusive. For every two indexes inside that interval, such that one is smaller or equal to the other, then the corresponding value of the first must also be smaller or equal to the value in the second index. Additionally, to define a sorted array predicate, we may use an interval from 0 to the length of the array, effectively meaning that all indexes are inside that interval.

Besides the concept of sorted array, we must also define the concept of permutation, as a way to ensure that all elements are retained from one iteration to the other, this will be done through the number of occurrences, as as in the previous libraries:

GOSPEL
```
let[@logic][@ghost] occ v a =
  let r = ref 0 in
  for i = 0 to Array.length a - 1 do
  (*@ invariant 0 <= !r <= i
      invariant !r > 0 -> Array.mem v a
      invariant !r = 0 -> (forall k. 0 <= k < i -> a[k] <> v) *)
    if a.(i) = v then r := !r + 1
  done;
  !r
(*@ res = occ v a
    ensures 0 <= res <= Array.length a
    ensures res > 0 <-> Array.mem v a *)

(*@ predicate permut (a1 a2: int array) =
    forall x. occ x a1 = occ x a2 *)
```

To count the number of occurrences in an array, we may use a for loop. This comes with the cost of having to define invariants based on the post conditions we want to express, these being that the number of occurrences is non-negative, an may be at most the length of the array, in which case every element of the array would be `v`, the value we are looking for. The second post condition states that if the result is a positive number then the elements belongs in the array and vice-versa. The first loop invariant is based on the first post condition, at a given point of the iteration `r`, the reference that stores the result, is non-negative and may be at most `i`, which represents the current index at the start of an iteration, or the next index at the of an iteration (that is why there is no plus 1 after `i`). The other two invariants serve to prove the second post-condition. If `r` holds a positive number then the element must belong to the array. Otherwise, if `r`, in a given iteration, holds 0, then it means that up until that point, no such value was found within the interval of array that has been traversed. The `permut` predicate only sees a slightly change, which is the type of its arguments, since we are now using arrays instead of lists.

4.3.2 Swap-based sorting

One technique commonly present in sorting algorithms, for arrays, is to swap two elements at a time, for instance, with the following function:

OCaml
```
let swap (a: int array) i j =
  let t = a.(i) in
  a.(i) <- a.(j);
  a.(j) <- t
```

Since we are dealing with data that can be mutated, it is important to ensure exactly where and how the data was changed. To achieve this, we suggest the creation of a new predicate, called exchange:

GOSPEL
```
(*@ predicate exchange (a1 a2: int array) (i j: int) =
    Array.length a1 = Array.length a2 &&
    0 <= i < Array.length a1 &&
    0 <= j < Array.length a1 &&
    a1.(i) = a2.(j) &&
    a1.(j) = a2.(i) &&
    (forall k. 0 <= k < Array.length a1 && k <> i -> k <> j ->
      a1.(k) = a2.(k)) *)
```

Given two arrays of the same length (ideally the same memory before and after the changes or a copy that has seen some changes) and two indexes, that fit inside the arrays, *i.e.* they are within 0 (inclusive) and the length of the array (exclusive). These two arrays are considered an exchange from each other if the previous conditions are met, as well as the element in index `i` in each of the arrays is equal to the element in index `j` of the other array. Moreover, every other index, other than `i` and `j` must remain the same between the two arrays. As one might suspect, this predicate is quite strong, and if we notice closely, any two arrays that are an exchange from one another are also a permutation, since they only differ in two positions, which have their values swapped. As such, we must correlate these two predicates:

GOSPEL
```
(*@ axiom exchange_occ : forall a1 a2: int array, i j: int.
    exchange a1 a2 i j -> permut a1 a2 *)
```

With this, we can now proceed to the specification of the `swap` function:

GOSPEL + OCaml
```
let swap (arr: int array) i j = (* ... *)
(*@ requires 0 <= i < Array.length arr
    requires 0 <= j < Array.length arr
    ensures exchange arr (old arr) i j
    ensures permut arr (old arr) *)
```

To avoid invalid accesses we must restrict the values of `i` and `j` to be within the boundaries of the array `arr`. Furthermore, we use our newly created predicate that combines effortlessly with this function to ensure that only two indexes are changed and they come in the form of a swap between the corresponding values. Furthermore, this is a permutation, as previously mentioned.

4.3.3 Insertion Sort

Insertion sort is a classical sorting algorithm, and its main characteristic is that it takes an element and it will check position by position where that element belongs. In a functional setting this is done by traversing the sorted sub-list and stopping once the current element in the sub-list is greater than the value to insert. In an imperative context, this can be done by swapping. We assume that the first element is sorted by default and start from the second value onwards:

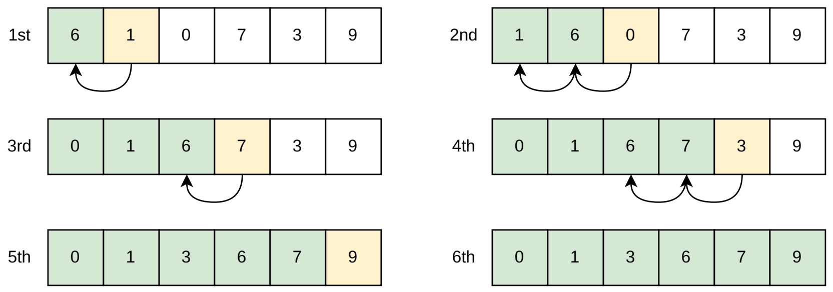


Figure 4.6: Selection sort example

So, we start by processing the second element, this amounts to checking the position immediately before it, and if it is lesser than that value, we perform the swap operation. This process is repeated iteratively for the value that we are placing (represented in yellow), until it reaches a value that it is smaller or equal, or once it reaches the beginning of the array.

This behaviour can be coded as:

OCaml
```
let in_sort arr =
  for i = 1 to (Array.length arr) - 1 do
    let j = ref i in
    while !j > 0 && arr.(!j - 1) > arr.(!j) do
      swap arr !j (!j - 1);
      j := !j - 1
    done
  done
```

As previously mentioned, we start by assuming that index 0 is already sorted, that is whay the `for` loop start at index 1. The element in index i is the value currently being processed (represented as yellow in the figure). So, create a pointer to the current position of the element we are processing, which starts as i, and may change due to swapping. After this, we begin the process of going backwards to find the correct position within previously sorted elements (all elements from 0, inclusive, to i, exclusive). This can be achieved with a `while` loop with two conditions, one to check if the loop has ended, note that this can only be performed if there is an element before index `j`, in other words, `j` must be at least one since it is "behind" on position, and the other condition is that the index immediately before `j` must be strictly greater than the value in `j` so that we can perform the swap operation. If these conditions are met, we can swap the value in index `j` with the value in index `j-1`, and then decrease `j` by 1.

Verification-wise, this algorithm is not as simple as it might seem from a computational perspective. However, our verification library for arrays and the swapping function are already large steps in this endeavour. So, starting with the function itself:

GOSPEL + OCaml

```
let in_sort arr = (* ... *)
(*@ ensures sorted arr
    ensures permut arr (old arr) *)
```

These two **ensures** conditions are quintessential in any sorting algorithm. Since, we are no longer using recursive functions, unlike the previous sections, we no longer need to prove termination here, at least, and since we are dealing with mutable memory, we may use it in the post condtions, rather than a result variable. Moving on to the outer loop (`for` loop):

GOSPEL + OCaml

```
for i = 1 to (Array.length arr) - 1 do
(*@ invariant sorted_sub arr 0 i
    invariant permut arr (old arr) *)
  (* ... *)
done;
```

This loop reflects the post condtions in the main function, although to a partial degree, in particular, concerning the `sorted` predicate, since only a subset of the array will be sorted. This subsection corresponds to the elements of the left side of the loop variable, `i`, in other words, every index between 0, inclusive, and i exclusive, since from `i`, inclusive, onwards the elements have not been processed yet. This corresponds to using the `sorted_sub` predicate from 0 to i. Finally, the inner loop (`while` loop) can be specified as:

GOSPEL + OCaml

```
while !j > 0 && arr.(!j - 1) > arr.(!j) do
(*@ variant !j
    invariant 0 <= !j <= i
    invariant forall p q.
      0 <= p <= q <= i -> q <> !j -> arr.(p) <= arr.(q)
    invariant permut arr (old arr) *)
  (* ... *)
done
```

Reference `j` is the loop variable and is responsible for every access, directly or indirectly, as such, to avoid accessing invalid memory positions, we must bound `!j`. Since `j` references `i` at the start and decreases every iteration, it is clear that `!j <= i`. The left hand side of the equation in the first invariant is obtained from the fact that if the element to be processed is smaller than every other before it, then it will be placed in index 0, at the start of that iteration `j` will point to 1, and, at the end of that iteration it decreased by 1, which updates it to 0 and breaks the loop, therefore `0 <= !j <= i`. Moreover, termination of the `while` loop can be proven by the value pointed to by `j`, since it is a positive value that decreases every iterarion and remains non-negative throughout the loop. Additionally, we must also reflect the conditions on the code block above, similarly to before, since this is what allows to provethe condtions on that block step-by-step. Due to its simplicity, the `permut` predicate remains the same all throughout. However, the `sorted` predicate is a bit more complex.

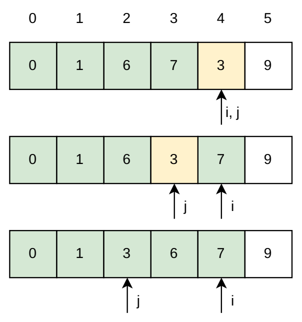


Figure 4.7: Selection sort example

If we extend our interval to include i, we can see that in most situations it is sorted, in fact, the only index that is not sorted coincides with the value of `!j`. Therefore, we may define an adaptation of the sorted predicate that excludes the `!j`, as that is exactly what we do: for any two values between 0 and i (both inclsuve), such that the largest is different from `!j`, then the corresponding value of the lowest index is lesser or equal to the corresponding value of the highest index. In the midst of this confusing condition a question may arise, should `p` not differ from `!j` too? The answer is, surprisingly, no, since if we consider `p = !j`, then everything that comes after `p` has been previously swapped, therefore the corresponding values are strictly higher that `arr.(!j)`.

4.4 Exercises

Implement and specify the:

1. Functional insertion sort
2. Tail recursive insertion sort
3. Polymorphic functional insertion sort
4. Optimized tail recursive selection sort (Hint: Use `::` instead of `@`. Beware of how data is stored).
5. Imperative selection sort (Hint: Use a swap-based implementation)
6. Polymorphic functional merge sort
7. Unoptimized tail recursive quick sort (Hint: Always go the same side first)
8. Polymorphic functional quick sort

Chapter 5

Data Structures

The next step is to discuss the verification of data structures. This is because a data structure implementation usually contains several functions, which makes them longer case studies. Especially compared to the examples in previous chapters. To verify data structures more advanced verification techniques are required. This chapter introduces new and advanced topics, such as type invariants and refinement proofs.

5.1 Refinement Proofs

Most mainstream programming languages contain several features to reduce code complexity, for instance, import systems that allow to separate a single program into several files, some of which may be interfaces. Such files are a form of abstraction that allow to declare high-level documentation, such as function signatures, *i.e.* their name and typing information. Interfaces, also known as signatures in the context of OCaml, can be implemented by multiple distinct classes, or modules, using OCaml's naming convention. This allows for different behaviour across these modules, while maintaining a common structure. A module that adheres to a signature must implement every single declaration in it, while strictly maintain the original type and naming information. Although, the module is not limited by its signature. New functions or other elements may be added on top of what was pre-established. In case a module does not comply with the signature it adheres to, then a type error is raised. This is checked by the compiler, and can be seen as a form "lightweight" verification. Note that OCaml modules that do not explicitly adhere to a pre-established signature, by the programmer, will be subjected to type inference, during compile time, to determine their own signature.

This idea has been reused in the context of formal verification: interfaces can be annotated with a specification language. Modules must comply with the corresponding signature, both in terms of the typing information and the logical properties. The process of verifying if a module logically respects its signature is known as proof by refinement [15], which is available in Cameleer and Why3. A proof by refinement can be seen as an application of the consequence rule from Hoare Logic [10]:

$$\frac{P_i \Longrightarrow P_m \qquad \{P_m\}\ f\ \{Q_m\} \qquad Q_m \Longrightarrow Q_i}{\{P_i\}\ f\ \{Q_i\}}\ \textit{CONSEQUENCE}$$

Where P_i and Q_i represent, respectively, the pre-conditions and post-conditions of function f defined in the interface's annotation. Meanwhile, P_m and Q_m denote, respectively, the pre-conditions and post-conditions of function f defined in the implementation's annotation. By analysing the rule, one can state that for the refinement proof to be valid, P_i must logically imply P_m, and Q_m must imply Q_i. This property allows reinforcing the pre-conditions, while weakening the post-conditions of function f.

The refinement of type invariants, in turn, is not a direct application of Hoare Logic. Instead, the refinement proof, for a type invariant I_i in the interface, and a type invariant I_m in the corresponding implementation, is the logical conjunction between three conditions: (1) $I_m \Longrightarrow I_i$, (2) type compliance, and (3) the fields declared in the interface must be a subset of the fields found in the implementation. Moreover, ghost [6] fields in the interface, may become concrete fields in the implementation.

A simple and intuitive example of a proof by refinement can be seen below:

GOSPEL + OCaml

```
module type I = sig
  val inc: int -> int
  (*@ r = inc x
      requires x >= 0
      ensures r > x *)
end

module Inc : I = struct
  let inc x = x + 1
  (*@ r = inc x
      requires true
      ensures r = x + 1 *)
end
```

The `Inc` module respects signature `I`. Thus, it must implement the `inc` function, with the exact name, typing information, and comply with its logical properties, based on the rule of consequence. It is possible to observe that one may strengthen the pre-conditions and weaken the post-conditions in the signature, when compared to its implementation in the `Inc` module.

5.2 Type Invariants

Record types can also be annotated with logical conditions that reflect the properties between the various fields. These conditions are known as type invariants, and must hold after creating or modifying an instance of that type. As such, type invariants lead to more verification conditions (VCs) in functions that receive, as a parameter, return, create and/or modify instances of an annotated record type. Moreover, in `Why3`, a set of type invariants, associated with a given type, generate one or more standalone proof goals. To prove these VCs, one must prove it with a witness, an instance of said type, typically with default-/empty values. Providing a witness in `WhyML` is easily achieved with the `by ...` construct. However, this construct is not available in `GOSPEL`, which significantly complicates discharging such proof goals. Purely functional witnesses, in our experience, tend to be proven automatically with the SMT solvers, assuming that the invariants are not inconsistent between themselves. Whereas, records that contain arrays as fields tend to be quite challenging for this class of provers. One (flawed) solution is to use a solver from a different family of automated theorem provers, for instance `Eprover` [17], which is more suited to deal with existential quantifiers. Despite this, it may not necessarily be capable of discharging every type invariant by itself. Another flawed solution is to manually provide a witness. This is exactly what the `by ...` construct in `WhyML` does. However, since it is not available in `GOSPEL`, manually providing a witness becomes inconvenient, although possible, due to scoping issues, especially when dealing with generic types. The final solution, which is also flawed, is to ignore the witness proof. These are quite technical by nature, and are not at all interesting, in our opinion, from a logical standpoint. In the context of resizeable arrays, we shall exemplify what a simple witness proof looks like. By contrast, we shall omit, from this textbook, the one found in circular queues, due to its complexity, although it still is provided in the online repository.

5.3 List Zipper

A zipper [11] is an iteration technique that can be applied to purely functional data structures for more fine-grained traversal. This technique, usually consists in finding an alternative representation to recursive data types. Such types do not offer the possibility to go backwards, which may not be adequate for some use cases. Hence, the zipper offers the possibility to revisit previous nodes and explore other paths (when applied to trees, for instance). The idea of a zipper applied to a list is similar to that of a cursor in a text editor. One may place the cursor, by moving left or right, for instance, before doing any alterations to the contents. In `OCaml`, one may use the module system to create their own library. This is particularly useful to define new data structures. In this case study, we shall consider the following signature (similar to an interface):

OCaml
```
module type Zipper = sig
  type 'a zipper

  exception OutOfBounds

  val empty: unit -> 'a zipper
  val zipper_of_list: 'a list -> 'a zipper
  val zipper_to_list: 'a zipper -> 'a list
  val move_right: 'a zipper -> 'a zipper
  val move_left: 'a zipper -> 'a zipper
  val insert: 'a zipper -> 'a -> 'a zipper
  val delete_left: 'a zipper -> 'a zipper
  val delete_right: 'a zipper -> 'a zipper
end
```

This signature includes two constructors, one for the empty zipper, and the other, `zipper_of_list`, transforms a list into a zipper. The inverse operation is also possible, to convert a zipper into a list. Moreover, we are also able to move the cursor left or right, and make local changes, such as inserting an element or deleting the element to the left or to the right of the cursor. Additionally, we have declared an exception to handle invalid accesses, when altering the contents of the zipper.

Just from the previous descriptions alone, we should be able to devise logical contracts that any implementation of a zipper must comply. Typically, the first step when annotating a signature is to outline the logical model of complex types. Note that not all types may need a logical module. In this case, the zipper can be considered complex enough to provide and module. Additionally, to be able to describe the logical properties of the operations available in the module, we need to, at least, know its high-level structure. Which can be represented by a list and the cursor. This can be easily explained from the fact that the zipper is meant to be a more flexible data structure that uses a cursor to control traversal, but from a logical perspective, the data can still be viewed as a list:

GOSPEL + OCaml
```
type 'a zipper
(*@ model cursor: int
    model view: 'a list *)
```

In the previous chapters, we have purposefully refrained from discussing a pertinent topic when verifying lists and arrays. It is well-known that linked lists are arrays are quite similar on a high-level, despite having fundamental differences, especially in OCaml. From a logical perspective, both are linear data structures with similar sets of operations. Verification tools, often, offer a logical data type, known as sequences, to abstract and congregate both data structures. While it is not necessary to use sequences in this case study, GOSPEL offers sequences with a comfortable syntax when accessing elements in a list. Moreover, within our tool set, lists and arrays are automatically converted to sequences, when necessary. Starting with the constructors:

GOSPEL + OCaml
```
val empty: unit -> 'a zipper
(*@ res = empty ()
    ensures 0 <= res.cursor <= Sequence.length res.view
    ensures res.view = Sequence.empty *)
```

```
val zipper_of_list: 'a list -> 'a zipper
(*@ res = zipper_of_list l
    ensures 0 <= res.cursor <= Sequence.length l
    ensures res.view = l *)
```

Independently of the constructor used, the cursor should not be out of bounds, *i.e.* it should be a non-negative value that is at most the length of the view, which would represent being placed after the last element. Note that view is a common name used to describe a data structure used as a logical representation. The choice of providing a range of values for the cursor, rather than a concrete, allows implementations of this signature to not be tied to a single value. Although, the most natural choice, in our opinion, would be 0. The second post-condition serves to check if the elements inside view are the ones expected. Similarly, when extracting a zipper to a list, the result should correspond to the view:

GOSPEL + OCaml

```
val zipper_to_list: 'a zipper -> 'a list
(*@ res = zipper_to_list z
    ensures res = z.view *)
```

Following the order pre-established in the signature, we arrive at the traversal functions: `move_right` and `move_left`. These functions are meant for traversal only, hence these should not change the contents of the zipper, `i.e.` the view of the result is the same of the initial zipper (`z`). When moving a position, we are effectively changing the position of the cursor. If the cursor is moving to the right side, then an increment of one is expected. If moving to the left, then the cursor is a decrement by one. We must also account of out of bounds accesses, if exception `OutOfBounds` is raised when moving to the right, then it means that the cursor was already placed at the rightmost position. Likewise, if it is raised when moving left, it means that the cursor was already at 0 (leftmost index):

GOSPEL + OCaml

```
val move_right: 'a zipper -> 'a zipper
(*@ res = move_right z
    raises OutOfBounds -> z.cursor = Sequence.length z.view
    ensures res.cursor = z.cursor + 1
    ensures res.view = z.view *)

val move_left: 'a zipper -> 'a zipper
(*@ res = move_left z
    raises OutOfBounds -> z.cursor = 0
    ensures res.cursor = z.cursor - 1
    ensures res.view = z.view *)
```

Inserting, removing or changing elements in a verified data structure are operations that require precise and carefully written annotations for successful proofs. Namely, to ensure that the data stays consistent after modifications. For now, let us consider the `insert` operation. When inserting a single element to a zipper, it is expected that it only grows in length by one unit. Moreover, the cursor should increment by one and accompany the new element. This can prevent subtle errors, such as repeatedly inserting elements onto the same index, making the previous shift one index to the right each time, instead of maintaining their original order. As briefly mentioned before, this operation inserts the element into the current position of the cursor, and triggers a right shift of the elements that were to the right of the cursor. The elements to the left remain in their original positions. These three behaviours are described in the three last post-conditions:

GOSPEL + OCaml
```
val insert: 'a zipper -> 'a -> 'a zipper
(*@ res = insert z x
    ensures res.cursor = z.cursor + 1
    ensures Sequence.length res.view = 1 + Sequence.length z.view
    ensures res.view[z.cursor] = x
    ensures forall i. 0 <= i < z.cursor -> res.view[i] = z.view[i]
    ensures forall i. z.cursor < i < Sequence.length z.view ->
      res.view[i+1] = z.view[i] *)
```

When deleting an element from the zipper, we offer the possibility of deleting to the left side or to the right side of the cursor. In both cases, as expected, the length of the view should decrement by only a single unit. Starting with the right side. If the cursor is placed after the rightmost element, then it means we cannot perform this operation. Thus, we raise the `OutOfBounds` exception in such event. Similarly to the insert operation, the elements on the left side of the cursor remain unchanged and in the same place. Starting from the index of the cursor (inclusive) until the end of the view, then every element in the result's view is the same as the one found in the next index of the original zipper:

GOSPEL + OCaml
```
val delete_right: 'a zipper -> 'a zipper
(*@ res = delete_right z
    raises OutOfBounds -> z.cursor = Sequence.length z.view
    ensures Sequence.length res.view = Sequence.length z.view - 1
    ensures forall i. 0 <= i < z.cursor ->
      res.view[i] = z.view[i]
    ensures forall i. z.cursor <= i < Sequence.length res.view ->
      res.view[i] = z.view[i+1] *)
```

The specification of `delete_left` follows a similar strategy. If the cursor is originally placed on a given position, then the element to be deleted is placed in the index prior to that. This means that from 0 to the index of the element to be deleted, exclusive, every element remains unchanged. Assuming that the cursor is not placed on index 0, where there are no elements to be deleted, thus raising the `OutOfBounds` exception. After that index to the end of the view (exclusive), the element in the result is equal to the one found in the next index of the original index:

GOSPEL + OCaml
```
val delete_left: 'a zipper -> 'a zipper
(*@ res = delete_left z
    raises OutOfBounds -> z.cursor = 0
    ensures Sequence.length res.view = Sequence.length z.view - 1
    ensures forall i. 0 <= i < z.cursor - 1 ->
      res.view[i] = z.view[i]
    ensures forall i. z.cursor - 1 <= i < Sequence.length res.view ->
      res.view[i] = z.view[i+1] *)
```

Now that we have discussed the signature in-depth, it is time to move on to the implementation. We shall start with the implementation of the `zipper` type:

GOSPEL + OCaml

```
module ListZipper : Zipper = struct

  type 'a zipper = {
    left: 'a list;
    right: 'a list;
    cursor: int; [@ghost]
    view: 'a list; [@ghost]
  }
  (*@ invariant Sequence.(==) view (Sequence.append (Sequence.rev left) right)
      invariant 0 <= cursor <= Sequence.length view
      invariant cursor = List.length left *)

  (* ... *)

end
```

As we have previously mentioned, linked lists in OCaml do not allow traversing backwards, hence the need for an alternative representation of the data. This can be achieved by using two lists, one that contains the elements to the left of the cursor, and the other with the elements on the right side. For efficiency purposes, the `left` list is reverse. This allows for quick access to element immediately to the left of the cursor, since it would be the head of the `left` list. This property is outlined in the first **`invariant`** clause of the `zipper` type. A type invariant is similar to a loop invariant, in the sense that when a zipper is created it must comply with its invariant, and these conditions should also be preserved when performing modifications its contents. Contrary to loop invariants, which are limited to the scope of the loop, type invariants generate verification conditions wherever that type is used. Therefore, a crucial part of certifying data structure operations is to preserve type invariants. For the `zipper` type, other invariants include the cursor ranging from between 0 and the length of the view (inclusive), and that it is equal to the length of the `left` list. The reasoning behind the former invariant was previously explained, and is meant to restrict invalid accesses. The latter can be explained with the fact that the `left` list represent the elements before the cursor, hence, it is natural that the value of the cursor is the index immediately after them, or in other words, the length of `left`.

Implementation-wise, the `empty` constructor initializes all lists with the empty list, and the cursor as 0, which is the only sensible and valid value here:

GOSPEL + OCaml

```
let empty () = { left = []; right = []; cursor = 0; view = [] }
(*@ res = empty ()
    ensures res.cursor = 0
    ensures res.view = Sequence.empty *)

let zipper_of_list l = { left = []; right = l; cursor = 0; view = l }
(*@ res = zipper_of_list l
    ensures res.cursor = 0
    ensures res.view = l *)
```

The `zipper_of_list` function, on the other hand, initializes both `right` and `view` with `l`, the list received in the parameters. The reasoning behind this choice is twofold. (1) If we were to place the elements on the `left` field, then we would have to reverse the list to maintain consistency with `view`, this would increase the time complexity from $\Theta(1)$ to $\Theta(n)$. (2) The `right` field represents the elements yet to visit by the cursor. This is the most natural and intuitive option.

To convert a zipper back to a list, it simply amounts to reversing the `left` list and appending it to the `right` list. This can be concisely achieved with the operation `List.rev_append`, which does exactly that: reversing the first parameter, and appending the result to the second:

GOSPEL + OCaml

```
let zipper_to_list z = List.rev_append z.left z.right
(*@ res = zipper_to_list z
    ensures res = z.view *)
```

The traversing functions simply amount to moving the head of the corresponding list to the opposite list. For instance, if we move right, then the head of the `right` list is placed on the head of the `left` list. Remember that the `left` list is reversed, so the head is, in fact, the element next to the cursor. The view remains unaltered during the traversal in both directions. The cursor is increment by one, when moving to the right:

GOSPEL + OCaml

```
exception OutOfBounds

let move_right z =
  match z.right with
  | [] -> raise OutOfBounds
  | x::r ->
    { left = x :: z.left; right = r; cursor = z.cursor + 1; view = z.view }
(*@ res = move_right z
    raises OutOfBounds -> z.cursor = Sequence.length z.view
    ensures res.cursor = z.cursor + 1
    ensures res.view = z.view *)
```

Conversely, when moving to the left, the head of the `left` list is placed at the head of the `right` list, and the cursor is decremented by one:

GOSPEL + OCaml
```
let move_left z =
  match z.left with
  | [] -> raise OutOfBounds
  | x::l ->
    { left = l; right = x :: z.right; cursor = z.cursor - 1; view = z.view }
(*@ res = move_left z
    raises OutOfBounds -> z.cursor = 0
    ensures res.cursor = z.cursor - 1
    ensures res.view = z.view *)
```

When an element is inserted the cursor is expected to be placed after it. So, this can be achieved by placing the new element at the head of the `left` list. The `right` list remains unchanged, and the cursor is incremented by one. The most significant change is the `view`, which can be replaced with the operation `List.rev_append`, whose operands are the `left` list with the new element at its head and the right list:

GOSPEL + OCaml
```
let insert z x = {
  z with left = x::z.left; cursor = z.cursor + 1;
    view = List.rev_append (x :: z.left) z.right
}
(*@ res = insert z x
    ensures res.cursor = z.cursor + 1
    ensures res.view[z.cursor] = x
    ensures Sequence.length res.view = 1 + Sequence.length z.view
    ensures forall i. 0 <= i < z.cursor -> res.view[i] = z.view[i]
    ensures forall i. z.cursor < i < Sequence.length z.view ->
      res.view[i+1] = z.view[i] *)
```

Verification-wise, the annotation is the same as the one defined in the signature. There are not any other interesting concrete details regarding this specific implementation that need to be verified, and are not captured by the refinement proof or the type invariant. Similarly, the specification on the delete functions does not change. Regarding `delete_right`'s implementation, one may use pattern matching on the `right` list to identify the case where there is no element to the right of the cursor. In such event, we raise the `OutOfBounds` exception. In case we can successfully delete to the right, then, we create a new zipper, where the new `right` list does not contain the head of the previous. Once more, to produce the new `view`, we use `List.rev_append` with the update lists, in this case only `right`. The cursor does not need to be updated since we are deleting on the right side of the cursor:

GOSPEL + OCaml
```
let delete_right z =
  match z.right with
  | [] -> raise OutOfBounds
  | _::r -> { z with right = r; view = List.rev_append z.left r }
(*@ res = delete_right z
    raises OutOfBounds -> z.cursor = Sequence.length z.view
    ensures Sequence.length res.view = Sequence.length z.view - 1
    ensures forall i. 0 <= i < z.cursor -> res.view[i] = z.view[i]
    ensures forall i. z.cursor <= i < Sequence.length res.view ->
      res.view[i] = z.view[i+1] *)
```

Meanwhile, for `delete_left`, we use pattern matching on the `left` list to check if it is empty. If so, then we raise the `OutOfBounds` exception, since there is no element to delete. Otherwise, the new zipper will have an updated `left` list, without its previous head. Additionally, the cursor is also decremented by once, to reflect that is sits in between the `left` and `right` lists. Finally, the view is also updated with the new `left` and `right` lists, which, in this case, only `left` was updated:

GOSPEL + OCaml

```
let delete_left z =
  match z.left with
  | [] -> raise OutOfBounds
  | _::l ->
    { z with left = l; cursor = z.cursor - 1; view = List.rev_append l z.right }
(*@ res = delete_left z
    raises OutOfBounds -> z.cursor = 0
    ensures Sequence.length res.view = Sequence.length z.view - 1
    ensures forall i. 0 <= i < z.cursor - 1 -> res.view[i] = z.view[i]
    ensures forall i. z.cursor - 1 <= i < Sequence.length res.view ->
      res.view[i] = z.view[i+1] *)
```

5.4 Tree Zipper

Zippers can also be applied to binary trees. As expected, we have to find an alternative representation for finer-grained traversal. What is considered as finer-grained traversal depends on the context of the problem at hand. This, naturally leads to different representations being more suitable based on the characteristics of the problem. For this section, we shall consider the Same Fringe problem: given two binary trees, determine if these have the exact same in-order traversal. This consists in visiting the left subtree first, followed by the current node, and, finally, the right subtree.

The traditional solution consists in creating a list for each tree, following the in-order traversal, and then comparing the two lists. This approach is considered naive [5], since the algorithm needs to fully traverse both trees produce the lists, and only then compare the elements of the list. For instance, consider the case where the trees differ in just a single element, namely the right child of the root. This would be highly inefficient. A potentially more adequate approach is to traverse the elements on the left spine, and store the right subtrees of each. From there, we compare each element on the left spine, starting from the left most to the root, and the respective right subtree, following a similar strategy of using its left spine. This has the advantage of not traversing the entire tree all at once, unless the tree is left-skewed. Moreover, in cases where the two tree differ, the algorithm may stop early without visiting all elements. The worst-case scenario is if the rightmost elements of the two trees are different.

The previous paragraph gives us a clue on how to define an alternative representation for a tree, that would be considered as our zipper. Given the nature of this problem, the essential operation is the comparison between two zippers. Besides this operation, we, naturally, need to define the constructor functions, namely one for the empty zipper, and another to transform a binary tree into a zipper. Moreover, we shall adopt a polymorphic implementation using the previously studied techniques:

GOSPEL + OCaml

```
module type ComparableType = sig
  type t

  val eq : t -> t -> bool
  (*@ res = eq x y
      ensures res <-> x = y *)
end

module type Zipper = sig
  type elt
  type tree
  type zipper

  (*@ function inorder (t: tree) : elt list *)
  (*@ function zip_inorder (z: zipper) : elt list *)

  val empty : unit -> zipper
  (*@ z = empty ()
      ensures zip_inorder z = [] *)

  val mk_zipper : zipper -> tree -> zipper
  (*@ res = mk_zipper z t
      ensures zip_inorder res = inorder t @ zip_inorder z *)

  val eq_zipper : zipper -> zipper -> bool
  (*@ res = equal z1 z2
      ensures res <-> zip_inorder z1 = zip_inorder z2 *)
end
```

In the *Same Fringe* problem, we are expected to check if two trees have the same in-order traversal. So, to ensure the correctness of our solution, we also need to be able to get the in-order traversal of a zipper, since it is an alternative representation of a tree. Naturally, the in-order traversal of an empty zipper is the empty list. The `mk_zipper` function is not as obvious. When converting a tree into a zipper, it can be attached to another zipper. This is explained from the fact that elements are not processed all at once. Thus, at the start, the left spine of a tree is transformed into a zipper, while the respective right subtrees are stored. At some point, each of those right subtrees might be processed, and since it has higher priority than the remaining elements, it is appended to the front of the zipper, hence the post condition of `mk_zipper`, stating that the in-order traversal of the resulting zipper is equal to concatenating the resulting list of elements resulting from traversing the tree `t`, received from the parameters, with the in-order traversal of the previous zipper. Finally, and as expected, two zippers are equal if their in-order traversal is equal. Moving on to the implementation of the tree zipper:

GOSPEL + OCaml

```
module TreeZipper (C: ComparableType) : Zipper = struct

  type elt = C.t
  type tree = E | N of tree * elt * tree
  type zipper = (elt * tree) list

  let empty () = []

  (* Continued next... *)

end
```

As we have previously alluded, the `zipper` type corresponds to a list of elements (on the left spine of a tree) and their respective right subtree. To achieve this, we have used tuples, containing the element itself, as the first member, and the right subtree as the second. As such, the empty zipper is simply the empty list. Note that the list is ordered from the leftmost element to the root.

GOSPEL

```
(*@ function rec inorder (t: tree) : elt list =
      match t with
      | E -> []
      | N l x r -> inorder l @ (x::inorder r) *)

(*@ function rec zip_inorder (z: zipper) : elt list =
      match z with
      | [] -> []
      | (x, r)::e -> x::(inorder r @ zip_inorder e) *)
```

The in-order traversal corresponds to visiting the left subtree first, then the root element, and, finally, the right subtree. This is easily, and, intuitively achieved for trees, as seen in the `inorder` auxiliary logical definition. Meanwhile, for zippers, although not as intuitive, it can easily be achieved by appending the leftmost element (`x`) at the head of the concatenation of the in-order traversal of the corresponding right subtree, and the recursive call on the remaining elements of the zipper.

Given that our ultimate goal is to compare two zippers, by their in-order traversal, then the data should be ordered from leftmost to the root element, to simplify the equality test. Moreover, since the starting point of the tree to covert is its root, this means that we are filling the list backwards, which allows us to use the `::` operator, which has $\Theta(1)$ time complexity. The order of elements both simplifies the algorithm itself and the associated time complexity:

GOSPEL + OCaml

```
let rec mk_zipper (z: zipper) = function
  | E -> z
  | N (l, x, r) -> mk_zipper ((x, r)::z) l
(*@ res = mk_zipper z t
    variant t
    ensures zip_inorder res = inorder t @ zip_inorder z *)
```

The function `mk_zipper` consists in appending the current node and its right subtree to the previous zipper, and making the recursive call on the left subtree. From a verification perspective, we maintain the post-condition from the certified signature. Additionally, we have to prove the termination of this recursive function, which can de done so with the structure of `t`, the binary tree received in the parameters.

Given the external structure of a zipper, *i.e.* a list, when comparing two zippers, we must compare element by element. However, the internal structure of each element complicates this process. Each element is composed by a value of an arbitrary type `elt`, and the right subtree associated with that node. We can compare directly those values, with the equality test from the `ComparableType` signature, and in case these differ, the function terminates early. However, the same can not be said about the subtrees, since these might have different structures, but alongside with the remaining elements in the zipper, have the same in-order traversal. To fix this problem, we combine the right subtree, or rather, its left spine into a zipper with the tail of the current zipper. Note that the right subtree is originally placed on the left of the elements that composed the current zipper. While the two nodes that were compared were the leftmost elements of the trees being compared. Hence, `mk_zipper` places the right subtree in front of the current zipper:

GOSPEL + OCaml

```
let rec eq_zipper z1 z2 =
  match z1, z2 with
  | [], [] -> true
  | (x1, r1)::e1, (x2, r2)::e2 ->
    C.eq x1 x2 && eq_zipper (mk_zipper e1 r1) (mk_zipper e2 r2)
  | _ -> false
(*@ res = eq_zipper z1 z2
    variant List.length (zip_inorder z1)
    ensures res <-> zip_inorder z1 = zip_inorder z2 *)
```

Verification-wise, we maintain the post-condition fron the certified signature, once more. The termination proof of this function is not as obvious, however. The length of one of the zippers does not suffice. This is can be explained from the fact that `mk_zipper` may increase the size of the current zipper, since many elements of the original tree are “hidden” in the right subtree, rather than being directly on the zipper. In this scenario the termination proof is not valid, since it must decrease every recursive call. Instead, and since we are not interested in performance in logical environments, we may use the left of the list of elements that make the in-order traversal of one of the zippers. If we notice closely, every recursive call each zippers does lose one element, that being the first value of the pair, or rather, its leftmost element. This also means that `zip_inorder z1` successively loses one element, that being the one at the head of the resulting list.

The final piece of this puzzle is to solve the *Same Fringe* problem. To do so, we use a modular approach and define a functorized client module that receives as a parameter another module that adheres to the certified signature `Zipper`:

GOSPEL + OCaml
```
module SameFringe (Z: Zipper) = struct

  let same_fringe t1 t2 =
    let e = Z.empty () in
    Z.eq_zipper (Z.mk_zipper e t1) (Z.mk_zipper e t2)
  (*@ res = same_fringe t1 t2
      ensures res <-> Z.inorder t1 = Z.inorder t2 *)

end
```

Solving the *Same Fringe* problem, with the `Zipper` signature, amounts to transforming each tree into a zipper, on top of an empty zipper, since there are no previous elements, and comparing the two zippers. Once more, to solve the *Same Fringe* problem, one must determine if two given trees have the same in-order traversal. This is exactly the post-condition of the *same_fringe* function. This condition is easily proven based on the post conditions of the `Zipper` operations. When the zipper passed in the parameters of `mk_zipper` is empty, then `zip_inorder` of the result is equal to the `inorder` of the tree. Since, the empty zipper has an empty in-order traversal. Moreover, in this scenario, when testing the equality of the in-order traversal of the two zippers, as per the post-condition of `eq_zipper`, we can replace those values with the `inorder` of the original trees, which is exactly the post-condition of `same_fringe`.

5.5 Resizeable Arrays

The built-in arrays found in `OCaml` are not dynamic. This means that we cannot expand or shrink the maximum amount of data that can be stored at request, since the memory blocks are pre-allocated (contiguously) when the array is created. This limitation can be answered with a custom-made data structure known as resizeable array. Besides common array operations, the data structure includes a resize function, which encompasses both shrinking and expanding. Following this brief description, we defined the following signature:

OCaml
```
module type RArray = sig
  type elt
  type rarray
  exception OutOfBounds
  val make : int -> elt -> rarray
  val length : rarray -> int
  val get : rarray -> int -> elt
  val set : rarray -> int -> elt -> unit
  val resize : rarray -> int -> unit
end
```

This signature contains the indispensable operations any array must have, in particular, initialization (`make`), knowing its length, as well as access and mutation of a given index (`get` and `set`, respectively). The two last operations mentioned above must successfully handle invalid indices, thus, we opted to define an exception, `OutOfBounds`, for that matter. Furthermore, we also define, indirectly, a new generic type, `elt`, that denotes the type of the elements of the array. And, finally, the last operation is exactly its core function, that being `resize`.

Implementation-wise, resizeable arrays are built on top of `OCaml` arrays. This is the most natural strategy, since the two share several characteristics, and resizeable arrays are effectively a slightly more versatile version of arrays. So, this raises the question: how to competently resize an array? If we need more memory that the amount currently allocated, then there is no alternative other than creating a bigger array and copy the original contents. If we wish to shrink the array, repeating the previous strategy is also a possibility, although not necessarily a good one. For instance, consider the scenario where we shrink the array, and later expand the array to a size in between the value after shrinking and the original value. This means that we would need to copy the contents of the array twice, which is undesired given the added linear time complexity. To avoid this problem, we could have kept the original memory and simply limited the maximum index accessible. To achieve this solution, we need to store a variable that holds the maximum index, besides the array itself. In `OCaml`, when instantiating an array, we also need a default value, which will be placed in every memory block. This leads us to define a signature for a polymorphic type with a default value:

OCaml
```
module type PolymorphicType = sig
  type t
  val default : t
end
```

This value is only meant to be placeholder, hence, there is no need to annotate it with any kind of logical conditions. Our implementation of resizeable arrays makes use of this signature, as follows:

OCaml
```
module ResizeableArray (P: PolymorphicType) : RArray = struct
  type elt = P.t

  type rarray = {
    default : elt;
    mutable size: int;
    mutable data: elt array;
  }
end
```

The `rarray` type is a record with mutable fields. The `size` and `data` fields are expected to change during the `resize` operation. Whereas, the `default` does not necessarily need to change.

Moving on to the `rarray`'s operations, `make` and `length` are quite simple to implement. The first receives, as parameters, the size of the array and the default value, to return a newly created resizeable array.

OCaml
```
let make n d = {
  default = d;
  size = n;
  data = Array.make n d
}
```

The second operation returns the length of the resizeable array, which is defined as the `size` variable, since it holds the maximum index accessible from outside the module:

OCaml
```
let length a = a.size
```

The `get` and `set` functions also work as one might expect. However, we must check for the incorrect accesses with the `size` variable, instead of the length of `data`. We perform this test, and in case of failure we raise an exception, namely `OutOfBounds`:

OCaml
```
exception OutOfBounds

let get a i =
  if 0 <= i && i < a.size then a.data.(i)
  else raise OutOfBounds

let set a i v =
  if i < 0 || a.size <= i then raise OutOfBounds
  else a.data.(i) <- v
```

Finally, the most complex operation in this data structure is `resize`. Given that there is a difference between the length of the accessible memory and the available memory, there are three possible cases based on the new size: (1) this value exceeds the available memory; (2) this value exceeds the accessible memory, but is within the available memory; (3) this value is smaller (or equal) than the accessible memory. Case (2) is trivial, we simply update `size` to the new value. Also note that this step happens regardless of the case. Case (1) implies allocating new memory to accommodate the desired higher capacity. In case (3), we will also deallocate between the new size and the previous. This step is not necessarily required, however, for demonstration purposes related to the proof, we have decided to deallocate said memory range.

OCaml
```
let resize a s =
  let l = Array.length a.data in
  if s > l then begin
    let a’ = Array.make (max s (2*l)) a.default in
    Array.blit a.data 0 a’ 0 a.size;
    a.data <- a’
  end else if s < a.size then begin
    for i = s to a.size - 1 do
      a.data.(i) <- a.default
    done;
  end;
  a.size <- s
```

In the `if` branch, we can see the implementation of case (1). When `s` is larger than `l` (length of the available memory), we must allocate a bigger array. To avoid potential naive resizes in succession, for instance, if the user repeatedly resizes the array with increments of one, we may allocate more memory than stated. However, this has the problem of reserving too much memory that can potentially be unused. So, to define the size of the new array, we follow a conservative approach. We reserve the maximum value between the desired size (`s`) and double of `l`. The double of the current memory is a good heuristic: it is based on the user memory allocation patterns, instead of a fixed increment, and 2 is the smallest integer we could possibly use in multiplication to obtain a larger value. Moreover, the desired size (`s`), itself, may be larger than double of `l`. The `Array.blit` function takes a source array, a starting position for the source array, a destination array, a starting index for the destination array, and the amount of elements to copy, as parameters. This function copies the contents of array `a.data`, starting from 0 to `a.size - 1`, to the new array, `’a`, starting from index 0. Once the new array has been filled, we update the `data` field of record `a`. Meanwhile on the `else-branch`, every element starting from `s` (inclusive) until the current size value (exclusive), is set to the default value stored in the record. Note that from (current) `a.size` forward, the memory blocks are already filled with the default value. When `a.size` is equal to `Array.length a.data`, this is easy to see, since there are no elements after `a.size`. If `a.size` is smaller than `s`, and `s` is smaller than the length of `data`, then this means that the `resize` operation was performed at least once. In this scenario, there is no need to deallocate memory, since we are technically expanding the array. Furthermore, if `s` lies between `a.size` and the length of `data`, this means that the array’s sizes has been changed at least once, which guarantees that the indexes between those two values had already been filled with the default value. Finally, as previously mentioned, we update the `a.size` field with the desired memory length (`s`), across all cases.

Our verification efforts start with the signature, namely the type invariant, as expected. The logical model of type `rarray` is composed by two fields, both mutable, those being `size` to denote the virtual size of the `rarray`, and `data` to store the elements as an array. In terms of invariants, we only need to guarantee that the value of `size` is non-negative and does not exceed the size of the array. When devising annotated signatures, we must abstract from concrete implementation details. In this implementation, we deallocate the elements starting from `size` (inclusive) until the end of the array. This is a concrete implementation detail that does not define a resizeable array. Other implementations of this data structure may opt for not simulating deallocation when shrinking the array. Thus, we annotated the `rarray`, in the signature, as follows:

GOSPEL + OCaml
```
type rarray
(*@ mutable model size: int
    mutable model data: elt array
    invariant 0 <= size <= Array.length data *)
```

The `make` and `length` functions are annotated as seen below, without much surprise:

GOSPEL + OCaml
```
val make : int -> elt -> rarray
(*@ res = make n d
    requires n >= 0
    ensures res.size = n *)

val length : rarray -> int
(*@ res = length a
    ensures res = a.size *)
```

The `make` function receives an integer value, as a parameter, that should be non-negative, since there is no such thing as allocating negative memory. This represents the virtual size of the newly created resizeable array, which may not necessarily be equal the real number memory blocks allocated. Some implementations could potentially reserve more memory upfront. Similarly, the `length` function returns the virtual memory. The `get` function is also very standard:

GOSPEL + OCaml
```
val get : rarray -> int -> elt
(*@ res = get a i
    raises OutOfBounds -> i < 0 || i >= a.size
    ensures res = a.data.(i) *)
```

It raises the `OutOfBounds` exception when the index provided in the parameters is invalid, and the result must correspond, in fact, to the value of the associated index. Meanwhile, the `set` operation is slightly more complex, although, standard, as well:

GOSPEL + OCaml
```
val set : rarray -> int -> elt -> unit
(*@ set a i v
    modifies a
    raises OutOfBounds -> i < 0 || i >= a.size
    ensures a.data.(i) = v
    ensures forall k. 0 <= k < Array.length a.data -> k <> i ->
      a.data.(k) = old a.data.(k) *)
```

This function modifies the contents of the `data` field of a resizeable array, and more generally, the resizeable array itself. Moreover, the value in the index supposed to be changed, must be equal to the provided value in the parameters, after execute. Every other position is expected to remain unchanged. Similar to `get`, this operation raises `OutOfBounds` is the index is invalid.

The `resize` function, similar to the constructor, must receive a non-negative value for the new virtual size of the resizeable array, and after execution, the virtual size is expected to be this value, received in the parameters:

GOSPEL + OCaml
```
val resize : rarray -> int -> unit
(*@ resize a s
    modifies a
    requires s >= 0
    ensures a.size = s
    ensures forall k. 0 <= k < min s (old a.size) ->
      a.data.(k) = old a.data.(k) *)
```

Furthermore, during this process, we should not lose any element from the start of the array until the old virtual size (exclusive). Unless, when shrinking the resizeable array. All cases can be encompassed with the minimum value between `s` and `old a.size`. The `size` field is expected to change in every scenario, and `data` may change if needed, hence the need for the clause `modifies a`.

Moving onto our resizeable array module, we can expand upon the type invariant from the module with the condition that all elements starting from the index of virtual size (1st non-accessible position) until the end of the `data` array:

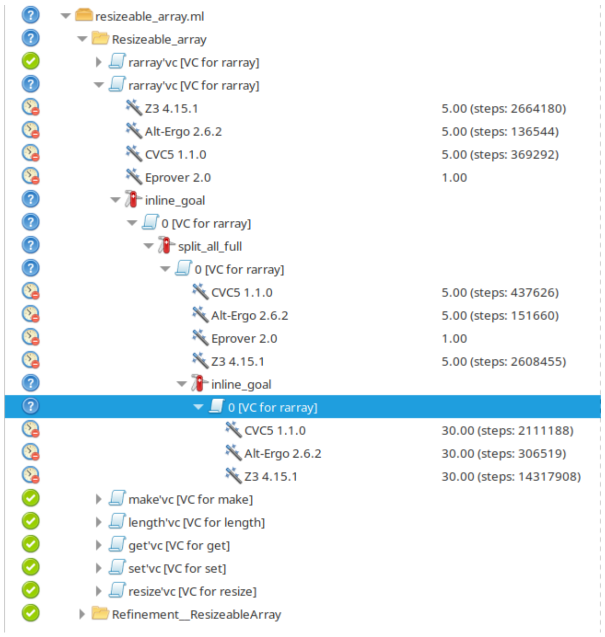
GOSPEL + OCaml
```
type rarray = (* ... *)
(*@ invariant 0 <= size <= Array.length data
    invariant forall k: int. size <= k < Array.length data ->
      data.(k) = default *)
```

However, these invariants are not discharged, even with Eprover or the maximum time limit in Why3.

To help our solvers, we need to provide a witness. Once more, creating witnesses in GOSPEL is not at all practical, and moving forward, those will not be studied.

The failed proof goal can be seen in the Why3 IDE and corresponds to the following logical condition:

WhyML
```
goal rarray'vc :
  exists default:t, size1:int, data1:my_array t.
```



```
resizeable_array.ml
  Resizeable_array
    rarray'vc [VC for rarray]
    rarray'vc [VC for rarray]
      Z3 4.15.1                          5.00 (steps: 2664180)
      Alt-Ergo 2.6.2                     5.00 (steps: 136544)
      CVC5 1.1.0                         5.00 (steps: 369292)
      Eprover 2.0                        1.00
      inline_goal
        0 [VC for rarray]
          split_all_full
            0 [VC for rarray]
              CVC5 1.1.0                 5.00 (steps: 437626)
              Alt-Ergo 2.6.2             5.00 (steps: 151660)
              Eprover 2.0                1.00
              Z3 4.15.1                  5.00 (steps: 2608455)
              inline_goal
                0 [VC for rarray]
                  CVC5 1.1.0             30.00 (steps: 2111188)
                  Alt-Ergo 2.6.2         30.00 (steps: 306519)
                  Z3 4.15.1              30.00 (steps: 14317908)
    make'vc [VC for make]
    length'vc [VC for length]
    get'vc [VC for get]
    set'vc [VC for set]
    resize'vc [VC for resize]
  Refinement__ResizeableArray
```

Figure 5.1: Failed invariant proof

```
(0 <= size1 /\ size1 <= data1.length) &&
(forall k:int. size1 <= k /\ k < data1.length ->
  get data1 k = default)
```

This means that we have to prove that an array with elements of type `t`, such that there is an integer value `size1` between 0 and its length, both inclusive, as well as, every element from index `size1` to the end of the array is equal to default. That much is obvious, from what we have been discussing until now. So, how can we prove this condition. Well, we could approach this problem by using Why3's internal tactics, however, this is considered interactive verification, which is not our objective. To maintain full automation, we must provide some sort of logical element that guarantees this condition, or a logical equivalent, beforehand, for instance, a lemma:

GOSPEL + OCaml

```
let[@lemma] witness () = (* TODO *)
(*@ res = witness ()
    ensures exists d:elt, s:int.
      0 <= s <= Array.length res /\
      (forall k: int. s <= k < Array.length res -> res.(k) = d) *)
```

In fact, we can actually simplify the aforementioned logical condition by instanciating a concrete array, named `res`. This allows to remove one of the existially quantified varaibles, and its most problematic. The remaining two are simple enough. Now, we only have to provide an instance of an array that complies with this conditions. And, there is not better candidate than the empty array:

GOSPEL + OCaml
```
module ResizeableArray (P: PolymorphicType) : RArray = struct
  type elt = P.t

  let[@lemma] witness () = Array.make 0 P.default
  (*@ ... *)

  type rarray = (* ... *)
  (*@ ... *)

  (* Remaining functions are to be presented next *)
end
```

In this listing we provide the expected structure for the witness to be in scope in order to prove the type invariant. To creant an empty array, using `Array.make`, one needs to provide a default value. This value is obtained from module `P`, which adheres to the `PolymorphicType` signature.

Due to their simplicity, the `length`, `get` and `set` functions have the same exact specification from the signature. Due to this, we shall skip these functions. Regarding `make`, we can have slightly stricter post-conditions:

GOSPEL + OCaml
```
let make n d = (* ... *)
(*@ res = make n d
    requires n >= 0
    ensures res.default = d
    ensures res.size = n
    ensures forall k. 0 <= k < Array.length res.data ->
      res.data.(k) = d *)
```

We can also state that the `default` field of the record is equal to the value received from the parameters, `d`, and that every element of `data` is filled with said element. Similar to the previous functions, `resize`'s (outer) annotation is exactly the same as the one found in the signature. However, we still need to provide a loop invariant to the loop found in its body:

GOSPEL + OCaml
```
for i = s to a.size - 1 do
(*@ invariant forall k. 0 <= k < s -> a.data.(k) = (old a.data).(k)
    invariant forall k. s <= k < i -> a.data.(k) = a.default
    invariant forall k. a.size <= k < Array.length a.data ->
      a.data.(k) = a.default *)
  a.data.(i) <- a.default
done;
```

Once more, when shrinking, must guarantee that all elements between the new and old virtual sizes are deallocated (*i.e.* replaced with the default value). The elements before the index of value `s` remain unchanged. Similarly, elements from the index of value `a.size` (inclusive) also remain unchanged, however, we can provide a stronger invariant. These positions of the array have already been filled the default so, we we can exactly this. With this only the indices between the two sizes are left. We can simply ignore whatever is place from loop variable (`i`) forward, since its conents will, eventually, be replaced with the default value. Therefore, from `s` (inclusive) to `i` (exclusive), the corresponding memory blocks are filled with `a.default`.

5.6 Persistent Queue

Queues are very versatile abstract data types, in the sense that these can be implement in several distinct ways. Some implmentations may fall on the purely function style, while others may make use of mutability. In this section, we shall discuss a purely functional queue with amortized operations, to ensure some level of time complexity optimization. However, and prior to that, we shall present a certified signature of the operations that are implemented:

OCaml

```
module type Queue = sig
  type 'a queue

  exception Empty

  val empty : unit -> 'a queue
  val length : 'a queue -> int
  val is_empty : 'a queue -> bool
  val enqueue : 'a -> 'a queue -> 'a queue
  val head : 'a queue -> 'a
  val dequeue : 'a queue -> 'a * 'a queue
  val transfer : 'a queue -> 'a queue -> 'a queue

end
```

Since we will be implementing a purely functional queue, we are not limited by size constraints. In this case study, we are concerned with basic operations, such as creating an empty queue, checking its length or if it is empty, adding an element, checking the first element, removing element, and combining two queues. Now, we must annotate these functions:

GOSPEL + OCaml

```
type 'a queue
(*@ model view: 'a list *)

(*@ function length (q: 'a queue) : int *)
```

We can model our queue type as a polymorphic list, as queues from a logical perspective are relatively simple, and can easily be seen as a list of elements. Additionally, we define an auxiliary function to calculate the length of a queue, which can be used as follows:

GOSPEL + OCaml

```
val empty : unit -> 'a queue
```

```
(*@ res = empty ()
    ensures length res = 0 *)

val length : 'a queue -> int
(*@ res = length q
    ensures res = length q *)

val is_empty : 'a queue -> bool
(*@ res = is_empty q
    ensures res <-> length q = 0 *)
```

Note that despite using the same name, for both the concrete and logical functions, these do not overlap nor cause conflicts. The specifications on these are quite intuitive, an empty queue must not have elements, `is_empty` tests if the que has not elements, and both versions of `length` must have the same value for a given queue. Similarly:

GOSPEL + OCaml

```
val enqueue : 'a -> 'a queue -> 'a queue
(*@ res = enqueue x q
    ensures length res = 1 + length q
    ensures res.view = Sequence.append q.view (Sequence.singleton x) *)

val head : 'a queue -> 'a
(*@ res = head q
    raises Empty -> length q = 0
    ensures res = q.view[0] *)
```

The `enqueue` function inserts an element at the end of the queue, as expected, since it follows a first in first out policy. As such, the resulting queue must have one more element than the original, and its view is updated accordingly. Meanwhile, the `head` operation assumes the existance of at least one element, and returns the head element of the queue, without removing it. If this function is called on a queue that does not have elements, then it raises the `Empty` exception.

Removing an element in a queue, modeled as a list, amounts to returning its tail, Additionally, we also return its head, after performing the `dequeue` operation. Once more, this function assumes the presence of at least one element, otherwise it raises the `Empty` exception:

GOSPEL + OCaml

```
val dequeue : 'a queue -> 'a * 'a queue
(*@ (v, res) = dequeue q
    raises Empty -> length q = 0
    ensures v = q.view[0]
    ensures res.view = q.view[1 ..] *)
```

The post-conditions serve to confirm the values of the resulting pair, namely that `v` is the head of the original queue, and `res` is the the subsequence starting from 1 until the end of the view, or, in other words, its tail.

The last operation in the `Queue` signature is `transfer`, which joins two queues, to form a bigger queue. The elements of the first place are placed, hence the following post-conditions:

GOSPEL + OCaml
```
val transfer : 'a queue -> 'a queue -> 'a queue
(*@ res = transfer q1 q2
    ensures length res = length q1 + length q2
    ensures res.view = Sequence.append q2.view q1.view *)
```

To achieve an amortized implementation, a simple list does not suffice to implemented a queue. Instead, we can use two lists, the first holds the elements at the front of the queue, for quick insertion, and the second holds the elements at the back of the queue, in reverse order, so that any elements can easily be added to the end of the queue, with $\Theta(1)$ time complexity:

GOSPEL + OCaml
```
module PersistentQueue : Queue = struct

  exception Empty

  type 'a queue = {
    front : 'a list;
    rear : 'a list;
    size : int;
    view : 'a list; [@ghost]
  }
  (*@ invariant size = Sequence.length view
      invariant Sequence.(==) view (Sequence.append front (Sequence.rev rear))
      invariant front = Sequence.empty -> rear = Sequence.empty *)

  (*@ function length (q: 'a queue) : int =
        q.size *)

  (* To be completed *)

end
```

The amortized aspect of the queue, which will be discussed in depth soon, is also taken into consideration in the type invariants. If the `front` list is empty, then `rear` is also empty. This means that if the queue contains any elements, then at least one is in the `front` list, to ensures $\Theta(1)$ removal in most scenarios. Otherwise, the queue is empty. Additionally, we also included a `size` field, for quick access. This field is used to calculate the logical `length` of a queue. An empty queue is initialized as such:

GOSPEL + OCaml
```
let empty () = { front = []; rear = []; size = 0; view = [] }
(*@ res = empty ()
    ensures res.view = Sequence.empty *)

let length q = q.size
```

```
(*@ res = length q
    ensures res = length q *)

let is_empty q = q.size = 0
(*@ res = is_empty q
    ensures res <-> q.view = Sequence.empty *)
```

All three list fields, `front`, `rear` and `view` are initialized with the empty list, whereas `size` is initialized with zero, as expected. The `length` function returns the `size` field, whereas, `is_empty` checks if the value of that field is 0. Moving on to the `enqueue` function:

GOSPEL + OCaml

```
let enqueue x q =
  if is_empty q then
    { q with front = [x]; size = q.size + 1; view = [x] }
  else
    { q with rear = x :: q.rear; size = q.size + 1; view = q.view @ [x] }
(*@ res = enqueue x q
    ensures res.view = Sequence.append q.view (Sequence.singleton x) *)
```

The behaviour of this function changes whether the queue is empty. If so, the we replace the value of `front`, which should be empty, with `[x]`, where `x` is the value to insert. This is done to respect the invariant that states that a queue with any elements must have at least one in the `front` list. In this scenario, we can also directly update the view to `[x]`. Otherwise, we place `x` at the head of the `rear` queue. Remember that it is reversed. Meanwhile, we concatenate `[x]` at the end of `view`. In both scenarios, the size is incremented by 1. The post-condition is akin as to how we update the `field`, with the small difference that it uses the equivalent operations for the sequence logical type, rather than lists.

The `head` function returns the first element of the queue, which corresponds to the element placed at the head of the `front` list. This function assumes that there is at least one element, otherwise it raises `Empty`:

GOSPEL + OCaml

```
let head q =
  match q.front with
  | [] -> raise Empty
  | h::t -> h
(*@ res = head q
    raises Empty -> length q = 0
    ensures res = q.view[0] *)
```

Specification-wise, it is the same as the signature, and overall, quite intuitive. Before discussing the `dequeue` function, we present an auxiliary function and a lemma. Starting with the `tail` function. to obtain the tail of a list:

GOSPEL + OCaml

```
let tail = function
  | [] -> assert false
  | _ :: l -> l
```

```
(*@ res = tail param
    requires param <> Sequence.empty
    ensures Sequence.(==) res param[1 ..]
    ensures Sequence.length res = Sequence.length param - 1 *)
```

This function has the prerequisite of the list not being empty. Furthermore, it ensures that its result represents the same sequence of elements, without the first element. Whereas, the lemma reminds the SMT solvers that the `List.rev` and `Sequence.rev` operations are equivalent:

GOSPEL + OCaml

```
let[@lemma] rec comm_rev_of_list (l: 'a list) =
  match l with
  | [] -> ()
  | _ :: r -> comm_rev_of_list r
(*@ ensures Sequence.(==) (of_list (List.rev l)) (Sequence.rev (of_list l)) *)
```

This lemma is proven by induction, with a lemma function. The base case is obvious, since `List.rev []` is the same as `Sequence.rev []`, hence we return unit, *i.e.* `()`. The recursive case simply makes the call to its tail, without any additional steps. Note that `of_list` is an auxiliary function from GOSPEL's standard library to convert a list to a sequence, although in most situations it can be done automatically.

The amortization process occurs within the `dequeue` function. In total, there are three plausible scenarios: (1) the queue is empty, which promptly raises the `Empty` exception; (2) if there is only an element in the `front` list; and (3) if there are multiple elements in the `front` list. There is an additional fourth case, which is absurd given one of the type invariants, however, the OCaml compiler is not aware of such conditions, and prefers to have a pattern matching that covers all possible cases. This is the scenario where the `front` list is empty:

GOSPEL + OCaml

```
let dequeue q =
  match q.front, q.rear with
  | [], [] -> raise Empty
  | [x], r ->
    x, { front = List.rev r; rear = []; size = q.size-1;
          view = tail q.view }
  | x::ft, r ->
    x, { front = ft; rear = r;
          size = q.size-1; view = tail q.view }
  | [], _ -> assert false
(*@ (v, res) = dequeue q
    raises Empty -> length q = 0
    ensures Sequence.(==) q.view (Sequence.cons v res.view) *)
```

The amortization process happens on scenario (2), *i.e.* when we remove the only element in the `front` list. To respect the type invariant, we must populate it once more. Since we cannot readily access, with good time complexity, the new first element of the queue, which would be the last element of `rear`, since it is ordered backwards. Instead of performing a $\Theta(n)$ operation to just fetch it, we can update the `front` field by reversing the `rear` list, for the sime time complexity, and avoid having to repeatedly traverse `rear`, while it has elements, every time a `dequeue` operation is called, since `enqueue` only places elements in `front` if the queue is empty. By updating `front` with `List.rev r`, then it means that `rear` is noew empty. Meanwhile, size is decremented by one, and the `view` field is updated with its own tail. In scenario (3) there is not amortization, we only remove the head of the `front` field, and perform the same updates to `size` and `view`, as in the previous case, while [rear] stays the same.

Finally, we present the transfer implementation:

GOSPEL + OCaml

```
let rec transfer q1 q2 =
  try
    let (h, t) = dequeue q1 in
    transfer t (enqueue h q2)
  with Empty -> q2
(*@ res = transfer q1 q2
    variant Sequence.length q1.view
    ensures res.view = Sequence.append q2.view q1.view *)
```

This function makes use of both `enqueue` and `dequeue` to transfer the elements of one queue to the other. Once the `Empty` exception is raised, after an unsuccessful dequeue, due to having no elements, the recursion is stopped, and the function returns the new queue with all elements. The elements of the first queue are successively transfered to the second queue, in order.

5.7 Circular Queue

In this section, we present a totaly contrasting queue implmentation, as to the one presented just before. A circular queue has a fixed size, and is typically implemeted using arrays. With this example, we want to show that two opposite implementations can be proven by refinement using the same certified signature. Well, in reality there are some slight differences on the OCaml level due to type checking, and concrete details such as limited space. The latter could easily be solve, either by also limiting the size of functional queues or automatically resize circular queues. Neither of which are totaly natural, hence we have refrained from either option, and prefered to present a slightly modified signature. Although, from a logical perspective they are nearly the same:

GOSPEL + OCaml

```
module type PolymorphicType = sig
  type t
  val default : t
end

module type Queue = sig
  type elt
  type queue
  exception Empty
  val create : int -> elt -> queue
  val length : queue -> int
  val is_empty : queue -> bool
  val is_full : queue -> bool
  val clear : queue -> unit
  val enqueue : elt -> queue -> unit
  val head : queue -> elt
  val dequeue : queue -> elt
end
```

Additionally, we have removed the `transfer` operation also due to potentially neededing to resize, as well as, simulating polymorphism due to technical difficulties that will be explained later. Despite these major changes, on the implementation level, the logical definitions remain mostly similar:

GOSPEL + OCaml
```
type queue
(*@ mutable model view: elt list *)

(*@ function length (q: queue) : int *)

(*@ predicate full (q: queue) *)
```

We still use a list as a logical model for our `queue` type, which is also complemented with a new predicate to indicate when it is full. Rather than having a function that returns the empty queue, we now have a function that creates a circular queue of fixed such, with that size being strictly positive, since there is no possibility to resize:

GOSPEL + OCaml
```
val create : int -> elt -> queue
(*@ res = create n d
    requires n > 0
    ensures length res = 0 *)

val length : queue -> int
(*@ res = length q
    ensures res = length q *)
```

The `length` function does not change. Similar to `is_empty`:

GOSPEL + OCaml
```
val is_empty : queue -> bool
(*@ res = is_empty q
    ensures res <-> length q = 0 *)

val is_full : queue -> bool
(*@ res = is_full q
    ensures res <-> full q *)
```

Morover, we also included a function to test if the queue is full, namely `is_full`, whose result is equivalent to the aforementioned `full` predicate, and a function to clear the contents of the queue:

```
val clear : queue -> unit                          GOSPEL + OCaml
(*@ clear q
    modifies q
    ensures length q = 0 *)
```

Unlike the previous implementation, this version needs to account for being full at the time of inserting an element, which is restricted with a pre-condition:

```
val enqueue : elt -> queue -> unit                 GOSPEL + OCaml
(*@ enqueue x q
    requires not full q
    modifies q
    ensures length q = 1 + old (length q)
    ensures q.view = Sequence.append (old q.view) (Sequence.singleton x) *)
```

Furthermore, we now have to use the `old` keyword in the `enqueue` and `dequeue` operations, since we are mutating our data structure. That is not the case for the `head` function, since we are only reading the first element of the queue:

```
val head : queue -> elt                            GOSPEL + OCaml
(*@ res = head q
    raises Empty -> length q = 0
    ensures res = q.view[0] *)

val dequeue : queue -> elt
(*@ v = dequeue q
    raises Empty -> length q = 0
    modifies q
    ensures v = old q.view[0]
    ensures q.view = old q.view[1 ..] *)
```

There are two main ways to implement a circular queue. The first is to store the indices of the first and last elements. The second is to store the index of the first element and its length. Both solutions are plausible, and are equally efficient in terms of time and space complexities. Note that one may even combine both solutions. For this case study, we have used the second option:

```
module CircularQueue (P: PolymorphicType) : Queue = struct     GOSPEL + OCaml
  type elt = P.t

  type queue = {
    data: elt array;
    mutable first: int;
    mutable len: int;
    mutable view: elt list; [@ghost]
  }
  (*@ invariant 0 <= first < Array.length data
      invariant 0 <= len <= Array.length data
      invariant len = List.length view
      invariant let size = Array.length data in
        forall i. 0 <= i < len ->
        (first + i < size -> view[i] = data[first + i]) /\
        (0 <= first + i - size -> view[i] = data[first + i - size]) *)

    (* To be continued... *)

end
```

Our type `queue` is formed by an array of elements, a pointer to the index of the first, a pointer to the length of the queue, and a ghost field, named `view`, that establishes our logical model of the queue as a list. In terms of type invariants, there are a few interesting properties that we need to account for. The `first` field must be a valid index to avoid out of bounds accesses, *i.e.* it must be a value between 0, inclusive, and the length of the array (`data`), exclusive. Similarly, `len` may not exceed the length of the array or be a negative number. Moreover, `len` must be equal to the length of `view`, to ensure that we are storing the elements correctly. The final invariant is also its strongest. It serves the purpose of establishing a relation between the elements of `data` and `view`. Namely, we have to state that `data[first]` corresponds to `view[0]` and so on. However, due to the circular property of this queue implementation, the elements in `data` are not necessarily placed contiguously. For instance, the first few elements of the queue may be placed near the end of the array, while the remaining are placed from index 0 onwards. To establish the logical correspondence between the elements of `data` and `view`, one solution is to find the i^{th} element of the queue, starting with the first. To achieve this, we use an offset based on the length of the queue, *i.e.* that ranges from 0, inclusive, to `len`, exclusive. The i^{th} element of the queue is obtained from summing the offset to the index of the first element. However, this may exceed the length of the `data` array, hence we have to check for its length. If `first + i < size`, where `i` is the offset and `size` is the length of the array, then `view[i] = data[first + i]`. If `first + i >= size`, we can subtract size to “loop” around the array. So, `view[i] = data[first + i - size]`, assuming that `first + i - size`, to avoid both out of bounds accesses and ensure that the two cases are disjoint.

For instance, consider a queue, with a capacity of 5 elements, a length of 4, and with first index equal to 3. This means that the first element of the queue is in index 3, the second in index 4, the third in index 0, and the final element is in index 2:

Forall $i \in \{0 \ldots \texttt{Array.length data} - 1\}$, then:

$i = 0 \implies \texttt{first} + i < \texttt{size} \equiv 3 + 0 < 5$, then $\texttt{view}[0] = \texttt{data}[3]$

$i = 1 \implies \texttt{first} + i < \texttt{size} \equiv 3 + 1 < 5$, then $\texttt{view}[1] = \texttt{data}[4]$

$i = 2 \implies \texttt{first} + i >= \texttt{size} \equiv 0 < 3 + 2 - 5$, then $\texttt{view}[2] = \texttt{data}[0]$

$i = 3 \implies \texttt{first} + i >= \texttt{size} \equiv 0 < 3 + 3 - 5$, then $\texttt{view}[3] = \texttt{data}[1]$

Unfortunately, these invariants are quite strong and need a witness. Once more, in Cameleer and GOSPEL, finding witnesses is not as simple as in WhyML, and the one we have found is, honestly, too complex and in our opinion does not justify the effort of manually providing witnesses. As such, we will refrain from displaying and explaining it here. Although, it can still be seen in the gallery found inside the textbook's website. Furthermore, due to technical questions related to the witness, we had to use simulated polymorphism, rather than natural polymorphism using OCaml's innate generic types.

The `length` function can be obtained from the `len` field of a queue:

GOSPEL + OCaml

```
(*@ function length (q: queue) : int =
      q.len *)

(*@ predicate full (q: queue) =
      length q = Array.length q.data *)
```

Whereas, the predicate `full` is obtained from checking if the length of the queue is equal to the length of the `data` array, meaning that are no more free memory blocks available. Similarly, the `length` and `is_full` functions are defined as follows:

GOSPEL + OCaml

```
let length q = q.len
(*@ res = length q
    ensures res = length q *)

let is_empty q = length q = 0
(*@ res = is_empty q
    ensures res <-> length q = 0 *)

let is_full q = length q = Array.length q.data
(*@ res = is_full q
    ensures res <-> full q *)
```

From a verification perspective, we can simply establish a relation between the result of these functions with their logical counterparts. Analogously, `is_empty` compares the length of the queue to 0, as one might expect.

An empty queue can be created using a positive integer, since there is not resizing capability, and a default value:

GOSPEL + OCaml

```
let create n d = {
  data = Array.make n d;
  first = 0;
  len = 0;
  view = []
}
(*@ res = create n d
    requires n > 0
    ensures Array.length res.data = n
    ensures res.view = [] *)
```

As post-conditions, we must ensure that the array has the desired capacity, and that `view` is empty. Note that we do not need to specify anything about the elements of this array, since this implementation is not concerned with the contents of memory blocks that are not occupied by the queue.

Clearing a circular queue amounts to reseting its length to zero, and updating the `view` to be the empty list. Additionally, we may also reset the pointer to the first index, although this remains optional:

GOSPEL + OCaml

```
let clear q =
  q.first <- 0;
  q.len <- 0;
  q.view <- []
(*@ clear q
    modifies q.first, q.len, q.view
    ensures length q = 0
    ensures q.view = [] *)
```

Verification-wise, we must ensure, as post-conditions, that the affected fields hold indeed the mandatory values.

GOSPEL + OCaml

```
let enqueue x q =
  q.view <- q.view @ [x];
  let i = q.first + q.len in
  let n = Array.length q.data in
  q.data.(if i >= n then i - n else i) <- x;
  q.len <- q.len + 1
(*@ enqueue x q
    requires not full q
    modifies q.data, q.len, q.view
    ensures length q = 1 + old (length q)
    ensures q.view = (old q.view) @ (Cons x []) *)
```

Queues follow the first in first out policy, which means that new elements are inserted at the back of the queue, as it can be seen when we update `view`. We can use the same strategy, as found in the type invariants, to calculate the index of the next element, assuming that the queue is not full. By summing `len` to `first`, we obtain the index of next element, although it may fall outside of the range of the array, in which case we have to subtract the length of the array. Note that in GOSPEL `[x]` produces an error, hence the use of `Cons x []`.

Accessing the head of the queue, assuming that there is at least one element, is quite easy given that we are using arrays and have a pointer to that element:

GOSPEL + OCaml
```
let head q =
  if is_empty q then raise Empty
  else q.data.(q.first)
(*@ res = head q
    raises Empty -> length q = 0
    ensures res = q.view[0] *)

let tail = function
  | [] -> assert false
  | _ :: l -> l
(*@ res = tail param
    requires param <> Sequence.empty
    ensures Sequence.(==) res param[1 ..]
    ensures Sequence.length res = Sequence.length param - 1 *)
```

Furthemore, we also reutilize the `tail` function from before in this `dequeue` implementation:

OCaml
```
let dequeue q =
  begin if length q = 0 then raise Empty end;
  q.view <- tail q.view;
  let h = q.data.(q.first) and n = Array.length q.data in
  q.len <- q.len - 1;
  q.first <- q.first + 1;
  if q.first = n then q.first <- 0;
  h
```

Naturally, this operation can only be performed if there is at least one element in the queue. In such cases, we update the `view` field with the tail of its previous content. At the end of this function we return the value of the first element of the queue before its removal. We can safely remove the value from the queue by simply decrementing the queue length by 1, and to increment the `first` field. However, it is important to check if it reaches the value of length of the array. In which case we can reset it to 0, to avoid out of bounds accesses. The `dequeue` function can be annotated as follows:

GOSPEL + OCaml
```
let dequeue q = (* ... *)
(*@ res = dequeue q
    raises Empty -> length q = 0
    modifies q.first, q.len, q.view
    ensures length q = (old (length q)) - 1
    ensures match old q.view with
      | [] -> false
      | Cons h t -> res = h /\ q.view = t *)
```

We must ensure that the length of the queue decrements by one, that the result is the head of the old `view`, and that the new `view` is the tail of the old `view`. In case the old `view` was emtpy we assert false, since that case is already excluded by the pre-condition.

5.8 Binary Search Tree

Binary search trees are ordered versions of standard binary trees. To make this work, one must establish that the left child is smaller than its parent, while the right child is greater. Some implementations may allow duplicates and perform self-balancing, but in our case, we shall simplify and not allow duplicates nor perform self-balancing. For this data structure, we will verify and implement this OCaml signature:

OCaml

```
module type BST = sig
  type elt
  type t

  val empty : unit -> t
  val insert : elt -> t -> t
  val mem : elt -> t -> bool
  val min_elt : t -> elt option
  val remove_min : t -> elt option * t
  val remove : elt -> t -> t

end
```

Operations in this signature include creating an empty binary search tree, inserting an element, checking the membership of an element, finding the smallest element, removing the smallest element, and removing any given element. Additionally, we may also define an auxiliary function for comparison of two elements, which will be useful later:

GOSPEL + OCaml

```
val cmp : elt -> elt -> int [@@logic]
(*@ axiom is_pre_order: is_pre_order cmp *)
```

The `cmp` function is equipped with an axiom stating that it is a preorder relation, in other words, it is reflexive and transitive. Note that the first `is_pre_order` represents the name of this axiom, while the second is a predicate from Cameleer's standard library. Other logical auxiliary definitions include:

GOSPEL

```
(*@ function occ (x: elt) (t: t) : int *)

(*@ predicate empty (t: t) *)

(*@ predicate belongs (x: elt) (t: t) *)

(*@ predicate bst (t: t) *)
```

A function to determine the occurrences of a given number in a binary tree, a predicate that checks if a binary tree is empty, a predicate that indicates if an element is a member of a binary tree, and, lastly, if a binary tree complies with the binary search tree definition. These will be used in the annotations of the data structure operations, such as:

GOSPEL + OCaml

```
val empty : unit -> t
(*@ r = empty ()
    ensures empty r
    ensures bst r *)

val mem : elt -> t -> bool
(*@ r = mem x t
    requires bst t
    ensures r <-> belongs x t *)
```

The `empty` constructor must return the empty tree, which is bst, by definition. The `mem` test must be logically equivalent to the `belongs` predicate.

The `insert` function expects a binary search tree as input and returns a binary search tree:

GOSPEL + OCaml

```
val insert : elt -> t -> t
(*@ r = insert x t
    requires bst t
    ensures forall y. y <> x -> occ y r = occ y t
    ensures belongs x r
    ensures bst r *)
```

The output binary search tree is guaranteed to have the element that was meant to be inserted. While the number of occurrences of every other element stays the same.

The result of `min_elt` is wrapped around an option type. This is an alternate possibility, instead of stating, as a pre-condition, that the binary search tree must not be empty:

GOSPEL + OCaml

```
val min_elt : t -> elt option [@@logic]
(*@ r = min_elt t
    requires bst t
    ensures match r with
      | None -> empty t
      | Some v -> belongs v t /\
        forall x: elt. belongs x t /\ x <> v -> cmp v x < 0 *)
```

In case the result is a `None` value, then it means that `t` is empty. Otherwise, `v` must be an element of t, such that it is smaller than every other element that belongs in `t`. We have also marked this function with the `[@@logic]` tag, as it can be used to verify `remove_min`:

GOSPEL + OCaml

```
val remove_min : t -> elt option * t
(*@ o, res = remove_min t
    requires bst t
    ensures bst res
    ensures o = min_elt t
    ensures match o with
      | None -> empty t
      | Some min ->
        not belongs min res /\
        (forall x. x <> min -> occ x res = occ x t) *)
```

This function returns a pair of elements, the first is an option type containing the minimum value, and the second is the resulting binary search tree. The value of the option type must be exactly equal to `min_elt t`, where `t` is the input tree. Moreover, we need to ensure that not other element is removed, using the number of occurrences. This principle also applies to the generic remove function:

GOSPEL + OCaml

```
val remove : elt -> t -> t
(*@ r = remove x t
    requires bst t
    ensures bst r
    ensures occ x r = occ x t || occ x r = occ x t - 1
    ensures forall y: elt. y <> x -> occ y r = occ y t *)
```

Once more, the final result must be a binary search tree that does not contain the element to be removed. With this specification, we allow input trees that may not have the element to be removed, thus the number of occurrences of that element is such scenarios remains the same.

Moving on to the implementation, to simulate polymorphism, we need a type equipped with a comparison function that respects the preorder predicate:

GOSPEL + OCaml

```
module type PreOrderType = sig
  type t

  val cmp : t -> t -> int [@@logic]
  (*@ axiom is_pre_order: is_pre_order cmp *)
end

module BSTImpl (P : PreOrderType) : BST = struct

  type elt = P.t
  type t = E | N of t * elt * t

  let[@logic] cmp e1 e2 = P.cmp e1 e2

  (*@ predicate empty (t: t) = t = E *)

  (* To be continued soon... *)
```

This type is inside the `PreOrderType` signature, which in turn is used as the parameter of a functorized module named `BSTImpl`. This module implements the certified signature, `BST`, that we have presented above. We define type `elt` as the type inside `PreOrderType`, and define a shortcut for the operation inside that signature, to respect `BST`, which requires the inclusion of a logical function, named `cmp`, equipped with preorder property. Furthemore, we also define the binary tree type, named `t`, and a predicate that determine if a tree is empty.

One way to define the `belongs` predicate is through the number of occurrences, which should be strictly positive:

GOSPEL + OCaml

```
(*@ function rec occ (x: elt) (t: t) : int =
      match t with
      | E -> 0
      | N l v r -> occ x l + occ x r + (if cmp x v = 0 then 1 else 0) *)
(*@ ensures 0 <= result *)

(*@ predicate belongs (x: elt) (t: t) = occ x t > 0 *)

(*@ predicate unique (x: elt) (t: t) = occ x t = 1 *)
```

Furthemore, we introduce the concept of `unique` element, meaning that there is only one instance of it in a tree. This concept is rather important since our implementation of a binary search tree does not allow duplicates:

GOSPEL + OCaml

```
(*@ predicate bst (t: t) = match t with
      | E -> true
      | N l v r ->
        (forall lv. belongs lv l -> cmp lv v < 0) &&
        (forall rv. belongs rv r -> cmp rv v > 0) &&
        bst l && bst r *)
```

The `bst` predicate is defined as every element to the left of a node being strictly smaller than it, whereas the elements to its right are strictly greater. This definition is recursively applied to every subtree.

An empty tree is represented by the `E` constructor of type `t`:

GOSPEL + OCaml

```
let empty () = E
(*@ r = empty ()
    ensures empty r
    ensures bst r *)

let rec mem x = function
  | E -> false
  | N (l, v, r) ->
    let c = cmp x v in
    c = 0 || mem x (if c < 0 then l else r)
(*@ r = mem x t
    requires bst t
    variant t
    ensures r <-> belongs x t *)
```

Given that binary search trees are sorted, we can use that to our advantage when traversing the tree. For instance, when checking for the membership of an element. If we are given an non-empty node, then we can compare the value of that node to our desired value. If they are the same the function terminates, otherwise, we can go left if the desired value is smaller than the current node or right, if greater. This ideia can also be applied when inserting a value:

GOSPEL + OCaml
```
let rec insert x = function
  | E -> N (E, x, E)
  | N (l, y, r) ->
    if cmp x y = 0 then N (l, y, r)
    else if cmp x y < 0 then N (insert x l, y, r)
    else N (l, y, insert x r)
(*@ r = insert x t
    requires bst t
    variant t
    ensures forall y. y <> x -> occ y r = occ y t
    ensures unique x r
    ensures bst r *)
```

If the element to be inserted differs from the value of the current node, we make a recursive call left or right, accordingly. However, if the values match, the we simply return the current node and do not insert the value, to avoid duplicates. Hence, we use as a post-condition the `unique` predicate, which implicitly denotes membership, and complies with the proof by refinement.

Once more, when traversing a binary search tree, we can make use of its properties to achieve the best time complexity. The minimum element is in its leftmost node. Thus, if we receive a nont-empty binary search tree, the recursion will stop when the left subtree is empty:

GOSPEL + OCaml
```
let [@logic] rec min_elt = function
  | E -> None
  | N (E, v, __) -> Some v
  | N (l, _, _) -> min_elt l
(*@ r = min_elt t
    variant t
    requires bst t
    ensures match r with
      | None -> empty t
      | Some v -> unique v t /\
        forall x: elt. belongs x t /\
        x <> v -> cmp v x < 0 *)
```

We make use of the `unique` predicate, once again, to implicitly denote its membership. Moreover, we need a stronger predicate here to verify another function that use `min_elt` indirectly, through the `remove_min` function:

GOSPEL + OCaml

```
let rec remove_min = function
  | E -> None, E
  | N (E, v, r) -> Some v, r
  | N (l, v, r) ->
    let m, l' = remove_min l in m, N (l', v, r)
(*@ (o, res) = remove_min t
    variant t
    requires bst t
    ensures bst res
    ensures o = min_elt t
    ensures match o with
      | None -> empty t
      | Some min -> not belongs min res /\
        (forall x. x <> min -> occ x res = occ x t) *)
```

This function uses similar patterns to `min_elt`, where the two base cases are if the tree itself is emtpy, or the left subtree is empty. The latter returns a pair of the minimum value, and the right subtree. Alternatively, the recursive case is obtained by first making a recursive call to the left subtree to obtain the minimum value and the left subtree without that element. These values are then used in the returning value, the minimum value is one element of the pair, and the other is an updated node with the new left subtree.

Removing the minimum element is simple, since it never has a left child. This means that its direct right child can take its spot in the new tree. The same cannot be said about removing any arbitrary element:

GOSPEL + OCaml

```
let rec remove x = function
  | E -> E
  | N (l, v, r) ->
    let c = cmp x v in
    if c = 0 then certified_merge l r
    else if c < 0 then N (remove x l, v, r)
    else N (l, v, remove x r)
(*@ r = remove x t
    variant t
    requires bst t
    ensures bst r
    ensures occ x r = occ x t || occ x r = occ x t - 1
    ensures forall y: elt. y <> x -> occ y r = occ y t *)
```

We need to safely merge the two subtrees, using `certified_merge`, presented below. Note that it is also possible that the element that we are trying to remove is not a member of the binary search tree, thus the number of occurrences of `x` may stay the same. One possible way to implement `certified_merge` is as follows:

GOSPEL + OCaml

```
let certified_merge t1 t2 =
match t1, t2 with
| E, t | t, E -> t
| _ ->
  let (Some m2, t2') = remove_min t2 in N (t1, m2, t2')
  (*@ r = certified_merge t1 t2
      requires bst t1 /\ bst t2
      requires forall x y: elt. belongs x t1 /\ belongs y t2 -> cmp x y < 0
      ensures bst r
      ensures forall x: elt. belongs x t1 || belongs x t2 <-> belongs x r
      ensures forall x: elt. occ x r = occ x t1 + occ x t2 *)
```

Given two binary search trees, such that the elements of one are strictly smaller than the elements of the other, then we can remove the minimum element of the second, and connect the two trees, since this minimum element is greater than every element in the first tree, but smaller than every other element in the second. This function is quite appropriate when removing a node with two subtrees. In this scenario, the elements of the left subtree are guaranteed to be smaller than the ones in the right subtree, since we are dealing with binary search trees. Hence, by choosing the smallest element of the right subtree, we can safely reconnect these two subtrees, after removing their original root.

5.9 Linked List

Coming soon

5.10 Exercises

Implement and specify the:

1. Queue, implemented as a list
2. Stack, implemented as a list
3. Set, implemented as a list. Operations: addition, membership test, and check if a set is a subset of another.

Chapter 6

Selected Topics

This chapter will be available soon. For now, we leave readers with a few suggestions on what to do next in the form of exercises.

6.1 Advanced Arithmetic

Given the introductory nature of chapter 2, many advanced mathematical problems do not fit there. In this chapter, we briefly introduce a few more arithmetic algorithms, with increased difficulty, namely Delannoy numbers.

6.1.1 Delannoy Numbers

The Delannoy number $D(n, m)$ is the number of possible paths in the integer grid $\mathbb{N}^2$ that go from the origin $(0, 0)$ to the point (n, m) by using any combination of elementary steps that are vertical (adding $(0, 1)$), diagonal (adding $(1, 1)$) or horizontal (adding $(1, 0)$).

By a simple case analysis on the last step, we see that a path to (n, m) is either a path to $(n-1, m)$ followed by a horizontal step, a path to $(n-1, m-1)$ followed by a diagonal step, or a path to $(n, m-1)$ followed by a vertical step. This leads to the following recursive definition:

$$D(n, m) = \begin{cases} 1, & \text{if } n = 0 \land m \geq 0 \\ 1, & \text{if } n \geq 0 \land m = 0 \\ D(n-1, m) + D(n-1, m-1) + D(n, m-1), & \text{if } n \geq 1 \land m \geq 1 \end{cases}$$

Implementing a recursive function that calculates $D(n, m)$, is far from ideal. Such function has an exponential time complexity due to its three recursive calls. Moreover, these calls frequently overlap each other:

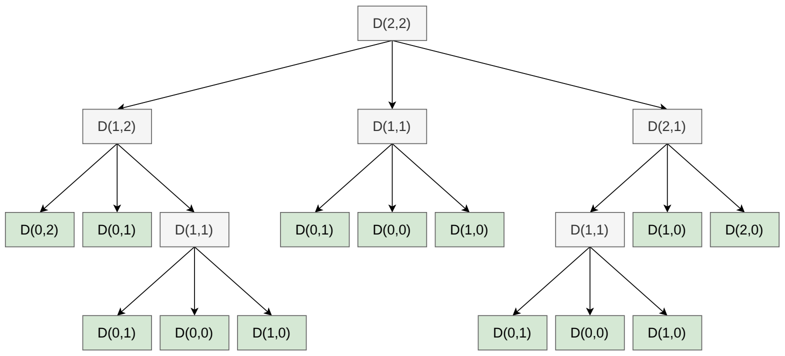


Figure 6.1: Recursive calls for $D(2, 2)$

When $n = 2$ and $m = 2$, we can see, from the figure above, that it needs 18 additional calls to calculate the final result. It, inclusively, has 3 repeated and non-terminating (marked as gray) recursive calls, those being of $D(1, 1)$. For larger numbers of n and m, this can easily lead to stack overflows.

The standard solution to this problem is to use a matrix to store previous Delannoy numbers, which can be quickly accessed to calculate the next number:

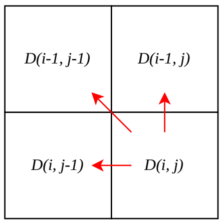


Figure 6.2: Matrix access to calculate $D(i, j)$

It consists in, firstly, creating a matrix with dimensions of $n + 1$ by $m + 1$. Secondly, the first row and first column are filled with 1, since $D(0, ...) = 1$ and $D(..., 0) = 1$. Then, the remainder of the matrix is filled, typically, line by line, and within each line by column. To calculate these cells, we need to access the elements "above" it, to its "left" and in its "upper left diagonal", as seen in the figure above.

This solution is an obvious upgrade performance-wise to the recursive version. However, it can still be optimized using Dynamic Programming to use less space. In particular, we can reduce its space complexity from $\Theta(n * m)$ to $\Theta(m)$. If one notices closely, to calculate the values of the current line, only the previous line is necessary. Thus, every other line is unnecessary. The Dynamic Programming version can be implemented as:

OCaml

```
let delannoy_array n m =
  let a = Array.make (m+1) 1 in
  for i = 1 to n do
    let x = ref 1 in
    for j = 1 to m do
      let y = a.(j-1) + a.(j) + !x in
      x := a.(j);
      a.(j) <- y
    done
  done;
  a.(m)
```

Similarly to the matrix-based solution, the array must also contain extra space, this time in the form of a single cell. This memory block, which will be the first, will always have the value of 1, since it corresponds to $D(..., 0)$. Every other memory block will be gradually replaced, from "left" to "right". This means that, during an array traversal, the array will contain, simultaneously, values from the current iteration (the ones already visited), and from the previous one (the ones yet to visit, and the current). From figure 6.2, one can see that, at any given time, we need both $D(i, j-1)$ and $D(i-1, j-1)$, however, only $D(i, j-1)$ is stored in the array in iteration j, for line i. This calls for an additional value to be stored, in the form of an auxiliary reference. Thus, this can be visually represented as:

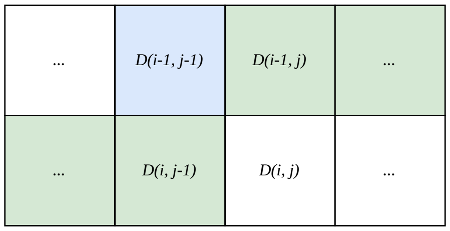


Figure 6.3: Calculating $D(i, j)$ using an array

The cells in green represent the contents of the array, compared to the matrix-based solution, while the blue cell represents the content of the auxiliary reference. Assuming that the reference is named `x`, and the array is named `a`, then, to calculate $D(i, j)$, we have to sum the values of `a.(j-1)` ($D(i, j-1)$), `a.(j)` ($D(i-1, j)$) and `!x` ($D(i-1, j-1)$). This is then followed by updating `x` with the value of `a.(j)` and `a.(j)` with $D(i, j)$, to prepare for the next iteration. By the end of the process, we are guaranteed to have $D(n, m)$ in `a.(m)`. In the matrix-based solution, $D(n, m)$ was the last value to be calculated. This solution is no different in that regard.

Similar to the Fibonacci problem, in 2.4, the advised approach to verify this algorithm is using a logical definition of the Delannoy numbers:

GOSPEL
```
(*@ function rec d (n: int) (m: int) : int =
    if n <= 0 || m <= 0 then 1
    else d (n-1) m + d (n-1) (m-1) + d n (m-1) *)
(*@ variant n + m *)
```

Given the necessity of logical functions being pure, we may use the recursive definition, as expected. To prove the termination of this function, it is important to notice that in a given recursive call at least of the arguments decrements by one. Thus, to encompass the three cases we may sum their values, since it will necessarily decrease in the next iteration. Moving on to the algorithm:

GOSPEL + OCaml
```
let delannoy_array n m = (* ... *)
(*@ res = delannoy_array n m
    requires n >= 0 /\ m >= 0
    ensures res = d n m *)
```

The annotation of the function itself is quite intuitive. We need to state, as pre-conditions, that n and m are non-negative, since we cannot create arrays with negative size and/or indexes. The post-condition is, as expected, that the result must correspond to $D(n, m)$. Starting with the outer loop:

GOSPEL + OCaml
```
for i = 1 to n do
(*@ invariant forall k. 0 <= k <= m -> a[k] = d (i-1) k *)
  (* ... *)
done
```

As previously mentioned, the memory blocks of array will be gradually replaced from the values of the previous line with the current. So, by the end of an iteration, the values in the array correspond to a given line of the matrix. The index of this line is behind by one unit, compared to the current index (i). Hence, the values of k range from 0 to m, to describe the width of a matrix line, while the value of the array in position k corresponds to $D(i-1, k)$. Finally, the inner loop can be annotated, as such:

GOSPEL + OCaml
```
for j = 1 to m do
(*@ invariant !x = d (i-1) (j-1)
    invariant forall k. 0 <= k < j -> a[k] = d i k
    invariant forall k. j <= k <= m -> a[k] = d (i-1) k *)
```

The previously presented invariant is only partially true inside the inner loop, given that we are updating the array. From the current column onward, the values in the array correspond to previous line, *i.e.*, $D(i-1, k)$. Meanwhile, the positions that have already been visited (from 0 to j (exclusive)), thus we use the current line, represented by i. Moreover, we also have to remember the SMT solvers that the reference x hold the value of $D(i-1, j-1)$.

6.1.2 Binomial Coefficients

Pascal's triangle is a triangular arrangement of numbers where each number is the sum of the two numbers directly above it, for instance:

$$\begin{array}{ccccccccccc}
 & & & & & 1 & & & & & \\
 & & & & 1 & & 1 & & & & \\
 & & & 1 & & 2 & & 1 & & & \\
 & & 1 & & 3 & & 3 & & 1 & & \\
 & 1 & & 4 & & 6 & & 4 & & 1 & \\
1 & & 5 & & 10 & & 10 & & 5 & & 1
\end{array}$$

The triangle always starts with the number 1, and the first and last element of each row is 1. We can imagine that on the outside of each row there are omitted zeros:

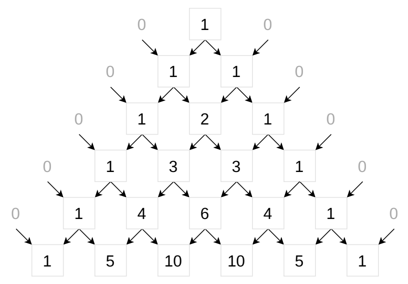


Figure 6.4: Pascal Triangle

The elements of the Pascal's triangle are also known as binomial coefficients. A binomial coefficient is a relation between two integers n and k, such that $n \geq k \geq 0$, and is defined as:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Or, alternatively, as the sum of other binomial coefficients:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Intuitively, n represents the index of the row (note that the first row's index is 0), whereas k represents the column's index. The index of the column ranges from 0 to the index of the row, both inclusively. Thus, the previous relation denotes that $\binom{n}{k}$ is the sum of the two elements directly above it in Pascal's triangle.

Following this definition, we can define a recursive function to calculate $\binom{n}{k}$:

$$\binom{n}{k} = \begin{cases} 1, & \text{if } k = 0 \land n \geq 0 \\ 1, & \text{if } k = n \land n \geq 0 \\ \binom{n-1}{k-1} + \binom{n-1}{k}, & \text{if } 0 < k < n \end{cases}$$

Just from this short mathematical overview, it is possible to come up several possible ways to implement a program to calculate $\binom{n}{k}$. Most of these are flawed. Recursively computing the previous binomial coefficients will generate repeated calls. Meanwhile, directly computation the binomial coefficient using the factorials, even with an efficient implementation, can easily compute numbers that are too large. Another solution is to store the whole triangle. Similar to Delannoy numbers, we only need the previous row. Actually, we only need part of the previous row, up until the desired column. This can be explained from the fact that to calculate $\binom{n}{k}$, we need the values of $\binom{n-1}{k-1}$ $\binom{n-1}{k}$, and never values after k. Thus, although not perfect, this solution can be implemented as such in OCaml:

OCaml

```ocaml
let binom n k =
  let a = Array.make (k+1) 0 in
  a.(0) <- 1;
  for i = 1 to n do
    let j = ref (min i k) in
    while !j > 0 do
      a.(!j) <- a.(!j) + a.(!j-1);
      j := !j - 1;
    done;
  done;
  a.(k)
```

Given that the number of columns increases by one per row, and to avoid reallocating and copying memory every iteration, we should initialize the array with the maximum amount of memory blocks needed. To calculate $\binom{n}{k}$, we only need the values of the columns from 0 to k. Hence, we need an array with size $k+1$. This array is initialized the value 0, to avoid errors. Then, we fill the first position with the value of one. This corresponds to the value of $\binom{0}{0}$, or visually, the top of the triangle. Then, we go row by row, starting from index 1 to index n, and fill the values by column. During each row, to avoid filling incorrect memory blocks or accessing indexes out of bounds, we must calculate the index of the last column accessible in that iteration:

In the first iteration ($n = 1$), we can only access the first and second positions of the array. This restriction is obtained from the minimum value of the current iteration (i), which also represents the index of the maximum column for the corresponding row, and the value of k.

Based on the recursive expression, to calculate $\binom{n}{k}$, we need to know the values of $\binom{n-1}{k-1}$ and $\binom{n-1}{k}$. This means that at the start of an iteration, the array

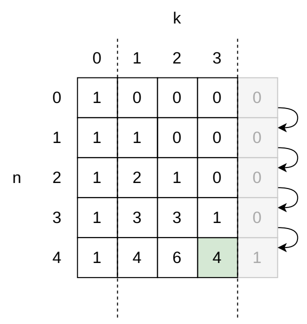


Figure 6.5: Filling the array for $\binom{4}{3}$

will be filled with the values of row $i-1$. So, if we fill the arrays from left to right, we would be overwriting one of values necessary for the next column. Instead, we should fill the array from right to left. This way, there is no dependency between the value being changed and the previous positions. Moreover, in the first few iterations, the algorithm is expected to make use of the zeros placed by omission when making the array. This behaviour is similar to what was previously described, at the start, where we could imagine that there is an extra layer of zeros in each side. In this case, we only consider the right side, since $\binom{n}{0}$ is 1 for each positive value of n, and, thus, we can skip filling index 0 of the array, which will always correspond to this value.

From a verification point of view, the behaviour described in the previous paragraph regarding the zeros is quite challenging. From the formula, we know that $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$, however, by the definition a binomial coefficient is always a positive number, thus the solvers cannot deduct $\binom{n}{k}$ if one of the components is 0. To solve this problem, we can relax the definition of the binomial coefficient to include the exceptional cases:

$$\binom{n}{k} = \begin{cases} 1, & \text{if } k = 0 \wedge n \geq 0 \\ 1, & \text{if } k = n \wedge n \geq 0 \\ 0, & \text{if } 0 \leq n < k \\ \binom{n-1}{k-1} + \binom{n-1}{k}, & \text{if } 0 < k < n \end{cases}$$

The relaxed definition of the binomial coefficients can be written, as an auxiliary logical function, in GOSPEL as seen below:

```
(*@ function rec bcoeff (n: int) (k: int) : int =                GOSPEL
    if k > n then 0
    else if k = 0 || k = n then 1
    else bcoeff (n-1) (k-1) + bcoeff (n-1) k *)
(*@ requires 0 <= k /\ 0 <= n
    variant n *)
```

It is important to remember that both n and k, even in the relaxed definition, are always non-negative. This explains the preconditions. The variant is obtained from the fact that both recursive calls correspond to values of the previous line in Pascal's triangle. Or in other words, n decrements by one every recursive call, and stays non-negative throughout the process. With this logical function, we can effectively express the expected result of the `binom` function presented above:

GOSPEL + OCaml
```
let binom n k = (* ... *)
(*@ res = binom n k
    requires 0 <= k <= n
    ensures res = bcoeff n k *)
```

Despite using a relaxed definition of logical coefficients for the auxiliary logical function, this does not mean that our algorithm should accept values of k, such that $0 \leq n < k$. Here, we should strictly abide by the regular mathematical definition. This relaxation is only meant to ease the verification process, as we will soon see by the loop invariants. We shall start with the outer loop.

GOSPEL + OCaml
```
for i = 1 to n do
(*@ invariant forall x. 0 <= x <= k -> a[x] = bcoeff (i-1) x *)
  (* ... *)
done;
```

After every iteration of this loop, it is expected that the array is filled with the values of the row corresponding. If we were using the regular definition, we would have to determine the minimum value between the desired column and the number of columns in the current row. This by itself is not particularly troublesome or even challenging. However, by relaxing the definition, we can state that every single element of the array is a (relaxed) binomial coefficient, including the values that have been filled with 0. In these situations, the index of some column will exceed the index of the row, hence, 0 is correctly place on the corresponding array positions. This property, of every element of the array being an extended binomial coefficient, is quite important to ensure the inner loop's invariants:

GOSPEL + OCaml
```
while !j > 0 do
(*@ variant !j
    invariant !j <= min i k
    invariant forall x. 0 <= x <= !j -> a[x] = bcoeff (i-1) x
    invariant forall x. !j < x <= k -> a[x] = bcoeff i x *)
done;
```

Reference `j` is initialized with the index of the column where we are going to stop iteration for a given row. This corresponds to the minimum value between the index of last column in the i^{th} row (it is i itself) or the desired column's index (no need to go further). This value is distinct from the array capacity, in case the row contains less than k columns. Independently of the starting value of reference `j`, every index after it (if any exist) is already filled with the corresponding binomial coefficients. Those array positions are filled with the value 0, since $k > i$ in that scenario. As the value of `j` decreases every iteration, the array is gradually filled with the values of new row (i). Meanwhile, the elements yet to explore retain the values of the current row $(i-1)$. The value of `j` never exceeds its initial value, so to ensure that behaviour, we must describe it in an invariant, to avoid out of bound accesses. Note that this invariant is implicitly complemented by the loop condition, where it states that `j` does not go below zero. Moreover, the loop termination is proved by `j` itself.

6.2 Tools In Collaboration

Despite the variety of programming styles, algorithmic techniques and language constructs used in this book, we have narrowed ourselves, thus far, to the OCaml fragment supported by Cameleer tool. In particular, this decision does not allow us to tackle directly the verification of case studies that resort to dynamically allocated memory or floating-point computations, for instance. Explicitly combining multiple verification tools in a cohesive case study with different parts opens the door to many interesting applications. However, one prerequisite is to have good inter-tool support. Fortunately, that is the case within the OCaml deductive verification sphere.

One possible application of this idea, and one of the first that may come to mind, is to have a sorting module that is used in another module before applying the Binary Search algorithm. For this experiment, we will consider the previously presented Insertion Sort implementation, found in section 4.3.3. Additionally, let us consider that it is stored in a file named `insertion_sort.ml`. In Why3, it is possible to import external OCaml files. This immediately allows combining independently certified modules in both different tools, as well as programming and specification languages. However, this poses the question: does the second tool recognize the logical contracts from the first? If so, then the use of this technique is not as restrictive. Surprisingly, that is the case with our toolset, Why3 is able to recognize both OCaml files and GOSPEL contracts. Moreover, one may also import logical definitions:

WhyML
```
module Client

  use int.Int
  use int.ComputerDivision
  use ref.Ref
  use ocamlstdlib.Stdlib
  use import insertion_sort.Insertion_sort as IS

  let binary_search (a: array int) (v: int) : int
    requires { IS.sorted a }
    ensures { (* omitted *) }
  = (* omitted *)

  let main (a: array int) (v: int) : bool
    ensures { result <-> Array.mem v a }
  = IS.in_sort a;
    0 <= binary_search a v < Array.length a

end
```

To import an OCaml file into a WhyML program, we must use the `use ocamlstdlib.Stdlib` import due to translation purposes, mostly associated with arrays. Additionally, we also have to include our OCaml file: `use import insertion_sort.Insertion_sort as IS`. To understand this import, we must before understand WhyML's naming conventions. The identifier before the dot (.) refers to the name of file, while the second identifier refers to the name of the module. For instance, consider `use int.Int` and `use int.ComputerDivision`, `int` refers to a file named `int.mlw` found in its standard library, while `Int` and `ComputerDivision` refer to modules found inside that file. When importing an OCaml file, Why3 implicitly creates a module over the file with the same name, but capitalizes the first letter (in case it was not already).

Verification-wise, we can see that in the `main` function, we receive a potentially unsorted array of integers. This array is then sorted, using the `in_sort` function, found in our OCaml file, so that it can be used in as a parameter in a Binary Search application. The `binary_search` function uses the `sorted` predicate defined in GOSPEL from the OCaml file as a pre-condition to check if the array is sorted. For demonstration purposes, we have used our GOSPEL predicate to show that logical entities can also be imported. However, for more sceptical readers, a logically equivalent predicate defined in WhyML would also work. To run the a WhyML program named `bsearch.mlw`, one can use the command `why3 ide bsearch.mlw -L .`, where the flag `-L .` is used to recognize external libraries from the same directory.

A further experiment with this example can be to test Why3's extraction mechanism. In particular, we can extract the previous WhyML file to OCaml, using the command `why3 extract -D ocaml64 bsearch.mlw -L .`:

OCaml
```
let binary_search (a: (Z.t) my_array) (v: Z.t) : Z.t =
  let exception QtReturn of (Z.t) in
  try
    let l = ref Z.zero in
    let u = ref (Z.sub a.length Z.one) in
    while Z.leq !l !u do
      let m = Z.add !l (Z.div (Z.sub !u !l) (Z.of_string "2")) in
      if Z.lt (get a m) v
      then l := Z.add m Z.one
      else
        begin
          if Z.gt (get a m) v then u := Z.sub m Z.one
          else raise (QtReturn m)
        end
    done;
    Z.of_string "-1"
  with
  | QtReturn r -> r

let main (a: (Z.t) my_array) (v: Z.t) : bool =
  in_sort a;
  let q1_ = binary_search a v in
    Z.leq Z.zero q1_ && Z.lt q1_ a.length
```

Currently, the extraction mechanism is a bit outdated and uses unnatural constructs at times. For instance, it uses the **zartih** library instead of "common" integers. However, there is potential in this mechanism for interesting applications.

6.3 Time Complexity Verification

This textbook combines Algorithm Design and Formal Verification. However, one important aspect of Algorithm Design that we do not cover to a great extent is complexity analysis. This is explained by the challenges that one faces when trying to verify time and space complexities. There are no appropriate mechanisms to do so in Cameleer. However, it is possible to simulate time complexity analysis using a counter variable. Take the base linear search implementation, from section 3.1, as an example:

OCaml

```
let linear_search a v =
  let exception Break of int in
  try
    for i = 0 to Array.length a - 1 do
      if a.(i) = v then raise (Break i)
    done;
    -1
  with Break i -> i
```

To check its correctness, one can annotate the `linear_search` function above with:

GOSPEL + OCaml

```
let search a v =
  let exception Break of int in
  try
    for i = 0 to Array.length a - 1 do
    (*@ invariant forall k. 0 <= k < i -> a.(k) <> v *)
      if a.(i) = v then raise (Break i)
    done;
    -1
  with Break i -> i
```

```
(*@ res = search a v
    ensures res >= 0 -> a.(res) = v
    ensures res = -1 -> forall k. 0 <= k < Array.length a -> a.(k) <> v *)
```

So, the first step to verify time complexity is to declare the reference that will count the number of iterations:

```ocaml
let linear_search a v =
  let n = Array.length a in
  let[@ghost][@logic] time = ref 0 in
  let exception Break of int in
  try
    for i = 0 to n - 1 do
      time := !time + 1;
      if a.(i) = v then raise (Break i)
    done;
    -1
  with Break i -> i
```

The reference, `time`, is initialized with 0 to avoid errors. In fact, counters are quite fickle for the purpose of verifying time complexity. A counter contains an exact value, which contrasts with time complexity values, which are relative and express the magnitude of an operation, time-wise. This means that $\Theta(1)$ operations must be handled carefully, otherwise, the counter will exceed the actual time complexity. In this case, only the loop body, as a whole, is considered as a $\Theta(1)$. For instance, if we were to also increment `time` by one after raising the `Break` exception, this would mean that for an array where the element to be found is on the last index, the `counter` variable would hold `n+1`, where `n` is the length of the array.

Both `n` and `time` could be omitted in this example, but we decided to explicitly define them for more clarity. So, to verify time complexity we must compare the respective values. There are two possible scenarios in linear search, either the element is found or not. If the element was not found, then it means the whole array was traversed and the time complexity is $\Theta(n)$. Whereas, if the value was found, then it could have been found earlier. In this case, the time complexity has an upper of n, *i.e.* it is $O(n)$. These cases can be checked as follows:

```ocaml
let linear_search a v =
  let n = Array.length a in
  let[@ghost][@logic] time = ref 0 in
  let exception Break of int in
  try
    for i = 0 to n - 1 do
      time := !time + 1;
      if a.(i) = v then raise (Break i)
    done;
    assert (!time = n); (* worst case: Theta(n) *)
    -1
  with Break i ->
    assert (!time <= n); (* average case: O(n) *)
    i
```

Remember that assertions also generate verification conditions (VCs) inside the Why3 IDE, hence this can be seen as a form of verifying time complexity. Θ is obtained by comparing with the equality sign, while O and Ω are obtained by using the $\leq$ and $\geq$ signs, depending on the order of the operands. Alternatively, one could also return the time variable to check these conditions on the `linear_search` annotation. Using the assertion strategy, all we have to do is update the loop invariant to:

GOSPEL + OCaml

```
for i = 0 to n - 1 do
(*@ invariant forall k. 0 <= k < i -> a.(k) <> v
    invariant !time = i *)
  (* ... *)
done;
```

6.4 Exercises

Formal Verification is a vast field with numerous tools and techniques to learn. We encourage the readers to explore its many facets further, which may include:

- Verifying an algorithm that is not present in this book (beware of tool limitations)
- Learning a new automated deductive verification tool
- Trying a proof assistant (e.q. Rocq, Isabelle, etc.)
- Trying model checking
- Learning Separation Logic

N.B. This is by no means a comprehensive list nor a definitive roadmap. Instead, our intention is to provide a few suggestions on where to go next within this upmost interesting field, in our opinion. Readers who wish to continue studying Formal Verification should survey the current tools and techniques to decide their next step, according to their own interests and curiosity.

Bibliography

[1] H. Barbosa et al. "cvc5: A Versatile and Industrial-Strength SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*. 2022. URL: https://doi.org/10.1007/978-3-030-99524-9_24 (cit. on p. 2).

[2] A. Charguéraud et al. "GOSPEL - Providing OCaml with a Formal Specification Language". In: *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings*. Ed. by M. H. ter Beek, A. McIver, and J. N. Oliveira. Vol. 11800. Lecture Notes in Computer Science. Springer, 2019, pp. 484–501. URL: https://doi.org/10.1007/978-3-030-30942-8_29 (cit. on p. 4).

[3] E. Clarke. *Lecture 1: Propositional Logic*. https://www.cs.cmu.edu/~emc/15414-f12/lecture/propositional_logic.pdf. Accessed: 2025-09-23 (cit. on p. 4).

[4] S. Conchon et al. "Alt-Ergo 2.2". In: *SMT Workshop: International Workshop on Satisfiability Modulo Theories*. Oxford, United Kingdom, 2018-07. URL: https://inria.hal.science/hal-01960203 (cit. on p. 2).

[5] J. Filliâtre and S. Conchon. *Apprendre à programmer avec OCaml: Algorithmes et structures de données*. Noire. Eyrolles, 2014. ISBN: 9782212291551. URL: https://books.google.pt/books?id=aTy9BAAAQBAJ (cit. on pp. iii, 90).

[6] J. Filliâtre, L. Gondelman, and A. Paskevich. "The Spirit of Ghost Code". In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. Ed. by A. Biere and R. Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 1–16. URL: https://doi.org/10.1007/978-3-319-08867-9%5C_1 (cit. on p. 82).

[7] J.-C. Filliâtre and C. Pascutto. "Ortac: Runtime Assertion Checking for OCaml". In: *RV'21 - 21st International Conference on Runtime Verification*. Los Angeles, CA, United States, 2021-10. URL: https://inria.hal.science/hal-03252901 (cit. on p. 9).

[8] J. Filliâtre and A. Paskevich. “Why3 - Where Programs Meet Provers”. In: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. Ed. by M. Felleisen and P. Gardner. Vol. 7792. Lecture Notes in Computer Science. Springer, 2013, pp. 125–128. URL: https://doi.org/10.1007/978-3-642-37036-6_8 (cit. on p. 2).

[9] R. W. Floyd. “Assigning Meanings to Programs”. In: *Proceedings of Symposium on Applied Mathematics* 19 (1967), pp. 19–32. URL: http://laser.cs.umass.edu/courses/cs521-621.Spr06/papers/Floyd.pdf (cit. on p. 6).

[10] C. A. R. Hoare. “An axiomatic basis for computer programming”. In: *Commun. ACM* 12.10 (1969-10), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: https://doi.org/10.1145/363235.363259 (cit. on pp. 6, 82).

[11] G. P. Huet. “The Zipper”. In: *J. Funct. Program.* 7.5 (1997), pp. 549–554. DOI: 10.1017/S0956796897002864. URL: https://doi.org/10.1017/s0956796897002864 (cit. on p. 83).

[12] Z. Manna and J. McCarthy. “Properties of Programs and Partial Function Logic”. In: 1969. URL: https://api.semanticscholar.org/CorpusID:119587093 (cit. on p. 20).

[13] L. de Moura and N. Bjørner. “Z3: An Efficient SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and J. Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3 (cit. on p. 2).

[14] M. Pereira and A. Ravara. “Cameleer: A Deductive Verification Tool for OCaml”. In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*. Ed. by A. Silva and K. R. M. Leino. Vol. 12760. Lecture Notes in Computer Science. Springer, 2021, pp. 677–689. DOI: 10.1007/978-3-030-81688-9_31. URL: https://doi.org/10.1007/978-3-030-81688-9_31 (cit. on p. 4).

[15] M. J. P. Pereira. “Tools and Techniques for the Verification of Modular Stateful Code. (Outils et techniques pour la vérification de programmes impératives modulaires)”. PhD thesis. University of Paris-Saclay, France, 2018. URL: https://tel.archives-ouvertes.fr/tel-01980343 (cit. on p. 82).

[16] D. Richeson. *What is the difference between a theorem, a lemma, and a corollary?* https://divisbyzero.com/2008/09/22/what-is-the-difference-between-a-theorem-a-lemma-and-a-corollary/. Accessed: 2025-09-23 (cit. on p. 6).

[17] S. Schulz, S. Cruanes, and P. Vukmirović. "Faster, Higher, Stronger: E 2.3". In: *Proc. of the 27th CADE, Natal, Brasil.* Ed. by P. Fontaine. LNAI 11716. Springer, 2019, pp. 495–507 (cit. on pp. 2, 83).

[18] R. Sedgewick and K. Wayne. *Algorithms.* Addison-Wesley, 2011. ISBN: 9780321573513. URL: `https://books.google.pt/books?id=MTpsAQAAQBAJ` (cit. on p. iii).

[19] M. Wooldridge. *Lecture 10: First-Order Logic.* `https://www.cs.ox.ac.uk/people/michael.wooldridge/teaching/soft-eng/lect10.pdf`. Accessed: 2025-09-23 (cit. on p. 5).

[20] M. Wooldridge. *Lecture 7: Propositional Logic.* `https://www.cs.ox.ac.uk/people/michael.wooldridge/teaching/soft-eng/lect07.pdf`. Accessed: 2025-09-23 (cit. on p. 4).

Appendix A

Solutions

A.1 Chapter 2

1. Euclidean division that supports negative numbers:

GOSPEL + OCaml

```
let sign n =
  if n >= 0 then 1
  else -1
(*@ r = sign n
    ensures n >= 0 -> r = 1
    ensures n < 0 -> r = -1 *)

let euclidean_div x y =
  let r = ref x in
  let q = ref 0 in
  let sx = sign x and sy = sign y in
  while not (0 <= !r && !r < abs y) do
  (*@ invariant x = y * !q + !r
      invariant x >= 0 -> 0 <= !r
      invariant x < 0 -> !r < abs y
      variant !r * sx + abs y *)
    r := !r - sx *  abs y;
    q := !q + 1 * sy * sx;
  done;
  (!q, !r)
(*@ (q, r) = euclidean_div x y
    requires y <> 0
    ensures x = y * q + r
    ensures 0 <= r < abs y *)
```

Explanation: The first step to verify this problem is to understand how the negative numbers affect the algorithm, with a concrete example:

	$y = 6$	$y = -6$
$x = 17$	$17 = 0 * 6 + 17$ $17 = 1 * 6 + 11$ $17 = 2 * 6 + 5$ Trend: $q\uparrow$, $r\downarrow$	$17 = 0 * (-6) + 17$ $17 = (-1) * (-6) + 11$ $17 = (-2) * (-6) + 5$ Trend: $q\downarrow$, $r\downarrow$
$x = -17$	$-17 = 0 * 6 - 17$ $-17 = (-1) * 6 - 11$ $-17 = (-2) * 6 - 5$ $-17 = (-3) * 6 + 1$ Trend: $q\downarrow$, $r\uparrow$	$-17 = 0 * (-6) - 17$ $-17 = 1 * (-6) - 11$ $-17 = 2 * (-6) - 5$ $-17 = 3 * (-6) + 1$ Trend: $q\uparrow$, $r\uparrow$

As one may observe, when the dividend x is positive, the remainder, r, otherwise r increases. The trend of the quotient, q, depends on both x and y, when both have the same sign q increases, otherwise, q decreases. For this effect, we define and specify the `sign` function:

GOSPEL + OCaml
```
let sign n =
  if n >= 0 then 1
  else -1
(*@ r = sign n
    ensures n >= 0 -> r = 1
    ensures n < 0 -> r = -1 *)
```

By defining additional variables with the sign of each of the inputs, we can accurately model the behaviours of the remainder and quotient:

GOSPEL + OCaml
```
let euclidean_div x y =
  let r = ref x in
  let q = ref 0 in
  let sx = sign x and sy = sign y in
  while not (0 <= !r && !r < abs y) do
    r := !r - sx *  abs y;
    q := !q + 1 * sy * sx;
  done;
  (!q, !r)
```

Another change is the loop condition, the remainder independently of starting as a positive or negative value, it will converge to a value between 0 (inclusive) and the absolute value of the dividend, y (exclusive). These changes reflect in the specification of the `while` loop, while the `euclidean_div` function remains intact:

GOSPEL + OCaml

```
while not (0 <= !r && !r < abs y) do
(*@ invariant x = y * !q + !r
    invariant x >= 0 -> 0 <= !r
    invariant x < 0 -> !r < abs y
    variant !r * sx + abs y *)
  (* ... *)
done;
```

With these changes, the previous `variant` clause (`!r`) becomes obsolete, since the remainder may grow in value when x is negative. So, we need to find a formula, possibly related to the remainder, that always decreases and remains non-negative throughout the loop. The solution that we propose stems from the observation that the remainder has different behaviours depending on the signal of x, so by multiplying these two values we obtain an ever-decreasing expression. However, if x is a negative, then so will r be until just before the last iteration, which implies that $\text{sign}(x) * r < 1$ exactly when that happens. As such, we have to find a suitable upper bound, the most natural choice would the absolute value of y, since the final value of r resides in the interval from 0 to y, so, this leads to $\text{sign}(x) * r + |y| > 0$.

2. Functional Euclidean division:

GOSPEL + OCaml

```
let rec eudiv_aux (x: int) y q r =
  if r < y then (q, r)
  else eudiv_aux x y (q+1) (r-y)
(*@ (q', r') = eudiv_aux x y q r
    requires y > 0
    requires r >= 0
    requires x = y * q + r
    ensures 0 <= r' < y
    ensures x = y * q' + r'
    variant r *)

let eudiv x y = eudiv_aux x y 0 x
(*@ (q, r) = eudiv x y
    requires x >= 0
    requires y > 0
    ensures x = y * q + r
    ensures 0 <= r < y *)
```

Explanation: In this exercise we propose an implementation does not support negative numbers, however, we also encourage readers to try more exercises on their own. The main function, `eudiv`, serves to initiate the values of the quotient as 0 and the remainder as the dividend. As such, the specification of this function does not differ from the imperative versions. In this algorithm the recursion strategy does not change significantly from the imperative version, which leads to the `eudiv_aux` function:

```
let rec eudiv_aux (x: int) y q r =                                    OCaml
  if r < y then (q, r)
  else eudiv_aux x y (q+1) (r-y)
```

The annotations on this function change slightly from the `while` loop in the imperative version, due to the special properties of the `invariant` clause: the condition must be true before and after every iteration. This type of clause is only available within loops, as such, we have to adapt it into pre- and post-conditions:

```
let rec eudiv_aux (x: int) y q r = (* ... *)                  GOSPEL + OCaml
(*@     (q', r') = eudiv_aux x y q r
    requires y > 0
    requires r >= 0
    requires x = y * q + r
    ensures 0 <= r' < y
    ensures x = y * q' + r'
    variant r *)
```

Given the functional nature of this implementation, we now have, as parameters, the previous candidates for the quotient and remainder, q and r, respectively, as well as the next candidates, q' and r', respectively. Both of these pairs of candidates must respect the Euclidean lemma, and due to the immutability of the variables, q and r must respect the condition right from the start, while q' and r' are only available at the end, therefore are used in post-conditions. We must also guarantee that y stays positive and r stays non-negative every iteration. The termination proof remains the same as the imperative version, which is r, since it decreases every iteration.

3. Iterative factorial function:

```
(*@ function rec fact (n: int) : int   =                      GOSPEL + OCaml
    if n = 0 then 1
    else n * fact (n-1) *)
(*@ requires n >= 0
    variant n *)

let rec factorial n =
  let r = ref 1 in
  for i = 1 to n do
  (*@ invariant !r = fact (i-1) *)
    r := !r * i
  done;
  !r
(*@ r = factorial n
    requires n >= 0
    ensures r = fact n *)
```

Explanation: Similar to the Fibonacci example, we must define a logical function to confirm the result of the factorial. This can be achieved using the mathematical definition, a recursive function. It is also worth noting this function is only applicable to non-negative numbers. The main function contains a `for` loop from 1 to n which allows calculating the factorial, if $n = 0$, then the loop is skipped and the result remains correct due to it being initialized as 1. The most notable condition in this example is the invariant: `!r = fact (i-1)`. This can be explained if we observe the behaviour when $i = 2$, if we were to use the condition `!r = fact i`, then, it would mean that at the start of the second iteration the reference r should contain the value of fact(2), which is not true since we are yet to multiply the previous result by 2, therefore the reference is one value behind.

4. Tribonacci sequence:

GOSPEL + OCaml

```
let[@ghost][@logic] rec trib n =
  if n = 0 then 0
  else if n <= 2 then 1
  else trib (n-1) + trib (n-2) + trib (n-3)
(*@ res = trib n
    requires n >= 0
    variant n *)

let tribonacci n =
  let z = ref 0 in
  let y = ref 1 in
  let x = ref 1 in
  for i = 0 to n - 1 do
  (*@ invariant !z = trib i
      invariant !y = trib (i+1)
      invariant !x = trib (i+2) *)
    let sum = !z + !y + !x in
    z := !y;
    y := !x;
    x := sum
  done;
  !z
(*@ res = tribonacci n
    requires n >= 0
    ensures res = trib n *)
```

N.B.: There is discussion on the first element of the Fibonacci sequence, and, even more so, on the first few elements of the Tribonacci sequence. However, from a verification perspective, this discussion is not really important, and what matters is to choose one variant and be consistent about it, or in other words, the logical function and the real implementation should denote the same variant for a successful verification process.

Explanation: As previously mentioned there are two ways to express a logical function, either as a GOSPEL function or an OCaml function with the `ghost` and `logic` tags. So to display both options, we opted to use the latter in this example. One possibility is to use the original Fibonacci example as basis, in which we need to add a new reference for the third value and update accordingly, with this we will calculate two values ahead, and must add an invariant clause to accompany this change.

5. Functional fast exponentiation:

GOSPEL + OCaml

```
(*@ function rec power (x n: int) : int =
    if n = 0 then 1
    else x * power x (n-1) *)
(*@ requires n >= 0
    variant n *)

let[@lemma] rec power_even (x: int) (n: int) =
  if n > 1 then power_even x (n-2)
(*@ requires n >= 0
    requires mod n 2 = 0
    variant n
    ensures power x n = (power (x * x) (div n 2)) *)

let[@lemma] power_odd (x: int) (n: int) =
  power_even x (n-1)
(*@ requires n >= 0
    requires mod n 2 = 1
    ensures power x n = x * (power (x * x) (div n 2)) *)

let rec exp x n =
  if n = 0 then 1
  else
    let r = exp (x * x) (n / 2) in
    if n mod 2 = 0 then r else x * r
(*@ r = exp x n
    requires n >= 0
    variant n
    ensures r = power x n *)
```

Explanation: We recommended using the same logical framework as before, i.e. the `power` logical function, the `power_even` and `power_odd` lemma functions. From an implementation standpoint, when the exponent n is 0, the function returns one, based on our logical definition. In the outer `else` branch, we first obtain the common factor between the two cases $(x^2)^{n/2}$, independently of the parity of n. The nested `if`-statement is used to calculate the final result, as when n is even, it suffices to return r by itself, however, when n is odd, we must also multiply x by r. The specification on the `exp` function is very similar to the imperative version, with the need for a `variant` clause. To prove termination we may use n, since it is a non-negative value that decreases every iteration until it reaches 0 in the end.

A.2 Chapter 3

1. Backwards linear search on arrays with a while loop:

GOSPEL + OCaml

```
let lsearch a v =
  let exception Break of int in
  try
    let i = ref (Array.length a - 1) in
    while !i >= 0 do
    (*@ invariant -1 <= !i < Array.length a
        invariant forall k. !i < k < Array.length a -> a.(k) <> v
        variant !i *)
      if v = a.(!i) then raise (Break !i)
      else i := !i - 1
    done;
    raise Not_found
  with Break i -> i
(*@ r = lsearch a v
    ensures a.(r) = v
    raises Not_found -> forall k. 0 <= k < Array.length a ->
      a.(k) <> v *)
```

Explanation: In this proposed solution we use the exception, however, any other of the taught techniques would be plausible (*i.e.* encoding with negative numbers or options). The main changes in the annotations are the ones on the `while` loop. The index variable i, varies from -1 (inclusive) and size(a) (exclusive), although it only reaches -1 when exiting the loop. This condition, alongside the loop condition guarantee that no invalid position of the array is accessed. Moreover, we need to exclude the previously check part of the array, since we are traversing from right to left the condition changes slightly, its lower limit is $!i$, which is exclusive, since at the start of the iteration we are yet to check index of the current $!i$. The upper limit is, naturally, the length of the array, also exclusive. To prove termination, we may use the value of $!i$ itself, since it decreases every iteration.

2. Recursive binary search on arrays:

GOSPEL + OCaml

```
(*@ predicate sorted (a: int array) = forall i j:int.
    0 <= i <= j < Array.length a -> a.(i) <= a.(j) *)
```

```
let rec bsearch a v l u =
  if l > u then None
  else begin
    let m = (l + u)/2 in
    if a.(m) < v && m+1 >= Array.length a then None
    else if a.(m) < v then bsearch a v (m+1) u
    else if a.(m) > v && m-1 < 0 then None
    else if a.(m) > v then bsearch a v l (m-1)
    else Some m
  end
(*@ res = bsearch a v l u
    variant u - l
    requires sorted a
    requires 0 <= l < Array.length a
    requires 0 <= u < Array.length a
    requires forall k. 0 <= k < l \/ u < k < Array.length a -> a.(k) <> v
    ensures match res with
      | None -> forall k. 0 <= k < Array.length a -> a.(k) <> v
      | Some i -> a.(i) = v *)
```

Explanation: Unlike the iterative version, where the lower and upper bounds are defined inside the body of the function, the recursive version takes them as parameters. This requires added attention to their values. There are two options. Either `l` (lower) and `u` (upper) are required, as a pre-condition, to be within the boundaries of the array or not. In the first case, it requires confirming, before each recursive call, that the updated boundary is still a valid index, otherwise, it is considered as not found. If the values are not restricted, then inside the function one must check if the current boundaries are valid, otherwise it must return the value encoded as not found. For this proposed solution, we opted for the first, due to it being slightly more concise.

A.3 Chapter 4

1. Functional insertion sort:

GOSPEL + OCaml

```
(* Verification library omitted *)

let rec insert x l =
  match l with
  | [] -> [x]
  | y::ls ->
    if x <= y then x::y::ls
    else y::(insert x ls)
(*@ r = insert x l
    requires sorted l
    ensures sorted r
    ensures permut r (x::l)
    variant l *)

let rec insertion_sort l =
  match l with
  | [] -> []
  | [x] -> [x]
  | x::ls -> insert x (insertion_sort ls)
(*@ r = insertion_sort l
    ensures sorted r
    ensures permut r l
    variant l *)
```

Explanation: First we define an auxiliary function, `insert`, that takes a value and a sorted list so that it places the value inside the list in the first place that respects the sorted predicate. The main function, `insertion_sort`, inserts the head of the list using the auxiliary function in a sorted permutation of its tail, obtained from the recursive call.

2. Tail recursive insertion sort:

GOSPEL + OCaml

```
(* Verification library omitted *)

let rec insert x acc l =
  match l with
  | [] -> acc @ [x]
  | y::ls ->
    if x <= y then acc @ x::l
    else insert x (acc @ [y]) ls
(*@ r = insert x acc l
    requires sorted acc
    requires forall k. List.mem k acc -> k <= x
    requires forall k y. List.mem k acc -> List.mem y l -> k <= y
    requires sorted l
    ensures sorted r
    ensures permut r (acc @ (x::l))
    variant l *)

let rec insertion_sort l acc =
  match l with
  | [] -> acc
  | x::ls -> insertion_sort ls (insert x [] acc)
(*@ r = insertion_sort l acc
    requires sorted acc
    ensures sorted r
    ensures permut r (l @ acc)
    variant l *)
```

Explanation: Starting with the main function, `insertion_sort`, we need an accumulator parameter to store the results of the auxiliary. Note that this allows to swap the order of the function calls when compared to the non-tail recursive version. In that version the main function's recursive call was used as a parameter to the auxiliary function, which would contribute significantly to introducing stack overflows. Specification-wise, `acc` will contain the intermediate results, which are sorted, therefore we need to express this in order to prove that the final result is sorted. Moreover, since the original input is spread in both `l` and `acc`, their concatenation will be a permutation of `r`, the final result. Moving on to the auxiliary function, `insert`, the accumulator will contain the previously visited elements, in the same order as before, so that we can remove list operations with function calls as arguments. As such, we need pre-conditions that state that `acc` is sorted, and its elements are smaller than the element to be inserted, `x`, and every element that has not been processed yet.

3. Polymorphic functional insertion sort:

GOSPEL + OCaml

```
module InsertionSort (E: Cmp) = struct

  (* Verification library omitted *)

  let rec insert x l =
    match l with
    | [] -> [x]
    | y::ls ->
      if E.leq x y then x::y::ls
      else y::(insert x ls)
  (*@ r = insert x l
      requires sorted l
      ensures sorted r
      ensures permut r (x::l)
      variant l *)

  let rec insertion_sort l =
    match l with
    | [] -> []
    | [x] -> [x]
    | x::ls -> insert x (insertion_sort ls)
  (*@ r = insertion_sort l
      ensures sorted r
      ensures permut r l
      variant l *)

end
```

Explanation: Using the non-recursive implementation, we simply have to replace any $\leq$ or $=$ operators with the corresponding operations from `E`, the module with the comparison functions.

4. Optimized tail recursive selection sort:

GOSPEL + OCaml

```
(* Verification library omitted *)

(* selection_aux remains unchanged *)

let[@lemma] rec rev_permut (l: int list) =
  let l' = List.rev l in
  match l with
  | [] -> ()
  | h::t -> assert (List.mem h l'); rev_permut t
(*@ ensures permut l (List.rev l) *)

let rec selection_sort acc = function
  | [] -> List.rev acc
  | [x] -> List.rev (x::acc)
  | x::ls ->
    let m, r = selection_aux x [] ls in
    assert (List.mem m (x::ls));
    selection_sort (m::acc) r
(*@ r = selection_sort acc l
    requires forall x y.
      List.mem x acc -> List.mem y l -> x <= y
    requires sorted (List.rev acc)
    ensures sorted r
    ensures permut r (l @ acc)
    variant List.length l *)
```

Explanation: The `@` operator has linear time complexity, since the elements are placed at the end of the first list, and we do not have direct access to the final element of the first list. Meanwhile, the `::` operator has constant time complexity, since we have direct access to the first element of the list, and we place an element at its head. So, the objective of this exercise was to optimize the previously presented selection sort implementation. However, this change comes with a cost, since we are now placing elements at the head of the accumulator, then it is in descending order, which contrasts with our desired to sort ascendingly. Therefore, we must revert the result in the final iteration. Moreover, this has consequences in the annotations, the pre-condition that uses the `sorted` predicate must now take in consideration that it is only sorted when reversed. The SMT solvers seem to have a bit of trouble when dealing with reversed lists, as such we have defined the `rev_permut` lemma function, where for each element of a given list we assert that it also can be found in the reversed list, thereby proving that the reverse of a list is a permutation.

5. Imperative selection sort:

GOSPEL + OCaml

```
(* Verification library omitted *)

let sel_sort a =
  let n = Array.length a - 1 in
  for i = 0 to n do
  (*@ invariant sorted_sub a 0 i
      invariant permut a (old a)
      invariant forall x y. 0 <= x < i && i <= y <= n -> a.(x) <= a.(y) *)
    let m = ref i in
    for j = i+1 to n do
    (*@ invariant i <= !m < j
        invariant forall k. i <= k < j -> a.(!m) <= a.(k) *)
      if a.(j) < a.(!m) then m := j
    done;
    swap a i !m
  done
(*@ ensures sorted a
    ensures permut a (old a) *)
```

Explanation: The outer loop serves the purpose to place the $(i+1)^{\text{th}}$ minimum value in the i^{th} position using the swap function. On the other hand, the inner loop is used to find the index containing the current minimum value from the remaining elements to be processed. Unsurprisingly, the specification on the main function ensures that the array, `a`, is sorted, by the end of the process, and a permutation of the original memory disposition. The outer loop reflects these conditions, in particular, a permutation is always a permutation, however, regarding the sorted predicate, only a subset of the array will be sorted during the computations, in particular the positions between 0, inclusive, and `i`, exclusive, since these are the indexes that have already been processed. The third invariant, on the outer loop, states that every element that already has been processed, *i.e.* the ones that are already sorted are greater or equal than those that have not been processed, this is true since at every step we are selecting the smallest number available, therefore, the numbers that have not been processed must be greater or equal. The first invariant on the inner loop bounds the index `!m` to the area that has already been search, `i.e.` starting from `i` (inclusive), which is the first unprocessed element, and `j` (exclusive), which is the current position. The second invariant states that the value of index `!m` must be lesser or equal to every other value we have seen in the unprocessed interval, which is true, since we are looking for the minimum value in that subset, and `!m` must correspond to the best minimum candidate we have seen so far.

6. Polymorphic functional merge sort:

GOSPEL + OCaml

```
module MergeSort (E: Cmp) = struct
```

```
  (* Verification library omitted *)

  let rec split (l: elt list) =
    match l with
    | [] -> ([], [])
    | x::ls ->
      let l1, lr = split ls in (lr, x::l1)
  (*@ (r1, r2) = split l
      ensures permut l (r1 @ r2)
      ensures List.length r2 = List.length r1 \/
        List.length r2 = List.length r1 + 1
      ensures List.length r2 + List.length r1 = List.length l
      variant List.length l *)

  let rec merge l1 l2 =
    match l1, l2 with
    | z, [] | [], z -> z
    | x::ls, y::rs ->
      if E.leq x y then x::(merge ls (y::rs))
      else y::(merge (x::ls) rs)
  (*@ r = merge l1 l2
      requires sorted l1
      requires sorted l2
      ensures sorted r
      ensures permut r (l1 @ l2)
      variant List.length l1 + List.length l2 *)

  let rec merge_sort l =
    match l with
    | [] -> []
    | [x] -> [x]
    | _ ->
    let (a, b) = split l in merge (merge_sort a) (merge_sort b)
  (*@ r = merge_sort l
      ensures sorted r
      ensures permut r l
      variant List.length l *)

end
```

Explanation: Simply amounts to replacing any $\leq$ or $=$ operators with the corresponding operations from E, the module with the comparison functions.

7. Unoptimized tail recursive quick sort:

GOSPEL + OCaml

```
(* Verification library omitted *)
(* quick_aux, permut_append_mem, permut_elems also omitted *)

let rec quick_sort acc = function
  | [] -> acc
  | [x] -> x::acc
  | p::ls ->
    assert (List.mem p (p::ls));
    let (left, right) = quick_aux p [] [] ls in
    let sorted_right = quick_sort acc right in
    quick_sort (p::sorted_right) left
(*@ r = quick_sort acc l
    requires forall x y. List.mem x acc -> List.mem y l -> x >= y
    requires sorted acc
    ensures sorted r
    ensures permut r (l @ acc)
    variant List.length l *)
```

Explanation: Since this is an unoptimized, we may define which side to go first, for instance, let's say right. This decision allows us to reduce the number of accumulators from two to one, since we prioritize one side over the other. This simplifies both the implementation and specification significantly, in particular, we reduce the number of pre-conditions from 5 to 2. It is important to notice that, since we have chosen to always go right first, our accumulator always contains values greater or equal to those in the list of elements yet to be processed, including the pivot (p) itself.

8. Polymorphic functional quick sort:

GOSPEL + OCaml

```
module MergeSort (E: Cmp) = struct

  (* Verification library omitted *)

  let rec quick_aux p = function
    | [] -> ([], [])
    | x :: r ->
      let ll, lr = quick_aux p r in
      if E.leq x p then x :: ll, lr
      else ll, x :: lr
  (*@ ll, lr = quick_aux p l
      variant l
      ensures forall x. List.mem x ll -> E.leq x p
      ensures forall x. List.mem x lr -> E.g x p
      ensures permut l (ll @ lr) *)

  let rec quick_sort l =
    match l with
    | [] -> []
    | [x] -> [x]
    | p :: ls ->
      let (left, right) = (quick_aux p ls) in
      (quick_sort left) @  p :: quick_sort right
  (*@ r = quick_sort l
      ensures sorted r
      ensures permut r l
      variant List.length l *)

end
```

Explanation: Simply amounts to replacing any $\leq$ or $=$ operators with the corresponding operations from E, the module with the comparison functions.

A.4 Chapter 5

1. Queue, implemented as a list:

GOSPEL + OCaml

```
module type Queue = sig
  type 'a queue
  (*@ model view: 'a list *)

  (*@ function length (q: 'a queue) : int *)

  exception Empty

  val empty : unit -> 'a queue
  (*@ res = empty ()
      ensures length res = 0 *)

  val size : 'a queue -> int
  (*@ res = size q
      ensures res = length q *)

  val is_empty : 'a queue -> bool
  (*@ res = is_empty q
      ensures res <-> length q = 0 *)

  val enqueue : 'a -> 'a queue -> 'a queue
  (*@ res = enqueue x q
      ensures length res = 1 + length q
      ensures res.view = Sequence.append q.view (Sequence.singleton x) *)

  val head : 'a queue -> 'a
  (*@ res = head q
      raises Empty -> length q = 0
      ensures res = q.view[0] *)

  val dequeue : 'a queue -> 'a * 'a queue
  (*@ (v, res) = dequeue q
      raises Empty -> length q = 0
      ensures v = (q.view)[0]
      ensures res.view = q.view[1 ..] *)

end
```

Explanation: Given the simplicity of our data type, we can model it directly as a list. Ideally, we should use sequences, however, due to type checking restrictions that will be discussed soon, we are not able to use them. Queues follow a first-in-first-out discipline. This means that we process elements by order of "arrival". So, new elements are added to the tail of the queue, whereas, the head of the queue is removed, when dequeueing. `Sequence.append q.view (Sequence.singleton x)` is the equivalent to `q.view @ [x]` for sequences. Meanwhile, `q.view[1 ..]` returns the sequence of elements `q.view` without the first element.

GOSPEL + OCaml

```
module ListQueue : Queue = struct
  type 'a queue = {
    view: 'a list
  }

  (*@ function length (q: 'a queue) : int = List.length q.view *)
  (*@ ensures result >= 0 *)

  exception Empty

  let empty () = { view = [] }
  (*@ res = empty ()
      ensures length res = 0 *)

  let size q = List.length q.view
  (*@ res = size q
      ensures res = length q *)

  let is_empty q = size q = 0
  (*@ res = is_empty q
      ensures res <-> length q = 0 *)

  let enqueue x q = { view = q.view @ [x] }
  (*@ res = enqueue x q
      ensures length res = 1 + length q
      ensures res.view = Sequence.append q.view (Sequence.singleton x) *)
  let head q =
    match q.view with
    | [] -> raise Empty
    | h::_ -> h
  (*@ res = head q
      raises Empty -> length q = 0
      ensures res = q.view[0] *)

  let dequeue q =
    match q.view with
    | [] -> raise Empty
    | h::t -> (h, { view = t })
  (*@ (v, res) = dequeue q
      raises Empty -> length q = 0
      ensures v = q.view[0]
      ensures res.view = q.view[1 ..] *)

end
```

Explanation: Since `view` is not a ghost field, and therefore not a logical value, the OCaml's type-checker cannot associate the list type with the sequence type. Furthermore, the proofs found in the module are exactly equal to the ones found in the signature.

2. Stack, implemented as a list:

GOSPEL + OCaml

```
module type Stack = sig
  type 'a stack
  (*@ model view: 'a list *)

  (*@ function length (q: 'a stack) : int *)

  exception Empty

  val empty : unit -> 'a stack
  (*@ res = empty ()
      ensures length res = 0 *)

  val size : 'a stack -> int
  (*@ res = size q
      ensures res = length q *)

  val is_empty : 'a stack -> bool
  (*@ res = is_empty q
      ensures res <-> length q = 0 *)

  val push : 'a -> 'a stack -> 'a stack
  (*@ res = push x q
      ensures length res = 1 + length q
      ensures res.view = Sequence.cons x q.view *)

  val head : 'a stack -> 'a
  (*@ res = head q
      raises Empty -> length q = 0
      ensures res = q.view[0] *)

  val pop : 'a stack -> 'a * 'a stack
  (*@ (v, res) = pop q
      raises Empty -> length q = 0
      ensures v = q.view[0]
      ensures res.view = q.view[1 ..] *)

end
```

Explanation: Stacks are quite similar to queues, the main difference is in their discipline. While queues follow a first-in-first-out model, stacks follow the last-in-first-out discipline. The only change is that elements are added to the head of the list, rather than their tail. The act of removing an element is still done in the head of the data structure. Thus, the only detail to be aware of is that `Sequence.cons x q.view` is the equivalent to `x::q.view` for sequences.

GOSPEL + OCaml

```
module ListStack : Stack = struct
  type 'a stack = {
    view: 'a list
  }

  (*@ function length (q: 'a stack) : int = List.length q.view *)
  (*@ ensures result >= 0 *)

  exception Empty

  let empty () = { view = [] }
  (*@ res = empty ()
      ensures length res = 0 *)

  let size q = List.length q.view
  (*@ res = size q
      ensures res = length q *)

  let is_empty q = size q = 0
  (*@ res = is_empty q
      ensures res <-> length q = 0 *)

  let push x q = { view = x::q.view }
  (*@ res = push x q
      ensures length res = 1 + length q
      ensures res.view = Sequence.cons x q.view *)

  let head q =
    match q.view with
    | [] -> raise Empty
    | h::_ -> h
  (*@ res = head q
      raises Empty -> length q = 0
      ensures res = q.view[0] *)

  let pop q =
    match q.view with
    | [] -> raise Empty
    | h::t -> (h, { view = t })
  (*@ (v, res) = pop q
      raises Empty -> length q = 0
      ensures v = q.view[0]
      ensures res.view = q.view[1 ..] *)

end
```

Explanation: Similarly to the previous example, we could not define the `view` field as a sequence in the signature due to the typechecker.

3. Set, implemented as a list:

GOSPEL + OCaml

```
module type EqualSig = sig
  type t

  val eq : t -> t -> bool
  (*@ res = eq x y
        ensures res <-> x = y *)
end

module type Set = sig
  type elt
  type set

  (*@ predicate belongs (x: elt) (s: set) *)
  (*@ predicate is_set (s: set)*)

  val empty: unit -> set
  (*@ res = empty ()
      ensures is_set res
      ensures forall x:elt. not belongs x res *)
  val mem: elt -> set -> bool
  (*@ res = mem v s
      requires is_set s
      ensures res <-> belongs v s *)

  val add: elt -> set -> set
  (*@ res = add v s
      requires is_set s
      ensures belongs v res
      ensures is_set res *)

  val is_subset: set -> set -> bool
  (*@ res = is_subset s1 s2
      requires is_set s1
      requires is_set s2
      ensures res <-> forall x:elt. belongs x s1 -> belongs x s2 *)

end
```

Explanation: To ensure polymorphism, we must simulate it with by using a signature equipped with any type `t`, and an equality test. This is because the set abstract data type does not allows for duplicates, and we must ensure that it does not by performing comparisons at the program level. When converting from OCaml's equality operator (`=`) to WhyML, it looses its polymorphic properties, unless it is used in logical definitions in GOSPEL. Thus, to regain polymorphism, we must simulate it. Similar to the binary search tree example, we may define a predicate rather than a type invariant, to avoid record types.

GOSPEL + OCaml

```
module ListSet (E: EqualSig) : Set = struct
  type elt = E.t
  type set = elt list

  (*@ predicate belongs (x: elt) (s: set) = List.mem x s *)

  (*@ function rec occ (v: elt) (s: set) : int =
      match s with
      | Nil -> 0
      | Cons h t -> occ v t + if E.eq v h then 1 else 0 *)
  (*@ ensures result >= 0
      ensures result = 0 -> not belongs v s
      ensures result > 0 -> belongs v s *)

  (*@ predicate is_set (s: set) = forall x: elt. occ x s <= 1 *)

  let empty () = let s: set = [] in s
  (*@ res = empty ()
      ensures is_set res
      ensures forall x:elt. not belongs x res *)

  let mem (v: elt) s = List.mem v s
  (*@ res = mem v s
      requires is_set s
      ensures res <-> belongs v s *)

  let add (v: elt) s =
    if mem v s then s
    else v::s
  (*@ res = add v s
      requires is_set s
      ensures belongs v res
      ensures is_set res *)

  (*@ lemma tail_set: forall h:elt, t:set. is_set (h::t) -> is_set t *)

  let rec is_subset (s1: set) (s2: set) =
    match s1 with
    | [] -> true
    | h::t -> List.mem h s2 && is_subset t s2
  (*@ res = is_subset s1 s2
      requires is_set s1
      requires is_set s2
      variant s1
      ensures res <-> forall x:elt. belongs x s1 -> belongs x s2 *)

end
```

Explanation: The `set` predicate is defined as every element only occurring at most once in the data structure, which needs an auxiliary definition, `occ`, to count occurences. For that reason, before adding an element to the set, we must check it already belongs. Finally, we need to define a lemma to ensure that the tail of a list set is also a set. This lemma allows to complete the verification conditions of the `is_subset` function.