Allan Blanchard

Introduction to C program proof with Frama-C and its WP plugin

November 2, 2024

zeste de savoir

Contents

1. Introduction

!

The source code of the tutorial is available on GitHub, as well as the solutions to the different exercises (including the Coq proofs of some properties).
If you find some errors, please do not hesitate to post an issue or a pull request at:
https://github.com/AllanBlanchard/tutoriel_wp

i

In this tutorial, some examples and some elements of organization are similar to the ones used in the TAP 2013 tutorial by Nikolai Kosmatov, Virgile Prevosto and Julien Signoles of the CEA LIST, since it is quite didactic. It also contains examples taken from *ACSL By Example* by Jochen Burghardt, Jens Gerlach, Kerstin Hartig, Hans Pohl and Juan Soto from the Fraunhofer. For formal aspects, I verified my statements and explanations using the course on Why3 given by Andrei Paskevich *at EJCP 2018* . The remaining ideas come from my personal experience with Frama-C and WP.

The versions of the tools considered in this tutorial are:

- Frama-C 30.0 Zinc
- Why3 1.7.2
- Alt-Ergo 2.6.0
- Coq 8.16.1 (for provided scripts, not used in the tutorial)
- Z3 4.8.10 (in one example, it is not absolutely necessary to follow this book)

Depending on the versions used by the reader some differences could appear in what is proved and what is not. Some presented features are only available in recent versions of Frama-C.

The only requirement to this tutorial is to have a basic knowledge of the C language, and at least to be familiar with the notion of pointer.

Despite its old age, C is still a widely used programming language. Indeed, no other language can pretend to be available on so many (hardware and software) platforms, its low-level orientation and the amount of time invested in the optimization of its compilers allows generating very light and efficient machine code (if the code allows it of course), and that there are a lot of experts in C language, which is an important knowledge base.

Furthermore, a lot of systems rely on a huge amount of code historically written in C, that needs to be maintained and sometimes fixed, as it would be far too costly to rewrite these systems.

But anyone who has already developed with C also knows that it is very hard to perfectly master this language. There are numerous reasons, but the complexity of the ISO C, and the fact that it is extremely permissive, especially about memory management, make the development of robust C program very hard, even for an experienced programmer.

However, the C language is often chosen for critical systems (avionics, railway, armament, ...) where it is appreciated for its good performances, its technological maturity and the predictability of its compilation.

In such cases, the needs in terms of test coverage of the source code become extremely high. Thus, the question “is our software tested enough?” becomes a question to which it is very hard to answer. Program proof can help us. Rather than testing all possible and (un)imaginable inputs of the program, we will *statically* and *mathematically* prove that there cannot be any problem at runtime.

The goal of this tutorial is to use Frama-C, a tool developed at CEA LIST, and WP, its deductive proof plugin, to learn the basics about C program proof. More than the use of the tool itself, the goal of this tutorial is to convince that it is more and more possible to write programs without any programming error, but also to sensitize to simple notions that allows to better understand and write programs.

Many thanks to the different beta-testers for their constructive feedback:

- Taurre
- barockobamo
- Vayel
- Aabu

I thank ZesteDeSavoir validators who helped me improve again the quality of this tutorial:

- Taurre (again)
- Saroupille
- Aabu (again)

Many thanks to Jens Gerlach for his help during the translation of this tutorial from French to English.
Thanks, finally, to the occasional reviewers on GitHub:

- Alex Lyr
- Rafael Bachmann
- @charlesseizilles
- Myriam Clouet
- @Costava
- Daniel Rocha
- @GaoTamanrasset
- André Maroneze
- @MSoegtropIMC
- @rtharston
- @TrigDevelopment
- Quentin Santos
- Ricardo M. Correia
- Basile Desloges

for their reviews and fixes.

2. Program proof and our tool for this tutorial: Frama-C

The goal of this first part is, in the first section, to introduce the idea of program proof without giving too many details, and then, in the second section, to give the necessary instructions to install Frama-C and some automatic provers that we will use in this tutorial.

2.1. Program proof

2.1.1. Ensure program reliability

It is often difficult to ensure that our programs have only correct behaviors. Moreover, it is already complex to establish good criteria that make us confident enough to say that a program works correctly:

- beginners simply "try" to use their programs and consider that these programs work if they do not crash (which is not an excellent indicator in C language),
- more advanced developers establish some test cases for which they know the expected result and compare the output they obtain,
- most companies establish complete test bases, that cover as much code as they can ; which are systematically executed on their code. Some of them apply test driven development,
- in critical domains, such as aerospace, railway or armament, source code needs to be certified using standardized processes with very strict criteria about coding rules and code covering by the test.

In all these ways to ensure that a program produces only what we expect it to produce, a word appears to be common: *test*. We *try* different inputs in order to isolate cases that are problematic. We provide inputs that we *estimate to be representative* of the actual use of the program (note that unexpected use cases are often not considered whereas there are generally the most dangerous ones) and we verify that the results we get are correct. But we cannot test *everything*. We cannot try *every* combination of *every* possible input of a program. It is then quite hard to choose good tests.

The goal of program proof is to ensure that, for any input provided to a given program, if it conforms to the specification, then the program will only well-behave. However, since we cannot test everything, we will formally, mathematically, establish a proof that our software can only exhibit specified behaviors, and that runtime-errors are not part of these behaviors.

A very well-known quote from Dijkstra precisely express the difference between test and proof:

> Program testing can be used to show the presence of bugs, but never to show their absence!
>
> *Dijkstra*

2.1.1.1. The developer's Holy Grail: the bug-free software

Every time we read news about attacks on computer systems, or viruses, or bugs leading to crashes in well known apps, there is always the one same comment "the bug-free/perfectly secure program does not exist". And, if this sentence is quite true, it is a bit misunderstood.

First, we do not really define what we mean by "bug-free". Creating software always relies at least on two steps: we establish a specification of what we expect from the program, and then we produce the source code of the program that must respect this specification. Moreover, we can add that our programming language itself defines what is the correct way to use it. On each of these aspects, an error can lead to the introduction of bugs that we can classify into three categories:

- the program is not correct, or does not have a defined behavior, according to the language specification (for example, a program accesses out of the bounds of an array when searching the index of the minimal value of the array),
- the program is not correct according to the specification that we defined for it (for example, we have defined that our program must find the index of the minimal value of an array, but in fact, due to an error, it does not check the last value of the array),
- the specification does not exactly reflect what we had in mind for our function, and consequently, the program does not do what we really wanted (for example, we have defined that our function must find the index of the minimal value of the array, but we have not specified that if there are multiple ones, we need the position of the first one, because it seemed too straightforward, but consequently, this is not what the program does).

Each of these categories can affect the safety and/or the security of our programs, but these two notions are not equivalent. Loosely speaking, in security a malicious entity that can attack the system, while in safety, we want to verify when used in a correct way, the system well behaves. Thus, safety is a first before we can get security[1].

In this tutorial, we will show how we can prove that implementations of our programs do not contain bugs related to the two first categories defined above, meaning that they conform to:

- the specification of our language,
- the specification that we have provided for them.

[1] Depending on the field you are working in, the term "safety" may have a very different sense. Namely, a *safe* system must be a system that can never cause injuries to human beings. And thus in this case, the opposite applies: we can never have safety if security is not guaranteed before. In this tutorial, we keep the definition: "safety = the program as a correct behavior when it is used in a correct way"

But what are the arguments of program proof, compared to program testing? First, the proof is complete, it cannot forget some corner case if the behavior is specified (program test cannot be complete, being exhaustive would be far too costly). Second, the obligation to formally specify with a logic formulation requires to exactly understand what we have to prove about our program.

One could cynically say that program proof shows that "the implementation does not contain bugs that do not exist in the specification" as we do not treat the third category of bugs defined above. But, well, knowing that "the implementation does not contain bugs that do not exist in the specification" is already a big step compared to knowing that "the implementation does not contain too many bugs that do not exist in the specification", as it already correspond to two entire categories of bugs that we rule out, bugs that can severely compromise the safety and security of our programs. And moreover, there also exist approaches to deal with the third category of bugs, allowing to analyze specifications to find errors or under-specified behaviors. For example, with model checking techniques, we can create an abstract model from the specification and produce the set of states that can be reached according to this model. By characterizing what is an error state, we can determine if reachable states are error states.

2.1.2. A bit of context

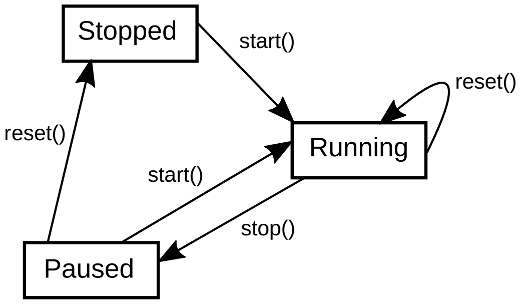
Formal methods, as we name them, allow in computer science to rigorously, mathematically, reason about programs. There exist a lot of formal methods that can take place at different levels from program design to implementation, analysis and validation, and for all systems that allow to manipulate information.

Here, we will focus on a method that allows to formally verify that our programs have only correct behaviors. We will use tools that are able to analyze a source code and to determine whether a program correctly implements what we want to express. The analyzer we will use provides a static analysis, that we can oppose to dynamic analysis.

In static analysis, the analyzed program is not executed. We reason on a mathematical model of the states it can reach during its execution. On the opposite, dynamic analyses such as program testing, require to execute the analyzed source code. Note that there exist formal dynamic analysis methods, for example automatic test generation, or code monitoring techniques that allow to instrument a source code to verify some properties about it during execution (correct memory use, for example).

Talking about static analyses, the model we use can be more or less abstract depending on the techniques, this is always an approximation of possible states of the program. The more the approximation is precise, the more the model is concrete, the more the approximation is vague, the more it is abstract.

To illustrate the difference between concrete and abstract model, we can have a look at the model of a simple chronometer. A very abstract model of a chronometer could be the one presented here:



We have a model of the behavior of our chronometer with the different states it can reach according to the different actions we can perform. However, we have not modeled how these states are implemented in the program (Is this a C enumeration? A particular program point in the source code?), nor how the computation of the elapsed time is done (A single variable? Multiple ones?). It would then be difficult to specify properties about our program. We could add some information:

- State stopped at 0 : time = 0s
- State running : time > 0s
- State stopped : time > 0s

Which gives us a more concrete model, but that is still not precise enough to ask interesting questions like: "is it possible for the program to continue updating the time variable while the chronometer is on the Stopped state?", as we do not model how the time measurement is updated by the chronometer.

On the opposite, with the source code of the program, we have a concrete model of the chronometer. The source code expresses the behavior of the chronometer since it will allow us to produce the executable. But this is still not the more concrete model! For example, the executable in machine code format, that we obtain after compilation, is far more concrete than our program.

The more a model is concrete, the more it precisely describes the behavior of our program. The source code more precisely describes the behavior than our diagram, but it is less precise than the machine code. However, the more the model is precise, the more it is difficult to have a global view of the defined behavior. Our diagram is understandable in the blink of an eye, the source code requires more time, and for the executable … Every single person that has already opened an executable with a text editor by error knows that it is not really pleasant to read[1].

When we create an abstraction of a system, we approximate it, in order to limit the knowledge we have about it and make our reasoning easier. A constraint we must respect, if we want our analysis to be correct, is to never under-approximate behaviors: we would risk removing a behavior that contains an error (and thus miss it during the analysis). However, when we over-approximate the behaviors of the program, we can add behaviors that cannot happen, and if we add too many of them, we could not be able to prove that our program is correct, since some of them could be faulty behaviors.

In our case, the model is quite concrete. Every type of instruction, of control structure, is associated to a precise semantics, a model of its behavior in a pure logic, mathematical, world.

The logic we use here is a variant of the Hoare logic, adapted to the C language and all its complex subtleties (which makes this model concrete).

2.1.3. Hoare triples

Hoare logic is a program formalization method proposed by Tony Hoare in 1969 in a paper entitled *An Axiomatic Basis for Computer Programming*. This method defines:

- axioms, that are properties we admit, such as "the skip action does not change the program state",
- rules to reason about the different allowed combinations of actions, for example "the skip action followed by the action A" is equivalent to "the action A".

The behavior of the program is defined by what we call "Hoare triples":

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

Where P and Q are predicates, logic formulas that express properties about the memory at particular program points. C is a list of instructions that defines the program. This syntax expresses the following idea: "if we are in a state where P is verified, after executing C and if C terminates, then Q is verified for the new state of the execution". Put in another way, P is a sufficient precondition to ensure that C will bring us to the postcondition Q. For example, the Hoare triples that corresponds to the skip action is the following one:

$$\{P\}\ \textbf{skip}\ \{P\}$$

When we do nothing, the postcondition is the precondition.

Along this tutorial, we will present the semantics of different program constructs (conditional blocks, loops, etc.) using Hoare logic. So, let us skip these details now since we will work on it later. It is not necessary to memorize these notions nor to understand all the theoretical background, but it is still useful to have some ideas about the way our tool works.

All of this gives us the basics that allow us to say "here is what this action does" but it does not give us anything to mechanize a proof. The tool we will use rely on a technique called weakest precondition calculus.

[1] There also exists formal methods which are interested in understanding how executable machine code work, for example in order to understand what malware does or to detect security breaches introduced during compilation.

2.1.4. Weakest precondition calculus

The weakest precondition calculus is a form of predicate transform semantics proposed by Dijkstra in 1975 in *Guarded commands, non-determinacy and formal derivation of programs*.

This title can appear complex, but actually the content of the article is in fact quite simple. We have seen before that Hoare logic gives us rules that explain the behavior of the different actions of a program, but it does not say how to apply these rules to establish a complete proof of the program.

Dijkstra reformulates the Hoare logic by explaining, in the triple $\{P\}\ C\ \{Q\}$, how the instruction, or the block of instructions, C transforms the predicate P in Q. This kind of reasoning is called *forward-reasoning*. We calculate from the precondition and from one or multiple instructions, the strongest postcondition we can reach. Informally, considering what we have in input, we calculate what we will get in output. If the postcondition we want is as strong or weaker, then we prove that there are no unexpected behaviors.

For example:

```c
int a = 2;
a = 4;
//calculated postcondition: a == 4
//expected postcondition  : 0 <= a <= 30
```

Ok, 4 is an allowed value for `a`.

The form of predicate transformer semantics which we are interested in works the opposite way, we speak about *backward-reasoning*. From the wanted postcondition and the instructions we are reasoning about, we find the weakest precondition that ensures this behavior. If our actual precondition is at least as strong, that is to say, if it implies the computed precondition, then our program is correct.

For example, if we have the instruction:

$\{P\}\ x := \mathrm{a}\ \{x = 42\}$

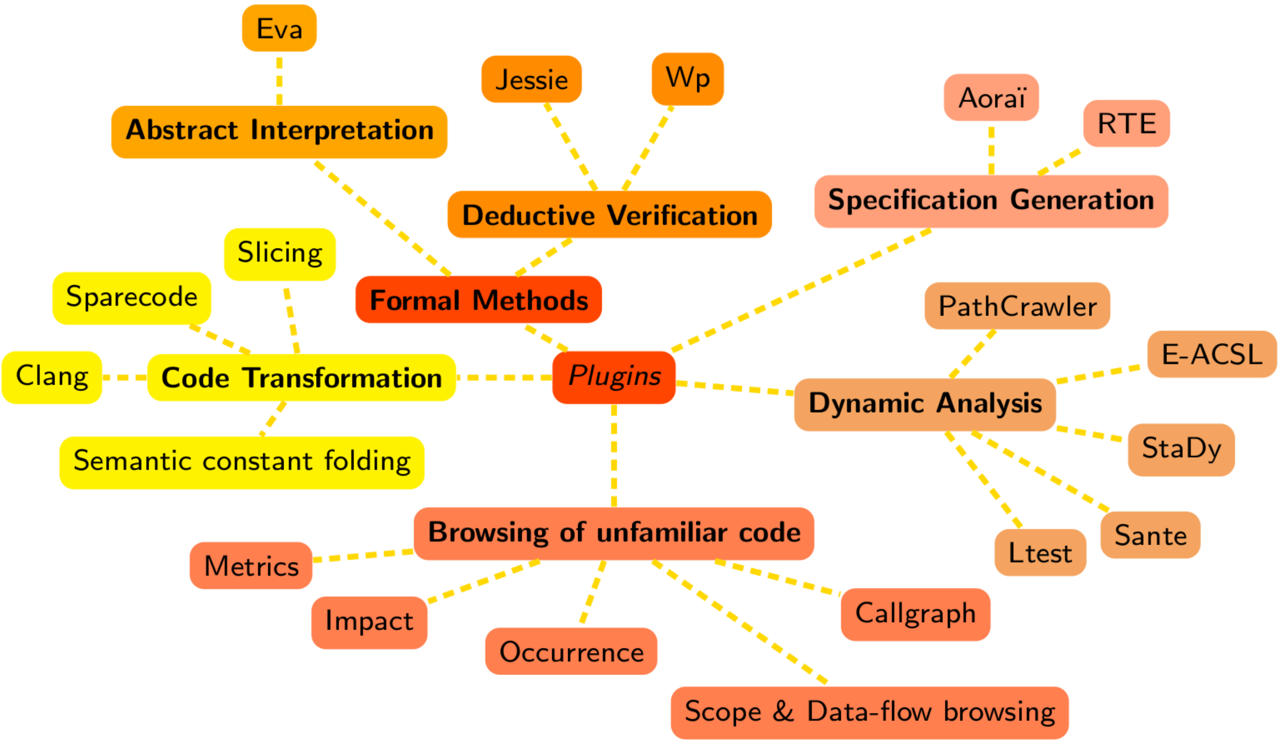
What is the weakest precondition to validate the postcondition $\{x = 42\}$? The rule will define that P is $\{\mathrm{a} = 42\}$.

For now, let us forget about it, we will come back to these notions as we use them in this tutorial to understand how our tools work. So now, we can have a look at these tools.

2.2. Frama-C

2.2.1. Frama-C? WP?

Frama-C (FRAmework for Modular Analysis of C code) is a platform dedicated to the analysis of C programs created by the CEA LIST and Inria. It is based on a modular architecture allowing to use different plugins. The default plugins comprise different static analyses (that do not execute source code), dynamic analyses (that requires code execution), or combining both. These plugins can collaborate together or not, either by communicating directly or by using the specification language provided by Frama-C.



This specification language is called ACSL (“Axel”) for ANSI C Specification Language and allows us to express the properties we want to verify about our programs. These properties are written using code annotations in comment sections. If one has already used Doxygen, it is quite similar, except that we write logic formulas and not text. During this tutorial, we will extensively write ACSL code, so let us just skip this for now.

The analysis we will use in this tutorial is provided by the WP plugin (for Weakest Precondition), a deductive verification plugin. It implements the technique we mentioned earlier: from ACSL annotations and the source code, the plugin generates what we call verification conditions, that are logic formulas that must be verified to be satisfiable or not. This verification can be performed manually or automatically, here we use automatic tools.

We will use a SMT solver (satisfiability modulo theory , we do not detail how it works). This solver is Alt-Ergo , that was initially developed by the Laboratoire de Recherche en Informatique d’Orsay, and is today maintained by OCamlPro.

2.2.2. Installation

Frama-C is a software developed on Linux and macOS. Its support is thus better on those operating system. Nevertheless, it is possible to install Frama-C on Windows and in theory, its use will be identical to its use on Linux. However:

- the tutorial presents the use of Frama-C on Linux (or macOS) and the author did not experiment the differences that could exist with Windows,
- in recent versions of Windows 10, a possibility is to use Windows Subsystem for Linux, in combination with a Xserver installed on Windows to get GUI,
- the "Bonus" section of this part could not be accessible on Windows.

2.2.2.1. Linux

2.2.2.1.1. Using package managers On Debian, Ubuntu and Fedora, there exist packages for Frama-C. In such a case, it is enough to type a command like:

```
apt-get/yum install frama-c
```

However, these repositories are not necessarily up to date with the latest version of Frama-C. Most of the tutorial should be usable for the version available on your Linux distribution if it is recent enough. However, some specific features could be missing.

Go to the section "Verify installation" to perform some tests about the installation.

2.2.2.1.2. Via Opam A second option is to use Opam, a package manager for Ocaml libraries and applications.

First, Opam must be installed (see its documentation). Then, some packages from the Linux distribution must be installed before installing Frama-C. On most systems, it is possible to ask Opam to install dependencies of the packages we want to install. For this, first install the depext tool of Opam:

```
opam install depext
```

And then use the depext tool to install the dependencies of Frama-C:

```
opam depext frama-c
```

If the depext tool does not support your distribution, the following libraries and software should be installed:

- GTK2 (development library)
- GTKSourceview 2 (development library)
- GnomeCanvas 2 (development library)
- autoconf

On recent versions of some distributions, GTK2 might not be available, in this case, or if you want GTK3 and not GTK2, the packages GTK2, GTKSourceview2 and GnomeCanvas2 must be replaced with GTK3 and GTKSourceview 3.

Once these packages are installed, we can install Frama-C and Alt-Ergo.

```
opam install frama-c
opam install alt-ergo
```

Note that for versions older than Potassium (Argon and before), if you want to use the Why3 platform as a backend (introduced later), the version 0.88.3 should be used for Why3.

Go to the section “Verify installation” to perform some tests about the installation.

2.2.2.1.3. Via “manual” compilation The packages we have listed in the Opam section are required (of course, Opam itself is not). It requires a recent version of Ocaml and its compiler (including a compiler to native code). It is also necessary to install Why3 (at least version 1.2.0), which is available either on Opam or on their website (Why3).

After having extracted the folder available here : https://frama-c.com/html/get-frama-c.html (Source distribution). Navigate to the folder and then execute the command line:

```
autoconf && ./configure && make && sudo make install
```

Go to the section “Verify installation” to perform some tests about the installation.

2.2.2.2. macOS

On macOS, the use of Homebrew and Opam is recommended to install Frama-C. The author does not use macOS, so here is a shameful copy and paste of the installation guide of Frama-C for macOS.

General macOS tools for OCaml:

```
> xcode-select --install
> open http://brew.sh
> brew install autoconf opam
```

Graphical User Interface:

```
> brew install gtk+ --with-jasper
> brew install gtksourceview libgnomecanvas graphviz
> opam install lablgtk ocamlgraph
```

Recommended for Frama-C:

```
> brew install gmp
> opam install zarith
```

Necessary for Frama-C/WP:

```
> opam install alt-ergo
> opam install frama-c
```

2.2.2.3. Windows

Currently, the best way to install Frama-C on Windows is to use the Windows Subsystem for Linux. Basically, once the Linux system is installed, one should install Opam and follow the instructions provided in the Linux section. To get a usable graphical user interface, an X server should be installed on Windows.

2.2.3. Verify installation

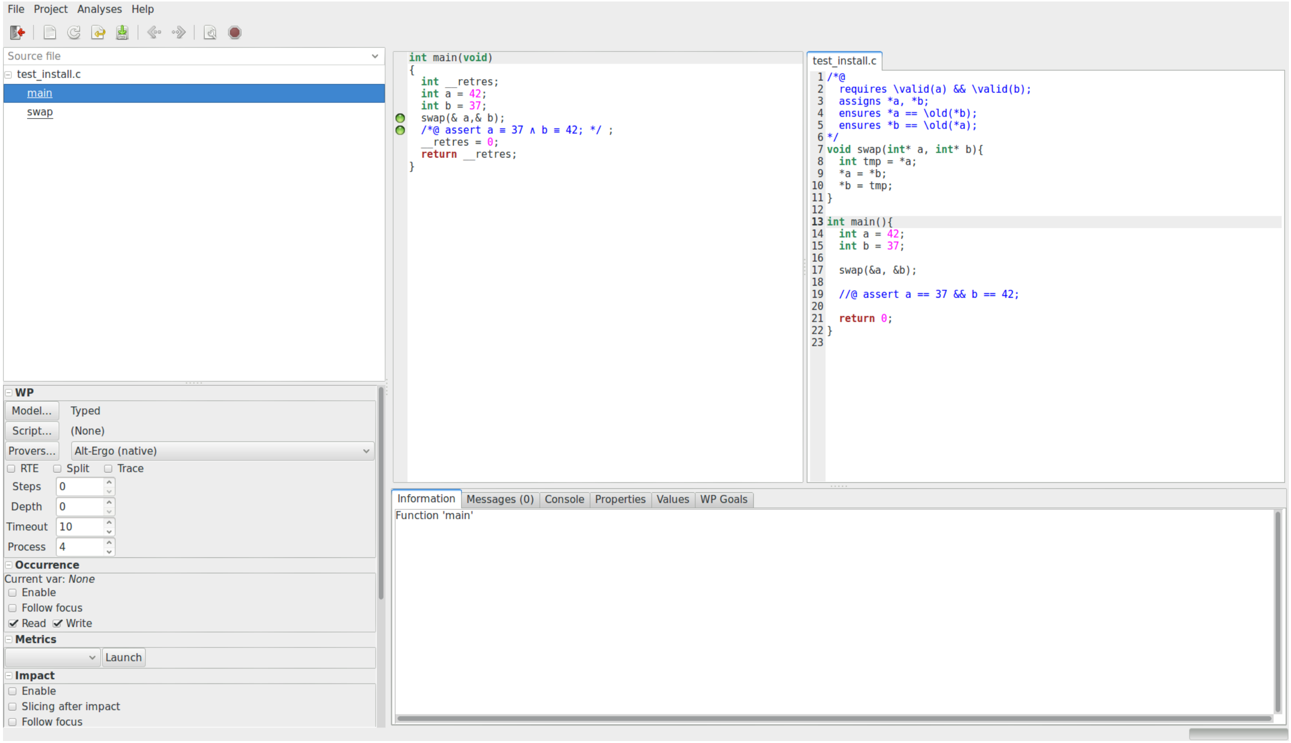
In order to verify that the installation has been correctly performed, we will use the following code:

```c
/*@
  requires \valid(a) && \valid(b);
  assigns *a, *b;
  ensures *a == \old(*b);
  ensures *b == \old(*a);
*/
void swap(int* a, int* b){
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(){
  int a = 42;
  int b = 37;

  swap(&a, &b);

  //@ assert a == 37 && b == 42;

  return 0;
}
```

Copy and paste this code in a file named `main.c`. Then, from a terminal, in the folder where the file has been created, we start Frama-C with the following command line:

```
frama-c-gui -wp -rte main.c
```

The following window should appear:



```
int main(void)
{
  int __retres;
  int a = 42;
  int b = 37;
  swap(& a,& b);
  /*@ assert a ≡ 37 ∧ b ≡ 42; */ ;
  __retres = 0;
  return __retres;
}
```

```
/*@
  requires \valid(a) && \valid(b);
  assigns *a, *b;
  ensures *a == \old(*b);
  ensures *b == \old(*a);
*/
void swap(int* a, int* b){
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(){
  int a = 42;
  int b = 37;

  swap(&a, &b);

  //@ assert a == 37 && b == 42;

  return 0;
}

```

WP
Model... Typed
Script... (None)
Provers... Alt-Ergo (native)
□ RTE □ Split □ Trace
Steps 0
Depth 0
Timeout 10
Process 4

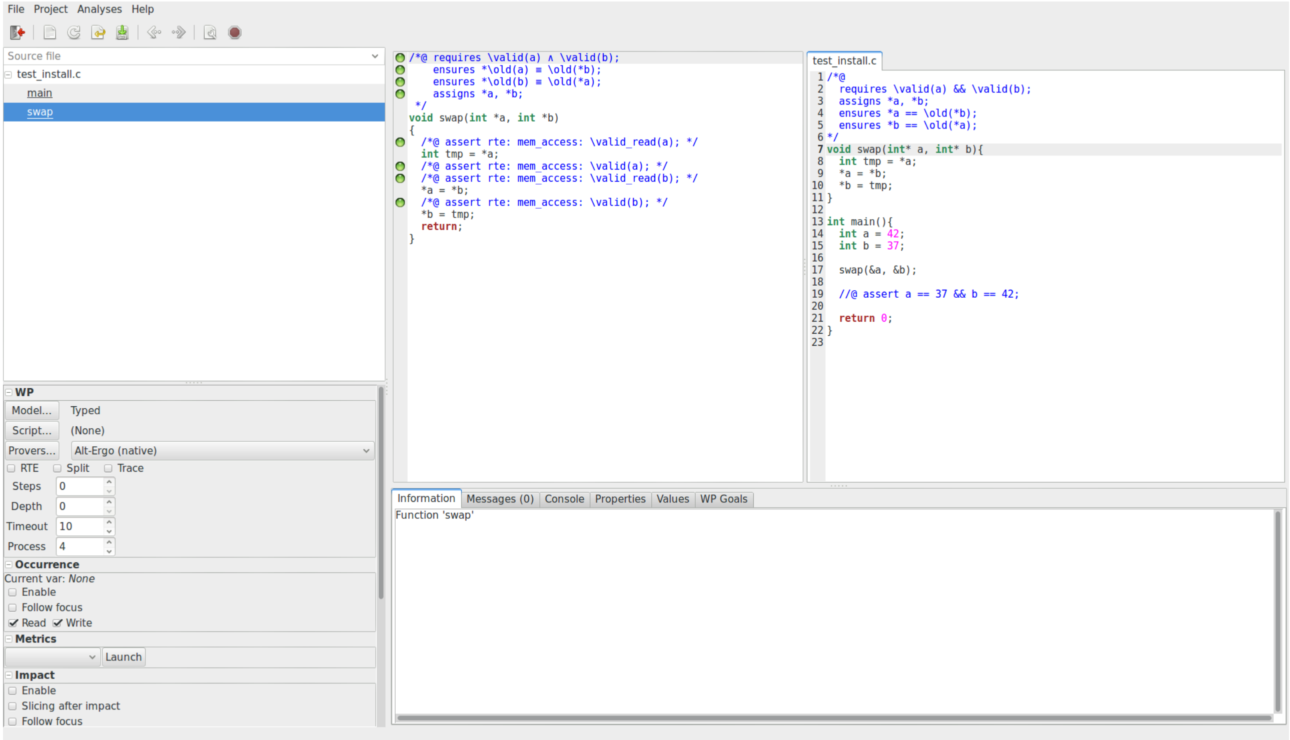
Occurrence
Current var: *None*
□ Enable
□ Follow focus
☑ Read ☑ Write

Metrics
Launch

Impact
□ Enable
□ Slicing after impact
□ Follow focus

Function 'main'

Clicking `main.c` in the left side-panel to select it, we can see its content (slightly) modified, and some green bullets on different lines as illustrated here:



Model... Typed
Script... (None)
Provers... Alt-Ergo (native)
□ RTE □ Split □ Trace
Steps 0
Depth 0
Timeout 10
Process 4

Current var: *None*
□ Enable
□ Follow focus
☑ Read ☑ Write

□ Enable
□ Slicing after impact
□ Follow focus

```
/*@ requires \valid(a) ∧ \valid(b);
    ensures *\old(a) ≡ \old(*b);
    ensures *\old(b) ≡ \old(*a);
    assigns *a, *b;
 */
void swap(int *a, int *b)
{
  /*@ assert rte: mem_access: \valid_read(a); */
  int tmp = *a;
  /*@ assert rte: mem_access: \valid(a); */
  /*@ assert rte: mem_access: \valid_read(b); */
  *a = *b;
  /*@ assert rte: mem_access: \valid(b); */
  *b = tmp;
  return;
}
```

test_install.c

```
/*@
  requires \valid(a) && \valid(b);
  assigns *a, *b;
  ensures *a == \old(*b);
  ensures *b == \old(*a);
*/
void swap(int* a, int* b){
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(){
  int a = 42;
  int b = 37;

  swap(&a, &b);

  //@ assert a == 37 && b == 42;

  return 0;
}

```

Function 'swap'

If this does not succeed, check that graphic libraries were not forgotten during installation, or the Alt-Ergo solver. If everything seems to be OK, forums can help (for example: StackOverflow - Frama-C ↗)

The graphical user interface of Frama-C does not allow source code edition.

For color blinds, it is possible to start Frama-C with another theme where color bullets are replaced:

```
frama-c-gui -gui-theme colorblind
```

2.2.4. (Bonus) Some more provers

This part is optional, nothing in this section should be particularly useful *in the tutorial*. However, when we start to be interested in proving more complex programs, it is often possible to reach the limits of Alt-Ergo, which is available by default, and we would thus need some other provers. For basic properties, almost all solvers have the same capabilities, for more complex ones, each solver has its predilection domains.

2.2.4.1. Why3

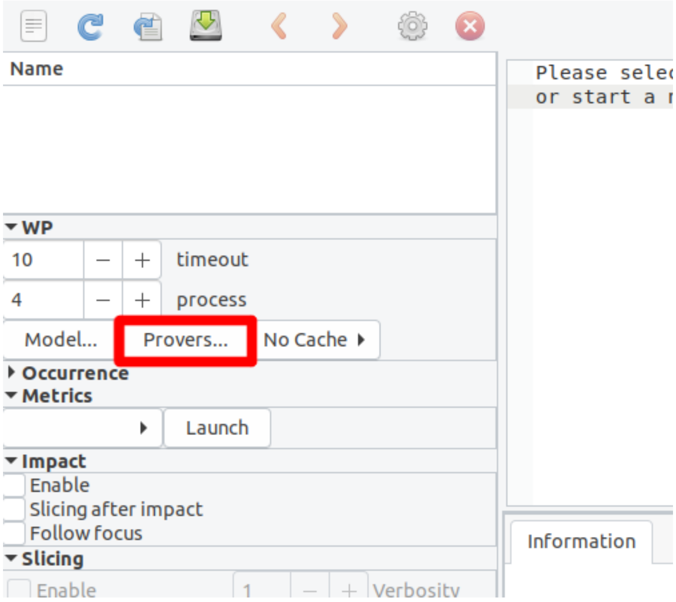
Why3 is a deductive proof platform developed by the LRI in Orsay. It provides a programming language and a specification language, as well as a module that allows to interact with a wide variety of automatic and interactive provers. This point is the one that interests us here. WP uses Why3 as a backend to talk with external provers.

On their website, we can find the list of supported provers . We recommend installing Z3 which is developed by Microsoft Research, and CVC4 which is developed by many research teams (New York University, University of Iowa, Google, CEA List). Those two provers are very efficient and somewhat complementary.

New provers can be installed any time after the installation of Frama-C. However, Why3 provers list must be updated:

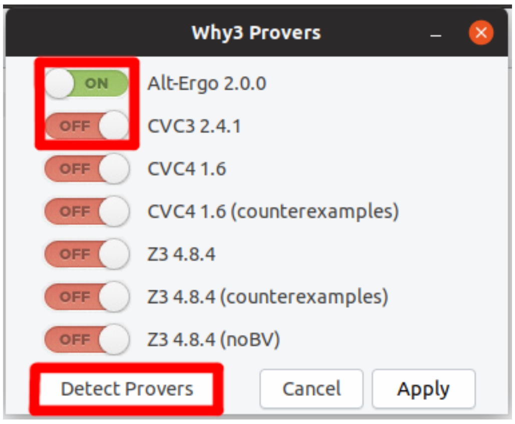
```
why3 config detect
```

And then activated in Frama-C. In the left-side panel, in the WP part, click "Provers...":



Please selec
or start a r

And then "Detect" in the window that appears. Once it is done, provers can be activated thanks to the button next to their name.



ON Alt-Ergo 2.0.0

OFF CVC3 2.4.1

OFF CVC4 1.6

OFF CVC4 1.6 (counterexamples)

OFF Z3 4.8.4

OFF Z3 4.8.4 (counterexamples)

OFF Z3 4.8.4 (noBV)

2.2.4.2. Coq

Coq, which is developed by Inria, is a proof assistant. Basically, we write the proofs ourselves in a dedicated language and the assistant verifies (using typing) that the proof is actually a valid proof.

Why would we need such a tool? Sometimes, the properties we want to prove can be too complex to be solved automatically by SMT solvers, typically when they require careful inductive reasoning with precise choices at each step. In this situation, WP allows us to generate verification conditions translated in Coq language, and to write the proof ourselves.

To learn Coq, we would recommend this tutorial .

> If Frama-C has been installed using the package manager of a Linux distribution, Coq could be automatically installed.

If one needs more information about Coq and its installation, this page can help: The Coq Proof Assistant .

Our tools are now installed and ready to be used.

The goal of this part, apart from the installation of our tools was to put in relief two main ideas:

- program proof is a way to ensure without executing them, that our programs only have correct behaviors, that are described by our specification,
- it is still our work to ensure that this specification is correct.

3. Function contract

It is time to enter the heart of the matter. Rather than starting with the basic notions of the C language, as we would do for a tutorial about C, we will start with functions. First because it is necessary to be able to write functions before starting this tutorial (to be able to prove that a code is correct, being able to write it correct is required), and then because it will allow us to directly prove some programs.

After this part about functions, we will on the opposite focus on simple notions like assignments or conditional structures, to understand how our tool really works.

In order to be able to prove that a code is valid, we first need to specify what we expect of it. Building the proof of our program consists in ensuring that the code we wrote corresponds to the specification that describes its job. As we previously said, Frama-C provides the ACSL language to let the developer write contracts about each function (but that is not its only purpose, as we will see later).

3.1. Contract definition

The goal of a function contract is to state the properties of the input that are expected by the function, and in exchange the properties that will be assured for the output. The expectation of the function is called the **precondition**. The properties of the output are called the **postcondition**.

These properties are expressed with ACSL. The syntax is relatively simple if one has already developed in C language since it shares most of the syntax of boolean expressions in C. However, it also provides:

- some logic constructs and connectors that do not exist in C, to ease the writing of specifications,
- built-in predicates to express properties that are useful about C programs (for example: a valid pointer),
- as well as some primitive types for the logic that are more general than primitive C types (for example: mathematical integer).

We will introduce along this tutorial a large part of the notations available in ACSL.

ACSL specifications are introduced in our source code using annotations. Syntactically, a function contract is integrated in the source code with this syntax:

```
/*@
  //contract
*/
void foo(int bar){

}
```

Notice the `@` at the beginning of the comment block, this indicates to Frama-C that what follows are annotations and not a comment block that it should simply ignore.

Now, let us have a look at the way we express contracts, starting with postconditions, since it is what we want our function to do (we will later see how to express precondition).

3.1.1. Postcondition

The postcondition of a function is introduced with the `ensures` clause. Let us illustrate its use with the following function that returns the absolute value of an input value. One of its postconditions is that the result (which is denoted with the keyword `\result`) is greater or equals to 0.

```
/*@
  ensures \result >= 0;
*/
int abs(int val){
  if(val < 0) return -val;
  return val;
}
```

(Notice the `;` at the end of the line).

But this it is not the only property to verify. We also need to specify the general behavior of a function returning the absolute value. That is: if the value is positive or 0, the function returns the same value, else it returns the opposite of the value.

We can specify multiple postconditions, first by combining them with an operator `&&` as we do in C, or by introducing a new `ensures` clause, as we illustrate here:

```
/*@
  ensures \result >= 0;
  ensures (val >= 0 ==> \result == val ) &&
          (val <  0 ==> \result == -val);
*/
int abs(int val){
  if(val < 0) return -val;
  return val;
}
```

This specification is the opportunity to present a very useful logic connector provided by ACSL and that does not exist in C: the implication $A \Rightarrow B$, that is written `A ==> B` in ACSL. The truth table of the implication is the following:

A	B	$A \Rightarrow B$
F	F	T
F	T	T
T	F	F
T	T	T

That means that an implication $A \Rightarrow B$ is true in two cases: either A is false (and in this case, we do not check the value of B), or A is true and then B must also be true. Note that it means that $A \Rightarrow B$ is equivalent to $\neg A \vee B$. The idea finally being "I want to know if when A is true, B also is. If A is false, I don't care, I consider that the complete formula is true". For example, "if it rains, I want to check that I have an umbrella, if it does not, I do not care, everything is fine".

Another available connector is the equivalence $A \Leftrightarrow B$ (written `A <==> B` in ACSL), and it is stronger. This is the conjunction of the implication in both ways $(A \Rightarrow B) \wedge (B \Rightarrow A)$. This formula is true in only two cases: A and B are both true, or both false (it can be seen as the negation of the exclusive or). Continuing with our example, "I do not only want to know that I have an umbrella when it rains, I also want to be sure that I have one only when it rains".

Let us give a quick reminder about all truth tables of usual logic connectors in first order logic ($\neg =$ `!`, $\wedge =$ `&&`, $\vee =$ `||`) :

A	B	$\neg A$	$A \wedge B$	$A \vee B$	$A \Rightarrow B$	$A \Leftrightarrow B$
F	F	T	F	F	T	T
F	T	T	F	T	T	F
T	F	F	F	T	F	F
T	T	F	T	T	T	T

We can come back to our specification. As our files become longer and contains a lot of specifications, it can be useful to name the properties we want to verify. So, in ACSL, we can specify a name (without spaces) followed by a `:` character, before stating the property. It is possible to put multiple levels of names to categorize our properties. For example, we could write this:

```c
/*@
  ensures positive_value: function_result: \result >= 0;
  ensures (val >= 0 ==> \result == val) &&
          (val < 0 ==> \result == -val);
*/
int abs(int val){
  if(val < 0) return -val;
  return val;
}
```

In most of this tutorial, we will not name the properties we want to prove, since they will be generally quite simple, and we will not have too many of them, names would not give us much information.

We can copy and paste the function `abs` and its specification in a file `abs.c` and use Frama-C to determine if the implementation is correct with respect to the specification. We can start the GUI of Frama-C (it is also possible to use the command line interface of Frama-C, but we will not use it during this tutorial) by using this command line:

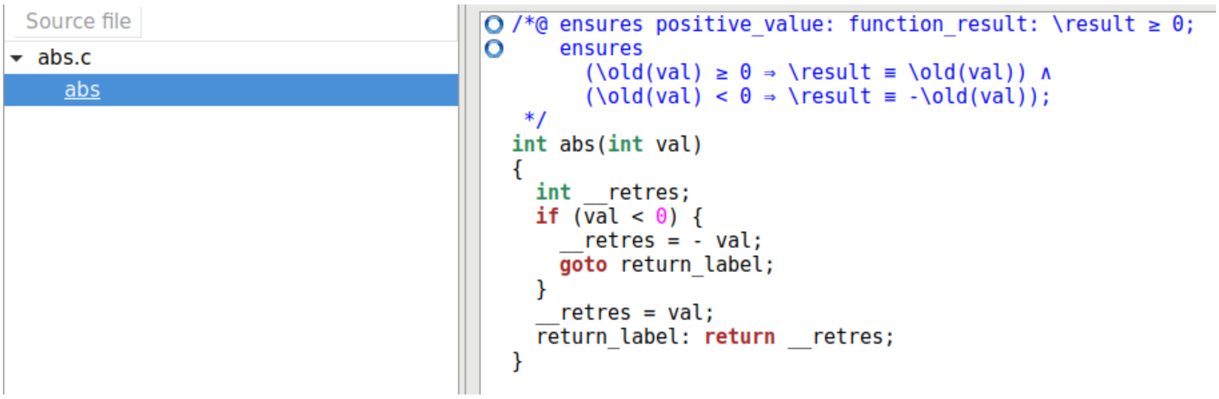
```
frama-c-gui
```

Or by opening it from the graphical environment.

It is then possible to click on the button “Create a new session from existing C files”, files to analyze can be selected by double-clicking it, the OK button ending the selection. Then, adding other files is done by clicking Files > Source Files.

Notice that it is also possible to directly open file(s) from the terminal command line passing them to Frama-C as parameter(s):

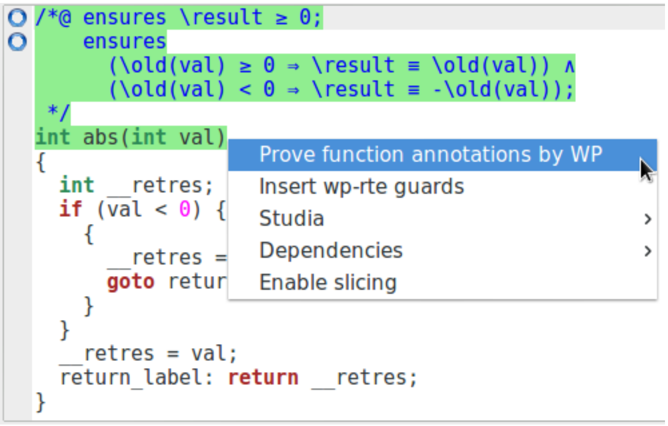
```
frama-c-gui abs.c
```



```
/*@ ensures positive_value: function_result: \result ≥ 0;
    ensures
      (\old(val) ≥ 0 ⇒ \result ≡ \old(val)) ∧
      (\old(val) < 0 ⇒ \result ≡ -\old(val));
 */
int abs(int val)
{
  int __retres;
  if (val < 0) {
    __retres = - val;
    goto return_label;
  }
  __retres = val;
  return_label: return __retres;
}
```

The window of Frama-C opens and in the panel dedicated to files and functions, we can select the `abs` function. At each `ensures` line, we can see a blue circle, it indicates that no verification has been attempted for these properties.

We ask the verification of the code by right-clicking the name of the function and “Prove function annotations by WP”:



```
/*@ ensures \result ≥ 0;
    ensures
      (\old(val) ≥ 0 ⇒ \result ≡ \old(val)) ∧
      (\old(val) < 0 ⇒ \result ≡ -\old(val));
 */
int abs(int val)
{
  int __retres;
  if (val < 0) {
    {
      __retres =
      goto retur
    }
  }
  __retres = val;
  return_label: return __retres;
}
```

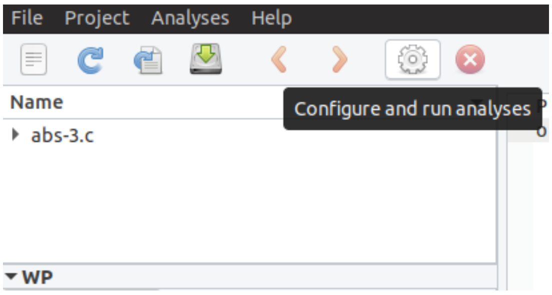
We can see that blue circles become green bullets, indicating that the specification is indeed ensured by the program. Furthermore, we can also prove properties one by one, by right-clicking on them and not on the name of the function. We also see that now, two additional annotations have been generated by Frama-C for our function:

- `exits \false`,
- `terminates \true`.

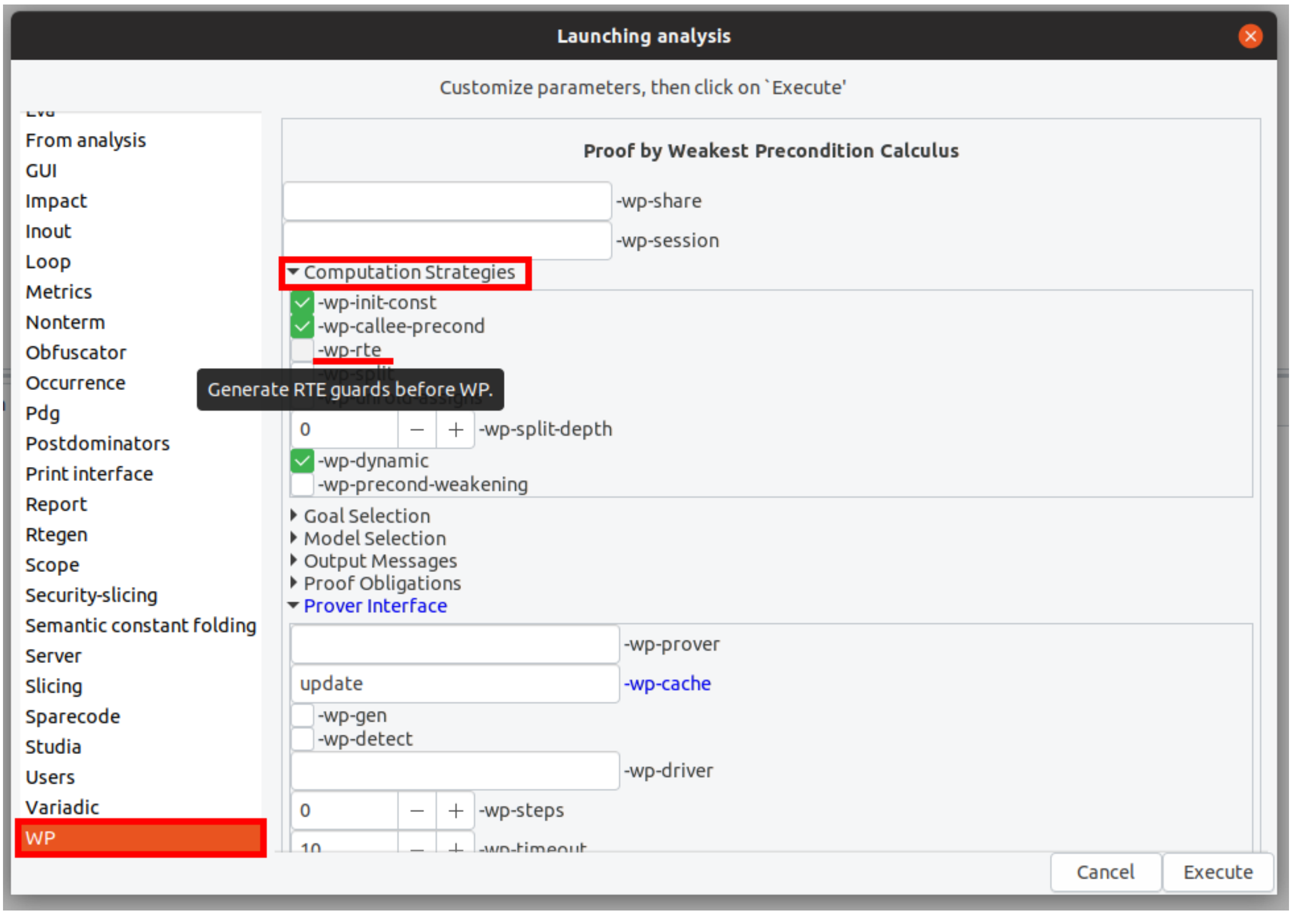
The first one tells that the function must not call the C `exit` function (or a function that might itself transitively call `exit` at some point), and that it will thus normally finish its execution by returning. The second tells that the function must not run forever. These annotations are generated by default, however one can add their own `exits` or `terminates` clause to change this behavior. We will explain this later, for now let us just ignore these annotations.

Is our code really bug free? WP gives us a way to ensure that a code conforms to a specification, but it does not directly check the absence of runtime errors (RTE) if we do not ask it. Another plugin of Frama-C, called RTE, can be used to generate some ACSL annotations that can be verified by the other plugins. Its purpose is to add, in the program, some controls to ensure that the program cannot create runtime errors (integer overflow, invalid pointer dereferencing, 0 division, etc.).

To activate these controls, we must activate this option in the configuration of WP. This is done by first clicking on the plugin configuration button:



And then adding the option `-wp-rte` in the options related to WP:



Launching analysis

Customize parameters, then click on `Execute'

Proof by Weakest Precondition Calculus

-wp-share

-wp-session

▾ Computation Strategies

☑ -wp-init-const

☑ -wp-callee-precond

□ -wp-rte

□ -wp-split

□ -wp-unfold-assigns

Generate RTE guards before WP.

0 − + -wp-split-depth

☑ -wp-dynamic

□ -wp-precond-weakening

-wp-prover

update -wp-cache

□ -wp-gen

□ -wp-detect

-wp-driver

0 − + -wp-steps

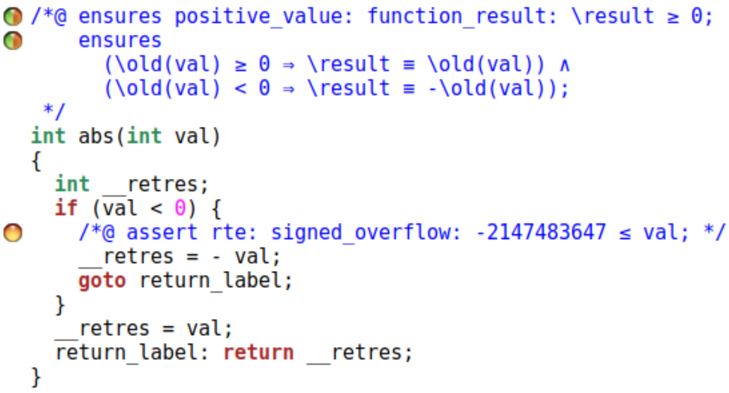
10 − + -wp-timeout

We can also ask WP to add the RTE controls in a function by right-clicking on its name and then click “Insert wp-rte guards”.

By now, `-wp-rte` will always be activated to verify examples except if we explicitly indicate the opposite.

Finally, we execute the verification again (we can also click on the “Reparse” button of the toolbar, it will delete existing proofs).

We can then see that WP fails to prove the absence of arithmetic overflow for the computation of `-val`. And, indeed, on our architectures, -`INT_MIN` (-2^{31}) > `INT_MAX` ($2^{31} - 1$).



```
/*@ ensures positive_value: function_result: \result ≥ 0;
    ensures
      (\old(val) ≥ 0 ⇒ \result ≡ \old(val)) ∧
      (\old(val) < 0 ⇒ \result ≡ -\old(val));
 */
int abs(int val)
{
  int __retres;
  if (val < 0) {
    /*@ assert rte: signed_overflow: -2147483647 ≤ val; */
    __retres = - val;
    goto return_label;
  }
  __retres = val;
  return_label: return __retres;
}
```

> We can notice that the overflow risk is real for us, since our computers (for which the configuration is detected by Frama-C) use the Two's complement implementation of integers, which does not define the behavior of overflows.

Here, we can see another type of ACSL annotation. By the line `//@ assert property ;`, we can ask the verification of a property at a particular program point. Here, the RTE plugin inserted it for us, since we have to verify that `-val` does not produce an overflow, but we can also add such an assertion manually in the source code.

In this screenshot, we can see two new colors for our bullets: green+brown and orange.

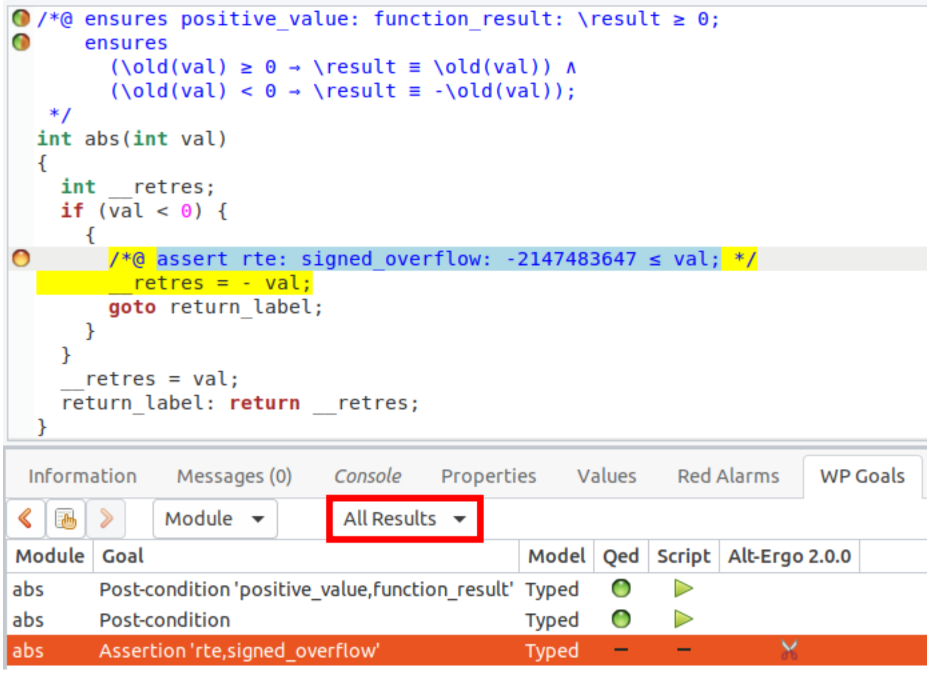
If the proof has not been entirely redone after adding the runtime error checks, these bullets must still be green. Indeed, the corresponding proofs have been realized without the knowledge of the property in the assertion, so they cannot rely on this unproved property.

When WP transmits a verification condition to an automatic prover, it basically transmits two kinds of properties : G, the goal, the property that we want to prove, and A_1 … A_n, the different assumptions we can have about the state of the memory at the program point where we want to verify G. However, it does not receive (in return) the properties that have been used by the prover to validate G. So, if A_3 is an assumption, and if WP did not succeed in getting a proof of A_3, it indicates that G is true, but only assuming that A_3 is true, for which no proof has been established so far.

The orange color indicates that no prover could determine if the property is verified. There are two possibles reasons:

- the prover did not have the information needed to terminate the proof,
- the prover did not have enough time to compute the proof and met a timeout (which can be configured in the WP panel).

In the bottom panel, we can select the “WP Goals” tab, it shows the list of verification conditions, and for each prover the result is symbolized by a logo that indicates if the proof has been tried and if it succeeded, failed or met a timeout. Note that it may require selecting “All Results” in the squared field to see all verification conditions.



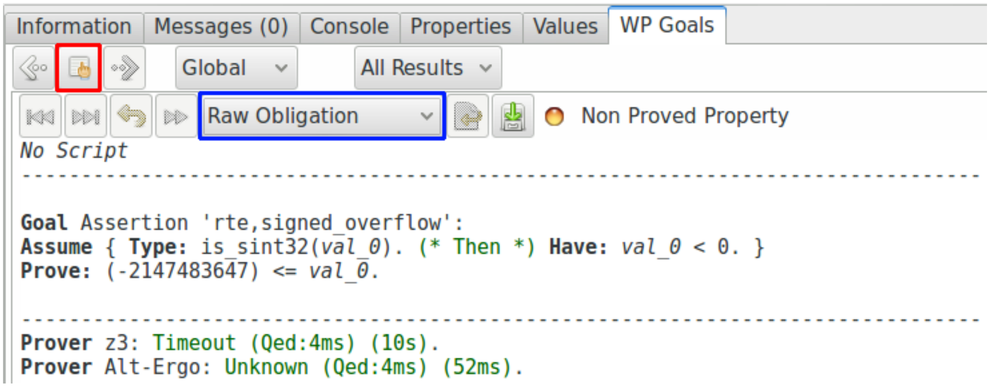
```
/*@ ensures positive_value: function_result: \result ≥ 0;
    ensures
      (\old(val) ≥ 0 → \result ≡ \old(val)) ∧
      (\old(val) < 0 → \result ≡ -\old(val));
 */
int abs(int val)
{
  int __retres;
  if (val < 0) {
    {
      /*@ assert rte: signed_overflow: -2147483647 ≤ val; */
      __retres = - val;
      goto return_label;
    }
  }
  __retres = val;
  return_label: return __retres;
}
```

Module	Goal	Model	Qed	Script	Alt-Ergo 2.0.0
abs	Post-condition 'positive_value,function_result'	Typed	●	▷	
abs	Post-condition	Typed	●	▷	
abs	Assertion 'rte,signed_overflow'	Typed	–	–	✂

In the first column, we have the name of the function the verification condition belongs to. The second column indicates the name of the verification condition. For example here, our postcondition is named `postcondition 'positive_value,function_result'`, we can notice that if we select a property in this list, it is also highlighted in the source code. Unnamed properties are automatically named by WP with the kind of wanted property. In the third column, we see the memory model that is used for the proof, we will not talk about it in this tutorial. Finally, the last columns represent the different provers available through WP.

In the list of provers, the first element is Qed. This is not really a prover. It is a tool used by WP to simplify properties before sending them to external provers. Then we find Script, this is a way to finish some proofs by hand when automatic solvers do not succeed in doing them. Finally, we find Alt-Ergo which is one of the solvers we use. Notice that for the RTE property, some scissors are indicated: it means that the solver was stopped due to timeout.

If we double-click on the property "absence of overflow" (highlighted in blue in the last screenshot), we can see the corresponding verification condition (if it is not the case, make sure that the value "Raw obligation" is selected in the blue squared field):



```
No Script
--------------------------------------------------------------------------------

Goal Assertion 'rte,signed_overflow':
Assume { Type: is_sint32(val_0). (* Then *) Have: val_0 < 0. }
Prove: (-2147483647) <= val_0.

--------------------------------------------------------------------------------
Prover z3: Timeout (Qed:4ms) (10s).
Prover Alt-Ergo: Unknown (Qed:4ms) (52ms).
```

This is the verification condition generated by WP about our property and our program, we do not need to understand everything here, but we can get the general idea. It contains (in the "Assume" part) the assumptions that we have specified and those that have been deduced by WP from the instructions of the program. It also contains (in the "Prove" part) the property that we want to verify.

What does WP do using these properties? In fact, it transforms them into a logic formula and then asks different provers if it is possible to satisfy this formula (to find for each variable, a value that can make the formula true), and it determines if the property can be proved. But before sending the formula to provers, WP uses a module called Qed, which is able to perform different simplifications about it. Sometimes, as this is the case for the other properties about the `abs` function, these simplifications are enough to determine that the property is true, in such a case, WP do not need the help of the automatic solvers.

When automatic solvers cannot ensure that our properties are verified, it is sometimes hard to understand why. Indeed, provers are generally not able to answer something other than "yes", "no" or "unknown", they are not able to extract the reason of a "no" or an "unknown". There exist tools that can explore a proof tree to extract this kind of information, currently Frama-C does not provide such a tool. Reading verification conditions can sometimes be helpful, but it requires a bit of practice to be efficient. Finally, one of the best way to understand the reason why a proof fails is to try to do it interactively with Coq. However, it requires to be quite comfortable with this language to be able to understand the verification conditions generated by WP, since these conditions need to encode some elements of the C semantics that can make them quite hard to read.

If we go back to our view of the verification conditions (see the red squared button in the previous screenshot), we can see that our hypotheses are not sufficient to determine that the property "absence of overflow" is true (which is actually impossible), so we need to add some hypotheses to guarantee that our function will well-behave: a precondition.

3.1.2. Precondition

Preconditions are introduced using `requires` clauses. As we could do with `ensures` clauses, we can compose logic expressions and specify multiple preconditions:

```
/*@
  requires 0 <= a < 100;
  requires b < a;
*/
void foo(int a, int b){

}
```

Preconditions are properties about the input (or about global variables) that we assume to be true when we analyze the function. We will verify that they are indeed true only at program points where the function is called.

In this small example, we can also notice a difference with C in the writing of boolean expressions. If we want to specify that `a` is between 0 and 100, we do not have to write

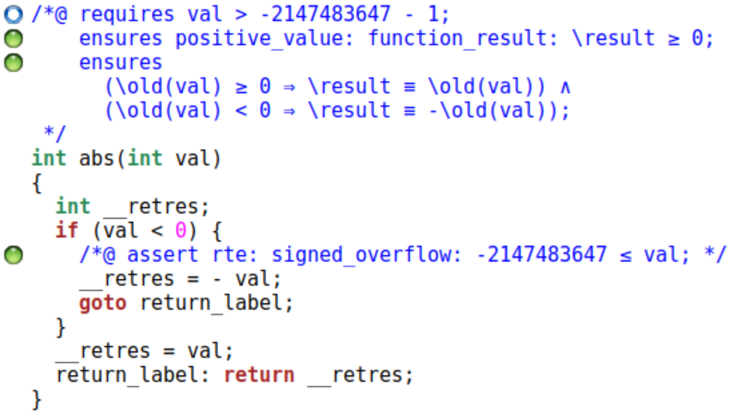
`0 <= a && a < 100`, we can directly write `0 <= a < 100` and Frama-C will perform necessary translations.

If we come back to our example about the absolute value, to avoid the arithmetic overflow, it is sufficient to state that `val` must be strictly greater than `INT_MIN` to guarantee that the overflow will never happen. Thus, we add it as a precondition of the function (notice that it is also necessary to include the header where `INT_MIN` is defined):

```c
#include <limits.h>

/*@
  requires INT_MIN < val;

  ensures \result >= 0;
  ensures (val >= 0 ==> \result == val) &&
          (val < 0 ==> \result == -val);
*/
int abs(int val){
  if(val < 0) return -val;
  return val;
}
```

! Reminder: The Frama-C GUI does not allow source code modification.

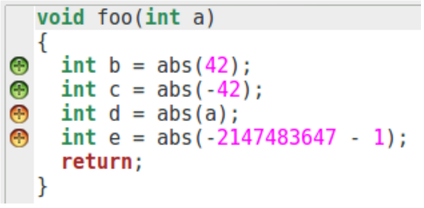
Once we have modified the source code with our precondition, we click on “Reparse” and we can ask again to prove our program. This time, everything is validated by WP, our implementation is proved:



```
/*@ requires val > -2147483647 - 1;
    ensures positive_value: function_result: \result ≥ 0;
    ensures
      (\old(val) ≥ 0 ⇒ \result ≡ \old(val)) ∧
      (\old(val) < 0 ⇒ \result ≡ -\old(val));
 */
int abs(int val)
{
  int __retres;
  if (val < 0) {
    /*@ assert rte: signed_overflow: -2147483647 ≤ val; */
    __retres = - val;
    goto return_label;
  }
  __retres = val;
  return_label: return __retres;
}
```

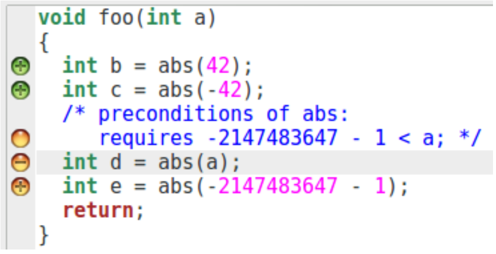
We can also verify that a function that would call `abs` conforms to the required precondition:

```c
void foo(int a){
  int b = abs(42);
  int c = abs(-42);
  int d = abs(a);       // Warning: "a" may be INT_MIN
  int e = abs(INT_MIN); // False: the parameter is INT_MIN
}
```



```
void foo(int a)
{
  int b = abs(42);
  int c = abs(-42);
  int d = abs(a);
  int e = abs(-2147483647 - 1);
  return;
}
```

Note that we can click on the bullet next to the function call to see the list of preconditions and check which ones are not validated. Here, there is only one precondition, but when there are multiple ones it is useful to check what is exactly the problem.



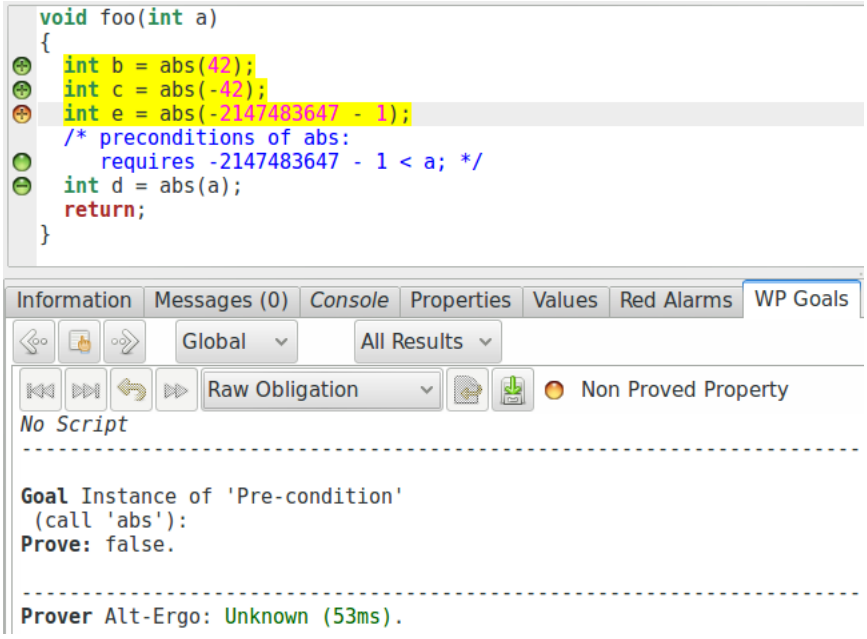
```
void foo(int a)
{
  int b = abs(42);
  int c = abs(-42);
  /* preconditions of abs:
     requires -2147483647 - 1 < a; */
  int d = abs(a);
  int e = abs(-2147483647 - 1);
  return;
}
```

We can modify this example by reverting the last two instructions. If we do this, we can see that the call `abs(a)` is validated by WP if it is placed after the call `abs(INT_MIN)`! Why?

We must keep in mind that the idea of the deductive proof is to ensure that if preconditions are verified, and if our computation terminates, then the postcondition is verified.

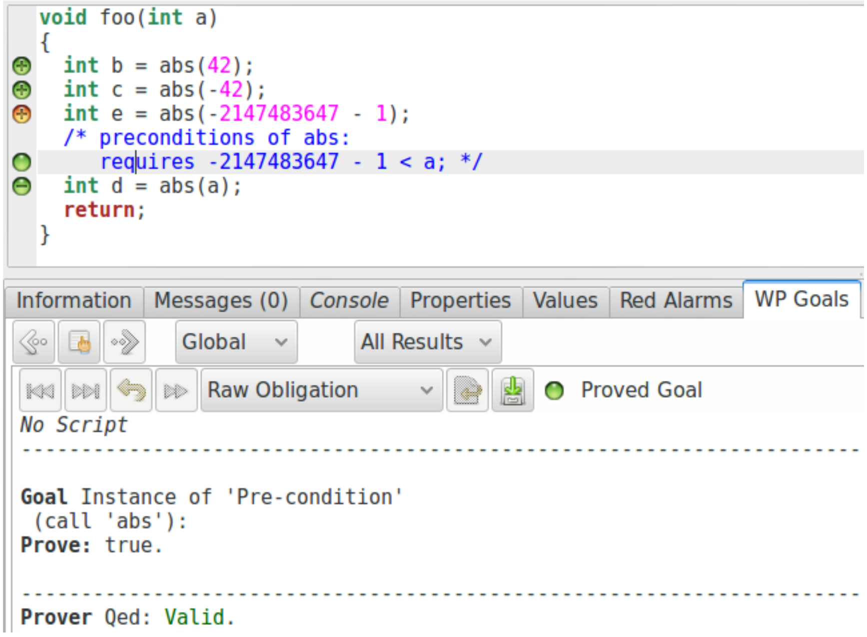
If we give to a function an input that surely breaks the precondition, we can deduce that everything can happen, including obtain false in postcondition. More precisely, here, after the call, we just suppose that the precondition is still true (as the function does not modify anything is memory), thus we suppose that `INT_MIN < INT_MIN` which is obviously false. Knowing this, we can prove absolutely everything because this “false” becomes an assumption of every call that follows. Knowing false, we can prove everything, because if we have a proof of false, then false is true, as well as true is true. So everything is true.

Taking our modified program, we can convince ourselves of this fact by looking at the verification conditions generated by WP for the bad call and the subsequent call that becomes verified:



```
void foo(int a)
{
  int b = abs(42);
  int c = abs(-42);
  int e = abs(-2147483647 - 1);
  /* preconditions of abs:
     requires -2147483647 - 1 < a; */
  int d = abs(a);
  return;
}
```

```
No Script
--------------------------------------------------------------------

Goal Instance of 'Pre-condition'
 (call 'abs'):
Prove: false.

--------------------------------------------------------------------
Prover Alt-Ergo: Unknown (53ms).
```



```
void foo(int a)
{
  int b = abs(42);
  int c = abs(-42);
  int e = abs(-2147483647 - 1);
  /* preconditions of abs:
     requires -2147483647 - 1 < a; */
  int d = abs(a);
  return;
}
```

```
No Script
--------------------------------------------------------------------

Goal Instance of 'Pre-condition'
 (call 'abs'):
Prove: true.

--------------------------------------------------------------------
Prover Qed: Valid.
```

We can notice that for function calls, the GUI highlights the execution path that leads to the call for which we want to verify the precondition. Then, if we have a closer look at the call `abs(INT_MIN)`, we can notice that, simplifying, Qed deduced that we try to prove “False”. Consequently, the next call `abs(a)` receives in its assumptions the property “False”. This is why Qed can immediately deduce “True”.

The second part of the question is then: why our first version of the calling function (`abs(a)` and then `abs(INT_MIN)`) did not have the same behavior, indicating a proof failure on the second call? The answer is simply that for the call `abs(a)` can we add the assumption `a < INT_MIN`, and while we do not have a proof that it is true, we do not have a proof that is false neither. So, while `abs(INT_MIN)` necessarily gives us the knowledge of “false”, the call `abs(a)` does not, since it can succeed.

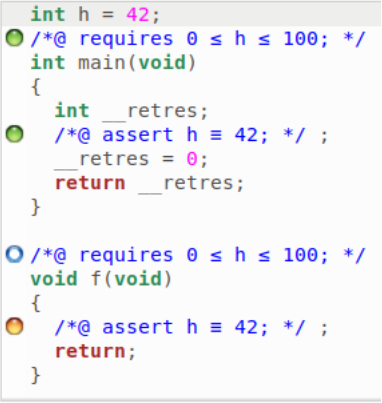
3.1.3. The particular case of the `main` function

The main function is generally not called directly by the program. However, it might have preconditions. In such a case, WP generates verification conditions for them. But, the context that is provided to the proof is not the same as what is provided for the other functions. Indeed, since the main function is called before any other function of the program, at this point, the global variables necessarily have the values that have been provided for their initialization.

Let us illustrate this with the following code:

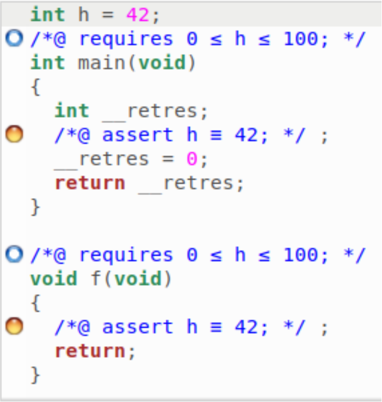
```c
int h = 42 ;

//@ requires 0 <= h <= 100 ;
int main(void){
  //@ assert h == 42 ;
}

//@ requires 0 <= h <= 100 ;
void f(void){
  //@ assert h == 42 ;
}
```

If we start WP on this example, the `requires` clause of the `main` function is proved, while no proof attempt has been done for the precondition of the `f` function since it is not called by the program. The precondition of the `main` function is proved because after initialization `h` has value 42 which is obviously between 0 and 100. Notice that the assertion in the `main` function is also proved because `h` is not changed before the assertion, while in the `f` function, the assertion is not proved since WP does not make any assumption about the values of the global variables.



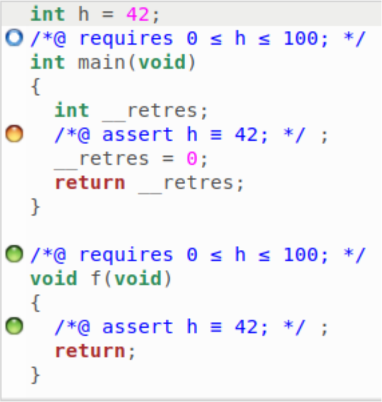
```c
int h = 42;
/*@ requires 0 ≤ h ≤ 100; */
int main(void)
{
  int __retres;
  /*@ assert h ≡ 42; */ ;
  __retres = 0;
  return __retres;
}

/*@ requires 0 ≤ h ≤ 100; */
void f(void)
{
  /*@ assert h ≡ 42; */ ;
  return;
}
```

Notice that WP makes this assumption about the `main` function because it is the default behavior of Frama-C itself. This behavior can be configured with the option `-lib-entry` that tells Frama-C to consider any function (including `main`) to be callable in the program. Thus, when we enable this option on command line for our example, no verification condition is created for the `requires` clause of the `main` function, and none of the assertions is proved.



```
  int h = 42;
○ /*@ requires 0 ≤ h ≤ 100; */
  int main(void)
  {
    int __retres;
🟠   /*@ assert h ≡ 42; */ ;
    __retres = 0;
    return __retres;
  }

○ /*@ requires 0 ≤ h ≤ 100; */
  void f(void)
  {
🟠   /*@ assert h ≡ 42; */ ;
    return;
  }
```

Finally, Frama-C enables with the possibility to configure which function must be considered as the main function. For example, if we call Frama-C with the option `-main=f` on the command line, the `requires` clause of `f` is proved as well as the assertion that it contains, no proof attempt is generated for the `requires` clause of the `main` function, and the assertion in `main` is not proved.



```
  int h = 42;
○ /*@ requires 0 ≤ h ≤ 100; */
  int main(void)
  {
    int __retres;
🟠   /*@ assert h ≡ 42; */ ;
    __retres = 0;
    return __retres;
  }

🟢 /*@ requires 0 ≤ h ≤ 100; */
  void f(void)
  {
🟢   /*@ assert h ≡ 42; */ ;
    return;
  }
```

3.1.4. Exercises

While these exercises are not absolutely necessary to read the next chapters of the tutorial we strongly suggest practicing them. Note that we also suggest to, at least, read the fourth exercise that introduces a notation that can be sometimes useful.

3.1.4.1. Addition

Write the postcondition of the following addition function:

```c
int add(int x, int y){
  return x+y ;
}
```

And run the command:

```
frama-c-gui your-file.c -wp
```

Once the function is successfully proved to respect the contract, run:

```
frama-c-gui your-file.c -wp -wp-rte
```

It should fail, adapt the contract by adding the right precondition.

3.1.4.2. Distance

Write the postcondition of the following distance function, by expressing the value of `b` in terms of `a` and `\result`:

```
int distance(int a, int b){
  if(a < b) return b - a ;
  else return a - b ;
}
```

And run the command:

```
frama-c-gui your-file.c -wp
```

Once the function is successfully proved to respect the contract, run:

```
frama-c-gui your-file.c -wp -wp-rte
```

It should fail, adapt the contract by adding the right precondition.

3.1.4.3. Alphabet Letter

Write the postcondition of the following function that return true if the character received in input is an alphabet letter. Use the equivalence operator `<==>`.

```
int alphabet_letter(char c){
  if( ('a' <= c && c <= 'z') || ('A' <= c && c <= 'Z') ) return 1 ;
  else return 0 ;
}

int main(){
  int r ;

  r = alphabet_letter('x') ;
  //@ assert r ;
  r = alphabet_letter('H') ;
  //@ assert r ;
  r = alphabet_letter(' ') ;
  //@ assert !r ;
}
```

And run the command:

```
frama-c-gui your-file.c -wp -wp-rte
```

All verification conditions should be proved, including the assertions in the main function.

3.1.4.4. Days of the month

Write the postcondition of the following function that returns the number of days in function of the received month (notice that we consider that they are numbered from 1 to 12), for February, we only consider the case when it has 28 days, we will see later how to solve this problem:

```
int day_of(int month){
  int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 } ;
  return days[month-1] ;
}
```

And run the command:

```
frama-c-gui your-file.c -wp
```

Once the postcondition is proved, run the command:

```
frama-c-gui your-file.c -wp -wp-rte
```

If it fails, complete the precondition in order to solve the problem.

You might have notice that writing the postcondition can be a bit painful. Let us try to simplify that. ACSL provide the notion of set, and the operator `\in` that can be used to check whether a value is in a set or not.

For example:

```
//@ assert 13 \in { 1, 2, 3, 4, 5 } ; // FALSE
//@ assert 3  \in { 1, 2, 3, 4, 5 } ; // TRUE
```

Modify the postcondition by using this notation.

3.1.4.5. Last angle of a triangle

This function receives two values of angle in input and returns the value of the last angle considering that the sum of the three angles must be 180. Write the postcondition that expresses that the sum of the three angles is 180.

```c
int last_angle(int first, int second){
  return 180 - first - second ;
}
```

And run the command:

```
frama-c-gui your-file.c -wp
```

Once the function is successfully proved to respect the contract, run:

```
frama-c-gui your-file.c -wp -wp-rte
```

If it fails, add the right precondition. Note that the values of the different angles should not be more than 180 degrees, including the resulting angle.

3.2. Well specified function

3.2.1. Correctly write what we expect

This is certainly the hardest part of our work. Programming is already an effort that consists in writing algorithms that correctly answer to our need. Specifying requests the same kind of work, except that we do not try to express *how* we answer to our need but *what* is exactly our need. To prove that our code implements what we need, we must be able to describe exactly what we need.

From now, let us use another example, the `max` function:

```c
int max(int a, int b){
  return (a > b) ? a : b;
}
```

The reader could write and prove their own specification. Let us use this one:

```c
/*@
  ensures \result >= a && \result >= b;
*/
int max(int a, int b){
  return (a > b) ? a : b;
}
```

If we ask WP to prove this code, it succeeds without any problem. However, is our specification really what we need? We can try to prove this calling code:

```
void foo(){
  int a = 42;
  int b = 37;
  int c = max(a,b);

  //@assert c == 42;
}
```

There, it fails. In fact, we can go further by modifying the body of the `max` function and notice that the following code is also correct with respect to the specification:

```
#include <limits.h>

/*@
  ensures \result >= a && \result >= b;
*/
int max(int a, int b){
  return INT_MAX;
}
```

While being a correct specification of `max`, our specification is however too permissive. We have to be more precise. We do not only expect the result to be greater or equal to both parameters, but also that the result is one of them:

```
/*@
  ensures \result >= a && \result >= b;
  ensures \result == a || \result == b;
*/
int max(int a, int b){
  return (a > b) ? a : b;
}
```

This specification can also be proved correct by WP, but now we can also prove the assertion in our function `foo`, and we cannot prove anymore an implementation that would just return the value `INT_MAX`.

3.2.2. Inconsistent preconditions

Producing a correct specification is crucial. Typically, by stating a false precondition, we can have the possibility to create a proof of false:

```
/*@
  requires a < 0 && a > 0;
  ensures \false;
```

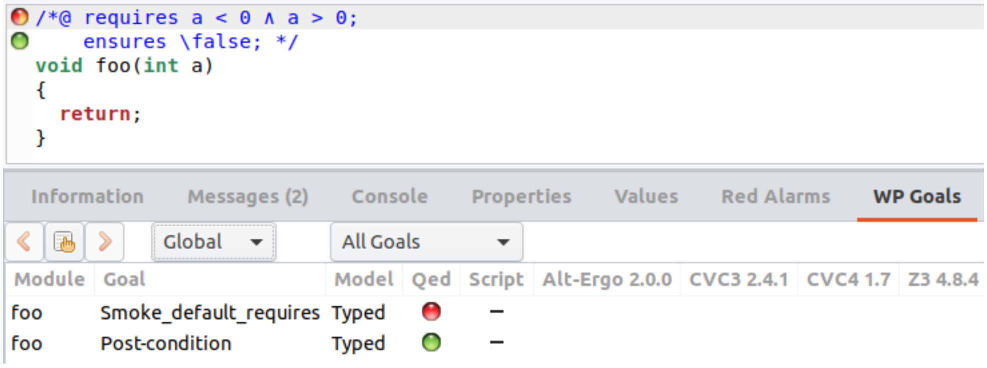
```
*/
void foo(int a){

}
```

If we ask WP to prove this function, it will accept it without any problem since the property we ask in precondition is necessarily false. However, we will not be able to give an input that satisfies the precondition.

For this particular kind of inconsistencies, a useful feature of WP is the "smoke tests" option of the plugin. It can be used to detect that some preconditions are unsatisfiable. For example, here, we can run this command line:

```
 frama-c-gui -wp -wp-smoke-tests file.c
```

and we obtain the following result in the GUI:

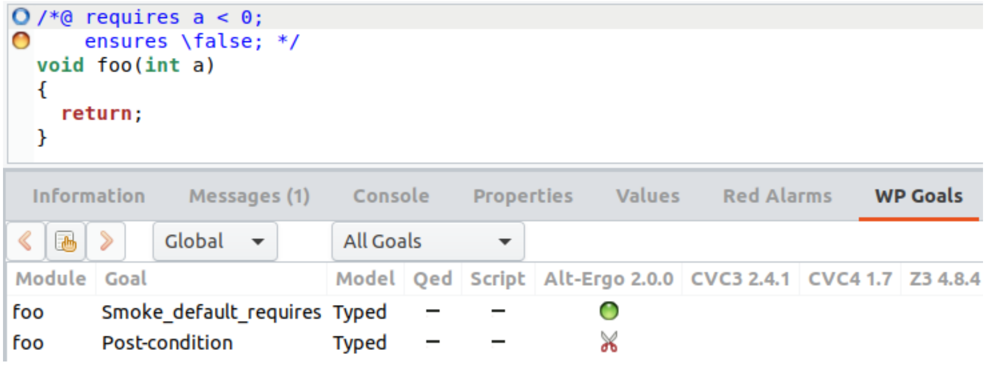


```
/*@ requires a < 0 ∧ a > 0;
    ensures \false; */
void foo(int a)
{
  return;
}
```

Module	Goal	Model	Qed	Script	Alt-Ergo 2.0.0	CVC3 2.4.1	CVC4 1.7	Z3 4.8.4
foo	Smoke_default_requires	Typed	●	—				
foo	Post-condition	Typed	●	—				

We can see a red and orange bullet on the precondition of the function, meaning that if there exists a reachable call to this function the precondition will be necessarily violated, and a red bullet in the list of goals, indicating that a prover succeeded in proving that this precondition is inconsistent.

Note that when the smoke tests succeed, for example if we fix the precondition like this:



```
/*@ requires a < 0;
    ensures \false; */
void foo(int a)
{
  return;
}
```

Module	Goal	Model	Qed	Script	Alt-Ergo 2.0.0	CVC3 2.4.1	CVC4 1.7	Z3 4.8.4
foo	Smoke_default_requires	Typed	—	—	●			
foo	Post-condition	Typed	—	—	✂			

it does not necessarily mean that the precondition is consistent, just that the prover was unable to prove that it was inconsistent.

Some notions we will see in this tutorial can expose us to the possibility to introduce subtle incoherence. So, we must always be careful when specifying a program.

3.2.3. Pointers

If there is one notion that we permanently have to confront with in C language, this is definitely the notion of pointer. Pointers are quite hard to manipulate correctly, and they still are the main source of critical bugs in programs, so they benefit of a preferential treatment in ACSL. In order to produce a correct specification of programs using pointers, it is necessary to detail the configuration of the considered memory.

We can illustrate with a swap function for C integers (this version can only be verified without RTE checking for now):

```
/*@
  ensures *a == \old(*b) && *b == \old(*a);
*/
void swap(int* a, int* b){
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
```

3.2.3.1. History of values in memory

Here, we introduce a first built-in logic function of ACSL: `\old`, that allows us to get the old (that is to say, before the call) value of a given element. So, our specification defines that the function must ensure that after the call, the value of `*a` is the old value of `*b` and conversely.

The `\old` function can only be used in the postcondition of a function. If we need this kind of information somewhere else, we use `\at` that allows us to express that we want the value of a variable at a particular program point. This function receives two parameters. The first one is the variable (or memory location) for which we want to get its value and the second one is the program point (as a C label) that we want to consider.

For example, we could write:

```
  int a = 42;
 Label_a:
  a = 45;

  //@assert a == 45 && \at(a, Label_a) == 42;
```

Of course, we can use any C label in our code, but we also have 6 built-in labels defined by ACSL that can be used:

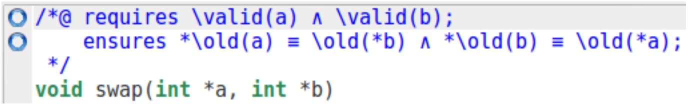
- `Pre` / `Old` : value before function call,
- `Post` : value after function call,
- `LoopEntry` : value at loop entry
- `LoopCurrent` : value at the beginning of the current step of the loop,

- `Here` : value at the current program point.

> The behavior of `Here` is, in fact, the default behavior when we consider a variable. Its use with `\at` generally allows us to ensure that what we write is not ambiguous, and is more readable, when we express properties about values at different program points in the same expression.

Whereas `\old` can only be used in function postconditions, `\at` can be used anywhere. However, we cannot use any program point with respect to the type annotation we are writing. `Old` and `Post` are only available in function postconditions, `Pre` and `Here` are available everywhere. `LoopEntry` and `LoopCurrent` are only available in the context of loops (which we will detail later in this tutorial).

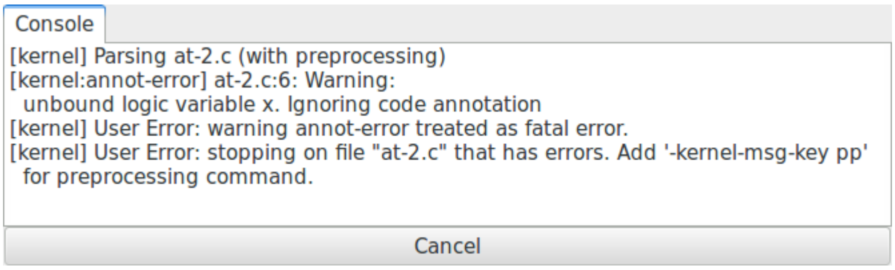
Note that one must take care to use `\old` and `\at` for values that make sense. This is why for example in a contract, all values received in input are put into `\old` when used in postcondition, the "new" value of the input variables not make any sense for the caller of the function as they are not accessible: they are local to the called function. For example, if we have look at the contract of the swap function transformed by Frama-C, we can see that in the postcondition, the pointers are enclosed into `\old`:



```
/*@ requires \valid(a) ∧ \valid(b);
    ensures *\old(a) ≡ \old(*b) ∧ *\old(b) ≡ \old(*a);
 */
void swap(int *a, int *b)
```

For the built-in `\at` function, we have to take care of this more explicitly. In particular, the specified label must make sense with respect to the scope of the value. For example, in the following program, Frama-C detects that we ask the value of the variable `x` at a program point where it does not exist:

```
void example_1(void){
 L: ;
  int x = 1 ;
  //@ assert \at(x, L) == 1 ;
}
```



```
[kernel] Parsing at-2.c (with preprocessing)
[kernel:annot-error] at-2.c:6: Warning:
  unbound logic variable x. Ignoring code annotation
[kernel] User Error: warning annot-error treated as fatal error.
[kernel] User Error: stopping on file "at-2.c" that has errors. Add '-kernel-msg-key pp'
  for preprocessing command.
```

However, in some other cases, we only reach a proof failure since determining that the value does not exist at some particular label cannot be done by a syntactic analysis. For example, if the variable is declared but undefined or if we want the value of a pointed value:

```
void example_2(void){
  int x ;
 L:
  x = 1 ;
  //@ assert \at(x, L) == 1 ;
}

void example_3(void){
 L: ;
  int x = 1 ;
  int *ptr = &x ;
  //@ assert \at(*\at(ptr, Here), L) == 1 ;
}
```

Here, it is easy to see the problem. However, the considered label is propagated to sub-expressions. Thus, sometimes, terms that seem to be innocent can have a surprising behavior if we do not keep this fact in mind. For example, in the following example:

```
/*@ requires x + 2 != p ; */
void example_4(int* x, int* p){
  *p = 2 ;
  //@ assert x[2]  == \at(x[2], Pre) ;
  //@ assert x[*p] == \at(x[*p], Pre) ;
}
```

The first assertion is proved, and while the second assertion seems to express the same property, it cannot be proved. Because, it in fact does not express the same property. The expression `\at(x[*p], Pre)` must be seen as `\at(x[\at(*p)], Pre)`, since the label is propagated to the sub-expression `*p`, for which we do not know the value at label `Pre` (since it is not specified).

For the moment, we do not need `\at`, but it can often be useful, if not essential, when we want to make our specification precise.

3.2.3.2. Pointers validity

If we try to prove that the swap function is correct (comprising the verification of absence of runtime errors), our postcondition is indeed verified but WP fails to prove that some runtime-error cannot happen, since we perform access to some pointers that we did not indicate to be valid pointers in the precondition of the function.

We can express that the dereferencing of a pointer is valid using the `\valid` predicate of ACSL which receives the pointer in input:

```
/*@
  requires \valid(a) && \valid(b);
  ensures  *a == \old(*b) && *b == \old(*a);
*/
void swap(int* a, int* b){
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
```

Once we have specified that the pointers we receive in input must be valid, dereferencing is assured to not produce undefined behaviors.

As we will see later in this tutorial, `\valid` can take more than one pointer in parameter. For example, we can give it an expression such as: `\valid(p + (s .. e))` which means "for all `i` between included `s` and `e`, `p+i` is a valid pointer". This kind of expression is extremely useful when we have to specify properties about arrays in specifications.

If we have a closer look at the assertions that RTE adds in the swap function when we ask the verification of absence of runtime errors, we can notice that there exists another version of the `\valid` predicate, denoted `\valid_read`. As opposed to `\valid`, the predicate `\valid_read` indicates that a pointer can be dereferenced, but only to read the pointed memory. This subtlety is due to the C language, where the downcast of a const pointer is easy to write but is not necessarily legal.

Typically, in this code:

```c
/*@ requires \valid(p); */
int unref(int* p){
  return *p;
}

int const value = 42;

int main(){
  int i = unref(&value);
}
```

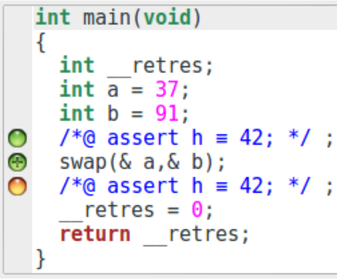
Dereferencing `p` is valid, however the precondition of `unref` is not verified by WP since dereferencing `value` is only legal for a read-access. A write access would result in an undefined behavior. In such a case, we can specify that the pointer `p` must be `\valid_read` and not `\valid`.

3.2.3.3. Side Effects

Our `swap` function is provable with regard to the specification and potential runtime errors, however is our specification precise enough? We can slightly modify our code to check this (we use `assert` to verify some properties at some particular points):

```c
int h = 42;

/*@
  requires \valid(a) && \valid(b);
  ensures  *a == \old(*b) && *b == \old(*a);
*/
void swap(int* a, int* b){
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(){
  int a = 37;
  int b = 91;

  //@ assert h == 42;
  swap(&a, &b);
  //@ assert h == 42;
}
```

The result is not exactly what we expect:



```
int main(void)
{
  int __retres;
  int a = 37;
  int b = 91;
  /*@ assert h ≡ 42; */ ;
  swap(& a,& b);
  /*@ assert h ≡ 42; */ ;
  __retres = 0;
  return __retres;
}
```

Indeed, we did not specify the allowed side effects for our function. In order to specify side effects, we use an `assigns` clause which is part of the postcondition of a function. It allows us to specify which **non-local** elements (we verify side effects) can be modified during the execution of the function.

By default, WP considers that a function can modify everything in memory. So, we have to specify what can be modified by a function. For example, our `swap` function can be specified to modify the values pointed by the received pointers:

```
/*@
  requires \valid(a) && \valid(b);

  assigns *a, *b;

  ensures  *a == \old(*b) && *b == \old(*a);
*/
void swap(int* a, int* b){
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
```

If we ask WP to prove the function with this specification, it is validated (including with the variable added in the previous source code).

Finally, we sometimes want to specify that a function is side effect free. We specify this by giving `\nothing` to `assigns` :

```
/*@
  requires \valid_read(a);
  requires *a <= INT_MAX - 5 ;

  assigns \nothing ;

  ensures \result == *a + 5 ;
*/
```

```
int plus_5(int* a){
  return *a + 5 ;
}
```

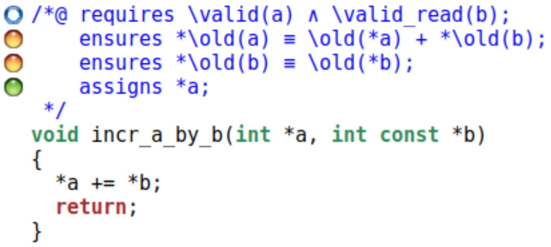
The careful reader will now be able to take back the examples we presented until now to integrate the right `assigns` clause.

3.2.3.4. Memory location separation

Pointers bring the risk of aliasing (multiple pointers can have access to the same memory location). For some functions, it does not cause any problem, for example when we give two identical pointers to the `swap` function, the specification is still verified. However, sometimes it is not that simple:

```
#include <limits.h>

/*@
  requires \valid(a) && \valid_read(b);
  assigns  *a;
  ensures  *a == \old(*a)+ *b;
  ensures  *b == \old(*b);
*/
void incr_a_by_b(int* a, int const* b){
  *a += *b;
}
```

If we ask WP to prove this function (let us ignore the verification of absence of RTEs for this example), we get the following result:



```
/*@ requires \valid(a) ∧ \valid_read(b);
    ensures *\old(a) ≡ \old(*a) + *\old(b);
    ensures *\old(b) ≡ \old(*b);
    assigns *a;
 */
void incr_a_by_b(int *a, int const *b)
{
  *a += *b;
  return;
}
```

The reason is simply that we do not have any guarantee that the pointer `a` is different of the pointer `b`. Now, if these pointers are the same,

- the property `*a == \old(*a) + *b` in fact means `*a == \old(*a) + *a` which can only be true if the old value pointed by `a` was 0, and we do not have such a requirement,
- the property `*b == \old(*b)` is not validated because we potentially modify this memory location.

?

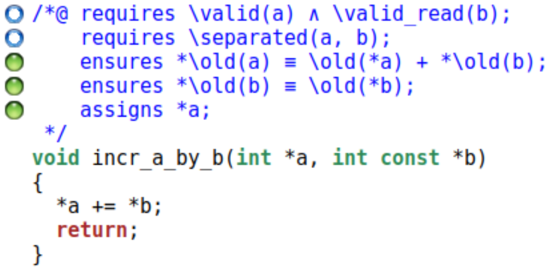
Why is the `assigns` clause validated?

The reason is simply that `a` is indeed the only modified memory location. If `a != b`, we only modify the location pointed by `a`, and if `a == b`, this is still the case: `b` is not another location.

In order to ensure that pointers refer to separated memory locations, ACSL provides the predicate `\separated(p1, ..., pn)` that receives in parameter a set of pointers and is true if and only if these pointers are non-overlapping. Here, we specify:

```c
#include <limits.h>

/*@
  requires \valid(a) && \valid_read(b);
  requires \separated(a, b);
  assigns  *a;
  ensures  *a == \old(*a)+ *b;
  ensures  *b == \old(*b);
*/
void incr_a_by_b(int* a, int const* b){
  *a += *b;
}
```

And this time, the function is verified:



```
○ /*@ requires \valid(a) ∧ \valid_read(b);
○     requires \separated(a, b);
●     ensures *\old(a) ≡ \old(*a) + *\old(b);
●     ensures *\old(b) ≡ \old(*b);
●     assigns *a;
   */
  void incr_a_by_b(int *a, int const *b)
  {
    *a += *b;
    return;
  }
```

We can notice that we do not consider the arithmetic overflow here, as we do not focus on this question in this section. However, if this function was part of a complete program, it would be necessary to define the context of use of this function and the precondition guaranteeing the absence of overflow.

3.2.4. Writing the right contract

Writing a specification that is precise enough can sometimes be a bit tricky. Interestingly, a good way to check if a specification is precise enough is to write tests. And in fact, this is basically what we have done for our examples `max` and `swap`. We have written a first version of the specification, and we have written some code with a call to the corresponding function to determine whether we could prove some properties that we expected to be easily provable from the contract of the function.

The most important idea is to determine the contract without taking in account the content of the function (at least, in a first step). Indeed, we are trying to prove the function, but maybe it contains a bug, so if we write the contract taking in account too directly its code, we have a risk to introduce the same bug, for example taking in account an erroneous conditional structure.

In fact, it is generally a good practice to work with someone else. One specifies the function and the other implements it (even if they previously agreed on a common textual specification).

Once the contract have been stated, we work on the specifications that are due to the constraints of our language and our hardware. That mostly concerns the precondition of the function. For example, the absolute value does not really have a precondition, this is our hardware that adds the condition we have given in precondition due to the two's complement on which it relies. As we will see in the chapter 7, verifying the absence of runtime errors can also impact the postcondition. For now, let us leave this.

3.2.5. Exercises

3.2.5.1. Division and remaining

Specify the postcondition of the following function, that computes the results of the division of `a` by `b` and its remaining and stores it into two memory locations `p` and `q`:

```c
void div_rem(unsigned x, unsigned y, unsigned* q, unsigned* r){
  *q = x / y ;
  *r = x % y ;
}
```

Run the command:

```
frama-c-gui your-file.c -wp
```

Once the function is successfully proved to respect the contract, run:

```
frama-c-gui your-file.c -wp -wp-rte
```

If it fails, complete the contract by adding the right precondition.

3.2.5.2. Reset on condition

Provide a contract for the following function that resets its first parameter if and only if the second is true. Be sure to express that the second parameter remains unmodified:

```c
void reset_1st_if_2nd_is_true(int* a, int const* b){
  if(*b) *a = 0 ;
}

int main(){
  int a = 5 ;
  int x = 0 ;

  reset_1st_if_2nd_is_true(&a, &x);
  //@ assert a == 5 ;
  //@ assert x == 0 ;

  int const b = 1 ;

  reset_1st_if_2nd_is_true(&a, &b);
  //@ assert a == 0 ;
  //@ assert b == 1 ;
}
```

Run the command:

```
frama-c-gui your-file.c -wp -wp-rte
```

3.2.5.3. Addition of pointed values

The following function receives two pointers as an input and returns the sum the pointed values. Write the contract of this function:

```
int add(int *p, int *q){
  return *p + *q ;
}

int main(){
  int a = 24 ;
  int b = 42 ;

  int x ;

  x = add(&a, &b) ;
  //@ assert x == a + b ;
  //@ assert x == 66 ;

  x = add(&a, &a) ;
  //@ assert x == a + a ;
  //@ assert x == 48 ;
}
```

Run the command:

```
frama-c-gui your-file.c -wp -wp-rte
```

Once the function and calling code are successfully proved, modify the signature of the function add as follows:

```
void add(int* p, int* q, int* r);
```

Now the result of the sum should be stored at `r`. Accordingly modify the calls in the main function as well as the code and the contract of `add`. `*p` and `*q` should remain unchanged.

3.2.5.4. Maximum of pointed values

The following function computes the maximum of the values pointed by `a` and `b`. Write the contract of the function:

```
1  int max_ptr(int* a, int* b){
2    return (*a < *b) ? *b : *a ;
3  }
4  
5  extern int h ;
6  
7  int main(){
8    h = 42 ;
9  
10   int a = 24 ;
11   int b = 42 ;
12 
13   int x = max_ptr(&a, &b) ;
14 
15   //@ assert x == 42 ;
16   //@ assert h == 42 ;
17 }
```

Run the command:

```
frama-c-gui your-file.c -wp -wp-rte
```

Once it is proved, modify the signature of the function as follows:

```
void max_ptr(int* a, int* b);
```

Now the function should ensure that after its execution `*a` contains the maximum of the input value, and `*b` contains the other value. Modify the code accordingly as well as the contract. Note that the variable `x` is not necessary anymore in the `main` function and that we can change the assertion on line 15 to reflect the new behavior of the function.

3.2.5.5. Order 3 values

The following function should order the 3 input values in increasing order. Write the corresponding code and specification of the function:

```
void order_3(int* a, int* b, int* c){
  // CODE
}
```

And run the command:

```
frama-c-gui your-file.c -wp -wp-rte
```

Remember that ordering values is not just ensuring that resulting values are sorted increasing order that each pointed value must be one of the original ones. All original values should still be there after the sorting operation: new values are a permutation of the original ones. To express this idea, one can rely on the set datatype. For example, this property is true:

```
//@ assert { 1, 2, 3 } == { 2, 3, 1 };
```

We can use this to express that the set of original values and final values is the same. However, that is not the only thing to consider, as a set only contains one occurrence of each value. So, if `*a == *b == 1`, `{ *a, *b } == { 1 }`. Thus, we also have to consider three other particular cases:

- all original values equal
- two original values equal and the last is greater
- two original values equal and the last is lower

That should set one more constraint on the final values.

As a helper, one could use the following test program:

```
  int a1 = 5, b1 = 3, c1 = 4 ;
  order_3(&a1, &b1, &c1) ;
  //@ assert a1 == 3 && b1 == 4 && c1 == 5 ;

  int a2 = 2, b2 = 2, c2 = 2 ;
  order_3(&a2, &b2, &c2) ;
  //@ assert a2 == 2 && b2 == 2 && c2 == 2 ;

  int a3 = 4, b3 = 3, c3 = 4 ;
  order_3(&a3, &b3, &c3) ;
  //@ assert a3 == 3 && b3 == 4 && c3 == 4 ;

  int a4 = 4, b4 = 5, c4 = 4 ;
  order_3(&a4, &b4, &c4) ;
  //@ assert a4 == 4 && b4 == 4 && c4 == 5 ;
}

```

If the specification is precise enough, each assertion should be proved. However, that does not mean that all cases have been considered by our tests, so do not hesitate do add other cases.

3.3. Behaviors

Sometimes, a function can have behaviors that can be quite different depending on the input. Typically, a function can receive a pointer to an optional resource: if the pointer is `NULL`, we have a certain behavior, which is different of the behavior expected when the pointer is not `NULL`.

We have already seen a function that have different behaviors: the `abs` function. Let us use it again to illustrate behaviors. We have two behaviors for the `abs` function: either the input is positive or it is negative.

Behaviors allow us to specify the different cases for postconditions. We introduce them using the `behavior` keyword. Each behavior is named. For a given behavior, we have different assumptions about the input of the function, they are introduced with the clause `assumes` (note that since they characterize the input, the keyword `\old` cannot be used there). However, the properties expressed by this clause do not have to be verified before the call, they can be verified and in this case, the postcondition specified in our behavior applies. The postconditions of a particular behavior are introduced using an `ensures` clause. Finally, we can ask WP to verify that behaviors are disjoint (to guarantee determinism) and complete (to guarantee that we cover all possible input).

Behaviors are disjoint if for any (valid) input of the function, it corresponds to the assumption (`assumes`) of a single behavior. Behaviors are complete if any (valid) input of the function corresponds to at least one behavior.

For example, for `abs` we can write the specification:

```c
#include <limits.h>

/*@
  requires val > INT_MIN;
  assigns  \nothing;

  ensures \result >= 0;

  behavior pos:
    assumes 0 <= val;
    ensures \result == val;

  behavior neg:
    assumes val < 0;
    ensures \result == -val;

  complete behaviors;
  disjoint behaviors;
*/
int abs(int val){
  if(val < 0) return -val;
  return val;
}
```

Note that declaring behaviors does not forbid to specify global postconditions. For example here, we have specified that for any behavior, the function must return a positive value.

Let us now slightly modify the assumptions of each behavior to illustrate the meaning of `complete` and `disjoint`:

- replace the assumption of `pos` with `val > 0`, in this case, behaviors are disjoint but incomplete (we miss `val == 0`),

- replace the assumption of `neg` with `val <= 0`, in this case, behaviors are complete but not disjoint (we have two assumptions corresponding to `val == 0`).

Even if `assigns` is a postcondition, indicating different assigns in each behavior is currently not well-handled by WP. If we need to specify this, we will:

- put our `assigns` before the behaviors (as we have done in our example) with all potentially modified non-local elements,
- add in postcondition of each behavior the elements that are in fact not modified by indicating their new value to be equal to the `\old` one.

Behaviors are useful to simplify the writing of specifications when functions can have very different behaviors depending on their input. Without them, specification would be defined using implications expressing the same idea but harder to write and read (which would be error-prone). On the other hand, the translation of completeness and disjointedness would be necessarily written by hand which would be tedious and again error-prone.

3.3.1. Exercises

3.3.1.1. Previous exercises

From previous sections, take back the examples:

- about the computation of the distance between to integers,
- "reset on condition",
- "days of the month",
- "Max of pointed values".

Considering that the contracts were:

```c
#include <limits.h>

/*@
  requires a < b  ==> b - a <= INT_MAX ;
  requires b <= a ==> a - b <= INT_MAX ;

  ensures a < b  ==> a + \result == b ;
  ensures b <= a ==> a - \result == b ;
*/
int distance(int a, int b){
  if(a < b) return b - a ;
  else return a - b ;
}

/*@
  requires \valid(a) && \valid_read(b) ;
  requires \separated(a, b) ;

  assigns *a ;

  ensures \old(*b) ==> *a == 0 ;
  ensures ! \old(*b) ==> *a == \old(*a) ;
  ensures *b == \old(*b);
*/
void reset_1st_if_2nd_is_true(int* a, int const* b){
  if(*b) *a = 0 ;
}

/*@
  requires 1 <= m <= 12 ;
  ensures m \in { 2 } ==> \result == 28 ;
  ensures m \in { 1, 3, 5, 7, 8, 10, 12 } ==> \result == 31 ;
  ensures m \in { 4, 6, 9, 11 } ==> \result == 30 ;
*/
int day_of(int m){
  int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 } ;
  return days[m-1] ;
}

/*@
  requires \valid(a) && \valid(b);
  assigns  *a, *b ;
  ensures  \old(*a) < \old(*b)  ==> *a == \old(*b) && *b == \old(*a) ;
  ensures  \old(*a) >= \old(*b) ==> *a == \old(*a) && *b == \old(*b) ;
*/
void max_ptr(int* a, int* b){
  if(*a < *b){
    int tmp = *b ;
    *b = *a ;
    *a = tmp ;
  }
}
```

Re-write them using behaviors.

3.3.1.2. Two other simple exercises

Produce the code and the specification of the two following functions and prove them. The specification should make use of behaviors.

First, a function that returns if a character is a vowel or a consonant, one should assume (and express) that the function receives a lowercase letter.

```
enum Kind { VOWEL, CONSONANT };

enum Kind kind_of_letter(char c){
  // ...
}
```

Then, a function that returns the quadrant of a given coordinate. When the coordinate is on an axis, arbitrary choose one of the quadrants.

```
int quadrant(int x, int y){
  // ...
}
```

3.3.1.3. Triangle

Complete the following functions that receive the lengths of the different sides of a triangle, and respectively return:

- if the triangle is scalene, isosceles, or equilateral,
- if the triangle is right, acute or obtuse.

```c
#include <limits.h>

enum Sides { SCALENE, ISOSCELE, EQUILATERAL };
enum Angles { RIGHT, ACUTE, OBTUSE };

enum Sides sides_kind(int a, int b, int c){
  // ...
}

enum Angles angles_kind(int a, int b, int c){
  //
}
```

Assuming (and it must be expressed) that:

- the received parameters indeed define a triangle,
- `a` is the hypotenuse of the triangle,

specify and prove that the functions do the right job.

3.3.1.4. Max of pointed values, returning the result

Take back the example “Max of pointed values” from the previous section, and more precisely, the version that returns the result. Considering that the contract was:

```c
/*@
  requires \valid_read(a) && \valid_read(b);
  assigns  \nothing ;
  ensures  *a <  *b ==> \result == *b ;
  ensures  *a >= *b ==> \result == *a ;
  ensures  \result == *a || \result == *b ;
*/
int max_ptr(int* a, int* b){
  return (*a < *b) ? *b : *a ;
}
```

1. Rewrite it using behaviors
2. Modify the contract of 1 in order to make the behaviors non-disjoint, except this property, the contract should remain verified,
3. Modify the contract of 1 in order to make the behaviors incomplete, add a new behavior that makes the contract complete again,

4. Modify the function of 1 in order to accept `NULL` pointers for both `a` and `b`. If both of them are null pointers, return `INT_MIN`, if one is a null pointer, return the value of the other, else, return the maximum of them. Modify the contract accordingly by adding new behaviors. Be sure that they are disjoint and complete.

3.3.1.5. Order 3

Take back the example "Order 3 values" from the previous section. Considering that the contract was:

```
/*@
  requires \valid(a) && \valid(b) && \valid(c) ;
  requires \separated(a, b, c);

  assigns *a, *b, *c ;

  ensures *a <= *b <= *c ;
  ensures { *a, *b, *c } == \old({ *a, *b, *c }) ;

  ensures \old(*a == *b < *c || *a == *c < *b || *b == *c < *a) ==> *a == *b ;
  ensures \old(*a == *b > *c || *a == *c > *b || *b == *c > *a) ==> *b == *c ;
*/
void order_3(int* a, int* b, int* c){
  if(*a > *b){ int tmp = *b ; *b = *a ; *a = tmp ; }
  if(*a > *c){ int tmp = *c ; *c = *a ; *a = tmp ; }
  if(*b > *c){ int tmp = *b ; *b = *c ; *c = tmp ; }
}
```

Rewrite it using behaviors. Note that you should have one general behaviors and 2 specific behaviors. Are these behaviors complete? Are they disjoint?

3.4. WP Modularity

For this last part, let us talk about function call composition, and have a closer look at WP. We will also have a look at the way we can split our programs in different files when we want to prove them using WP.

Our goal is to prove the `max_abs` function, that returns the maximum absolute value of two values:

```
int max_abs(int a, int b){
  int abs_a = abs(a);
  int abs_b = abs(b);

  return max(abs_a, abs_b);
}
```

Let us start by (over-)separating the declarations and definitions of the different functions we need (and have previously proved) into header/source files, that are `abs` and `max`. We obtain, for `abs`:

File abs.h :

```
#ifndef _ABS
#define _ABS

#include <limits.h>

/*@
  requires val > INT_MIN;
  assigns  \nothing;

  behavior pos:
    assumes 0 <= val;
    ensures \result == val;

  behavior neg:
    assumes val < 0;
    ensures \result == -val;

  complete behaviors;
  disjoint behaviors;
*/
int abs(int val);

#endif
```

File abs.c

```
#include "abs.h"

int abs(int val){
  if(val < 0) return -val;
  return val;
}
```

We can notice that we put our function contract in the header file. The goal is to be able to import the specification at the same time as the declaration when we need it in another file. Indeed, WP needs the contract of the function when it is called. First to prove that the precondition is verified (and thus that the call is legal), and second to get in return the postcondition that is useful to prove the right properties after the function call.

We can create a file using the same format for the `max` function. In both cases, we can open the source file (we do not need to specify header files in the command line) with Frama-C and notice that the specification is indeed associated to the function and that we prove it.

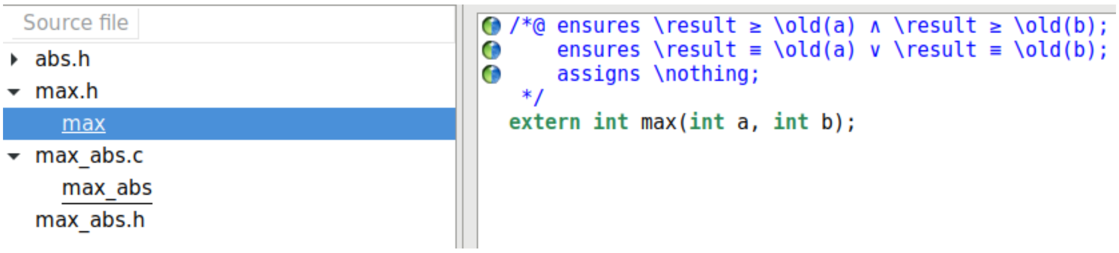
Now, we can prepare our files for the `max_abs` function with the header:

```
#ifndef _MAX_ABS
#define _MAX_ABS

int max_abs(int a, int b);

#endif
```

And its source file:

```c
#include <limits.h>
#include "max_abs.h"
#include "abs.h"
#include "max.h"

int max_abs(int a, int b){
  int abs_a = abs(a);
  int abs_b = abs(b);

  return max(abs_a, abs_b);
}
```

We can open the source file in Frama-C. If we look at the side panel, we can see that the header files we have included in `abs_max` correctly appear and if we look at the function contracts for them, we can see some blue and green bullets:



```
/*@ ensures \result ≥ \old(a) ∧ \result ≥ \old(b);
    ensures \result ≡ \old(a) ∨ \result ≡ \old(b);
    assigns \nothing;
 */
extern int max(int a, int b);
```

These bullets indicate that, since we do not have the implementation of the function, the postcondition of the function is assumed to be true. It is an important strength of the deductive proof of programs compared to some other formal methods: functions are verified in isolation from each other.

When we are not currently performing the proof of a function, its specification is considered to be correct: we do not try to prove it when we are proving another function, we only verify that the precondition is correctly established when we call it. It provides very modular proofs and specifications that are therefore more reusable. Of course, if our proof relies on the specification of another function, it must be provable to ensure that the proof of the program is complete. But, we can also consider that we trust a function that comes from an external library that we do not want to prove (or for which we do not even have the source code).

The careful reader could specify and prove the `max_abs` function. A possible answer is provided there:

```c
/*@
  requires a > INT_MIN;
  requires b > INT_MIN;

  assigns \nothing;

  ensures \result >= 0;
  ensures \result >= a && \result >= -a && \result >= b && \result >= -b;
  ensures \result == a || \result == -a || \result == b || \result == -b;
*/
int max_abs(int a, int b);
```

3.4.1. Exercises

3.4.1.1. Days of the month

Specify the function leap year that returns true if the year received as an input is leap. Use this functions to complete the function `days_of` in order to return the number of days of the month received as an input, including the right behavior when the year is leap for February.

```c
int leap(int y){
  return ((y % 4 == 0) && (y % 100 !=0)) || (y % 400==0) ;
}

int days_of(int m, int y){
  int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 } ;
  int n = days[m-1] ;
  // code
}
```

3.4.1.2. Alphanumeric character

Write and specify the different functions used by `is_alpha_num`. Provide a contract for each of them and provide the contract of `is_alpha_num`.

```c
int is_alpha_num(char c){
  return
    is_lower_alpha(c) ||
    is_upper_alpha(c) ||
    is_digit(c) ;
}
```

Declare an enumeration with values `LOWER`, `UPPER`, `DIGIT` and `OTHER`, and a function `character_kind` that returns, using the different functions `is_lower`, `is_upper`, `is_digit`, the kind of character received in input. Use behaviors to specify the contract of this function and be sure that they are disjoint and complete.

3.4.1.3. Order 3 values

Taking back the function `max_ptr` that orders two values, putting the maximum at the first location and the minimum at the second, write a function `min_ptr` that uses this function and produces the opposite operation. Use these functions to complete the four functions that orders 3 values. For each variant (increasing and decreasing), write it once using only `max_ptr` and once using only `min_ptr`. Write a precise contract for each of these functions and prove them.

```c
void max_ptr(int* a, int* b){
  if(*a < *b){
    int tmp = *b ;
    *b = *a ;
    *a = tmp ;
  }
}

void min_ptr(int* a, int* b){
  // use max_ptr
}

void order_3_inc_max(int* a, int* b, int* c){
  //in increasing order using max_ptr
}

void order_3_inc_min(int* a, int* b, int* c){
  //in increasing order using min_ptr
}

void order_3_dec_max(int* a, int* b, int* c){
  //in decreasing order using max_ptr
}

void order_3_dec_min(int* a, int* b, int* c){
  //in decreasing order using min_ptr
}
```

3.4.1.4. Give the change

The goal of this exercise is to write a function that computes the change on a purchase. The function `make_change` receives the amount to pay, the received amount of money, and a buffer to indicate how many of each note must be returned to the client.

For example, for an amount of 410 and a received amount of 500, the array should contain 1 in the cell `change[N50]` and 2 in the cell `change[N20]` after the call to the function.

If the received amount is less than the price, the function should return -1 (and oppositely, 0 on success).

```c
enum note { N500, N200, N100, N50, N20, N10, N5, N2, N1 };
int const values[] = { 500, 200, 100, 50, 20, 10, 5, 2, 1 };

int remove_max_notes(enum note n, int* rest){
  // ...
}

int make_change(int amount, int received, int change[9]){
  // ...

  int rest ;

  change[N500] = remove_max_notes(N500, &rest);
  // ...

  return 0;
}
```

The function `remove_max_notes` receives a note value and what remains to be changed (via a pointer), supposed to be greater than 0. It computes the maximum number of this note that can fit into the remaining, decreases it accordingly and returns the number of notes. The function `make_change` should then make use of this function to compute the change.

Write the code of these functions and their specification and prove their correctness. Note that this should not use loop as we still do not know how to deal with them.

3.4.1.5. Triangle

In this exercise, we want to gather the results of the functions we have written in the previous section to get the properties of triangles into a structure. The function `classify` receives three lengths `a`, `b`, and `c`, assuming that `a` is the hypotenuse. If these values do not correspond to a triangle, the function should return -1, and 0 if everything is OK. The properties are collected into a structure `info` received via a pointer.

```c
#include <limits.h>

enum Sides { SCALENE, ISOSCELE, EQUILATERAL };
enum Angles { RIGHT, ACUTE, OBTUSE };

struct TriangleInfo {
  enum Sides sides;
  enum Angles angles;
};

enum Sides sides_kind(int a, int b, int c){
  // ...
}

enum Angles angles_kind(int a, int b, int c){
  // ...
}

int classify(int a, int b, int c, struct TriangleInfo* info){
  // ...
}
```

Write, specify and prove all functions.

Note there are quite a lot of behaviors to list and specify. You can try to write a version that does not require `a` to be the hypotenuse, but it could be hard to finish the proof automatically with Alt-Ergo because there are quite a lot of combinations to consider.

During this part of the tutorial, we have studied how we can specify functions using contracts, composed of a pre and a postcondition, as well as some features ACSL provides to express those properties. We have also seen why it is important to be precise when we specify and how the introduction of behaviors can help us to write more understandable and safer specification.

However, we have not studied one important aspect: the proof of programs with loops. Before that, we should have a closer look at the way WP works.

4. Basic instructions and control structures

> This part is more formal than what we have seen so far. If the reader wishes to concentrate on the usage of the tool, they can skip the introduction and the first two sections (about the basic instructions and the bonus training) of this chapter. If what we presented so far has been difficult for the reader from a formal point of view, it is well possible to reserve the introduction and the two sections for a later reading.
> The sections on loops, however, are indispensable. We will highlight the more formal parts of these sections.

We will associate with every C programming construct:

- the corresponding inference rule together,
- its governing rule from the weakest precondition calculus and
- examples that show its usage.

Not necessarily in that order and sometimes only with a loose connection to the tool. Since the first rules are quite simple, we will discuss them in a fairly theoretical manner. Later on, however, our presentation will rely more and more on the tool, in particular when we begin dealing with loops.

4.0.1. Inference rules

An inference rule is of the form

$$\frac{P_1 \quad \dots \quad P_n}{C}$$

and means that in order to assure that the conclusion C is true, first the truth of the premises P_1, ..., and P_n has to be established. In case that the rule has no premises

$$\frac{}{C}$$

then nothing has to be assured in order to conclude the truth of C, and it is called an axiom.

On the other hand, in order to prove that a certain premise is true, it might be necessary to employ other inference rules which would lead to something like this:

$$\frac{\dfrac{}{P_1} \qquad \dfrac{\dfrac{}{P_{n_1}} \quad \dfrac{}{P_{n_2}}}{P_n}}{C}$$

This way, we obtain step by step the *deduction tree* of our reasoning. In our case, the premises and conclusions under consideration will in general be *Hoare triples*.

4.0.2. Hoare triples

We are now returning to concept of a Hoare triple:

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

In the beginning of this tutorial we have seen that this triple expresses the following: if the property P holds before the execution of C and if C terminates, then the property Q holds too. For example, if we take up again our (slightly modified) program for the computation of the absolute value:

```
/*@
  ensures \result >= 0;
  ensures (val >= 0 ==> \result == val ) && (val <  0 ==> \result == -val);
*/
int abs(int val){
  int res;
  if(val < 0) res = - val;
  else        res = val;

  return res;
}
```

The rules of Hoare logic tell us that in order to show that our program satisfies its contract we have to verify the properties shown in the braces. (We have omitted one postcondition in order to simplify the presentation.)

```
int abs(int val){
  int res;
// { P }
  if(val < 0){
// {  (val < 0) && P }
    res = - val;
// { \at(val, Pre) >= 0 ==> res == val && \at(val, Pre) < 0 ==> res == -val }
  } else {
// { !(val < 0) && P }
    res = val;
// { \at(val, Pre) >= 0 ==> res == val && \at(val, Pre) < 0 ==> res == -val }
  }
// { \at(val, Pre) >= 0 ==> res == val && \at(val, Pre) < 0 ==> res == -val }

  return res;
}
```

Yet, Hoare logic does not tell us how we can automatically obtain the precondition `P` of our program `abs`. Dijkstra's *weakest-precondition calculus*, on the other hand, allows us to compute from a given postcondition Q and a code snippet C the minimal precondition P that ensures Q after the execution of C. We are thus in a position to determine for our example `abs` the desired property `P`.

In this chapter, we present the different cases of the function *wp* which, starting from a given postcondition and a program (or statement), computes the *weakest* precondition that allows us to establish the validity of the postcondition. We will use the following notation to define the computation that corresponds to one or several statements:

$$wp(Instruction(s), Post) := WeakestPrecondition$$

The function wp will guarantee that the Hoare triple

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

really is a valid triple.

We will thereby often use ACSL assertions in order to represent the upcoming concepts:

```
//@ assert some_property ;
```

These assertions correspond in fact to possible intermediate steps for the properties in our Hoare triples. We can, for example, replace the properties of our function `abs` by corresponding ACSL assertions (we have omitted here the property `P` because it is just `true`):

```
int abs(int val){
  int res;
  if(val < 0){
    //@ assert val < 0 ;
    res = - val;
    //@ assert \at(val, Pre) >= 0 ==> res == val && \at(val, Pre) < 0 ==> res == -val ;
  } else {
    //@ assert !(val < 0) ;
    res = val;
    //@ assert \at(val, Pre) >= 0 ==> res == val && \at(val, Pre) < 0 ==> res == -val ;
  }
  //@ assert \at(val, Pre) >= 0 ==> res == val && \at(val, Pre) < 0 ==> res == -val ;

  return res;
}
```

4.1. Basic concepts

4.1.1. Assignment

Assignment is the most basic operation one can have in an imperative language (leaving aside the "do nothing" operation that is not particularly interesting). The weakest precondition calculus associates the following

$$wp(x = E, Post) := Post[x \leftarrow E]$$

Here the notation $P[x \leftarrow E]$ means "the property P where x is replaced by E". In our case this corresponds to "the postcondition $Post$ where x is replaced by E". The idea is that the postcondition of an assignment of E to x can only be true if replacing all occurrences of x in the formula by E leads to a property that is true. For example:

```
// { P }
x = 43 * c ;
// { x = 258 }
```

$$P = wp(x = 43 * c, \{x = 258\}) = \{43 * c = 258\}$$

The function wp allows us to compute, as weakest precondition of the assignment provided our expected postcondition, the formula $\{43*c = 258\}$, thus obtaining the following Hoare triple:

```c
// { 43*c = 258 }
x = 43 * c ;
// { x = 258 }
```

In order to compute the precondition of the assignment we have replaced each occurrence of x in the postcondition by the assigned value $E = 43 * c$. If our program were of the form:

```c
int c = 6 ;
// { 43*c = 258 }
x = 43 * c ;
// { x = 258 }
```

we could submit the formula $43*6 = 258$ to our automatic prover in order to determine whether it is really valid. The answer would of course be “yes” because the property is easy to verify. If we had, however, given the value 7 to the variable `c` the prover’s reply would be “no” since the formula $43 * 7 = 258$ is not true.

Taking into account the weakest precondition calculus, we can now write the inference rule for the Hoare triple of an assignment as

$$\frac{}{\{Q[x \leftarrow E]\} \quad x = E \quad \{Q\}}$$

We note that there is no premise to verify. Does this mean that the triple is necessarily true? Yes. However, it does not mean that the precondition is satisfied by the program to which the assignment belongs or that the precondition is at all possible. Here the automatic provers come into play.

For example, we can ask Frama-C to verify the following line:

```c
int a = 42;
//@ assert a == 42;
```

which is, of course, directly proven by Qed, since it is a simple application of the assignment rule.

We remark that according to the C standard, an assignment is in fact an expression. This allows us, for example, to write `if( (a = foo()) == 42)`. In Frama-C, an assignment will always be treated as a statement. Indeed, if an assignment occurs within a larger expression, then the Frama-C preprocessor, while building the abstract syntax tree, systematically performs a *normalization step* that produces a separate assignment statement.

4.1.1.1. Assignment of pointed value

In C, thanks to (because of?) pointers, we can have programs with aliases, meaning that two pointers can point to the same memory location. Our weakest precondition calculus should consider these cases. For example, let us consider this simple Hoare triple:

```
//@ assert p = q ;
*p = 1 ;
//@ assert *p + *q == 2 ;
```

This Hoare triple is correct, since `p` and `q` are in alias, modifying `*p` also modifies `*q`, thus both these expressions evaluate to 1 and the postcondition is true. However, let us apply the weakest precondition calculus from the postcondition:

$$\begin{aligned} wp(*p = 1, *p + *q = 2) &= (*p + *q = 2)[*p \leftarrow 1] \\ &= (1 + *q = 2) \end{aligned}$$

We get the weakest precondition: `1 + *q == 2`, and thus we could deduce that the weakest precondition is `*q == 1`, which is true, but does not allow us to conclude that the program is correct, since in our formula we do not have anything that models that `p == q ==> *q == 1`. In fact, here, we would like to be able to compute the weakest precondition like:

$$\begin{aligned} wp(*p = 1, *p + *q = 2) &= (1 + *q = 2 \lor q = p) \\ &= (*q = 1 \lor q = p) \end{aligned}$$

For this, we have to take care of aliasing. A common way to do this is to consider that the memory is one particular variable (let us name this variable M) on which we can perform two operations: get the element at a particular location l in memory (which returns an expression) and set the element at a particular location l to a new value v (which returns the new memory).

We denote:

- $get(M, l)$ with the notation $M[l]$
- $set(M, l, v)$ with the notation $M[l \mapsto v]$

And basically, the get operation can be seen as follows:

$$\begin{aligned} M[l1 \mapsto v][l2] = \quad & \text{if } l1 = l2 \text{ then } v \\ & \text{if } l1 \neq l2 \text{ then } M[l2] \end{aligned}$$

If there is no value associated to the location we use for a get, the value is undefined (thus, the memory is partial function). Of course, at the beginning of a function, the memory context can be populated with the memory locations for which a value is known to be defined.

Now, we can change a bit the weakest precondition calculus for assignment of pointed memory location. For this, we consider that we have an implicit variable M that models the memory, and we define the assignment of a memory location as an update of the memory such that now the corresponding pointer points to the written expression.

$$wp(*x = E, Q) := Q[M \leftarrow M[x \mapsto E]]$$

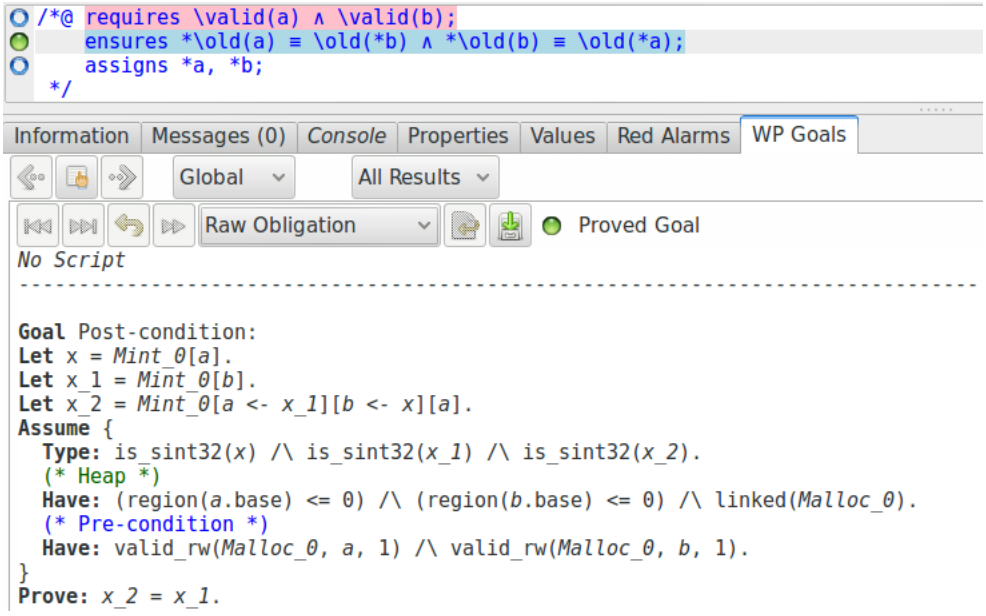
And evaluating a pointed value $*x$ in a formula now requires us to use the get operator to ask the right value. Thus, we can for example compute the weakest precondition of our previous program:

$$\begin{aligned}
wp(*p = 1, *p + *q = 2) &= (*p + *q = 2)[M \leftarrow M[p \mapsto 1]] && (1)\\
&= (M[p] + M[q] = 2)[M \leftarrow M[p \mapsto 1]] && (2)\\
&= (M[p \mapsto 1][p] + M[p \mapsto 1][q] = 2) && (3)\\
&= (1 + M[p \mapsto 1][q] = 2) && (4)\\
&= (1 + (\texttt{if}\ q = p\ \texttt{then}\ 1\ \texttt{else}\ M[q]) = 2) && (5)\\
&= (\texttt{if}\ q = p\ \texttt{then}\ 1 + 1 = 2\ \texttt{else}\ 1 + M[q] = 2) && (6)\\
&= (q = p \vee M[q] = 1) && (7)
\end{aligned}$$

1. we have to apply the rule of assignment for pointers, but for this we need to introduce M,
2. we replace pointer accesses in the formula by a call to *get* on M,
3. we apply the replacement asked by the assignment rule,
4. we use the definition of the *get* operator for the expression about p ($M[p \mapsto 1][p] = 1$)
5. we use the definition of the *get* operator for the expression about q ($M[p \mapsto 1][q] = \texttt{if}\ q = p\ \texttt{then}\ 1\ \texttt{else}\ M[q]$)
6. we perform some simplification to the formula ...
7. ... and finally conclude that either $M[q] = 1$ or $p = q$.

Then in our program, since we know that $p = q$, we can conclude that the program is correct.

The WP plugin does not exactly work like this. In particular, it depends on the memory model chosen for the proof that will make different assumption about the memory is organized. For the memory model we use, the typed memory model, in fact WP creates multiple variables for memory. However, let us have a look at the verification condition generated for the postcondition of the swap function:



```
/*@ requires \valid(a) ∧ \valid(b);
    ensures *\old(a) ≡ \old(*b) ∧ *\old(b) ≡ \old(*a);
    assigns *a, *b;
  */
```

```
No Script
--------------------------------------------------------------------------------

Goal Post-condition:
Let x = Mint_0[a].
Let x_1 = Mint_0[b].
Let x_2 = Mint_0[a <- x_1][b <- x][a].
Assume {
  Type: is_sint32(x) /\ is_sint32(x_1) /\ is_sint32(x_2).
  (* Heap *)
  Have: (region(a.base) <= 0) /\ (region(b.base) <= 0) /\ linked(Malloc_0).
  (* Pre-condition *)
  Have: valid_rw(Malloc_0, a, 1) /\ valid_rw(Malloc_0, b, 1).
}
Prove: x_2 = x_1.
```

We can see, in the beginning of the verification condition, that a variable `Mint_0` representing a memory of values of integer types have been created, and that this memory is updated and accessed using the operators we previously introduced (see the definition of the variable `x_2`).

4.1.2. Composition of statements

For a statement to be valid, its precondition must allow us by means of executing the said statement to reach the desired postcondition. Now we would like to execute several statements one after another. Here the idea is that the postcondition of the first statement is compatible with the required precondition of the second statement and so on for the third statement.

The inference rule that corresponds to this idea utilizes the following Hoare triples:

$$\frac{\{P\} \quad S1 \quad \{R\} \quad \{R\} \quad S2 \quad \{Q\}}{\{P\} \quad S1;\ S2 \quad \{Q\}}$$

In order to verify the composed statement $S1;\ S2$ we rely on an intermediate property R that is at the same time the postcondition of $S1$ and the precondition of $S2$. (Please note that $S1$ and $S2$ are not necessarily simple statements; they themselves can be composed statements.) The problem is, however, that nothing indicates us how to determine the properties P and R.

The weakest-precondition calculus now says us that the intermediate property R can be computed as the weakest precondition of the second statement. The property P, on the other hand, then is computed as the weakest precondition of the first statement. In other words, the weakest precondition of the composed statement $S1; S2$ is determined as follows:

$$wp(S1;\ \ S2, Post) := wp(S1, wp(S2, Post))$$

The WP plugin of Frama-C performs all these computations for us. Thus, we do not have to write the intermediate properties as ACSL assertions between the lines of codes.

```c
int main(){
  int a = 42;
  int b = 37;

  int c = a+b; // i:1
  a -= c;      // i:2
  b += a;      // i:3

  //@assert b == 0 && c == 79;
}
```

4.1.2.1. Proof tree

When we have more than two statements, we can consider the last statement as second statement of our rule and all the preceding ones as first statement. This way we traverse step by step backwards the statements in our reasoning. With the previous program this looks like:

$$\dfrac{\dfrac{\{P\} \quad i_1; \quad \{Q_{-2}\} \qquad \{Q_{-2}\} \quad i_2; \quad \{Q_{-1}\}}{\{P\} \quad i\_1; \quad i\_2; \quad \{Q_{-1}\}} \qquad \{Q_{-1}\} \quad i_3; \quad \{Q\}}{\{P\} \quad i\_1; \quad i\_2; \quad i\_3; \quad \{Q\}}$$

The weakest-precondition calculus allows us to construct the property Q_{-1} starting from the property Q and statement i_3 which in turn enables us to derive the property Q_{-2} from the property Q_{-1} and statement i_2. Finally, P can be determined from Q_{-2} and i_1.

Now that we can verify programs that consist of several statements it is time to add some structure to them.

4.1.3. Conditional rule

For a conditional statement to be true, one must be able to reach the postcondition through both branches. Of course, for both branches, the same precondition (of the conditional statement) must hold. In addition, we have that in the if-branch the condition is true while in the else-branch it is false.

We therefore have, as in the case of composed statements, two facts to verify (in order to avoid confusion we are using here the syntax $if\ \ B\ then\ S1\ else\ S2$):

$$\dfrac{\{P \wedge B\} \quad S1 \quad \{Q\} \qquad \{P \wedge \neg B\} \quad S2 \quad \{Q\}}{\{P\} \quad if \quad B \quad then \quad S1 \quad else \quad S2 \quad \{Q\}}$$

Our two premises are therefore that we can both in the if-branch and the else-branch reach the postcondition from the precondition.

The result of the weakest-precondition calculus for a conditional statement reads as follows:

$$wp(if\ \ B\ then\ S1\ else\ S2, Post) := (B \Rightarrow wp(S1, Post)) \wedge (\neg B \Rightarrow wp(S2, Post))$$

This means that the condition B has to imply the weakest precondition of $S1$ in order to safely arrive at the postcondition. Analogously, the negation of B must imply the weakest precondition of $S2$.

4.1.3.1. Empty `else` -branch

Following this definition, we obtain for the case of an empty else-branch the following rule by simply replacing the statement $S2$ by the empty statement `skip` .

$$\frac{\{P \wedge B\} \quad S1 \quad \{Q\} \qquad \{P \wedge \neg B\} \quad skip \quad \{Q\}}{\{P\} \quad if \quad B \quad then \quad S1 \quad else \quad skip \quad \{Q\}}$$

The triple for `else` is:

$$\{P \wedge \neg B\} \quad skip \quad \{Q\}$$

which means that we need to ensure:

$$P \wedge \neg B \Rightarrow Q$$

In short, if the condition B of `if` is false, this means that the postcondition of the complete conditional statement is already established before entering the else-branch (since it does not do anything).

As an example, we consider the following code snippet where we reset a variable c to a default value in case it had not been properly initialized by the user.

```c
int c;

// ... some code ...

if(c < 0 || c > 15){
  c = 0;
}
//@ assert 0 <= c <= 15;
```

Let

$wp(if \neg(c \in [0;15])\ then\ c := 0, \{c \in [0;15]\})$

$:= (\neg(c \in [0;15]) \Rightarrow wp(c := 0, \{c \in [0;15]\})) \wedge (c \in [0;15] \Rightarrow wp(skip, \{c \in [0;15]\}))$

$= (\neg(c \in [0;15]) \Rightarrow 0 \in [0;15]) \wedge (c \in [0;15] \Rightarrow c \in [0;15])$

$= (\neg(c \in [0;15]) \Rightarrow true) \wedge true$

The property can be verified: independent of the evaluation of $\neg(c \in [0;15])$, the implication will hold.

4.1.4. Bonus Stage - Consequence rule

It can sometimes be useful to strengthen a postcondition or to weaken a precondition. The former will often be established by us to facilitate the work of the prover, the latter is more often verified by the tool as the result of computing the weakest precondition.

The inference rule of Hoare logic is the following:

$$\frac{P \Rightarrow WP \quad \{WP\} \quad c \quad \{SQ\} \quad SQ \Rightarrow Q}{\{P\} \quad c \quad \{Q\}}$$

(We remark that the premises here are not only Hoare triples but also formulas to verify.)

For example, if our postcondition is too complex, it may generate a weaker precondition that is, however, too complicated, thus making the work of provers more difficult. We can then create a simpler intermediate postcondition SQ, that is, however, stricter and implies the real postcondition. This is the part $SQ \Rightarrow Q$.

Conversely, the calculation of the precondition will usually generate a complicated and often weaker formula than the precondition we want to accept as input. In this case, it is our tool that will check the implication between what we want and what is necessary for our code to be valid. This is the part $P \Rightarrow WP$.

We can illustrate this with the following code. Note that here the code could be proved by WP without the weakening and strengthening of properties because the code is very simple, it is just to illustrate the rule of consequence.

```
/*@
  requires P: 2 <= a <= 8;
  ensures  Q: 0 <= \result <= 100 ;
  assigns  \nothing ;
*/
int constrained_times_10(int a){
  //@ assert P_imply_WP: 2 <= a <= 8 ==> 1 <= a <= 9 ;
  //@ assert WP:         1 <= a <= 9 ;

  int res = a * 10;

  //@ assert SQ:         10 <= res <= 90 ;
  //@ assert SQ_imply_Q: 10 <= res <= 90 ==> 0 <= res <= 100 ;

  return res;
}
```

(Note: We have omitted here the control of integer overflow.)

Here we want to have a result between 0 and 100. But we know that the code will not produce a result outside the bounds of 10 and 90. So we strengthen the postcondition with an assertion that at the end `res`, the result, is between 0 and 90. The calculation of the weakest precondition of this property together with the assignment `res = 10 * a` yields a weaker precondition `1 <= a <= 9`, and we know that `2 <= a <= 8` gives us the desired guarantee.

When there are difficulties to carry out a proof on more complex code, then it is often helpful to write assertions that produce stronger, yet easier to verify, postconditions. Note that in the previous code, the lines `P_imply_WP` and `SQ_imply_Q` are never used because this is the default reasoning of WP. They are just here for illustrating the rule.

4.1.5. Bonus Stage - Constancy rule

Certain sequences of instructions may concern and involve different variables. Thus, we may initialize and manipulate a certain number of variables, begin to use some of them for a time, before using other variables. When this happens, we want our tool to be concerned only with variables that are susceptible to change in order to obtain the simplest possible properties.

The rule of inference that defines this reasoning is the following:

$$\frac{\{P\} \quad c \quad \{Q\}}{\{P \wedge R\} \quad c \quad \{Q \wedge R\}}$$

where c does not modify any variable in R. In other words: "To check the triple, let's get rid of the parts of the formula that involve variables that are not influenced by c and prove the new triple." However, we must be careful not to delete too much information, since this could mean that we are not able to prove our properties.

As an example, let us consider the following code (here gain, we ignore potential integer overflows):

```c
/*@
  requires a > -99 ;
  requires b > 100 ;
  ensures  \result > 0 ;
  assigns  \nothing ;
*/
int foo(int a, int b){
  if(a >= 0){
    a++ ;
  } else {
    a += b ;
  }
  return a ;
}
```

If we look at the code of the `if` block, we notice that it does not use the variable `b`. Thus, we can completely omit the properties about `b` in order to prove that `a` will be strictly greater than 0 after the execution of the block:

```c
/*@
  requires a > -99 ;
  requires b > 100 ;
  ensures  \result > 0 ;
  assigns  \nothing ;
*/
int foo(int a, int b){
  if(a >= 0){
    //@ assert a >= 0; // and nothing about b
    a++ ;
  } else {
    a += b ;
  }
  return a ;
}
```

On the other hand, in the `else` block, even if `b` is not modified, formulating properties only about `a` would render a proof impossible for humans. The code would be:

```c
/*@
  requires a > -99 ;
  requires b > 100 ;
  ensures  \result > 0 ;
  assigns  \nothing ;
*/
int foo(int a, int b){
  if(a >= 0){
    //@ assert a >= 0; // and nothing about b
    a++ ;
  } else {
    //@ assert a < 0 && a > -99 ; // and nothing about b
    a += b ;
  }
  return a ;
}
```

In the `else` block, knowing that `a` lies between -99 and 0, but knowing nothing about `b`, we could hardly know if the operation `a += b` produces a result that is greater than 0.

The WP plugin will, of course, prove the function without problems, since it produces by itself the properties that are necessary for the proof. In fact, the analysis which variables are necessary or not (and, consequently, the application of the constancy rule) is conducted directly by WP.

Let us finally remark that the constancy rule is an instance of the consequence rule

$$\frac{P \wedge R \Rightarrow P \quad \{P\} \quad c \quad \{Q\} \quad Q \Rightarrow Q \wedge R}{\{P \wedge R\} \quad c \quad \{Q \wedge R\}}$$

If the variables of R have not been modified by the operation (which, on the other hand, may modify the variables of P to produce Q), then the properties $P \wedge R \Rightarrow P$ and $Q \Rightarrow Q \wedge R$ hold.

4.1.6. Assertion

ACSL assertions needs to be proved (or used). Thus, they need their WP calculus rules. There are three different kinds of assertions in ACSL:

- `assert`
- `check`
- `admit`

Each kind of assertion has a different behavior.

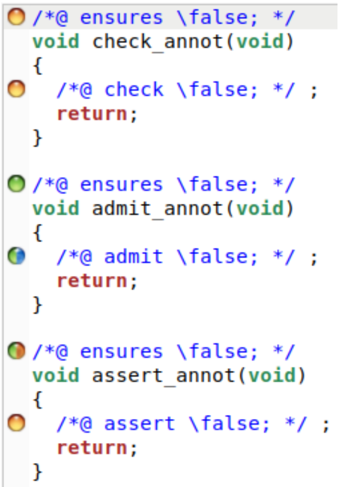
Let us start with the one we have used so far: `assert`. An annotation `assert P` has the following informal semantics: the property `P` must be verified at this program point, and then the fact that it is true at this program point is added as a hypothesis when proving properties that come later. Note that it might not be true, but it is still added to the proof context anyway. However, if it is not true, first, we cannot to prove it, and second, the properties that we are proved admitting this property are marked as “valid under hypothesis”.

The annotation `check P` means that the property `P` must be verified at this program point, but then it is not added to the proof context for the proof of the annotations that follow it.

Finally, the annotation `admit P` means that from this program point, the property `P` is considered to be true, thus added to the proof context of the annotations that follow it, even if we do not verify it at this program point.

Let us illustrate these different kind of annotations with this short example:

```
//@ ensures \false ;
void check_annot(void){
  //@ check \false ;
}

//@ ensures \false ;
void admit_annot(void){
  //@ admit \false ;
}

//@ ensures \false ;
void assert_annot(void){
  //@ assert \false ;
}
```



```
/*@ ensures \false; */
void check_annot(void)
{
  /*@ check \false; */ ;
  return;
}

/*@ ensures \false; */
void admit_annot(void)
{
  /*@ admit \false; */ ;
  return;
}

/*@ ensures \false; */
void assert_annot(void)
{
  /*@ assert \false; */ ;
  return;
}
```

In the `check_annot` function, neither the `check` nor the `ensures` clauses are proved. Each one have been tried separately (and failed).

In the `admit_annot` function, the `admit` annotation appear with a blue and green bullet (just like when we have a contract on a function declaration without a body), meaning that from this point, it is considered to be valid even it is not proved. Thus, the `ensures` clause is proved, since `\false` is assumed to be true.

Finally, in the `assert_annot` function, the `assert` annotation is not proved, but from this point it is assumed to be true. Thus, the `ensures` clause is proved. However, while in the `admit_annot` function it appears entirely valid (green bullet), here, it appears valid under hypothesis: the validity depends on the validity of a property that has not been proved yet (and will not).

We can encode these different behaviors with the following WP rules.

The rule for the `check` annotation is the following:

$$\mathit{wp}(\mathit{check}\ A, \mathit{Post}) := A \land \mathit{Post}$$

which means that we add to the properties we must verify another property, which is the property A.

The rule for the `admit` annotation is the following:

$$\mathit{wp}(\mathit{admit}\ A, \mathit{Post}) := A \Rightarrow \mathit{Post}$$

which means that assuming A, we must verify the postcondition (of the statement, which can be a property that has been computing using other WP rules).

Finally, the rule for the `assert` combines the previous ones. Indeed:

```
assert P;
```

is equivalent to:

```
check P; // first, verify that P is true
admit P; // now, we can assume that it is
```

Thus:

$$\mathit{wp}(\mathit{assert}\ A, \mathit{Post}) := \mathit{wp}(\mathit{check}\ A, \mathit{wp}(\mathit{admit}\ A, \mathit{Post})) \equiv A \land (A \Rightarrow \mathit{Post})$$

This allows to introduce some sort of cut in the proof, saying ”OK, let us first prove A and once it is done, we will prove that when A is true, Post is too”.

One could think the `admit` annotation is dangerous. It is. But it can be useful when one wants to debug a proof. In particular, in Chapter 7, we will see that `assert` annotations can be used to guide program proof. During the process, `admit` can be useful to try annotations without having to immediately prove them or to consider some annotations as valid to make the design phase faster. More rarely, it can be useful to explicit some hypotheses about hardware, or software platform, but for this, it is more common to use Frama-C kernel options or axioms (we will present axioms in Section 6.2).

4.1.7. WP plugin vs. Dijkstra's WP calculus

One might have notice from the beginning of this section that the *wp* function starts from the postcondition and, instruction after instruction, changes the formula until it reaches the beginning of the function, when it generates the verification condition (as briefly explained in Section 4.1.4), that we must verify.

However, one might have also noticed that from the very beginning of this book, in most examples, we do not have *one* verification condition to verify, but *several*. On a theoretical aspect, this is not a crucial difference, but on a practical aspect, it allows generating simpler verification conditions, thus easier to verify using SMT solvers.

In fact, the WP plugin generates several verification conditions in parallel during the WP calculus. Each time it meets a new instruction, the *wp* function is applied for this instruction on all verification conditions involved in the program path the instruction belongs to (to guarantee this, the conditional rule is also handled slightly differently, but we will not detail this aspect). However, when this step includes the proof of a property (for example, an `assert` annotation), instead of applying the rule by adding it as a conjunction, it is simply added separately to the set of verification conditions to prove. Intuitively: we start a new weakest precondition calculus in parallel from this program point.

Let us illustrate this on a toy example:

```c
/* run.config
  DONTRUN:
*/

int x ;
int y ;

// 8.
/*@ requires P(x) ;
    ensures P(x) ;
    ensures P(y) ;
*/
void toy(void){
  // 7.
  if(x > 0){
    // 6.
    x ++ ;
    // 5.
    //@ assert Q(x);
  } else {
    // 4.
    y ++ ;
    // 3.
    //@ check Q(y);
  }
  // 2.
  x = x * y ;
  // 1.
}
```

We start (in 1) with two verification conditions:

- `VC1: P(x)`,
- `VC2: P(y)`.

We apply the transformer of assignment on each of them, thus we reach 2 with:

- `VC1: P(x * y)`,
- `VC2: P(y)`.

In 2, we have to deal with the conditional. Instead of directly applying the standard WP rule, we split the analysis in two sets of verification conditions, initially equivalent. Let us consider the `else` branch, then the `then` branch.

From 2 to 3, we meet the `check` annotation, thus we add a new verification condition, we get the set:

- `VC1: P(x * y)`,
- `VC2: P(y)`,
- `VC3: Q(y)`.

Then, we apply the assignment rule, and we get in 4:

- `VC1: P(x * (y+1))`,
- `VC2: P(y+1)`,
- `VC3: Q(y+1)`.

From 2 to 5, we meet the `assert` annotation, thus we add this knowledge to all verification conditions, and we create a new one:

- `VC1: Q(x) ==> P(x * y)`,
- `VC2: Q(x) ==> P(y)`,
- `VC4: Q(x)`.

Then we apply the assignment rule, and we get in 6:

- `VC1: Q(x+1) ==> P((x+1) * y)`,
- `VC2: Q(x+1) ==> P(y)`,
- `VC4: Q(x+1)`.

Now we have to reconcile the two sets. Let us not enter into the details (just notice the introduced primed variables, the trick is here) of how it is done, we get, in 7, something like:

```
VC1:
  ((x <= 0 ==> y' == y+1 && x' == x) &&
   (x >  0 ==> y' == y   && x' == x+1 && Q(x+1))) ==>
     P(x' * y')

VC2:
  ((x <= 0 ==> y' == y+1) &&
   (x >  0 ==> y' == y    && Q(x+1))) ==>
     P(y')
VC3:
  (x <= 0) ==> Q(x+1)

VC4:
  (x >  0) ==> Q(x+1)
```

Finally, we meet the precondition, and we obtain in 8:

```
  VC1:
    P(x) ==>
      ((x <= 0 ==> y' == y+1 && x' == x) &&
       (x >  0 ==> y' == y   && x' == x+1 && Q(x+1))) ==>
         P(x' * y')

  VC2:
    P(x) ==>
      ((x <= 0 ==> y' == y+1) &&
       (x >  0 ==> y' == y    && Q(x+1))) ==>
         P(y')
  VC3:
    P(x) ==> (x <= 0) ==> Q(x+1)

  VC4:
    P(x) ==> (x >  0) ==> Q(x+1)
```

We can see that for each property that must be proved, we have built a separate verification condition that gathers the knowledge available along the path that leads to the program point where its verification is asked.

4.1.8. Exercises

4.1.8.1. A series of assignment

Compute by hand the weakest precondition of the following program:

```
/*@
  requires -10 <= x <= 0 ;
  requires 0 <= y <= 5 ;
  ensures -10 <= \result <= 10 ;
*/
int function(int x, int y){
  int res ;
  y += 10 ;
  x -= 5 ;
  res = x + y ;
  return res ;
}
```

Deduce that the program is correct with respect to its contract using the right rule.

4.1.8.2. Empty “then” branch in conditional

We have previously shown that when a conditional structure has an empty “else” branch, that means that the conjunction of the precondition and the negation of the condition must be enough to verify the postcondition of the complete conditional structure. For both of the following question, we only need the inference rules and no WP calculus.

Show that when, instead of the "else" branch, the "then" branch is empty, the postcondition is verified by the conjunction of the precondition and the condition (as only the "else" can change the memory state).

Show that when both branches are empty, the overall condition is just a skip operation.

4.1.8.3. Short circuit

C compilers implement short circuit for conditions. For example, that means that a code like this one (**without "else" block**) :

```c
if(cond1 && cond2){
  // code
}
```

can be written as:

```c
if(cond1){
  if(cond2){
    // code
  }
}
```

Show that on those two source code, the weakest precondition calculus generates an equivalent weakest precondition for equivalent for any code in the "then" block. Note that we assume the conditions to be pure expressions (without side effects).

4.1.8.4. A larger program

Compute by hand the weakest precondition of the following program:

```c
/*@
  requires -5 <= y <= 5 ;
  requires -5 <= x <= 5 ;
  ensures  -15 <= \result <= 25 ;
*/
int function(int x, int y){
  int res ;

  if(x < 0){
    x = 0 ;
  }

  if(y < 0){
    x += 5 ;
  } else {
    x -= 5 ;
  }

  res = x - y ;

  return res ;
}
```

Deduce that the program is correct with respect to its contract using the right rule.

4.2. Loops

Loops need a particular treatment in deductive verification of programs. These are the only control structures that will require important work from us. We cannot avoid this because without loops, it is difficult to write and prove interesting programs.

Before we look at the way we specify loop, we can answer to a rightful question: why are loops so complex?

4.2.1. Induction and invariant

The nature of loops makes their analysis complex. When we perform our reasoning, we need a rule to determine the precondition from a given sequence of instructions and a postcondition. Here, the problem is that we cannot *a priori* deduce how many times a loop iterates, and consequently, we cannot know how many times variables are modified.

We then proceed using an inductive reasoning. We have to find a property that is true before we start to execute the loop and that, if it is true at the beginning of an iteration, remains true at the end (and that is consequently true at the beginning of the next iteration). When the loop ends, we add the knowledge that the condition of the loop is false and that should allow us to deduce that the postcondition of the loop is verified.

This type of property is called a loop invariant. A loop invariant is a property that must be true before and after each loop iteration. And more precisely, each time the condition of the loop is checked. For example with the following loop:

```c
for(int i = 0 ; i < 10 ; ++i){ /* */ }
```

The property $0 \leq \texttt{i} \leq 10$ is a loop invariant. The property $-42 \leq \texttt{i} \leq 42$ is also an invariant (even if it is far less precise). The property $0 < \texttt{i} \leq 10$ is not an invariant because it is not true at the beginning of the execution of the loop. The property $0 \leq \texttt{i} < 10$ **is not a loop invariant**, it is not true at the end of the last iteration that sets the value of `i` to 10.

To verify an invariant I, WP then produces the following “reasoning”:

- verify that I is true at the beginning of the loop (establishment)
- verify that if I is true before an iteration, then I is true after (preservation).

4.2.1.1. Formal - Inference rule

Let us note the invariant I, the inference rule corresponding to loops is defined as follows:

$$\frac{\{I \wedge B\}\ c\ \{I\}}{\{I\}\ while(B)\{c\}\ \{I \wedge \neg B\}}$$

And the weakest precondition calculus is the following:

$$wp(while(B)\{c\}, Post) := I \wedge ((B \wedge I) \Rightarrow wp(c, I)) \wedge ((\neg B \wedge I) \Rightarrow Post)$$

Let us detail this formula:

- (1) The first I corresponds to the establishment of the invariant, in layman's terms, this is the "precondition" of the loop.
- The second part of the conjunction $((B \wedge I) \Rightarrow wp(c, I))$ corresponds to the verification of the operation performed by the body of the loop:
  - the precondition that we know of the loop body (let us note KWP, "Known WP") is $(KWP = B \wedge I)$. That is the fact we have entered the loop (B is true), and that the invariant is verified at this moment (I, is true before we start the loop by (1), and we want to verify that it will be true at the end of the body of the loop in (2)),
  - (2) it remains to verify that KWP implies the actual precondition* of the body of the loop $(KWP \Rightarrow wp(c, Post))$. What we want at the end of the loop is the preservation of the invariant I (B is maybe not true anymore however), formally $KWP \Rightarrow wp(c, I)$, that is to say $(B \wedge I) \Rightarrow wp(c, I)$,
  - * it corresponds to the application of the consequence rule previously explained.
- Finally, the last part $((\neg B \wedge I) \Rightarrow Post)$ expresses the fact that when the loop ends$(\neg B)$, and the invariant I has been maintained, it must imply that the wanted postcondition of the loop is verified.

In this computation, we can notice that the wp function does not indicate any way to obtain the invariant I. We have to specify ourselves this property about our loops.

4.2.1.2. Back to the WP plugin

There exist tools that can infer invariant properties (provided that these properties are simple, automatic tools remain limited). This is not the case for WP. We have to manually annotate our programs to specify the invariant of each loop. To find and write invariants for our loops will always be the hardest part of our work when we want to prove programs.

Indeed, when there are no loops, the weakest precondition calculus function can automatically provide the verifiable properties of our programs, this is not the case for loop invariant properties for which we do not have computation procedures. We have to find and express them correctly, and depending on the algorithm, they can be quite subtle and complex.

In order to specify a loop invariant, we add the following annotations before the loop:

```
int main(){
  int i = 0;

  /*@
    loop invariant 0 <= i <= 30;
  */
  while(i < 30){
    ++i;
  }
  //@ assert i == 30;
}
```

REMINDER : The invariant is: i $\leq$ 30 !

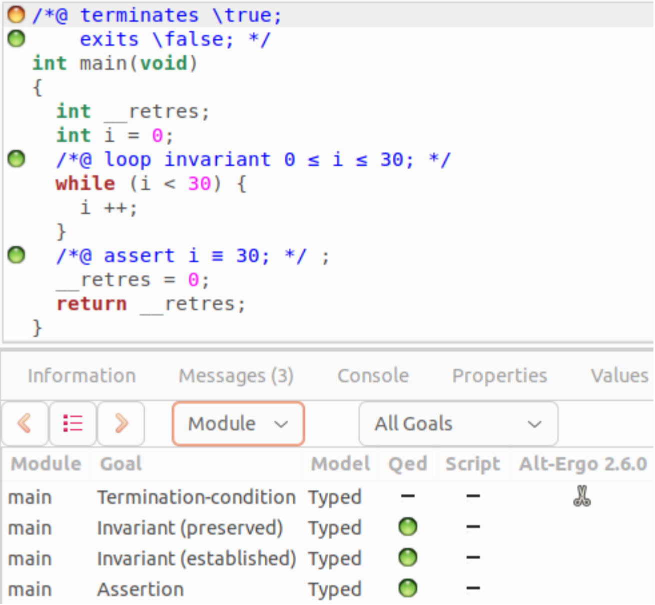
Why? Because along the loop, `i` is comprised between 0 and **included** 30. 30 is indeed the value that allows us to leave the loop. Moreover, one of the properties required by the weakest precondition calculus is that when the loop condition is invalidated, by knowing the invariant, we can prove the postcondition (Formally $(\neg B \wedge I) \Rightarrow Post$).

The postcondition of our loop is $\texttt{i} = 30$ and must be implied by $\neg\ \texttt{i} < 30\ \wedge\ 0 \leq \texttt{i} \leq 30$. Here, it is true since:

$$\texttt{i} \geq 30 \wedge 0 \leq \texttt{i} \leq 30 \Rightarrow \texttt{i} = 30$$

On the opposite, if we exclude the equality to 30, the postcondition would be unreachable.

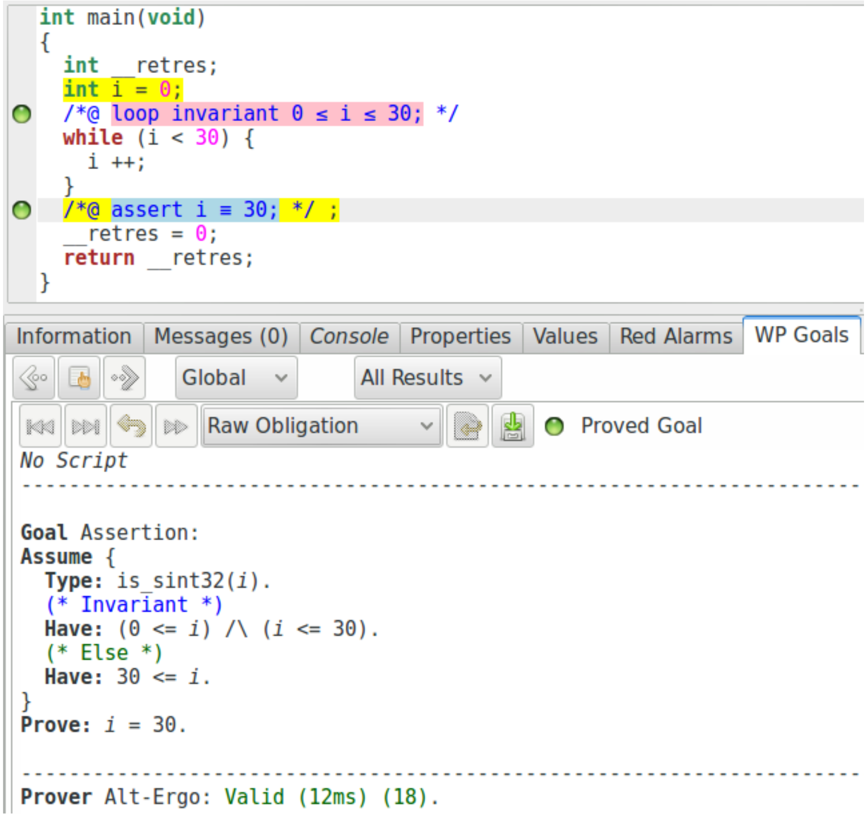
Again, we can have a look at the list of verification conditions in “WP Goals”:



```
/*@ terminates \true;
    exits \false; */
int main(void)
{
  int __retres;
  int i = 0;
  /*@ loop invariant 0 ≤ i ≤ 30; */
  while (i < 30) {
    i ++;
  }
  /*@ assert i ≡ 30; */ ;
  __retres = 0;
  return __retres;
}
```

Module	Goal	Model	Qed	Script	Alt-Ergo 2.6.0
main	Termination-condition	Typed	–	–	✂
main	Invariant (preserved)	Typed	●	–	
main	Invariant (established)	Typed	●	–	
main	Assertion	Typed	●	–	

We see that the termination condition is not proved, but let us ignore that right now. We notice that WP produces two different verification conditions: the establishment of the invariant and its preservation. WP produces exactly the reasoning we previously described to prove the assertion. The Qed simplifier is powerful enough to directly finish the proof (showing directly “True”). Using the option `-wp-no-simpl` at start, we can however see these details:



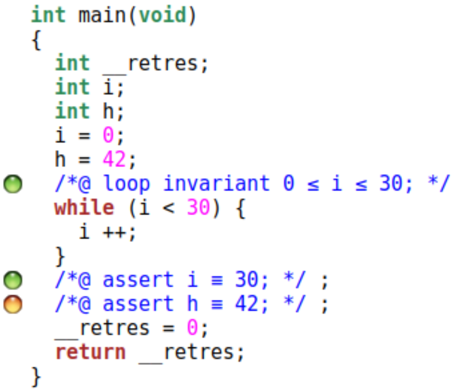
```
int main(void)
{
  int __retres;
  int i = 0;
  /*@ loop invariant 0 ≤ i ≤ 30; */
  while (i < 30) {
    i ++;
  }
  /*@ assert i ≡ 30; */ ;
  __retres = 0;
  return __retres;
}
```

```
No Script
--------------------------------------------------------------------

Goal Assertion:
Assume {
  Type: is_sint32(i).
  (* Invariant *)
  Have: (0 <= i) /\ (i <= 30).
  (* Else *)
  Have: 30 <= i.
}
Prove: i = 30.

--------------------------------------------------------------------
Prover Alt-Ergo: Valid (12ms) (18).
```

But is our specification precise enough?

```
int main(){
  int i = 0;
  int h = 42;

  /*@
    loop invariant 0 <= i <= 30;
  */
  while(i < 30){
    ++i;
  }
  //@assert i == 30;
  //@assert h == 42;
}
```

And the result is:



```
int main(void)
{
  int __retres;
  int i;
  int h;
  i = 0;
  h = 42;
  /*@ loop invariant 0 ≤ i ≤ 30; */
  while (i < 30) {
    i ++;
  }
  /*@ assert i ≡ 30; */ ;
  /*@ assert h ≡ 42; */ ;
  __retres = 0;
  return __retres;
}
```

It seems not.

4.2.2. The `assigns` clause … for loops

In fact, considering loops, WP **only** reasons about what is provided by the user to perform its reasoning. And here, the invariant does not specify anything about the way the value of `h` is modified (or not). We could specify the invariant of all program variables, but it would be a lot of work. ACSL simply allows adding `assigns` annotations for loops. Any other variable is considered to keep its old value. For example:

```c
int main(){
  int i = 0;
  int h = 42;

  /*@
    loop invariant 0 <= i <= 30;
    loop assigns i;
  */
  while(i < 30){
    ++i;
  }
  //@assert i == 30;
  //@assert h == 42;
}
```

This time, we can establish the proof that the loop correctly behaves. However, we cannot prove that it terminates. The loop invariant alone does not give enough information to perform such a proof. For example, in our program, we could modify the loop, removing the loop body:

```c
/*@
  loop invariant 0 <= i <= 30;
  loop assigns i;
*/
while(i < 30){

}
```

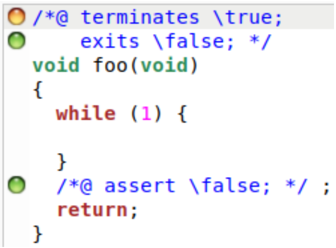
The invariant is still verified, but we cannot prove that the loop ends: it is infinite.

4.2.3. Partial correctness and total correctness - Loop variant

In deductive verification, we find two types of correctness, the partial correctness and the total correctness. In the first case, the formulation of the correctness property is “if the precondition is valid, and **if** the computation terminates, then the postcondition is valid”. In the second case, “if the precondition is valid, **then** the computation terminates and the postcondition is valid”. By default, WP only verify partial correctness, *but* asks the kernel to generate the `terminates` clause so that we are forced to verify the termination anyway (except if we voluntarily change the clause as explained in Section 4.4.3). Thus, if we try to verify the following program:

```
void foo(){
  while(1){}
  //@ assert \false;
}
```

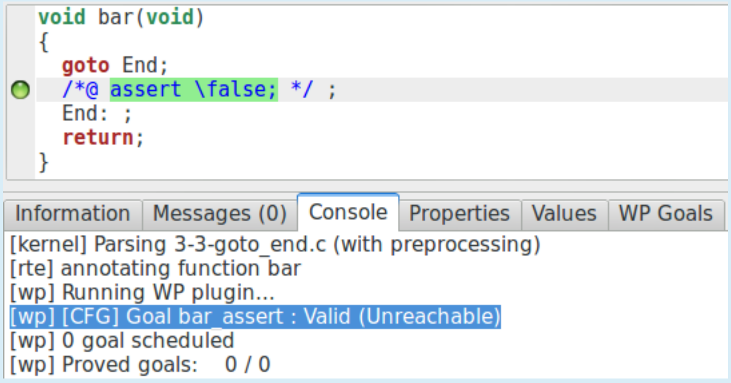
we get this result:



```
/*@ terminates \true;
    exits \false; */
void foo(void)
{
  while (1) {

  }
  /*@ assert \false; */ ;
  return;
}
```

The assertion "False" is proved! For a very simple reason: since the condition of the loop is "True" and no instruction of the loop body allows to leave the loop, it does not terminate. As we are proving the code with partial correctness, and as the execution does not terminate, we can prove anything about the code that follows the non-terminating part of the code. However, if the termination is not trivially provable, the assertion will probably not be proved. However, because we have not proved the termination of the loop, the `terminates` clause is not proved, and we see that it might be a problem.

Note that a (provably) unreachable assertion is always proved to be true:



```
void bar(void)
{
  goto End;
  /*@ assert \false; */ ;
  End: ;
  return;
}
```

```
[kernel] Parsing 3-3-goto_end.c (with preprocessing)
[rte] annotating function bar
[wp] Running WP plugin...
[wp] [CFG] Goal bar_assert : Valid (Unreachable)
[wp] 0 goal scheduled
[wp] Proved goals:    0 / 0
```

And this is also the case when we trivially know that an instruction produces a runtime error (for example dereferencing `NULL`), or inserting "False" in postcondition as we have already seen with `abs` and the parameter `INT_MIN`.

4.2.3.1. Proof of termination - Providing a measure

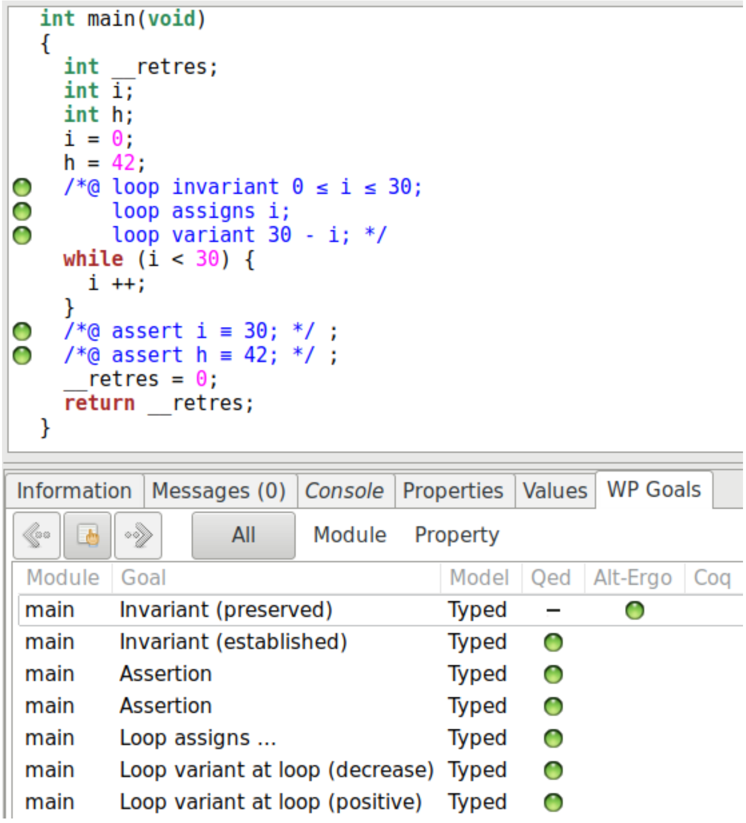
In program proof, when we need to prove termination of an algorithm, we introduce a notion of *measure*. A measure is an expression that must be strictly decreasing according to a given well-founded relation R. From a "step" of the computation to another, we want the measure to decrease according to R. In ACSL, by default, the measure is an integer expression that must be decreasing and positive: $R(x, y) \Leftrightarrow x > y \wedge x \geq 0$, but one can specify another relation (see Section 4.2.3.2).

For loops, the measure is specified through a loop variant, and the notion of "step" in the computation is the loop iteration. A loop variant is not a property but an expression that involves some variables modified by the loop and that provides an upper bound to the number of iterations that remains to be executed by the loop before any iteration. Thus, this expression is greater or equals to 0, and strictly decreases at each loop iteration. Now, since for any loop iteration the value decreases, but it remains positive, that means that the loop necessarily stops after a finite number of iterations.

If we take our previous example, we add the loop variant with this syntax:

```c
int main(){
  int i = 0;
  int h = 42;

  /*@
    loop invariant 0 <= i <= 30;
    loop assigns i;
    loop variant 30 - i;
  */
  while(i < 30){
    ++i;
  }
  //@assert i == 30;
  //@assert h == 42;
}
```

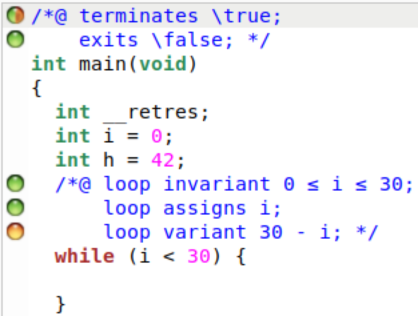
Again, we can have a look at the generated verification conditions:



```
  int main(void)
  {
    int __retres;
    int i;
    int h;
    i = 0;
    h = 42;
●   /*@ loop invariant 0 ≤ i ≤ 30;
●       loop assigns i;
●       loop variant 30 - i; */
    while (i < 30) {
      i ++;
    }
●   /*@ assert i ≡ 30; */ ;
●   /*@ assert h ≡ 42; */ ;
    __retres = 0;
    return __retres;
  }
```

Module	Goal	Model	Qed	Alt-Ergo	Coq
main	Invariant (preserved)	Typed	–	●	
main	Invariant (established)	Typed	●		
main	Assertion	Typed	●		
main	Assertion	Typed	●		
main	Loop assigns ...	Typed	●		
main	Loop variant at loop (decrease)	Typed	●		
main	Loop variant at loop (positive)	Typed	●		

WP generates two verification conditions for the loop variant: verify that the value of the expression specified in the variant is positive, and prove that it strictly decreases during the execution of the loop. We also see that with the loop variant, WP can prove that the function terminates. And if we delete the line of code that increments `i`, WP cannot prove anymore that `30 - i` strictly decreases, and the `terminates` clause is now “proved under hypothesis”.



```
/*@ terminates \true;
    exits \false; */
int main(void)
{
  int __retres;
  int i = 0;
  int h = 42;
  /*@ loop invariant 0 ≤ i ≤ 30;
      loop assigns i;
      loop variant 30 - i; */
  while (i < 30) {

  }
```

We will give more details about how the `terminates` clause is proved later. For now, note that, since the loop variant is an upper bound on the number of remaining iterations, being able to give a loop variant does not necessarily induce that we can give the exact number of remaining iterations of the loop, as we do not always have a so precise knowledge of the behavior of the program. We can for example build an example like this one:

```c
#include <stddef.h>

/*@
  ensures min <= \result <= max;
*/
size_t random_between(size_t min, size_t max);

void random_loop(size_t bound){
  /*@
    loop invariant 0 <= i <= bound ;
    loop assigns i;
    loop variant i;
  */
  for(size_t i = bound; i > 0; ){
    i -= random_between(1, i);
  }
}
```

Here, at each iteration, we decrease the value of the variable `i` by a value comprised between 1 and `i`. Thus, we can ensure that the value of `i` is positive and strictly decreases during each loop iteration, but we cannot tell how many loop iterations remain to be executed.

Note also that a loop variant only needs to be positive at the beginning of the execution of the block of the loop. Thus, in the following code:

```c
int i = 5 ;
while(i >= 0){
  i -= 2 ;
}
```

Even if `i` can be negative when the loop exits, it is still a variant since we do not start the execution of the block of the loop again.

4.2.3.2. General measure

> In this short section, we present a particular usage of loop variants. It uses an ACSL feature that is presented in the next part of the tutorial, yet the section should be understandable without having read the next part. Note that, since this way of using loop variants is rarely useful, it can be ignored in a first read.

Most of the time a simple integer measure is enough to express loop variants and prove termination of loops. However, in some situations, it might be hard to use. In such a case, ACSL allows the notion of generalized variant where one can specify an expression of any type, as long as they can provide a well-founded relation that correspond to this type. However, WP **does not** verify that the provided relation is well-founded, it must be proved in some other way. Such a measure is provided through the following syntax:

```
/*@ loop variant <term> for <Relation> ;
```

The `Relation` predicate must be a defined binary predicates of the type of `term`. Let us illustrate this with the following example:

```
1  /*@ ensures \result >= 0;
2      assigns \nothing ; */
3  int positive(void);
4  
5  struct pair { int x, y ; };
6  
7  /*@ predicate lexico(struct pair p1, struct pair p2) =
8        p1.x > p2.x && p1.x >= 0 ||
9        p1.x == p2.x && p1.y > p2.y && p1.y >= 0 ;
10 */
11 
12 //@ requires p.x >= 0 && p.y >= 0;
13 void f(struct pair p) {
14   /*@ loop invariant p.x >= 0 && p.y >= 0;
15       loop assigns p ;
16       loop variant p for lexico;
17   */
18   while (p.x > 0 && p.y > 0) {
19     if (positive()) {
20       p.x--;
21       p.y = positive();
22     }
23     else p.y--;
24   }
25 }
```

We will describe more precisely user-defined predicates in Section 5.2, for now let us consider that the ACSL code on lines 12–15 defines some kind of function that takes to structures in input and returns a boolean value. Here, instead of using an integer for our measure, we use a structure that contains two integers. The `lexico` predicate defines a lexicographical order over the structures of type `pair`. If we look at the generated verification condition, in the "Prove" part, we see that we must prove `P_lexico(p, p_1)`, that uses our user-defined predicate.

Note that unlike the default `loop variant` clause, this clause generates a single verification condition. Indeed, in WP, the default relation $R(x, y) \Leftrightarrow x > y \wedge x \geq 0$ is split into $x > y$ and $x \geq 0$, but for a user-defined relation, it is not necessarily possible to do so, thus WP never tries to do it.

4.2.4. Create a link between postcondition and invariant

Let us consider the following specified program. Our goal is to prove that this function returns the old value of `a` plus 10.

```
/*@
  requires 0 <= a <= 100 ;
  ensures \result == \old(a) + 10;
*/
int add_ten(int a){
  /*@
    loop invariant 0 <= i <= 10;
    loop assigns i, a;
    loop variant 10 - i;
  */
  for (int i = 0; i < 10; ++i)
    ++a;

  return a;
}
```

The weakest precondition calculus does not allow deducing information that is not part of the loop invariant. In a code like:

```
/*@
    ensures \result == \old(a) + 10;
*/
int add_ten(int a){
    ++a;
    ++a;
    ++a;
    //...
    return a;
}
```

By reading the instructions backward from the postcondition, we always keep all knowledge about `a`. On the opposite, as we previously mentioned, outside the loop, WP only considers the information provided by the invariant. Consequently, our `add_ten` function cannot be proved: the invariant does not say anything about `a`. To create a link between the postcondition and the invariant, we have to add this knowledge. See, for example:

```c
/*@
  requires 0 <= a <= 100 ;
  ensures \result == \old(a) + 10;
*/
int add_ten(int a){
  /*@
    loop invariant 0 <= i <= 10;
    loop invariant a == \at(a, Pre) + i; //< ADDED
    loop assigns i, a;
    loop variant 10 - i;
  */
  for (int i = 0; i < 10; ++i)
    ++a;

  return a;
}
```

> This need can appear as a very strong constraint. This is not really the case. There exists strongly automated analysis that can compute loop invariant properties. For example, without a specification, an abstract interpretation would easily compute `0 <= i <= 10` and `old(a) <= a <= \old(a)+10`. However, it is often more difficult to compute the relations that exist between the different variables of a program, for example the equality expressed by the invariant we have added and that is absolutely necessary to prove the postcondition of the function.

4.2.5. Multiple loop invariants

We can specify several `loop invariant` clauses on a loop. There are similarities in the way assertions and loop invariants are handled. Namely, just like when an assertion follows another, it can be proved using the previous one as a hypothesis, like in:

```c
assert A1: P(x);
assert A2: Q(x);
```

where `A2` can be proved with `A1` in hypothesis, the same happens with loop invariants. However, there are some subtleties. Let us consider the following situation:

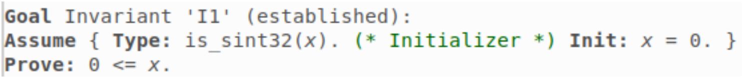
```c
int main(void){
  int x = 0 ;

  /*@ loop invariant I1: 0 <= x ;
      loop invariant I2: x <= 10 ;
      loop assigns x ;
  */
  while(x < 10) x++ ;
}
```

Here, we have 4 verification conditions to prove:

- `I1` is established,

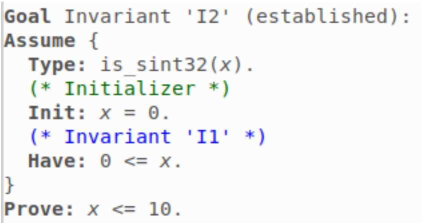
- `I2` is established,
- `I1` is preserved by each loop iteration,
- `I2` is preserved by each loop iteration.

For `I1` establishment, there is not much to say: WP generates a verification condition with all hypotheses that lead to the loop (so just the fact that `x` is 0) and we have to prove that `0 <= x` (we have used the option `-wp-no-let` to avoid immediate proof by Qed):



```
Goal Invariant 'I1' (established):
Assume { Type: is_sint32(x). (* Initializer *) Init: x = 0. }
Prove: 0 <= x.
```

For `I2`, it is quite similar *but* since we have proved that `I1` is established, we also add it as a hypothesis for proving that `I2` is established:



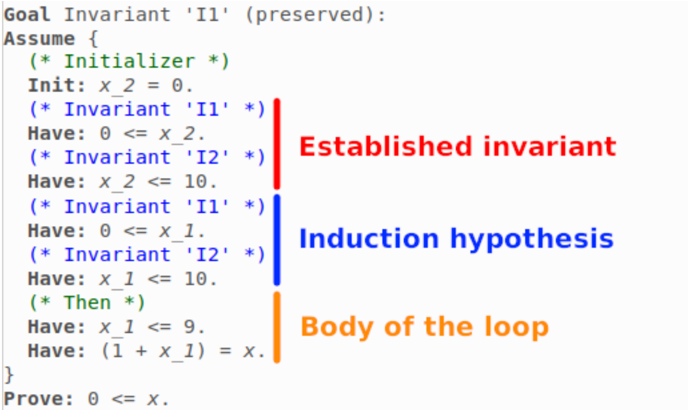
```
Goal Invariant 'I2' (established):
Assume {
  Type: is_sint32(x).
  (* Initializer *)
  Init: x = 0.
  (* Invariant 'I1' *)
  Have: 0 <= x.
}
Prove: x <= 10.
```

If there was a third loop invariant `I3`, the proof of its establishment would receive both the establishment of `I1` and `I2` as a hypothesis and so on.

Then, we have to prove that `I1` and `I2` are preserved by each loop iteration. For proving that `I1` is preserved, we assume that all loop invariants are verified before the loop iteration, and we prove that `I1` is still true at the end of the iteration. However, we can add more assumptions. In particular, we can add that all loop invariants have been established before the loop starts. Thus, we get:

- assuming that `I1` and `I2` were verified when the loop started,
- assuming also that all loop invariants are verified when the iteration starts,
- prove that `I1` is still true at the end of the iteration.

Note that we have slightly simplified the goal to focus on what is important here:

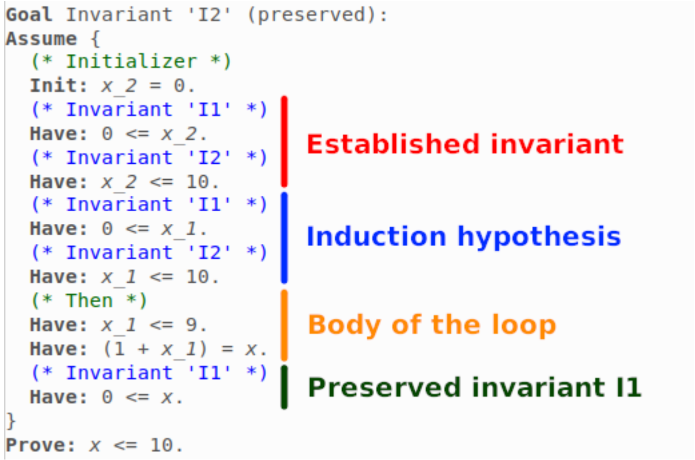


```
Goal Invariant 'I1' (preserved):
Assume {
  (* Initializer *)
  Init: x_2 = 0.
  (* Invariant 'I1' *)
  Have: 0 <= x_2.
  (* Invariant 'I2' *)
  Have: x_2 <= 10.
  (* Invariant 'I1' *)
  Have: 0 <= x_1.
  (* Invariant 'I2' *)
  Have: x_1 <= 10.
  (* Then *)
  Have: x_1 <= 9.
  Have: (1 + x_1) = x.
}
Prove: 0 <= x.
```

Finally, when proving that `I2` is preserved, we can collect even more hypotheses since we also know that `I1` is preserved thus:

- assuming that `I1` and `I2` were verified when the loop started,
- assuming also that all loop invariant are verified when the iteration starts,
- assuming that `I1` is preserved,
- prove that `I2` is still true at the end of the iteration.

Again, and so on when there are more `loop invariant` clauses.



```
Goal Invariant 'I2' (preserved):
Assume {
  (* Initializer *)
  Init: x_2 = 0.
  (* Invariant 'I1' *)
  Have: 0 <= x_2.
  (* Invariant 'I2' *)
  Have: x_2 <= 10.
  (* Invariant 'I1' *)
  Have: 0 <= x_1.
  (* Invariant 'I2' *)
  Have: x_1 <= 10.
  (* Then *)
  Have: x_1 <= 9.
  Have: (1 + x_1) = x.
  (* Invariant 'I1' *)
  Have: 0 <= x.
}
Prove: x <= 10.
```

4.2.6. Different kinds of loop invariants

Just like the `assert` clause that have the two variants `check` and `admit`, one can specify that some loop invariants must be only checked or admitted. The syntax is the following:

```
/*@ check loop invariant <property> ;
    admit loop invariant <property> ;
```

Again, this has an impact on generated verification conditions and related hypotheses.

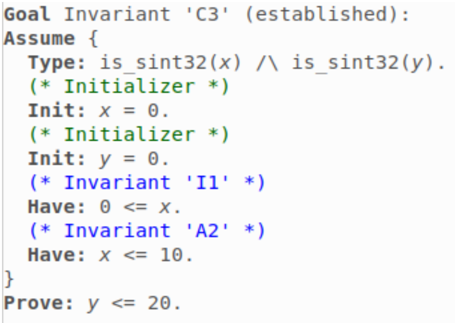
The behavior associated to the `admit loop invariant` clause does not involve any particular subtlety. It is assumed exactly in the same situations as the standard `loop invariant` clause, so that the only difference between the two clauses is the fact that WP never generates a verification condition to check that the `admit loop invariant` clause is indeed true.

The `check loop invariant` clause is slightly more complex to deal with. Since this clause must be checked to be true, WP generate verification conditions for it just like for standard `loop invariant` clause. Then, since it is only checked, one must not assume that it is true for proving other properties. However, for proving that the `check loop invariant` clause is preserved, one still has to assume that it is true at the beginning of the iteration but *only* for this proof of preservation (not for all other loop invariants).

Let us illustrate the behavior with the different verification conditions of `C3` and `I4` in the following code snippet (again, we use `-wp-no-let`, and the verification conditions have been slightly cleaned):

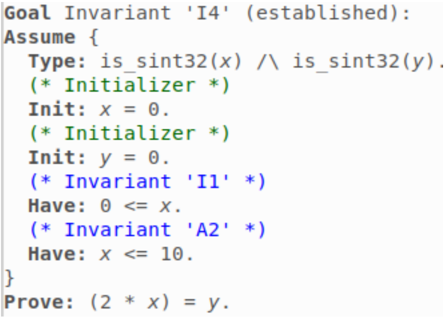
```c
int main(void) {
  int x = 0, y = 0 ;

  /*@       loop invariant I1: 0 <= x ;
      admit loop invariant A2: x <= 10 ;
      check loop invariant C3: y <= 20 ;
            loop invariant I4: y == 2 * x ;
      loop assigns x ;
  */
  while (x < 10) {
    x++ ;
    y += 2 ;
  }
}
```

The verification condition associated to the establishment of `C3` is quite simple, since it appears after the loop invariants `I1` and `A2` that must be both admitted (even if `A2` has not been proved by WP), we have those two properties in the hypotheses:



```
Goal Invariant 'C3' (established):
Assume {
  Type: is_sint32(x) /\ is_sint32(y).
  (* Initializer *)
  Init: x = 0.
  (* Initializer *)
  Init: y = 0.
  (* Invariant 'I1' *)
  Have: 0 <= x.
  (* Invariant 'A2' *)
  Have: x <= 10.
}
Prove: y <= 20.
```

The verification condition associated to the establishment of `I4` is more interesting: it still receives `I1` and `A2` in the hypotheses, but `C3` is not there because it is only checked and not admitted after that:

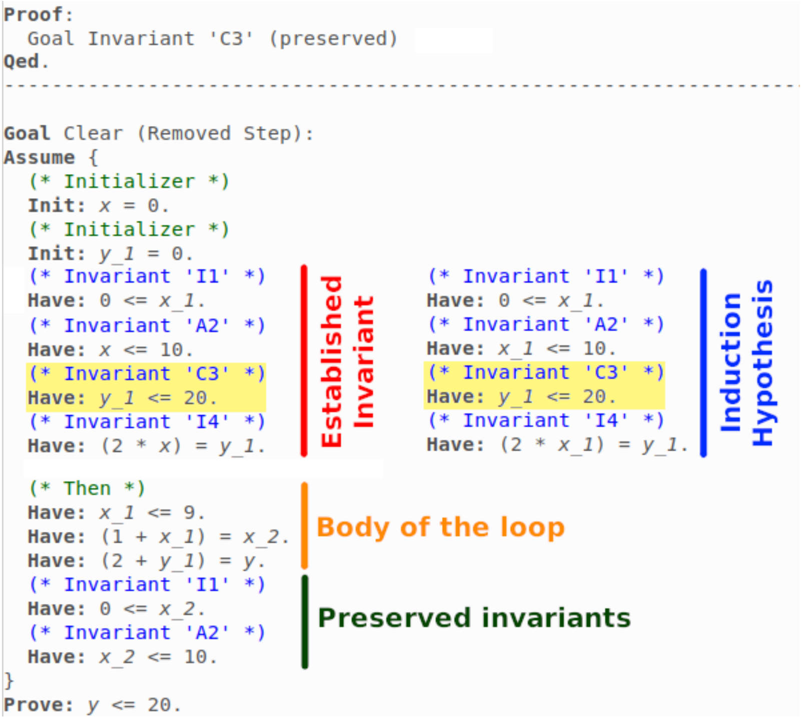


```
Goal Invariant 'I4' (established):
Assume {
  Type: is_sint32(x) /\ is_sint32(y).
  (* Initializer *)
  Init: x = 0.
  (* Initializer *)
  Init: y = 0.
  (* Invariant 'I1' *)
  Have: 0 <= x.
  (* Invariant 'A2' *)
  Have: x <= 10.
}
Prove: (2 * x) = y.
```

Then, we can have a look to the preservation of these invariants. For `C3`, in the hypotheses, we find:

- all the established invariants, including `C3`,

- all the invariants assumed (induction hypothesis), including `C3`,
- all the invariants that precede `C3`, assumed to be preserved.

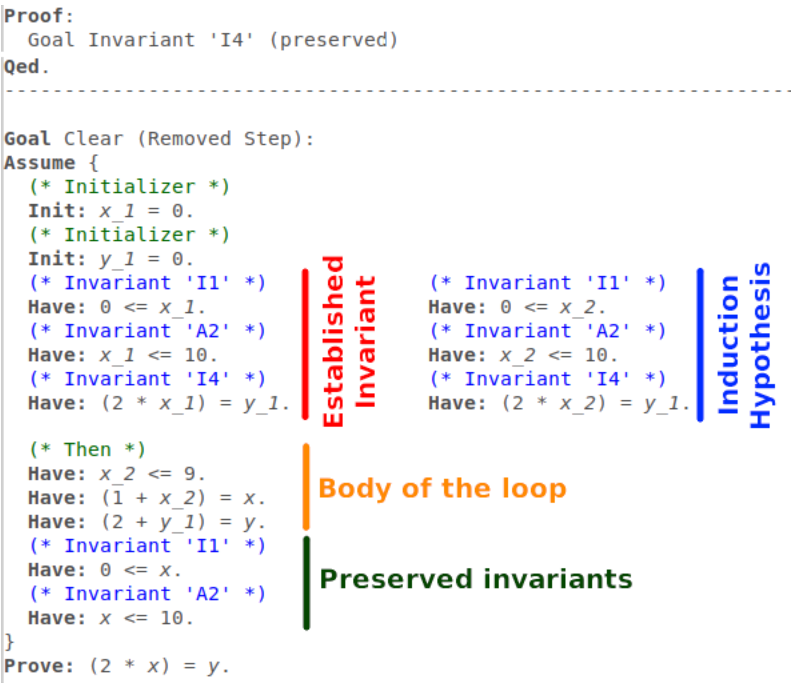


```
Proof:
  Goal Invariant 'C3' (preserved)
Qed.
----------------------------------------------------------------

Goal Clear (Removed Step):
Assume {
  (* Initializer *)
  Init: x = 0.
  (* Initializer *)
  Init: y_1 = 0.
  (* Invariant 'I1' *)
  Have: 0 <= x_1.
  (* Invariant 'A2' *)
  Have: x <= 10.
  (* Invariant 'C3' *)
  Have: y_1 <= 20.
  (* Invariant 'I4' *)
  Have: (2 * x) = y_1.

  (* Then *)
  Have: x_1 <= 9.
  Have: (1 + x_1) = x_2.
  Have: (2 + y_1) = y.
  (* Invariant 'I1' *)
  Have: 0 <= x_2.
  (* Invariant 'A2' *)
  Have: x_2 <= 10.
}
Prove: y <= 20.
```

```
(* Invariant 'I1' *)
Have: 0 <= x_1.
(* Invariant 'A2' *)
Have: x_1 <= 10.
(* Invariant 'C3' *)
Have: y_1 <= 20.
(* Invariant 'I4' *)
Have: (2 * x_1) = y_1.
```

On the opposite, if we look at the preservation of `I4`, in the hypotheses, we find:

- all the established invariants, excluding `C3`,
- all the invariants assumed (induction hypothesis), excluding `C3`,
- all the invariants that precede `I4`, assumed to be preserved, excluding `C3`.



```
Proof:
  Goal Invariant 'I4' (preserved)
Qed.
----------------------------------------------------------------

Goal Clear (Removed Step):
Assume {
  (* Initializer *)
  Init: x_1 = 0.
  (* Initializer *)
  Init: y_1 = 0.
  (* Invariant 'I1' *)
  Have: 0 <= x_1.
  (* Invariant 'A2' *)
  Have: x_1 <= 10.
  (* Invariant 'I4' *)
  Have: (2 * x_1) = y_1.

  (* Then *)
  Have: x_2 <= 9.
  Have: (1 + x_2) = x.
  Have: (2 + y_1) = y.
  (* Invariant 'I1' *)
  Have: 0 <= x.
  (* Invariant 'A2' *)
  Have: x <= 10.
}
Prove: (2 * x) = y.
```

```
(* Invariant 'I1' *)
Have: 0 <= x_2.
(* Invariant 'A2' *)
Have: x_2 <= 10.
(* Invariant 'I4' *)
Have: (2 * x_2) = y_1.
```

Understanding how loop invariant are used by WP depending on the order in which they appear is useful to prove complex invariants. The more we provide useful hypotheses for a proof, the easier the proof will be. In particular, Qed can do a lot of simplification by rewriting hypotheses using deduction rules. Thus, providing first the simplest invariants and then the most complex can help the proof process by first proving easy properties and then using the acquired knowledge to prove the most complex ones.

4.2.7. Early termination of loop

A loop invariant must be true each time the condition of the loop is checked. In fact, that also means that it must be true before an iteration, and after each **complete** iteration. Let us illustrate on an example this important idea.

```c
int main(){
  int i = 0;
  int h = 42;

  /*@
    loop invariant 0 <= i <= 30;
    loop assigns i;
    loop variant 30 - i;
  */
  while(i < 30){
    ++i;

    if(i == 30) break ;
  }
  //@assert i == 30;
  //@assert h == 42;
}
```

In this function, when the loop reaches the index 30, we break the loop before checking the condition again. While the invariant is verified, let us show that we can now further constrain it.

```c
int main(){
  int i = 0;
  int h = 42;

  /*@
    loop invariant 0 <= i <= 29;
    loop assigns i;
    loop variant 30 - i;
  */
  while(i < 30){
    ++i;

    if(i == 30) break ;
  }
  //@assert i == 30;
  //@assert h == 42;
}
```

Here we can see that we have excluded 30 from the range of values of `i` and everything is still verified by WP. This is particularly interesting because it does not apply only to the invariant, none of the loop properties need to be verified in this last iteration. For example, we can write this code that is still verified:

```c
int main(){
  int i = 0;
  int h = 42;

  /*@
    loop invariant 0 <= i <= 29;
    loop assigns i;
    loop variant 30 - i;
  */
  while(i < 30){
    ++i;

    if(i == 30){
      i = 42 ;
      h = 84 ;
      break ;
    }
  }
  //@assert i == 42;
  //@assert h == 84;
}
```

We can see that we can write the variable `h` even if it is not listed in the `loop assigns` clause, and that we can give the value 42 to `i` which does not respect the invariant, and also makes the expression of the variant negative. In fact, everything happens as if we had written:

```c
int main(){
  int i = 0;
  int h = 42;

  /*@
    loop invariant 0 <= i <= 29;
    loop assigns i;
    loop variant 30 - i;
  */
  while(i < 29){
    i++ ;
  }

  if(i < 30){
    ++i;

    if(i == 30){
      i = 42 ;
      h = 84 ;
    }
  }
  //@assert i == 42;
  //@assert h == 84;
}
```

This is an interesting scheme. It basically corresponds to any algorithm that searches, using a loop, a particular condition verified by an element in a given data-structure and stops when it founds it to perform some operations that are thus not really part of the loop. From a verification point of view, it allows us to simplify the contract of the loop: we know that the (potentially complex) operations performed just before we stop do not need to be considered when designing the invariant.

4.2.8. Exercises

4.2.8.1. Loop invariant

Write a suitable invariant for the following loop and prove it using the command.

```
frama-c -wp your-file.c
```

```
int x = 0 ;

while(x > -10){
  -- x ;
}
```

Is the property $-100 \leq x \leq 100$ a correct invariant? Explain why.

4.2.8.2. Loop variant

Write a suitable invariant and variant for the follow loop and prove it using the command:

```
frama-c -wp your-file.c
```

```
int x = -20 ;

while(x < 0){
  x += 4 ;
}
```

If your variant does not precisely state the number of remaining iteration, add a variable that records exactly the number of remaining iterations and use it as a variant. You might need to add an invariant.

4.2.8.3. Loop assigns

Write a suitable loop assigns clause for this loop such that the assertion on line 8 is proved as well as the `assigns` clause. Let us ignore runtime errors in this proof.

```
1  int h = 42 ;
2  int x = 0 ;
3  int e = 0 ;
4  while(e < 10){
  ++ e ;
  x += e * 2 ;
}
//@ assert h == 42 ;
```

Once the proof succeeds, completely remove the assigns clause and find another way to ensure that the assertion is verified using annotations (note that you can add a C label in the code). What do you deduce about the notion of `loop assigns` clause?

4.2.8.4. Early termination

Write a suitable contract for this loop such that the assertions on lines 9 and 10 are proved as well as the contract of the loop.

```
1   int i ;
2   int x = 0 ;
3   for(i = 0 ; i < 20 ; ++i){
4     if(i == 19){
5       x++ ;
6       break ;
7     }
8   }
9   //@ assert x == 1 ;
10  //@ assert i == 19 ;
```

4.3. More examples on loops

4.3.1. Writing loop annotations

Writing loop annotations requires a lot of work during program proof. There is no perfect way of writing them. In particular, finding the right invariant, and the right way to express it, is mostly a matter of experience. Nevertheless, a few simple ideas can help, so that at least we do not miss common mistakes that can make lose a lot of time during the process.

Before anything else, we write the `loop assigns` clause. It is generally easy to write (just look at the assignment instructions and function calls), and if it is incorrect we can prove wrong properties that can make us lose time. When writing them, we should not try to be too precise. WP cannot use efficiently something like `array[x .. y]` when `x` or `y` are themselves modified, so we generally prefer bounds that include them while being constant during the execution of the loop, if we need more precise information during the proof of the invariants, we will provide some additional invariants for this.

Then, we bound the different variables that are assigned, in particular indexes. These invariants are generally easy to guess, express and verify. We put these invariants at the beginning of the list of `loop invariant` clauses, since, as we explained in Sections 4.2.5 and 4.2.6, the order of the invariants is important, and these simple properties can be propagated by WP in the other loop invariants to make verification conditions simplifications.

For most loops, except the ones that rely on a complex condition, once this step is done, it is easy to provide the `loop variant` clause. We can just look at the variables that we have just bounded and what is the value that it will reach at the end of the loop. For example, if in the loop some `i` goes from 0 to `n`, `n - i` is a good candidate. For more complex loops, one can rely on ghost code (that we present in Section 6.3) to make some measure explicit and use it to provide a suitable loop variant.

Then, we add the "main" loop invariants of the loop, i.e. those that relate to the expected postcondition of the loop (that might be the postcondition of the function itself). For this, we can use the postcondition itself as a guide. If we have something like `ensures P(n);` and a loop that iterates `i` from 0 to `n`, `loop invariant P(i);` is probably a good candidate. Notice that in some situations, the `assumes` clause might also be interesting, typically when the result is a simple value that depends on the precondition state, as we will see in some later example. These invariants should be positioned at the end of the list of loop invariants.

We may (optionally) need some "glue" to prove the most interesting "main" loop invariant. For example, if we have voluntarily provided an imprecise `loop assigns` clause, we might need a loop invariant to explain that some parts are not modified, or we may need to explain to the solver that because some properties are true, then we can deduce that the "main" invariant is verified, etc. These loop invariants should be provided before the "main" invariant, but after the simple invariants about variables bounds. Ordering them might not be straightforward, so generally this part is mostly a "trial and error" process.

In the remaining of this section, we will illustrate this approach on some examples. Although in the process, it is advised to run the proof between each step, we will generally not go to this level of detail in the future examples.

4.3.2. Example with read-only arrays

The array is the most common data structure when we are working with loops. It is then a good example base to exercise with loops, and these examples allow to rapidly show interesting invariant and will allow us to introduce some important ACSL constructs.

We can for example illustrate with the search function that allows to find a value in an array. For now, let us focus on the contract of the function:

```
#include <stddef.h>

/*@
  requires \valid_read(array + (0 .. length-1));

  assigns  \nothing;

  behavior notin:
    assumes \forall size_t off ; 0 <= off < length ==> array[off] != element;
    ensures \result == NULL;

  behavior in:
    assumes \exists size_t off ; 0 <= off < length && array[off] == element;
    ensures array <= \result < array+length && *\result == element;

  disjoint behaviors;
  complete behaviors;
```

```c
*/
int* search(int* array, size_t length, int element){
  for(size_t i = 0; i < length; i++)
    if(array[i] == element) return &array[i];
  return NULL;
}
```

There are enough ideas in this example to introduce some important syntax.

First, as we previously presented, the `\valid_read` predicate (as well as `\valid`) allows us to specify not only the validity of a readable address but also to state that a range of contiguous addresses is valid. It is expressed using this syntax:

```c
//@ requires \valid_read(a + (0 .. length-1));
```

This precondition states that all addresses `a+0`, `a+1`, ..., `a+length-1` must be valid readable locations.

We also introduced two notations that are used almost all the time in ACSL, the keywords `\forall` (∀) and `\exists` (∃), the universal logic quantifiers. The first one allows to state that for any element, some property is true, the second one allows to say that there exists some element such that the property is true. If we comment a bit the corresponding lines in our specification, we can read them this way:

```c
/*@
// for all "off" of type "size_t", IF "off" is comprised between 0 and "length"
//                                 THEN the cell "off" in "a" is different from "element"
\forall size_t off ; 0 <= off < length ==> a[off] != element;

// there exists "off" of type "size_t", such that "off" is comprised between 0 and "length"
//                                      AND the cell "off" in "a" equals to "element"
\exists size_t off ; 0 <= off < length && a[off] == element;
*/
```

If we want to summarize the use of these keywords, we would say that on a range of values, a property is true, either about at least one of them or about all of them. A common scheme is to constrain this set using another property (here: `0 <= off < length`) and to prove the actual interesting property on this smaller set. **But using `\exists` and `\forall` is fundamentally different**.

With `\forall type a ; p(a) ==> q(a)`, the constraint `p` is followed by an implication. For any element where a first property `p` is verified, we have to also verify the second property `q`. If we use a conjunction, as we do for "exists" (which we will later explain), that would mean that all elements verify both `p` and `q`. In our previous example, it is clearly not the case as not all integers are comprised between 0 and `length`. Sometimes, it could be what we want to express, but it would then not correspond anymore to the idea of constraining a set for which we want to verify some other property.

With `\exists type a ; p(a) && q(a)`, the constraint `p` is followed by a conjunction. We say there exists an element such that it satisfies the property `p` at the same time it also satisfies `q`. If we use an implication, as we do for `\forall`, such an expression will always be true if `p` is not a tautology! Why? Is there an “a” such that p(a) implies q(a)? Let us take any “a” such that p(a) is false, then the implication is true.

Notice that in this example, the `assumes` clause of the `in` behavior is exactly the negation of the `assumes` clause of the `notin` behavior, this is the reason why the `disjoint` and `complete` clauses are proved, in fact it could have been expressed with:

```
/*@ ...
  behavior in:
    assumes !(\forall size_t off ; 0 <= off < length ==> array[off] != element);
  ...
*/
```

Now, let us talk about the loop annotations. The first step is to add the `loop assigns` clause. Here, it is simple: the loop only modifies the variable `i`. Thus, we need to bound `i`, it goes from `0` to `length`, this is our first loop invariant: `0 <= i <= length`. That also gives us the loop variant: `length - i`. Now, we can provide our “main” loop invariant. Here, it is related to the `assumes` clauses, and not to the `ensures` clauses. In particular, the interesting part of the loop is the fact that unless we meet the element that we search, it is not in the array, we exploit this starting from our `notin` assumes:

```
//@ \forall size_t off ; 0 <= off < length ==> array[off] != element;
```

The variable that reaches `length` at the end of the loop is `i`, thus:

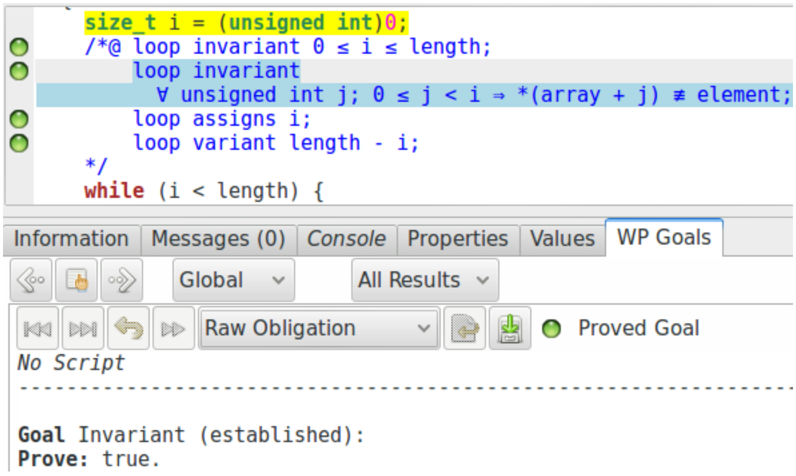
```
//@ loop invariant \forall size_t off ; 0 <= off < i ==> array[off] != element;
```

is certainly a good candidate. Thus, we have the following loop annotations:

```
/*@ loop invariant 0 <= i <= length;
    loop invariant \forall size_t j; 0 <= j < i ==> array[j] != element;
    loop assigns i;
    loop variant length - i; */
for(size_t i = 0; i < length; i++)
  if(array[i] == element) return &array[i];
```

And indeed, our final loop invariant defines the treatment performed by our loop, it indicates to WP what happens in the loop (or more precisely: what we learn) along the execution. Here, this formula indicates that at each iteration of the loop, we know that for each memory location between 0 and the next location to visit (`i` excluded), the memory location contains a value different from the element we are looking for.

The verification condition associated to the preservation of this invariant is a bit complex, and it is not interesting to precisely look at it, on the contrary, the proof that the invariant is established before executing the loop is interesting:



```
size_t i = (unsigned int)0;
/*@ loop invariant 0 ≤ i ≤ length;
    loop invariant
      ∀ unsigned int j; 0 ≤ j < i ⇒ *(array + j) ≢ element;
    loop assigns i;
    loop variant length - i;
*/
while (i < length) {
```

```
No Script
--------------------------------------------------------------------

Goal Invariant (established):
Prove: true.
```

We note that this property, while quite complex, is proved easily by Qed. If we look at the parts of the programs on which the proof relies, we can see that the instruction `i = 0` is highlighted and this is, indeed, the last instruction executed on `i` before we start the loop. And consequently, if we replace the value of `i` by 0 inside the formula of the invariant, we get:

```
//@ loop invariant \forall size_t j; 0 <= j < 0 ==> array[j] != element;
```

"For all j, greater or equal to 0 and strictly lower than 0", this part of the formula is necessarily false, our implication is then necessarily true.

4.3.3. Examples with mutable arrays

Let us present two examples with mutation of arrays. One with a mutation of all memory locations, the other with selective modifications.

4.3.3.1. Reset

Let us have a look at the function that resets an array of integers to 0.

```
#include <stddef.h>

/*@
  requires \valid(array + (0 .. length-1));
  assigns  array[0 .. length-1];
  ensures  \forall size_t j; 0 <= j < length ==> array[j] == 0;
*/
void reset(int* array, size_t length){
  /*@
    loop invariant 0 <= i <= length;
    loop invariant \forall size_t j; 0 <= j < i ==> array[j] == 0;
    loop assigns i, array[0 .. length-1];
    loop variant length-i;
  */
  for(size_t i = 0; i < length; ++i)
    array[i] = 0;
}
```

This time, the loop does modify the content of the array, thus we have to provide this in the `loop assigns` clause. Notice that we can use the notation `n .. m` for this. Furthermore, we do not say that the loop assigns the content from `0` to `i-1` (since `i` is modified, and WP cannot exploit this) but from `0` to `length-1`, this is less precise, but it can be used by WP out the loop. Finally, this time we directly relate the invariant to the postcondition, the goal of the function is to reset the array from 0 to `length`, at a given iteration, the loop has done it from 0 to `i`.

4.3.3.2. Search and replace

The last example we will detail to illustrate the proof of functions with loops is the algorithm “search and replace”. This algorithm selectively modifies values in a range of memory locations. It is generally harder to guide the tool in such a case, because on one hand we must keep track of what is modified and what is not, and on the other hand, the induction relies on this fact.

As an example, the first specification and loop annotations we can write for this function could be this one (which follows basically the same process we had for the previous example):

```c
#include <stddef.h>

/*@
  requires \valid(array + (0 .. length-1));
  assigns array[0 .. length-1];

  ensures \forall size_t i; 0 <= i < length && \old(array[i]) == old
             ==> array[i] == new;
  ensures \forall size_t i; 0 <= i < length && \old(array[i]) != old
             ==> array[i] == \old(array[i]);
*/
void search_and_replace(int* array, size_t length, int old, int new){
  /*@
    loop invariant 0 <= i <= length;
    loop invariant \forall size_t j; 0 <= j < i && \at(array[j], Pre) == old
                     ==> array[j] == new;
    loop invariant \forall size_t j; 0 <= j < i && \at(array[j], Pre) != old
                     ==> array[j] == \at(array[j], Pre);
    loop assigns i, array[0 .. length-1];
    loop variant length-i;
  */
  for(size_t i = 0; i < length; ++i){
    if(array[i] == old) array[i] = new;
  }
}
```

We use the logic function `\at(v, Label)` that gives us the value of the variable `v` at the program point `Label`. If we look at the usage of this function here, we see that in the invariant we try to establish a relation between the old values of the array and the potentially new values:

```
/*@
  loop invariant \forall size_t j; 0 <= j < i && \at(array[j], Pre) == old
                   ==> array[j] == new;
  loop invariant \forall size_t j; 0 <= j < i && \at(array[j], Pre) != old
                   ==> array[j] == \at(array[j], Pre);
*/
```

For every memory location that contained the value to be replaced, it now must contain the new value. All other values must remain unchanged. While we previously relied on the `assigns` clauses for this kind of properties, here, we do not have this possibility because while the ACSL language provides some facilities that we could use to write a very precise `assigns` clause, WP would not perfectly exploit it. Thus, we have to use an invariant and keep an approximation of the assigned memory location.

If we try to prove this invariant with WP, it fails, because the `assigns` specification is not precise enough. Thus, in such a case, we provide an additional loop invariant to detail in the assigned range what are the locations that have not been modified yet by the loop at some iteration:

```
for(size_t i = 0; i < length; ++i){
    //@assert array[i] == \at(array[i], Pre); // proof failure
    if(array[i] == old) array[i] = new;
}
```

We can add this information as an invariant:

```
  /*@
    loop invariant 0 <= i <= length;
    loop invariant \forall size_t j; i <= j < length
                     ==> array[j] == \at(array[j], Pre);
    loop invariant \forall size_t j; 0 <= j < i && \at(array[j], Pre) == old
                     ==> array[j] == new;
    loop invariant \forall size_t j; 0 <= j < i && \at(array[j], Pre) != old
                     ==> array[j] == \at(array[j], Pre);
    loop assigns i, array[0 .. length-1];
    loop variant length-i;
  */
  for(size_t i = 0; i < length; ++i){
    if(array[i] == old) array[i] = new;
  }
```

And this time the proof succeeds.

4.3.4. Exercises

For all these exercises, use the following command line to start verification:

```
frama-c-gui -wp -wp-rte -warn-unsigned-overflow -warn-unsigned-downcast your-file.c
```

4.3.4.1. Non-mutating: For all, Exists, ...

Currently, function pointers are not supported by WP. Let us consider that we have a function:

```c
/*@
  assigns \nothing ;
  ensures \result <==> \true ; // some property about value instead
*/
int pred(int value){
  // your code
}
```

Write your own code and corresponding postcondition and then write the following functions together with their contract and prove their correctness. Note that, as `pred` is a C function, it cannot be used in the specification of the functions you have to write, thus the property that you have considered for this predicate should be inlined in the contract of the different functions.

- `forall_pred` returns true if and only if `pred` is true for all elements ;
- `exists_pred` returns true if and only if `pred` is true for at least one element ;
- `none_pred` returns true if and only if `pred` is false for all elements ;
- `some_not_pred` returns true if and only if `pred` is false for at least one element.

The two last functions should be written by just calling the two first ones.

4.3.4.2. Non-mutating: Equality of ranges

Write, specify and prove the function `equal` that returns true if and only if two ranges of values equal. Write, using the equal function, the code of `different` that returns true if two ranges are different, your postcondition should contain an existential quantifier.

```c
int equal(const int* a_1, const int* a_2, size_t n){

}

int different(const int* a_1, const int* a_2, size_t n){

}
```

4.3.4.3. Binary Search

The following function searches for the position of a particular value in an array, assuming that this array is sorted. First, let us consider that the length of the array is an int and use int values to deal with indexes. One could note that there are two behavior: either the value exists in the array (and the result is the index of this value) or not (and the result is -1).

```c
#include <stddef.h>

/*@
  requires Sorted:
    \forall integer i, j ; 0 <= i <= j < len ==> arr[i] <= arr[j] ;
*/
int bsearch(int* arr, int len, int value){
  if(len == 0) return -1 ;

  int low = 0 ;
  int up = len-1 ;

  while(low <= up){
    int mid = low + (up - low)/2 ;
    if      (arr[mid] > value) up = mid-1 ;
    else if(arr[mid] < value) low = mid+1 ;
    else return mid ;
  }
  return -1 ;
}
```

This function is a bit tricky to prove, so let us provide some hints. First, this time the length is provided as an int, so we need to further restrict this value to make it coherent. Second, one of the invariant properties should state the bounds of `low` and `up`, but note that for both of them one of the bounds is not needed. Finally, the second invariant property should state that if some index of the array stores the value, this index must be correctly bounded.

Harder: Modify this function in order to receive `len` as a `size_t`. You will have to slightly modify the algorithm and the proof. As a hint, we advise making `up` an excluded bound for the search.

4.3.4.4. Mutating: Addition of vectors

Write, specify and prove the function that adds two vectors into a third one. Put arbitrary constraints to solve the integer overflow. Consider that the resulting vector is entirely separated from the input vectors, however the same vector should be usable for both input vectors.

```c
void add_vectors(int* v_res, const int* v1, const int* v2, size_t len){

}
```

4.3.4.5. Mutating: Reverse

Write, specify and prove the function that reverses a vector in place. Take care of the unmodified part of the vector at some iteration of the loop. Use the previously proved `swap` function.

```c
void swap(int* a, int* b);

void reverse(int* array, size_t len){

}
```

4.3.4.6. Mutating: Copy

Write, specify and prove the function `copy` that copies a range of values into another array, starting from the first cell of the array. First consider (and specify) that the two ranges are entirely separated.

```c
void copy(int const* src, int* dst, size_t len){

}
```

Harder: The true copy and copy backward functions

In fact, a strong separation is not necessary. The actual precondition must guarantee that if the two arrays overlap, the beginning of the destination range must not be in the source range:

```c
//@ requires \separated(&src[0 .. len-1], dst) ;
```

In essence, by copying the element in that order, we can shift elements from the end of a particular range to the beginning. However, that means that we have to be more precise in our contract: we do not guarantee anymore an equality with the values of the source array but with the *old* values of the source array. And we have also to be more precise in our invariant, first by also specifying the relation in regard to the previous state of the memory, and second by adding an invariant that shows that the source array is not modified from the `i`th we are visiting to the end.

Finally, it is also possible to write a function that copies the elements from the end to the beginning, in this case, again, arrays can overlap, but the condition is not exactly the same. Write, specify and prove the function `copy_backward` that copies elements in the reverse order.

4.4. Function calls

4.4.1. Calling a function

4.4.1.1. Formal - Weakest precondition calculus

When a function is called, the contract of this function is used to determine the precondition of the call. But one has to consider two important facts to express the weakest precondition calculus.

First, the postcondition of the called function f is not necessarily directly the precondition that was computed for the instructions that follow the call to f. For example, if we have a program: `x = f() ; c` and $wp(\texttt{c}, Q) = 0 \leq x \leq 10$, whereas the postcondition of the function `f` is $1 \leq x \leq 9$, we have to express some weakening between the actual precondition of `c` and the computed one. For this, we refer to the section 4.1.4, the idea is simply to verify that the postcondition of the function implies the computed precondition.

Second, in C, a function can have side effects. Thus, the values of the variables referenced in input are not necessarily the same as they were after the call to the function, and the contract may express some property about those values before and after the call. So, if we have labels in the postcondition, we must correctly replace them.

In order to define the weakest-precondition calculus of function calls, let us introduce some notation to make things clearer. For this, consider this example:

```c
/*@ requires \valid(x) && *x >= 0 ;
    assigns *x ;
    ensures *x == \old(*x)+1 ; */
void inc(int* x);

void foo(int* a){
  L1:
  inc(a) ;
  L2:
}
```

The weakest precondition of the function call asks us to consider the contract of the function that is called (here, in `foo`, when we call the `inc` function). Of course, before the call to the function we have to verify its precondition, so it is part of the weakest precondition. But, we also have to consider the postcondition of the function, else that would mean that we do not consider its effect.

Thus, it is important to notice that in the precondition, the considered memory state is the one where we compute the weakest precondition, whereas for the postcondition it is not the case, the considered memory state is the one that follows the call, while we need to explicitly mention the old state to speak about the values before the call. For example, considering the contract of `inc` when we call it in `foo`, `*x` in the precondition is `*a` at `L1`, while `*x` in the postcondition is `*a` at `L2`. Consequently, the pre and the postcondition must be considered slightly differently when it comes to mutable memory location. Note that for the value of the parameter `x` itself, there is no such consideration: this value cannot be modified by the call.

Now, let us define the weakest precondition of a function call. For this, we denote:

- $\vec{v}$ a vector of values $v_1, ..., v_n$ and v_i the i^{th} value,
- $\vec{t}$ the arguments provided to the function when we call it,
- $\vec{x}$ the parameters in the function definition,
- $\vec{a}$ the assigned values (seen from the outside, once instantiated),
- $here(x)$ a value in postcondition
- $old(x)$ a value in precondition

We name `f:Pre` the precondition of the function, and `f:Post` the postcondition:

$$\begin{aligned} wp(f(\vec{t}), Q) := \quad & \texttt{f:Pre}[x_i \leftarrow t_i] \\ \wedge \quad & \forall \vec{v}, \quad (\texttt{f:Post}[x_i \leftarrow t_i, here(a_j) \leftarrow v_j, old(a_j) \leftarrow a_j] \Rightarrow Q[here(a_j) \leftarrow v_j]) \end{aligned}$$

We can detail a bit the reasoning for each part of this formula.

First, note that in both pre and postcondition, each named parameter x_i is replaced with the corresponding argument ($[x_i \leftarrow t_i]$), as we said before we do not have to consider memory states there because those values cannot be changed by the function call. For example in the contract of `inc`, each `x` would be replaced by the argument `a`.

Then, in the part of the formula that corresponds to the postcondition, we can see that we introduce a $\forall \vec{v}$. The goal is here to model the fact that the function can write any value in each memory location that is assigned. So, for each of the assigned location a_j (that is for our call to `inc`, `*(&a)`), we generate a value v_j that is its value after the call. But, if we want to check that the postcondition gives us the right result, we cannot accept *any value* for each assigned location, we just want the ones *that allows to satisfy the postcondition*.

So these values are used to transform the postcondition of the function and verify that it implies the postcondition in input of the weakest precondition. This is done by replacing, for each assigned location a_j, its value *here* with the value v_j that it is supposed to get after the call ($here(a_j) \leftarrow v_j$). Finally, we have to replace each *old* value by its value before the call, and for each $old(a_j)$, it is simply a_j ($old(a_j) \leftarrow a_j$).

4.4.1.2. Formal - Example

Let us illustrate this on an example by applying the weakest precondition calculus to this short code, assuming the contract we previously proposed for the `swap` function.

```c
int a = 4 ;
int b = 2 ;

swap(&a, &b) ;

//@ assert a == 2 && b == 4 ;
```

We now compute the weakest precondition:

$$
\begin{aligned}
&wp(a = 4; b = 2; swap(\&a, \&b), a = 2 \land b = 4) = \\
&\quad wp(a = 4, wp(b = 2; swap(\&a, \&b), a = 2 \land b = 4)) = \\
&\quad wp(a = 4, wp(b = 2, wp(swap(\&a, \&b), a = 2 \land b = 4)))
\end{aligned}
$$

Let us first consider separately:

$$wp(swap(\&a, \&b), a = 2 \land b = 4)$$

From this `assigns` clause, we know that the assigned values are $*(\&a) = a$ and $*(\&b) = b$. (Let us shorten *here* with H and *old* with O).

$$\begin{aligned}&\texttt{swap:Pre}[x \leftarrow \&a,\ y \leftarrow \&b]\\ &\quad \wedge \forall v_a, v_b, (\texttt{swap:Post}\ \ [x \leftarrow \&a,\ y \leftarrow \&b,\\ &\qquad\qquad H(*(\&a)) \leftarrow v_a,\ H(*(\&b)) \leftarrow v_b,\\ &\qquad\qquad O(*(\&a)) \leftarrow *(\&a),\ O(*(\&b)) \leftarrow *(\&b)])\\ &\qquad \Rightarrow (H(a) = 2 \wedge H(b) = 4)[H(a)) \leftarrow v_a, H(b)) \leftarrow v_b])\end{aligned}$$

For the precondition, we get :

$$valid(\&a) \wedge valid(\&b)$$

For the postcondition part, let us first write the expression from which we start before any term replacement (and without the syntax for the replacement for the sake of conciseness):

$$H(*x) = O(*y) \wedge H(*y) = O(*x) \Rightarrow H(a) = 2 \wedge H(b) = 4$$

First we replace the pointers $(x \leftarrow \&a,\ y \leftarrow \&b)$:

$$H(*(\&a)) = O(*(\&b)) \wedge H(*(\&b)) = O(*(\&a)) \Rightarrow H(a) = 2 \wedge H(b) = 4$$

Then, the *here* values, with the quantified v_is $(H(a)) \leftarrow v_a, H(b)) \leftarrow v_b)$:

$$v_a = O(*(\&b)) \wedge v_b = O(*(\&a)) \Rightarrow v_a = 2 \wedge v_b = 4$$

And the *old* values, with the value before call $(O(*(\&a)) \leftarrow *(\&a),\ O(*(\&b)) \leftarrow *(\&b))$:

$$v_a = *(\&b) \wedge v_b = *(\&a) \Rightarrow v_a = 2 \wedge v_b = 4$$

We can now simplify this formula to:

$$v_a = b \wedge v_b = a \Rightarrow v_a = 2 \wedge v_b = 4$$

So, $wp(swap(\&a, \&b), a = 2 \wedge b = 4)$ is:

$$P : valid(\&a) \wedge valid(\&b) \wedge \forall v_a, v_b, \quad v_a = b \wedge v_b = a \Rightarrow v_a = 2 \wedge v_b = 4$$

Let us immediately simplify the formula by noticing that validity properties are trivially true here (since the variable are allocated on the stack just before):

$$P : \forall v_a, v_b, \quad v_a = b \wedge v_b = a \Rightarrow v_a = 2 \wedge v_b = 4$$

Let us now compute $wp(a = 4, wp(b = 2, P)))$, by first replacing b with 2 by the assignment rule:

$$\forall v_a, v_b, \quad v_a = 2 \wedge v_b = a \Rightarrow v_a = 2 \wedge v_b = 4$$

and then replacing a with 4 by the same rule:

$$\forall v_a, v_b, \quad v_a = 2 \wedge v_b = 4 \Rightarrow v_a = 2 \wedge v_b = 4$$

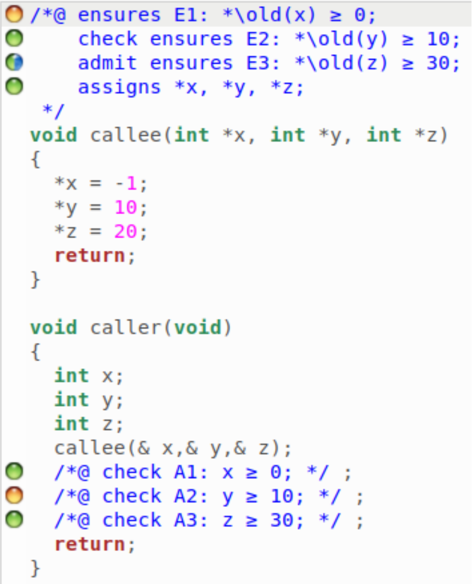
This last property is trivially true, thus the program is verified.

4.4.1.3. Function contracts: check and admit

Just like assertions and loop invariants, the `requires` and `ensures` clauses can be only checked or admitted.

Let us start with the `ensures` clause. When a function `f` has an `ensures` clause, it must be proved correct when verifying `f`. On the opposite, when we call `f`, we admit that it is true. Now, if `f` has a `check ensures` clause, we have to verify it when verifying `f`, but we *ignore* it when `f` is called. On the opposite, if `f` has an `admit ensures` clause, it is not verified when verifying the `f`, still we admit it when it is called. In the following example:

```
/*@ assigns *x, *y, *z ;
          ensures E1: *x >= 0 ;
    check ensures E2: *y >= 10 ;
    admit ensures E3: *z >= 30 ;
*/
void callee(int *x, int *y, int *z){
  *x = -1 ;
  *y = 10 ;
  *z = 20 ;
}

void caller(void){
  int x, y, z ;
  callee(&x, &y, &z);

  //@ check A1: x >= 0;
  //@ check A2: y >= 10 ;
  //@ check A3: z >= 30 ;
}
```

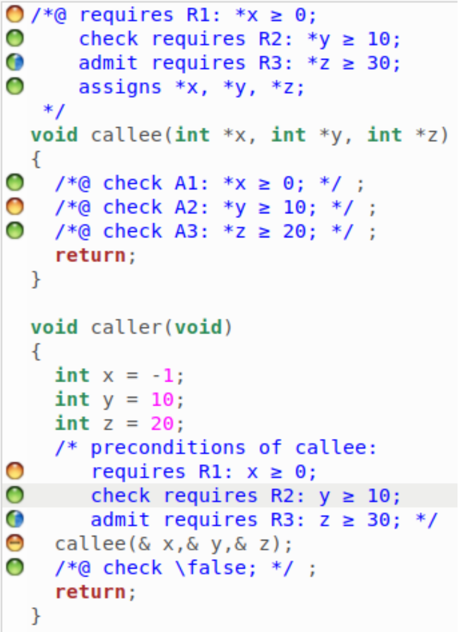


```
/*@ ensures E1: *\old(x) ≥ 0;
      check ensures E2: *\old(y) ≥ 10;
      admit ensures E3: *\old(z) ≥ 30;
      assigns *x, *y, *z;
  */
void callee(int *x, int *y, int *z)
{
  *x = -1;
  *y = 10;
  *z = 20;
  return;
}

void caller(void)
{
  int x;
  int y;
  int z;
  callee(& x,& y,& z);
  /*@ check A1: x ≥ 0; */ ;
  /*@ check A2: y ≥ 10; */ ;
  /*@ check A3: z ≥ 30; */ ;
  return;
}
```

We cannot prove the clause `E1` because `*x` is assigned to -1, but we can prove the check `A1` because when `f` is called, we admit that `E1` is true. We can prove the clause `E2` because `*y` is assigned to 10 that is indeed greater than or equals to 10. However, we cannot prove the check `A2` because this postcondition is ignored at call point. Finally, while `*z` is assigned to 20, we do not try to verify that `E3` is true (and it is not), yet WP does not warn about it: it is admitted, thus at call point, we can prove that `z` is greater than 30 even if it is not.

Now, we can present the behavior for the `requires` clause. When a function `f` has a `requires` clause, it is assumed to be true when verifying `f` and must be verified when `f` is called. When it is a `check requires`, the clause is verified at call point, yet we do not assume it when verifying `f`. Finally, when a function `f` has an `admit requires` clause, it is assumed to be true when verifying `f` without any verification of it at call point. In the following example:

```
/*@       requires R1: *x >= 0 ;
    check requires R2: *y >= 10 ;
    admit requires R3: *z >= 30 ;
    assigns *x, *y, *z ;
*/
void callee(int *x, int *y, int *z){
  //@ check A1: *x >= 0 ;
  //@ check A2: *y >= 10 ;
  //@ check A3: *z >= 20 ;
}

void caller(void){
  int x = -1, y = 10, z = 20 ;
  callee(&x, &y, &z);

  //@ check \false ;
}
```



```
/*@ requires R1: *x ≥ 0;
    check requires R2: *y ≥ 10;
    admit requires R3: *z ≥ 30;
    assigns *x, *y, *z;
 */
void callee(int *x, int *y, int *z)
{
  /*@ check A1: *x ≥ 0; */ ;
  /*@ check A2: *y ≥ 10; */ ;
  /*@ check A3: *z ≥ 20; */ ;
  return;
}

void caller(void)
{
  int x = -1;
  int y = 10;
  int z = 20;
  /* preconditions of callee:
     requires R1: x ≥ 0;
     check requires R2: y ≥ 10;
     admit requires R3: z ≥ 30; */
  callee(& x,& y,& z);
  /*@ check \false; */ ;
  return;
}
```

We cannot prove that `R1` is true at call point, yet we admit it in `callee` thus can prove `A1`. On the opposite, while `R2` is verified at call point, we cannot prove `A2` in `callee` since we ignore `R2` when verifying `f`. Finally, we do not try to verify `R3` at call point (and WP does not warn about it even if it is false), but we can prove that `A3` is verified because we assume `R3`.

There is a second important aspect with the behavior of the `requires` clause. The careful reader might have noticed in the previous example, that the `check \false` is verified. The reason is that the `requires` clauses are also assumed locally in the caller for normal and `admit` `requires` clauses. Thus, since in this example those `requires` clauses are false, we locally introduce "false" in the proof context.

4.4.1.4. What should we keep in mind about function contracts?

Functions are absolutely necessary to modular programming, and the weakest precondition calculus is fully compatible with this idea, allowing to reason about each function locally and compose proofs just as we compose function calls.

So as a reminder, we should just keep in mind the following general scheme:

```
/*@       requires bar_R
    check requires bar_CR ;
    admit requires bar_AR ;

    assigns ... ;

          ensures bar_E
    check ensures bar_CE ;
    admit ensures bar_AE ;
*/
void bar(...) ;

/*@       requires foo_R
    check requires foo_CR ;
    admit requires foo_AR ;

    assigns ... ;

          ensures foo_E
    check ensures foo_CE ;
    admit ensures foo_AE ;
*/
type foo(parameters...){
  // Here we suppose that foo_R and foo_AR hold


  // Here we must prove that bar_R and bar_CR hold
  bar(some parameters ...) ;
  // Here we assume that bar_E and bar_AE hold


  // Here we must prove that foo_E and foo_CE hold
  return ... ;
}
```

Note that for the last statement, with weakest precondition calculus, the idea is more to show that our precondition is strong enough to ensure that the code leads to our postcondition. However, first, this vision is simpler to understand, and second the WP plugin does not actually perform a strict weakest precondition calculus but a highly optimized one that does not follow exactly the same rules.

4.4.1.5. The particular case of `exits`

In C, one can call the `exit` function to stop the execution with a particular error code. In such a case, the code that follows the call will not be executed, and the `return` statement will not be reached. Thus, when the function exits, the postcondition is always verified since it is unreachable (just like when a function never calls a function that might exit, the `exits` clause might be anything):

```c
#include <stdlib.h>

/*@ exits \true ;
    ensures \false ;
*/
void this_function_exits(void){
  exit(1);
}
```

Here, we have just indicated that `\true` must be true when the function exits, but in fact one can indicate any property of interest. For example, one can indicate the value of the exit status:

```c
#include <stdlib.h>

/*@ exits \exit_status == 1 ;
    ensures \false ;
*/
void this_function_exits(void){
  exit(1);
}
```

Of course when one calls a function that might exit, the risk of exit action is propagated:

```c
#include <stdlib.h>

/*@ exits \exit_status == 1 ;
    ensures \false ;
*/
void this_function_exits(void){
  exit(1);
}

void does_this_function_exit(void){
  this_function_exits();
}
```

It is thus recommended indicating precisely what are the behaviors that can lead to a call to `exit` and to forbid calls to `exit` in the other cases:

```c
#include <stdlib.h>

/*@ behavior exits:
      assumes x == 0;
      exits \exit_status == 1 ;
      ensures \false ;
    behavior is_ok:
      assumes x != 0 ;
      exits \false ;
*/
void this_function_exits(int x){
  if(!x) exit(1);
}

void does_this_function_exit(void){
  this_function_exits(1);
}
```

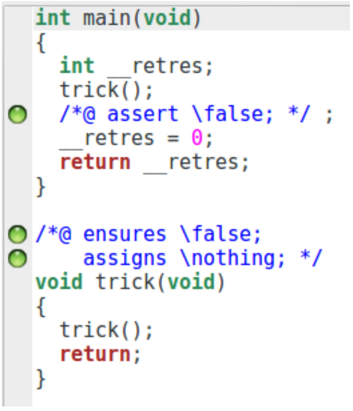
[Formal] WP calculus When we have to prove the postcondition of the function, we start the calculus from the postcondition. Thus, we start our computation from the single `return` statement created by Frama-C (although it is exactly like we would start from all the different `return` statements in the original program), and we reason backward along the program. When we meet a function call, we use the contract with the previously explained rule for function call, and the postcondition we consider at this point for reasoning is the one stated in the `ensures` clauses.

For proving `exits` clauses, it is different, but not that much. Instead of starting the calculus from `return` instructions, we start a new calculus from each function call (as it might be one of the exit points) and at this call, we use the `exit` clause as a postcondition, then when we continue the backward reasoning, we use the `ensures` as usual since if we have reached the function call where we started from, that means that previous calls did not exit.

4.4.2. Recursive functions

Just as it is easy to prove anything about the postcondition of a function if it contains an infinite loop, it is easy to prove anything about the postcondition of a function that does an infinite recursion:

```
/*@
  assigns \nothing ;
  ensures \false ;
*/
void trick(){
  trick() ;
}

int main(){
  trick();
  //@ assert \false ;
}
```



```
int main(void)
{
  int __retres;
  trick();
  /*@ assert \false; */ ;
  __retres = 0;
  return __retres;
}

/*@ ensures \false;
    assigns \nothing; */
void trick(void)
{
  trick();
  return;
}
```

We can see that the function and the assertion are proved. And indeed the proof is correct: we consider partial correctness, and we face a function that does not terminate: anything that follows a call to this function would be true. But again, we have a watchdog: the `terminates` clause is not proved.

Thus, the question is: what could we do in such a case? Again, we could use a measure (like in Section 4.2.3.1), to bound the depth of the recursive calls. In ACSL, it is expressed thanks to the `decreases` clause:

```
/*@ requires n >= 0 ;
    decreases n ;
*/
void ends(int n){
  if(n > 0) ends(n-1);
}
```

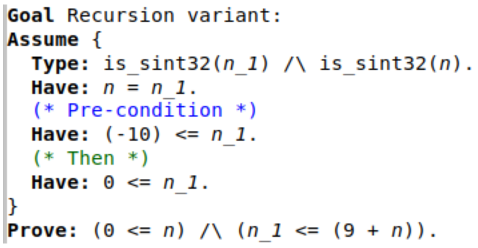
Just like the `loop variant` clause, the `decreases` clause expresses a notion of measure. That is, a positive integer expression (or an expression equipped with a relation) that strictly decreases. While, a `loop variant` clause represents an upper bound on the number of iteration, `decreases` represents an upper bound on the depth of the recursion (and not the number of function calls):

```
/*@ requires n >= 0 ;
    decreases n ;
*/
void ends_2(int n){
  if(n > 0) ends_2(n-1); // Ok: 0 <= n && n-1 < n
  if(n > 0) ends_2(n-1); // Ok: 0 <= n && n-1 < n, no need to be less than call on l.5
}
```

Note that, like for the `loop variant` clause where we verify the properties of the expression only when a new iteration might happen, the properties of the `decreases` expression are verified when the function is called again. It means that when we reach the maximum recursion depth, the expression might be negative:

```
/*@ requires n >= -10 ;
    decreases n ; */
void go_negative(int n){
  if(n >= 0) go_negative(n-10); // if n is 0, this call is rec(-10), it is fine
}
```

We can see the generated verification condition for the previous example by disabling goal simplifications (option `-wp-no-let`, we removed redundant information in the screenshot):



```
Goal Recursion variant:
Assume {
  Type: is_sint32(n_1) /\ is_sint32(n).
  Have: n = n_1.
  (* Pre-condition *)
  Have: (-10) <= n_1.
  (* Then *)
  Have: 0 <= n_1.
}
Prove: (0 <= n) /\ (n_1 <= (9 + n)).
```

Here, the condition `n - 10 < n` is formulated `n_1 (= n) <= 9 + n` because of the normalization of the formula.

When proving the correctness of the `decreases` clause of a particular function (and for a call to this same function), the expression is evaluated for two entities : the function under proof and the call instruction. The state where the expression is evaluated for the function under proof is `Pre`, the state where the expression is evaluated for the call instruction is `Here`. The value of the expression at call point must be less than the value of the expression in `Pre` state.

```
/*@ requires \valid(p) && *p >= 0 ;
    decreases *p ;
*/
void ends_ptr(int *p){
  if(*p > 0){
    (*p)-- ;
    ends_ptr(p); // Ok: 0 <= \at(*p, Pre) && *p < \at(*p, Pre)
  }
}
```

Of course, recursive functions can be mutually recursive thus the `decreases` clause can be used to bound recursive function calls in this situation. But, we want to do so *only* for functions that are indeed in the set of functions that are involved in the recursion. For this, WP computes the strongly connected components from the set of functions, this is called a *cluster*.

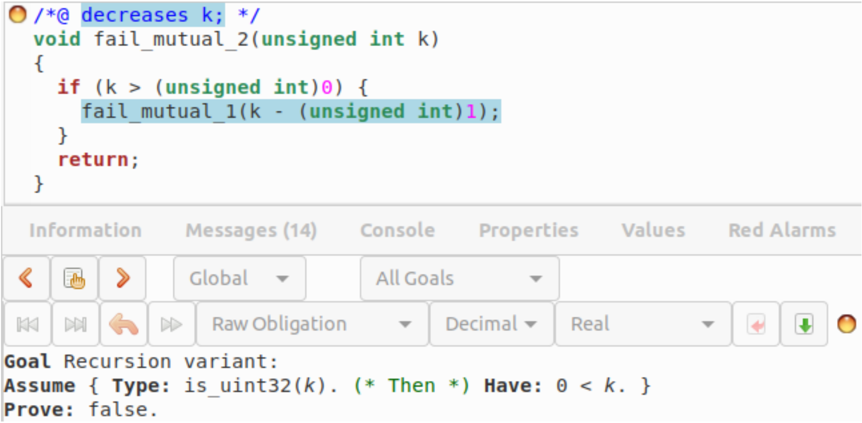
Thus, let us be more precise about how we proceed to verify of the correction of a `decreases` clause. When a function is (mutually) recursive, its specification must be equipped with such a clause to prove that it terminates. Verifying that the `decreases` clause of a function `f` gives a measure of the depth of recursion is done by checking for each call to a function *that belongs to the same cluster as* `f`, that the expression is indeed positive and decreasing. And thus, no verification condition is generated when calling a recursive function that does not belong to the same cluster:

```
//@ decreases v ;
void single(unsigned v){
  if(v > 0) single(v-1); // OK: 0 <= v && v-1 < v
}

//@ decreases k-1 ;
void mutual_2(unsigned k);

//@ decreases n ;
void mutual_1(unsigned n){
  if(n > 1) mutual_2(n-1); // OK: 0 <= n && (n-1)-1 < n
}

void mutual_2(unsigned k){
  if(k > 1) mutual_1(k-2); // OK: 0 <= k-1 && (k-2) < k-1
  single(k+1) ; // no verification needed, single is not in the cluster
}
```

Note that the computation of the cluster is done syntactically, for now we do not cover function pointers in this tutorial, but one can refer to the option `-wp-dynamic` in the WP manual.

Finally, if a function of a cluster does not have a `decreases` clause, a `\false` verification condition is generated and a warning is emitted by WP.

```
void fail_mutual_1(unsigned k);

//@ decreases k ;
void fail_mutual_2(unsigned k){
  if(k > 0) fail_mutual_1(k-1);
}
/* Warning:
  [wp] file.c:5: Warning:
    Missing decreases clause on recursive function fail_mutual_1, call must be unreachable
*/

void fail_mutual_1(unsigned k){
  if(k > 0) fail_mutual_2(k-1);
}
```



```
/*@ decreases k; */
void fail_mutual_2(unsigned int k)
{
  if (k > (unsigned int)0) {
    fail_mutual_1(k - (unsigned int)1);
  }
  return;
}
```

```
Goal Recursion variant:
Assume { Type: is_uint32(k). (* Then *) Have: 0 < k. }
Prove: false.
```

Just like `loop variant` clauses, one can provide another relation (see Section 4.2.3.2) for a `decreases` clause. The syntax is:

```
//@ decreases <term> for <Relation> ;
```

4.4.3. Specifying and proving function termination

A desirable property for a function is often that it should terminate. Indeed, in most programs, all functions must terminate, and when it is not the case and some functions can loop forever, it is highly common that there is a single function that is allowed to loop forever (so almost all functions in this program must terminate).

4.4.3.1. Syntax and description

ACSL provides the `terminates` clause to specify that a function must terminate when some property is verified in precondition. The syntax is:

```
//@ terminates condition ;
void function(void){
  // ...
}
```

It states that when `condition` is verified in precondition, then the function must terminate. For example, function `abs` must always terminate:

```
/*@ requires x > INT_MIN ;
    terminates \true ;
*/
int abs(int x){
  return (x < 0) ? -x : x ;
}
```

while function `main_loop` may not (note that with default options, the variant is not verified, we will explain why later):

```
int debug_steps = -1 ;

/*@ requires debug_steps >= -1 ;
    terminates debug_steps >= 0 ;
 */
void main_loop(void){
  /*@ loop invariant \at(debug_steps, Pre) == -1
                  || 0 <= debug_steps <= \at(debug_steps, Pre) ;
      loop assigns debug_steps ;
      loop variant debug_steps ; */
  while(1){
    if(debug_steps == 0) return ;
    else if(debug_steps > 0) debug_steps -- ;

    // actual code
  }
}
```

Let us emphasize on the fact that the function *may not* terminate, it is not *forced to loop forever*. For example, in the following function, the `terminates` clause is verified since *whenever the termination condition is verified* (never), the function terminates (always):

```
//@ terminates \false ;
void may_not_terminate(void){

}
```

If one really wants to verify that some function never terminates, it can be done by specifying that the function never returns and never exits. That is: the post-conditions states associated to these kinds of termination are unreachable:

```
unsigned counter ;

/*@ terminates \false;
    exits \false;
    ensures \false;
*/
void does_not_terminate(void){
  //@ loop assigns counter ;
  while(1){
    counter++;
  }
}
```

In ACSL, it is specified that when a function does not have a `terminates` clause, the default is `terminates \true`. This behavior is automatically enabled by WP at the kernel level. Thus, when WP starts the verification of a function, it asks the kernel to generate these annotations. This behavior can be disabled **for user defined functions** using the kernel option:

- `-generated-spec-custom terminates:skip`

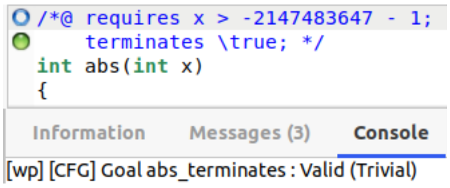
4.4.3.2. Verification

Verifying that a function terminates asks to verify that all reachable statements of the function terminate. Assignments trivially terminate, thus we do not have something particular to do for them. A conditional statement terminates if all statements in the different (reachable) branches terminate, thus we just have to verify that these statements terminate (or are unreachable). The remaining statements are loops and function calls. Thus, we have to verify that:

- all loops have a (verified) `loop variant` clause,
- all called functions terminate with the given parameters,
- all recursive functions have a (verified) `decreases` clause,
- (or that there are no loops nor calls in the function).

However, we only have to do so when the termination condition of the function is verified. So let us now explain what are the generated verification conditions and when does WP generate them.

When a function has a `terminates` clause, WP visits all statements and collects the loops that do not have a `loop variant` clause and the function calls. If there are none of them, the `terminates` clause is trivially verified.



```
/*@ requires x > -2147483647 - 1;
    terminates \true; */
int abs(int x)
{
```

```
[wp] [CFG] Goal abs_terminates : Valid (Trivial)
```

When there are such statements, their termination must be verified *when* the function must terminate (say, when `T`). Thus, the verification conditions are of the form `\at(T, Pre)` ⇒ `<statement termination>`. One should note that the premise of the implication is evaluated at the `Pre` state. Thus, in this code:

```
void call(int r);

//@ terminates r > 0 ;
void simple(int r){
  r -- ;
  call(r);
}
```

Even if `r` has been decremented, the verification of the termination of `call(r)` is done with `\at(r, Pre) > 0` as a premise. We will see in the next section the verification condition generated for a call.

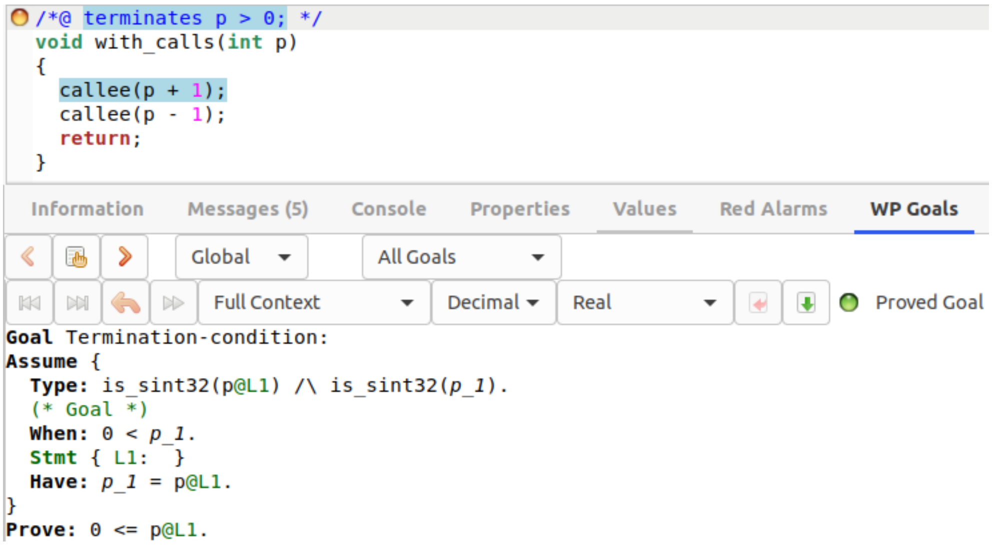
Note also that it means that when we reach a program point when `T` is false, the verification condition is always verified:

```
//@ terminates r > 0 ;
void cond_may_not_terminate(int r){
  if(r <= 0){ // if we enter here, r > 0 is false at Pre
    // here statements are not force to terminate: FALSE ==> P is always true
    while(1){}
  }
}
```

4.4.3.2.1. Function call Verifying that a function call terminates is done by verifying that when it is called, its termination condition is true. For example, from the following program:

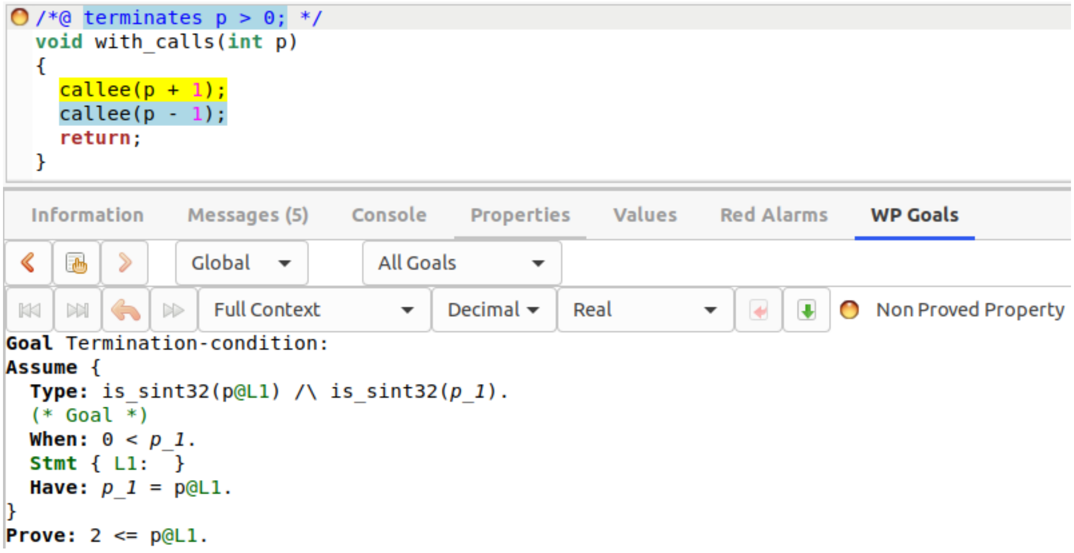
```
//@ terminates value > 0 ;
void callee(int value);

//@ terminates p > 0 ;
void with_calls(int p){
  // goal: p > 0 ==> p+1 > 0 ; (provable)
  callee(p+1);
  // goal: p > 0 ==> p-1 > 0 ; (not provable, the function may not terminate)
  callee(p-1);
}
```

We get the following verification conditions (by using `-wp-no-let` to disable simplifications):



```
/*@ terminates p > 0; */
void with_calls(int p)
{
  callee(p + 1);
  callee(p - 1);
  return;
}
```

```
Goal Termination-condition:
Assume {
  Type: is_sint32(p@L1) /\ is_sint32(p_1).
  (* Goal *)
  When: 0 < p_1.
  Stmt { L1:  }
  Have: p_1 = p@L1.
}
Prove: 0 <= p@L1.
```



```
/*@ terminates p > 0; */
void with_calls(int p)
{
  callee(p + 1);
  callee(p - 1);
  return;
}
```

```
Goal Termination-condition:
Assume {
  Type: is_sint32(p@L1) /\ is_sint32(p_1).
  (* Goal *)
  When: 0 < p_1.
  Stmt { L1:  }
  Have: p_1 = p@L1.
}
Prove: 2 <= p@L1.
```

Where the first call termination is indeed verified while the second is not.

4.4.3.2.2. Loop variant When a function contains a loop that does not have a `loop variant` clause, its termination cannot be verified, thus WP asks us to verify that when the termination condition is verified, this loop is unreachable.

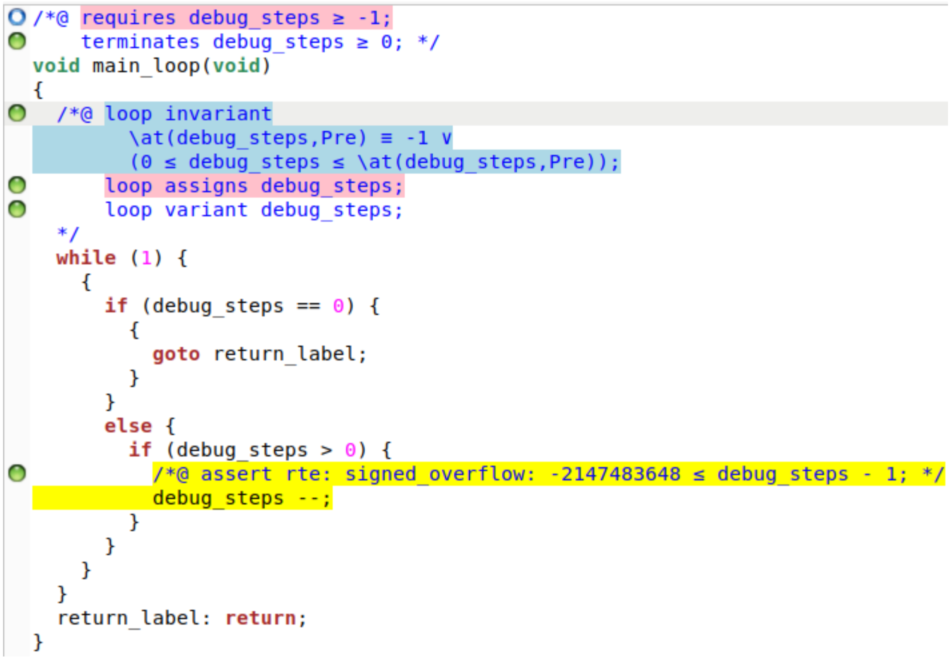
```
//@ terminates value > 0 ;
void with_loop(int value){
  if(value <= 0){
    //@ loop assigns \nothing ;
    while(1){
      // code
    }
  }
}
```

In the previous code, the loop does not have a `loop variant` clause, thus we have to verify `value > 0 ==> \false` at the loop location, which is OK: this code is unreachable when `value > 0`.

Finally, when a loop has a `loop variant` clause, it must be verified *only when* the function must terminate. So in the example we presented at the beginning of this section:

```
int debug_steps = -1 ;

/*@ requires debug_steps >= -1 ;
    terminates debug_steps >= 0 ;
 */
void main_loop(void){
  /*@ loop invariant \at(debug_steps, Pre) == -1
                || 0 <= debug_steps <= \at(debug_steps, Pre) ;
      loop assigns debug_steps ;
      loop variant debug_steps ; */
  while(1){
    if(debug_steps == 0) return ;
    else if(debug_steps > 0) debug_steps -- ;

    // actual code
  }
}
```

We have to verify that the loop variant is a positive decreasing value only when `debug_steps` is not -1. However, this is not the default behavior of WP (that always verify loop variants by default), this can be enabled using the option `-wp-variant-with-terminates` and in this case our function is entirely verified:



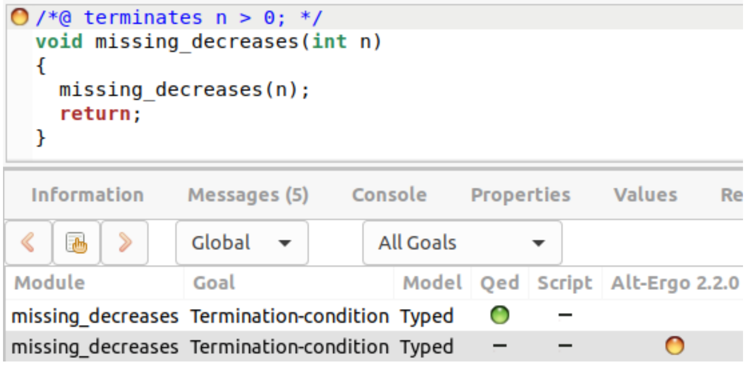
```
/*@ requires debug_steps ≥ -1;
    terminates debug_steps ≥ 0; */
void main_loop(void)
{
  /*@ loop invariant
        \at(debug_steps,Pre) ≡ -1 ∨
        (0 ≤ debug_steps ≤ \at(debug_steps,Pre));
      loop assigns debug_steps;
      loop variant debug_steps;
  */
  while (1) {
    {
      if (debug_steps == 0) {
        {
          goto return_label;
        }
      }
      else {
        if (debug_steps > 0) {
          /*@ assert rte: signed_overflow: -2147483648 ≤ debug_steps - 1; */
          debug_steps --;
        }
      }
    }
  }
  return_label: return;
}
```

4.4.3.2.3. Recursion

A recursive function should have a `decreases` clause when its specification states that it terminates. If such a clause is missing, a verification condition `\false` is generated.

```
//@ terminates n > 0 ;
void missing_decreases(int n){
  missing_decreases(n);
}
```

Note that this code generates two verification conditions:



```
/*@ terminates n > 0; */
void missing_decreases(int n)
{
  missing_decreases(n);
  return;
}
```

Module	Goal	Model	Qed	Script	Alt-Ergo 2.2.0
missing_decreases	Termination-condition	Typed	●	–	
missing_decreases	Termination-condition	Typed	–	–	●

The first one corresponds to the rule associated to a function call explained earlier. The second corresponds to the fact that the `decreases` clause is missing and is not verified.

Again in ACSL, the verification of the `decreases` clause verification is required only when the termination condition is verified in `Pre` state. The behavior of WP on this aspect is similar to the `loop variant`. By default, the verification is always tried, the ACSL specified behavior is enabled via the option `-wp-variant-with-terminates`.

```
/*@ requires n >= -1 ;
    terminates n >= 0 ;
    decreases n ; // is verified only with option -wp-variant-with-terminates
*/
void recursive(int n){
  if(n == -1) recursive(n) ;
  else if(n > 0) recursive(n - 1);
}
```

4.4.4. Exercises

4.4.4.1. Explain proof failures

In the following program, some verification conditions are not verified:

```
#include <limits.h>

/*@ requires x > INT_MIN ;
    assigns \nothing ;
    ensures x >  0 ==> \result == x ;
    ensures x <  0 ==> \result == -x ; */
int abs(int x){
  return x >= 0 ? x : -x ;
}

/*@ requires INT_MIN <= b - a <= INT_MAX;
    ensures a < b  ==> a + \result == b ;
    ensures b <= a ==> a - \result == b ;
*/
int distance(int a, int b){
  return abs(b - a) ;
}

/*@
  requires a < b  ==> b - a <= INT_MAX ;
  requires b <= a ==> a - b <= INT_MAX ;

  assigns \nothing ;

  ensures a < b  ==> a + \result == b ;
  ensures b <= a ==> a - \result == b ;
*/
int old_distance(int a, int b){
  if(a < b) return b - a ;
  else return a - b ;
}

extern int old ;
extern int new ;

/*@ requires INT_MIN <= b - a <= INT_MAX; */
void test(int a, int b){
  old = old_distance(a, b);
  new = distance(a, b);
  //@ assert old == new ;
}
```

Explain why they are not verified and propose a way to fix the specification so that everything is verified.

4.4.4.2. Explain termination proof results

In the following program:

```c
#include <limits.h>
#include <stddef.h>

/*@ requires x > INT_MIN ;
    terminates \true ;
    assigns \nothing ;
    ensures x >= 0 ==> \result == x ;
    ensures x <  0 ==> \result == -x ;
*/
int abs(int x){
  return x >= 0 ? x : -x ;
}

/*@ requires INT_MIN < b - a <= INT_MAX ;
    terminates \true ;
    assigns \nothing ;
    ensures a < b  ==> a + \result == b ;
    ensures b <= a ==> a - \result == b ;
*/
int distance(int a, int b){
  return abs(b - a) ;
}

/*@ requires \valid_read(a + (0 .. len-1)) && \valid_read(b + (0 .. len-1));
    requires \valid(result + (0 .. len-1));
    requires \separated(a + (0..len-1), b + (0..len-1), result + (0..len-1));
    requires \forall integer i ; 0 <= i < len ==> 0 <= b[i] - a[i] <= INT_MAX ;

    terminates \true ;

    assigns result[0 .. len-1] ;

    ensures \forall integer i ; 0 <= i < len ==> a[i] + result[i] == b[i] ;
*/
void distances(int const* a, int const* b, int* result, size_t len){
  /*@ loop invariant 0 <= i <= len ;
      loop invariant \forall integer k ; 0 <= k < i ==> a[k] + result[k] == b[k] ;
      loop assigns i, result[0 .. len-1];
  */
  for(size_t i = 0 ; i < len ; ++i){
    result[i] = distance(a[i], b[i]);
  }
}

struct client ;

/*@ requires \valid(c) ;
    terminates \true ;
    assigns *c ;
*/
void initialize(struct client *c);

/*@ requires \valid(c) ;
    terminates \false ;
    assigns *c ;
    ensures \result \in { 0 , 1 } ;
*/
int connect(struct client *c);

/*@ requires \valid(c);
    terminates \true ;
    assigns *c ;
    ensures \result \in { 0 , 1 } ;
*/
```

```c
int prepare(struct client *c){
  initialize(c);
  return connect(c);
}

/*@ terminates x > 0 ; */
void terminates_f1(int x){
  if(x <= 0) for(;;);
}

/*@ requires x > 0 ;
    terminates \true ;
*/
void terminates_f2(int x){
  if(x <= 0) for(;;);
}
```

Explain why termination clauses are verified or not. Modify the specification so that all of them are verified.

4.4.4.3. Search

Specify and prove the following recursive search function:

```c
#include <stddef.h>
#include <limits.h>

unsigned search(int* array, unsigned length, int element){
  if(length == 0) return UINT_MAX ;
  if(array[length-1] == element) return length - 1 ;
  else return search(array, length-1, element);
}
```

The specification should include the termination condition.

4.4.4.4. Sum integers

The following program computes the sum of the integers between `fst` and `lst`:

```c
int sum(int fst, int lst, int acc){
  if (fst == lst) return acc ;
  else return sum(fst+1, lst, fst+acc) ;
}
```

Prove that this function terminates. Proving the correct behavior of the function or the absence of runtime errors is not asked.

4.4.4.5. Power

The following program computes the power of `x` to `n`:

```
/*@
  requires \true ; // to complete
  terminates \true ; // to complete
  decreases 0 ; // to complete
*/
int rec_power(int x, int n){
  if(n == 0) return 1 ;
  else if(n % 2 == 0) return rec_power(x * x, n / 2) ;
  else return x * rec_power(x, n - 1) ;
}
```

Prove that this function terminates. Proving the correct behavior of the function or the absence of runtime errors is not asked.

In this part, we have seen how assignment and control structure are translated to a logic view of our program. We have spent quite a lot of time on loops because they represent the main difficulty we have to face when we want to specify and prove a program by deductive verification. The loop annotations allow us to express as precisely as possible their behavior.

In the next part of this tutorial, we will see more precisely the logic constructs provided by ACSL. They are important because they give us a way to write more abstract specification, that are easier to understand and to prove.

5. ACSL - Properties

From the beginning of this tutorial, we have used different predicates and logic functions provided by ACSL: `\valid`, `\valid_read`, `\separated`, `\old` and `\at`. There are others built-in predicates, but we will not present them all, the reader can refer to the documentation (ACSL implementation) (note that everything is not necessarily supported by WP).

ACSL allows us to do something more than "just" specify our code using existing predicates and functions. We can define our own predicates, functions, relations, etc. Doing this, we can have more abstract specifications. It also allows us to factor specifications (for example defining what is a valid array), which have two pleasant consequences: our specifications are more readable and more understandable, and we can reuse existing proofs to ease the proof of new programs.

5.1. Some logical types

ACSL provides different logic types that allow us to write properties in a more abstract, mathematical world. Among the types that can be useful, some are dedicated to numbers, and allow expressing properties or functions without having to think about constraints due to the size of the representation of primitive C types in memory. These types are `integer` and `real`, which respectively represent mathematical integers and reals (that are modeled to be as close to the reality we can, but this notion cannot be perfectly handled).

From now, we will often use integers instead of classical C `int`s. The reason is simply that a lot of properties and definitions are true regardless the size of the machine integer we have as input.

On the other hand, we will not talk about the differences that exist between `real` and `float/double`. It would require speaking about precise numerical calculus, and about proofs of programs that rely on such calculus which could deserve an entire dedicated tutorial.

5.2. Predicates

A predicate is a property about different objects that can be true or false. To sum up, we are writing predicates from the beginning of this tutorial in precondition, postcondition, assertion and loop invariant. ACSL allows us to name these predicates, as we could do for a boolean function in C, for example. An important difference, however, is that predicates (as well as logic functions that we will see later) must be pure. For example, they cannot produce side effects by modifying a pointed value.

These predicates can receive some parameters. Moreover, they can also receive some C labels that will allow us to establish relations between different program points.

5.2.1. Syntax

Predicates are introduced using ACSL annotations. The syntax is the following:

```
/*@
  predicate named_predicate { Lbl0, ..., LblN }(type0 arg0, ..., typeN argN) =
    //a logic relations between all these things
*/
```

For example, we can define the predicate that checks whether an integer in memory is changed between two particular program points:

```
/*@
  predicate unchanged{L0, L1}(int* i) =
    \at(*i, L0) == \at(*i, L1);
*/
```

Keep in mind that passing a value to a predicate is done, as it is done in C, by value. We cannot write this predicate by directly passing `i` in parameter:

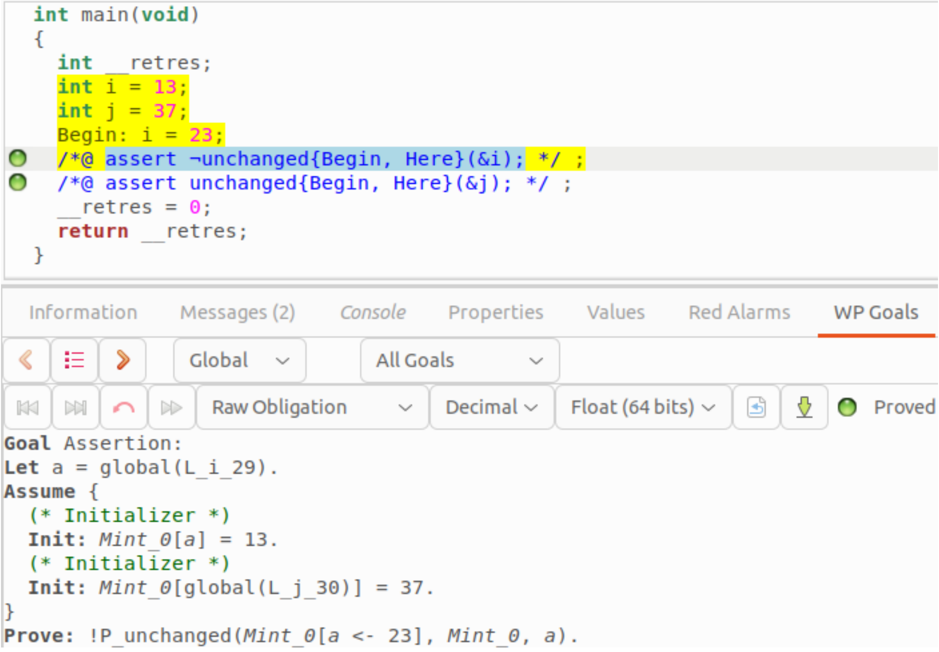
```
/*@
  predicate unchanged{L0, L1}(int i) =
    \at(i, L0) == \at(i, L1);
*/
```

Since `i` is just a copy of the received variable.

We can verify this code using our predicate:

```
int main(void){
  int i = 13;
  int j = 37;

 Begin:
  i = 23;

  //@assert ! unchanged{Begin, Here}(&i);
  //@assert   unchanged{Begin, Here}(&j);
}
```

We can also have a look at the verification conditions generated by WP and notice that, even it is slightly (syntactically) modified, the predicate is not unrolled by WP. The provers will determine themselves whether they need to use the definition of the predicate to establish the proof.



```
int main(void)
{
  int __retres;
  int i = 13;
  int j = 37;
  Begin: i = 23;
  /*@ assert ¬unchanged{Begin, Here}(&i); */ ;
  /*@ assert unchanged{Begin, Here}(&j); */ ;
  __retres = 0;
  return __retres;
}
```

```
Goal Assertion:
Let a = global(L_i_29).
Assume {
  (* Initializer *)
  Init: Mint_0[a] = 13.
  (* Initializer *)
  Init: Mint_0[global(L_j_30)] = 37.
}
Prove: !P_unchanged(Mint_0[a <- 23], Mint_0, a).
```

As we said earlier, one important use of predicates (and logic functions) is to make our specifications more readable and to factor it. An example can be to write a predicate that expresses the validity of an array in reading or writing. It allows us to avoid writing the complete expression every time we need it and to make it readable quickly:

```
/*@
  predicate valid_range_rw(int* t, integer n) =
    n >= 0 && \valid(t + (0 .. n-1));

  predicate valid_range_r(int* t, integer n) =
    n >= 0 && \valid_read(t + (0 .. n-1));
*/

/*@
  requires 0 < length;
  requires valid_range_r(array, length);
  //...
*/
int* search(int* array, size_t length, int element);
```

In this specification, we do not give an explicit label to predicates for their definition, nor for their use. For the definition, Frama-C automatically creates an implicit label. At predicate use, the given label is implicitly `Here`. The fact we do not explicitly define the label in the definition of a predicate does not forbid to explicitly give a label when we use it.

Of course, predicates can be defined in header files in order to produce a utility library for specification for example.

5.2.1.1. Predicate overloading

It is possible to overload predicates as long as the types of the parameters are different or the number of parameters changes. For example, we can redefine the `valid_range_r` as a predicate that takes in parameters both the beginning and the end of the range to consider.

Then, we can write an overloaded version that uses the previous one for the particular case of ranges that starts at 0:

```
/*@
  predicate valid_range_r(int* t, integer beg, integer end) =
    end >= beg && \valid_read(t + (beg .. end-1)) ;

  predicate valid_range_r(int* t, integer n) =
    valid_range_r(t, 0, n) ;
*/

/*@
  requires 0 < length;
  requires valid_range_r(array, length);
  //...
*/
int* search(int* array, size_t length, int element);
```

5.2.2. Abstraction

Another important use of predicates is to define the logical state of our data structures when programs start to be more complex. Our data structures must usually respect an invariant (again) that each manipulation function must maintain in order to ensure that the data structure will always remain coherent and usable through future calls.

It allows us to ease the reading of specifications. For example, we can define the specification required to ensure the safety of a fixed size stack. It could be done as illustrated here (note that we do not provide the definition of the predicates as it is not the purpose of our example, the careful reader could consider this as an exercise):

```
#include <stddef.h>
#define MAX_SIZE 42

struct stack_int{
  size_t top;
  int    data[MAX_SIZE];
};

/*@
  predicate valid_stack_int(struct stack_int* s) = \true ; // to define
  predicate empty_stack_int(struct stack_int* s) = \true ; // to define
  predicate full_stack_int(struct stack_int* s) =  \true ; // to define
*/

/*@
  requires \valid(s);
  assigns *s;
  ensures valid_stack_int(s) && empty_stack_int(s);
*/
void initialize(struct stack_int* s);

/*@
  requires valid_stack_int(s) && !full_stack_int(s);
  assigns  *s;
  ensures valid_stack_int(s);
*/
void push(struct stack_int* s, int value);

/*@
```

```
  requires valid_stack_int(s) && !empty_stack_int(s);
  assigns \nothing;
*/
int  top(struct stack_int* s);

/*@
  requires valid_stack_int(s) && !empty_stack_int(s);
  assigns *s;
  ensures valid_stack_int(s);
*/
void pop(struct stack_int* s);

/*@
  requires valid_stack_int(s);
  assigns \nothing;
  ensures \result == 1 <==> empty_stack_int(s);
*/
int  is_empty(struct stack_int* s);


/*@
  requires valid_stack_int(s);
  assigns \nothing;
  ensures \result == 1 <==> full_stack_int(s);
*/
int  is_full(struct stack_int* s);
```

Here, the specification does not express functional properties. For example, we do not specify that when we perform the push of a value, and then we ask for the top of the stack, we get the same value. But we already have enough details to ensure that, even if we cannot prove that we always get the right result (behaviors such as “if I push v, top returns v”), we can still guarantee that we do not produce runtime errors (if we provide correct predicates for the stack, and prove that the implementation of our functions ensures that no runtime errors can occur).

5.2.3. Exercises

5.2.3.1. Days of the month

Taking back the solution of the exercise 3.4.1.1 about days of the month, write a predicate to express that a year is a leap year and adapt the contracts using it.

5.2.3.2. Alphanumeric character

Taking back the solution of the exercise 3.4.1.2 about alphanumeric characters, write predicates to express that a character is an upper letter, lower letter, and a digit. Adapt the contracts of the different functions using them.

5.2.3.3. Max of 3 values

The following function returns the max of 3 input values:

```c
int max_of(int* a, int* b, int* c){
  if(*a >= *b && *a >= *c) return *a ;
  if(*b >= *a && *b >= *c) return *b ;
  return *c ;
}
```

Write a predicate to express that a value is one of three pointed values at a given memory state:

```c
/*@
  predicate one_of{L}(int value, int *a, int *b, int *c) =
    // ...
*/
```

Use the set notation. Write a contract to the function and prove that it is verified.

5.2.3.4. Binary Search

Taking back the solution of the exercise 4.3.4.3 about the binary search function with unsigned types, write a predicate that expresses that an array is sorted on a range of values starting at `begin` and ending at `end` (excluded). Overload this predicate in order to make `begin` optional with a default value of 0. Define a predicate that checks if an element is in a range of values of an array starting at index `begin` and ending at `end` (excluded), again overload this predicate to make the first bound optional.

Use those two predicates to simplify the contract of the function. Note that both behaviors `assumes` clause should be modified.

5.2.3.5. Search and replace

Taking back the example 4.3.3.2, about the search and replace function, write predicates that express that in some range of an array starting at index `begin` and ending at `end` (excluded), values

- remain unchanged between two labels,
- are replaced with some new value when it equals to some old value, then left unchanged

Overload both predicates to make the first bound optional. Use the obtained predicates to simplify the contract and loop invariant of the function.

5.3. Logic functions

Logic functions are meant to describe functions that can only be used in specifications. It allows us, first, to factor those specifications and, second, to define some operations on `integer` or `real` with the guarantee that they cannot overflow since they involve mathematical types.

Like predicates, they can receive different labels and values in parameter.

5.3.1. Syntax

To define a logic function, the syntax is the following:

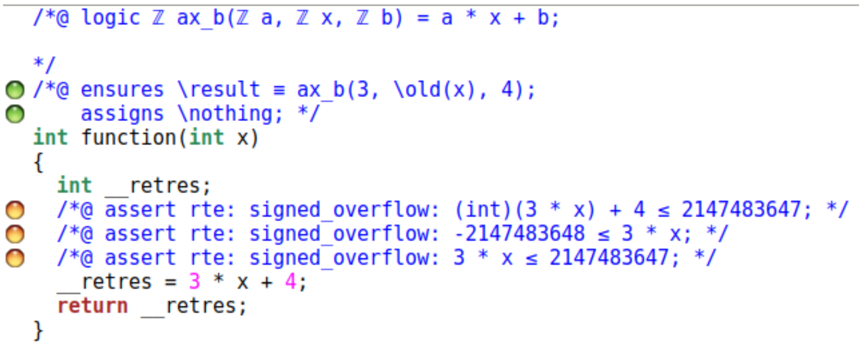
```
/*@
  logic return_type my_function{ Label0, ..., LabelN }( type0 arg0, ..., typeN argN ) =
    formula using the arguments ;
*/
```

We can for example define a mathematical linear function using a logic function:

```
/*@
  logic integer ax_b(integer a, integer x, integer b) =
    a * x + b;
*/
```

And it can be used to prove the source code of the following function:

```
/*@
  assigns \nothing ;
  ensures \result == ax_b(3, x, 4);
*/
int function(int x){
  return 3*x + 4;
}
```



```
/*@ logic ℤ ax_b(ℤ a, ℤ x, ℤ b) = a * x + b;

*/
/*@ ensures \result ≡ ax_b(3, \old(x), 4);
    assigns \nothing; */
int function(int x)
{
  int __retres;
  /*@ assert rte: signed_overflow: (int)(3 * x) + 4 ≤ 2147483647; */
  /*@ assert rte: signed_overflow: -2147483648 ≤ 3 * x; */
  /*@ assert rte: signed_overflow: 3 * x ≤ 2147483647; */
  __retres = 3 * x + 4;
  return __retres;
}
```

This code is indeed proved but some runtime-errors seems to be possible. We can add some constraint in precondition so that the result can be stored into a C integer:

```
/*@
  requires INT_MIN <= ax_b(3, x, 4) <= INT_MAX;
  assigns \nothing ;
  ensures \result == ax_b(3, x, 4);
*/
int function(int x){
  return 3*x + 4;
}
```

Some runtime errors are still possible. Indeed, while the bound provided for `x` by our logic function is defined for the complete computation, it does not say anything about the value obtained in the intermediate computation. For example here, the fact that `3 * x + 4` is not lower than `INT_MIN` does not guarantee that this is the case for `3 * x`. We can imagine two ways to solve this problem, this choice should be guided by the project in which this function would be used.

Either we further restrict the input values:

```
/*@
  requires INT_MIN <= 3*x ;
  requires INT_MIN <= ax_b(3, x, 4) <= INT_MAX;
  assigns \nothing ;
  ensures \result == ax_b(3, x, 4);
*/
int restricted(int x){
  return 3*x + 4;
}
```

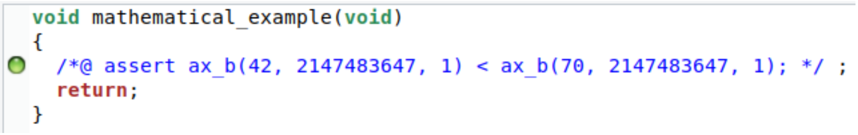
Or we can modify the source code so that the risk of overflow does not appear anymore:

```
/*@
  requires INT_MIN <= ax_b(3, x, 4) <= INT_MAX;
  assigns \nothing;
  ensures \result == ax_b(3, x, 4);
*/
int function_modified(int x){
  if(x > 0)
    return 3 * x + 4;
  else
    return 3 * (x + 2) - 2;
}
```

Note that, as in specifications, computations are done using mathematical integers. We then do not need to care about some overflow risk when using the `ax_b` function:

```
void mathematical_example(void){
  //@ assert ax_b(42, INT_MAX, 1) < ax_b(70, INT_MAX, 1) ;
}
```

which is correctly discharged by WP, that does not generate any alarm related to overflows:



```
void mathematical_example(void)
{
  /*@ assert ax_b(42, 2147483647, 1) < ax_b(70, 2147483647, 1); */ ;
  return;
}
```

5.3.2. Recursive functions and limits of logic functions

Logic functions (as well as predicates) can be recursively defined. However, such an approach will rapidly show some limits in their use for program proof. Indeed, when the automatic solver reasons on such logic properties, if such a function is met, it is necessary to evaluate it. SMT solvers are not meant to be efficient for this task, thus it is generally costly, producing too long proof resolution and eventually timeouts.

We can have a concrete example with the factorial function, using logic and using C language:

```
/*@
  logic integer factorial(integer n) = (n <= 0) ? 1 : n * factorial(n-1);
*/

/*@
  assigns \nothing ;
  ensures \result == factorial(n) ;
*/
int facto(int n){
  if(n < 2) return 1 ;

  int res = 1 ;
  /*@
    loop invariant 2 <= i <= n+1 ;
    loop invariant res == factorial(i-1) ;
    loop assigns i, res ;
    loop variant n - i ;
  */
  for(int i = 2 ; i <= n ; i++){
    res = res * i ;
  }
  return res ;
}
```

Without checking overflows, this function is easy and fast to prove. If we add runtime error checking, we see that there is a possibility of overflow on the multiplication.

On `int`, the maximum value for which we can compute factorial is 12. If we go further, it overflows. We can then add this precondition:

```
/*@
  requires n <= 12 ;
  assigns \nothing ;
  ensures \result == factorial(n) ;
*/
int facto(int n){
```

If we ask for a proof on this input, Alt-ergo will probably fail, whereas Z3 can compute the proof in less than a second. The reason is that in this case, the heuristics that are used by Z3 consider that it is a good idea to spend a bit more time on the evaluation of the function.

Logic functions can then be defined recursively but without some more help, we are rapidly limited by the fact that provers needs to perform evaluation or to “reason” by induction, two tasks for which they are not efficient. This can limit our possibilities for program proofs, but we will see later that we can get rid of these problems.

5.3.3. Exercises

5.3.3.1. Distance

Specify and prove the following program:

```c
int distance(int a, int b){
  if(a < b) return b - a ;
  else return a - b ;
}
```

For this, define two logic functions `abs` and `distance`. Use these functions to write the specification of the function.

5.3.3.2. Square

Write the body of the `square` function. Specify and prove the program. Use a `square` logic function.

```c
int abs(int x){
  return (x < 0) ? -x : x ;
}

unsigned square(int x){
  return 0 ; // to complete
}
```

Take care of the types of the variables and do not over-constrain the input of the function. Furthermore, when verifying the absence of runtime errors, do not forget to provide the options `-warn-unsigned-overflow` and `-warn-unsigned-downcast`.

5.3.3.3. Iota

Here is a possible implementation of the iota function:

```c
#include <limits.h>
#include <stddef.h>

void iota(int* array, size_t len, int value){
  if(len){
    array[0] = value ;

    for(size_t i = 1 ; i < len ; i++){
      array[i] = array[i-1]+1 ;
    }
  }
}
```

Write a logic function that returns the input value increased by one. Prove that after the execution of `iota`, the first value of the array is the input value and that each value of the array corresponds to the value that precedes it increased by one (using the previously defined logic function).

5.3.3.4. Vector add

In the following program, the `vec_add` function adds the second vector in input into the first one. Write a contract for the function `show_the_difference` that expresses, for each value of the vector `v1` the difference between the pre and the postcondition. For this, define a logic function `diff` that returns the difference between the value of a memory location at a label `L1` and the value at a label `L2`.

```c
#include <stddef.h>
#include <limits.h>

/*@
  predicate unchanged{L1, L2}(int* ptr, integer a, integer b) =
    \forall integer i ; a <= i < b ==> \at(ptr[i], L1) == \at(ptr[i], L2) ;
*/

/*@
  requires \valid(v1 + (0 .. len-1));
  requires \valid_read(v2 + (0 .. len-1));
  requires \separated(v1 + (0 .. len-1), v2 + (0 .. len-1));
  requires
    \forall integer i ; 0 <= i < len ==> INT_MIN <= v1[i]+v2[i] <= INT_MAX ;

  assigns v1[0 .. len-1];

  ensures
    \forall integer i ; 0 <= i < len ==> v1[i] == \old(v1[i]) + v2[i] ;
  ensures
    \forall integer i ; 0 <= i < len ==> v2[i] == \old(v2[i]) ;
*/
void vec_add(int* v1, const int* v2, size_t len){
  /*@
    loop invariant 0 <= i <= len ;
    loop invariant
      \forall integer j ; 0 <= j < i ==> v1[j] == \at(v1[j], Pre) + v2[j] ;
    loop invariant unchanged{Pre, Here}(v1, i, len) ;
    loop assigns i, v1[0 .. len-1] ;
    loop variant len-i ;
  */
  for(size_t i = 0 ; i < len ; ++i){
    v1[i] += v2[i] ;
  }
}

void show_the_difference(int* v1, const int* v2, size_t len){
  vec_add(v1, v2, len);
}
```

Re-express the `unchanged` predicate using the logic function you have defined.

5.3.3.5. The sum of the N first integers

The following function computes the sum of the N first integers. Write a recursive logic functions that returns the sum of the N first integers and write a specification for the C function expressing that it computes the same value as provided by the logic function.

```c
int sum_n(int n){
  if(n < 1) return 0 ;

  int res = 0 ;
  for(int i = 1 ; i <= n ; i++){
    res += i ;
  }
  return res ;
}
```

Try to verify the absence of runtime errors. The integer overflow is not so simple to get rid of. However, write a precondition that should be enough to prove the function (remember that the sum of the N first integers can be expressed with a really simple formula ...). It will certainly not be enough to directly prove the absence of overflow, but we will see how to provide such an information in the next section.

5.4. Lemmas

Lemmas are general properties about predicates or functions. These properties can be proved in isolation of the rest of the proof of a program by automatic or (more often) interactive provers. Once this proof is done, the information that it states can be safely used to simplify the reasoning in other, more complex proofs, without having to prove it again. For example, if we state a lemma L that says $P \Rightarrow Q$ in any case, if at some point in another proof, we have to prove Q, and we know P, we can directly conclude by using the lemma L without having to perform again the reasoning that brings us from P to Q.

In the previous section, we said that recursive function can make proof harder for SMT solvers. In such a case, lemmas can help us. We can write by ourselves the proofs that require inductive reasoning for some properties that we state as lemmas, and these lemmas can be used efficiently by SMT solvers to perform the other proofs, related to our programs.

5.4.1. Syntax

Again, we introduce lemmas using ACSL annotations. The syntax is following:

```
/*@
  lemma name_of_the_lemma { Label0, ..., LabelN }:
    property ;
*/
```

This time, the properties we want to express do not depend on received parameters (except for labels). So we express these properties for universally quantified variables. For example, we can state this lemma, which is true, even if it is trivial:

```
/*@
  lemma lt_plus_lt:
    \forall integer i, j ; i < j ==> i+1 < j+1;
*/
```

This proof can be performed using WP. The property is, of course, proved using only Qed.

5.4.2. Example: properties of linear functions

We can come back to our linear functions and express some interesting properties about them:

```
/*@
  lemma ax_b_monotonic_neg:
    \forall integer a, b, i, j ;
      a <  0 ==> i <= j ==> ax_b(a, i, b) >= ax_b(a, j, b);
  lemma ax_b_monotonic_pos:
    \forall integer a, b, i, j ;
      a >  0 ==> i <= j ==> ax_b(a, i, b) <= ax_b(a, j, b);
  lemma ax_b_monotonic_nul:
    \forall integer a, b, i, j ;
      a == 0 ==> ax_b(a, i, b) == ax_b(a, j, b);
*/
```

For these proofs, Alt-ergo, will probably not be able to discharge all generated verification conditions. In this case, Z3 will certainly succeed in finishing the remaining ones. We can then write the following example code:

```
/*@
  requires INT_MIN <= a*x <= INT_MAX ;
  requires INT_MIN <= ax_b(a,x,4) <= INT_MAX ;
  assigns \nothing ;
  ensures \result == ax_b(a,x,4);
*/
int function(int a, int x){
  return a*x + 4;
}

/*@
  requires INT_MIN <= a*x <= INT_MAX ;
  requires INT_MIN <= a*y <= INT_MAX ;
  requires a > 0;
  requires INT_MIN <= ax_b(a,x,4) <= INT_MAX ;
  requires INT_MIN <= ax_b(a,y,4) <= INT_MAX ;
  assigns \nothing ;
*/
void foo(int a, int x, int y){
  int fmin, fmax;
  if(x < y){
    fmin = function(a,x);
    fmax = function(a,y);
  } else {
    fmin = function(a,y);
    fmax = function(a,x);
  }
  //@assert fmin <= fmax;
}
```

If we do not give the lemmas provided earlier, Alt-ergo will not be able to prove the proof that `fmin` is lower or equals to `fmax`. With the lemmas it is however very easy for it since the property is the simply an instance of the lemma `ax_monotonic_pos`, the proof is then trivial as our lemma is considered to be true when are not currently proving it. Note that on this generalized version of `function`, Z3 will be probably more efficient to prove the absence of runtime errors.

5.4.3. Example: arrays and labels

Later in this tutorial, we will see some kind of definitions for which it is sometimes hard to reason about for SMT solvers when some mutations happen in memory. Thus, we will often need lemmas to state relations about the content of the memory between labels.

For now, let us illustrate with a first simple example. Consider the two following predicates:

```
/*@
  predicate sorted(int* array, integer begin, integer end) =
    \forall integer i, j ; begin <= i <= j < end ==> array[i] <= array[j] ;

  predicate unchanged{L1, L2}(int *array, integer begin, integer end) =
    \forall integer i ; begin <= i < end ==>
      \at(array[i], L1) == \at(array[i], L2) ;
*/
```

One could for example want to state that when an array is sorted, and some mutations happen in memory (creating a new memory state), but the content of the array remains unchanged, then the array is still sorted. This can be done with the following lemma:

```
/*@
  lemma unchanged_sorted{L1, L2}:
    \forall int* array, integer b, integer e ;
      sorted{L1}(array, b, e) ==>
      unchanged{L1, L2}(array, b, e) ==>
        sorted{L2}(array, b, e) ;
*/
```

We state this lemma for two labels `L1` and `L2`, and express that if any range in any array is sorted at `L1`, and unchanged from `L1` to `L2`, then it is still sorted at `L2`.

Note that this lemma is easily proved by SMT solvers. We will see later some examples where it is not so easy to get a proof.

5.4.4. Check lemma

Like assertions, it is possible to use a `check` version of lemmas:

```
/*@
  check lemma name_of_the_lemma { Label0, ..., LabelN }:
    property ;
*/
```

A `check lemma` annotation asks WP to generate a verification condition. But, contrarily to standard lemmas, the knowledge that the corresponding property is verified will not be added to the context of the proofs. Thus, SMT solvers will not be able to use it for other proofs. This kind of annotations can be useful to gain confidence about global annotations, testing whether they can be used in some situations, or on the opposite, that some unwanted properties cannot be proved.

There is no `admit lemma` annotation, these are `axiom` annotations that are described in Section 6.2.

5.4.5. Exercises

5.4.5.1. Multiplication property

Write a lemma that states that for three integers x, y and z, if x is greater than or equal to 0, if z is greater than or equal to y, then $x * z$ is greater than or equal to $x * y$.

This lemma will not be proved by SMT solvers. We provide a solution and the corresponding Coq proof on the GitHub repository of this book.

5.4.5.2. Locally sorted to globally sorted

The following program contains a function that requires an array to be sorted in the sense that each element is lower than or equal to the element that follows it and calls the binary search function.

```
/*@
  lemma element_level_sorted_implies_sorted:
    \true ; // to complete
*/

/*@
  requires \valid_read(arr + (0 .. len-1));
  requires element_level_sorted(arr, 0, len) ;
  requires in_array(value, arr, len);

  assigns \nothing ;

  ensures 0 <= \result < len ;
  ensures arr[\result] == value ;
*/
size_t bsearch_callee(int* arr, size_t len, int value){
  return bsearch(arr, len, value);
}
```

Take back your proved binary search function from the exercise 5.2.3.4. You might notice that the precondition of the binary search function is stronger than what we know in precondition of the `bsearch_callee` function. However, our precondition implies that the array is globally sorted. Write a lemma that states that if an array is `element_level_sorted` then it is `sorted`. This lemma will probably not be proved by SMT solvers, all remaining properties should be.

We provide a solution and the corresponding Coq proof on the GitHub repository of this book.

5.4.5.3. Sum of the N first integers

Take back your solution to the exercise 5.3.3.5 about the sum of the N first integers. Write a lemma that states that the result of the call to the logic function is $n * (n + 1)/2$. This lemma will not be proved by SMT solvers.

We provide a solution and the corresponding Coq proof on the GitHub repository of this book.

5.4.5.4. Shift transitivity

The following program is composed of two functions. The first one is the `shift_array` function that shifts the elements of an array with a given offset (named `shift`). The second performs two successive shifts on the same array.

```c
#include <stddef.h>
#include <stdint.h>

/*@
  predicate shifted_cell{L1, L2}(int* p, integer shift) =
    \at(p[0], L1) == \at(p[shift], L2) ;

  predicate shifted{L1, L2}(int* arr, integer fst, integer last, integer shift) =
    \true ; // to complete

  predicate unchanged{L1, L2}(int *a, integer begin, integer end) =
    \true ; // to complete

  lemma shift_ptr { TO_COMPLETE }:
    \true ; // to complete

  lemma shift_transivity{ TO_COMPLETE }:
    \true ; // to complete
*/

void shift_array(int* array, size_t len, size_t shift){
  for(size_t i = len ; i > 0 ; --i){
    array[i+shift-1] = array[i-1] ;
  }
}

/*@
  requires \valid(array+(0 .. len+s1+s2-1)) ;
  requires s1+s2 + len <= SIZE_MAX ;
  assigns array[s1 .. s1+s2+len-1];
  ensures shifted{Pre, Post}(array, 0, len, s1+s2) ;
*/
void double_shift(int* array, size_t len, size_t s1, size_t s2){
  shift_array(array, len, s1) ;
  shift_array(array+s1, len, s2) ;
}
```

Complete the predicates `shifted` and `unchanged`. The `shifted` predicate should use `shifted_cell`. The `unchanged` predicate should use `shifted`. Complete the contract of the `shift_array` function and prove it using WP.

Express two lemmas about the `shifted` property.

The first one `shift_ptr` should state that shifting a range `fst+s1` to `last+s1` of an `array` with an offset `s2` is equivalent to shifting the range `fst` to `last` of the memory location `array+s1` with an offset `s2`.

The second one should state that when the elements of an array are shifted a first time with an offset `s1` and then a second time with an offset `s2`, then the complete shift corresponds to a single shift with an offset `s1+s2`.

The lemma `shift_ptr` will not be proved by SMT solvers, we provide a solution and the corresponding Coq proof on the GitHub repository of this book. All remaining properties should be proved automatically.

5.4.5.5. Shift sorted range

The following program is composed of two functions. The function `shift_and_search` shifts the element of an array and then performs a binary search.

```
#include <stddef.h>

/*@
  predicate sorted(int* arr, integer begin, integer end) =
    \true ; // to complete

  predicate in_array(int value, int* arr, integer begin, integer end) =
    \true ; // to complete
*/

/*@
  predicate shifted_cell{L1, L2}(int* p, integer shift) =
    \at(p[0], L1) == \at(p[shift], L2) ;
*/

size_t bsearch(int* arr, size_t beg, size_t end, int value);
void shift_array(int* array, size_t len, size_t shift);

/*@
  lemma shifted_still_sorted{ TO_COMPLETE }:
    \true ; // to complete
  lemma in_array_shifted{ TO_COMPLETE }:
    \true ; // to complete
*/

/*@
  requires sorted(array, 0, len) ;
  requires \valid(array + (0 .. len));
  requires in_array(value, array, 0, len) ;

  assigns array[1 .. len] ;

  ensures 1 <= \result <= len ;
*/
unsigned shift_and_search(int* array, size_t len, int value){
  shift_array(array, len, 1);
  return bsearch(array, 1, len+1, value);
}
```

Take back your proved binary search function from the exercise 5.2.3.4, modify the binary search function, its contract and its proof in order to be able to search in any range.

Use the `shift_array` function proved in the previous exercise.

Complete the contract of the function `shift_and_search`. You might notice that the precondition that requires the array to be sorted is not validated, nor the postcondition of the caller. First, complete the lemma `shifted_still_sorted` that should state that if a range is sorted at some label and then shifted, the resulting range is still sorted, the precondition should now be proved. Then, complete the lemma `in_array_shifted` that should state that if a value is in a range of values that is then shifted, it is still in the resulting new range. The postcondition of the caller should now be proved.

The lemmas do not have to be proved by SMT solvers. We provide a solution and the corresponding Coq proofs on the GitHub repository of this book.

In this part of the tutorial, we have seen different ACSL constructs that allow us to factor our specifications and to express general properties that can be used by our solvers to make their task easier.

All techniques we have talked about are safe, since they do not *a priori* allow us to write false or contradictory specifications. At least if the specification only uses such logic constructions and if every lemma, precondition (at call site), every postcondition, assertion, variant and invariant are correctly proved, the code is correct.

However, sometimes, such constructions are not enough to express all properties we want to express about our programs. The next constructions we will see give us some new possibilities about it, but it will be necessary to be really careful using them since an error would allow us to introduce false assumptions or silently modify the program we are verifying.

6. ACSL - Logic definitions and ghost code

In this part of the tutorial, we will present three important notions of ACSL:

- inductive definitions,
- axiomatic definitions,
- ghost code.

In some cases, these notions are absolutely needed to ease the process of specification and, more importantly, proof. On one hand they force some properties to be more abstract when an explicit modeling would involve too much computation during proof. On the other hand, they force some properties to be explicitly modeled when they are harder to reason about when they are implicit.

Using these notions, we expose ourselves to the possibility to make our proof irrelevant if we make mistakes writing specification with it. Inductive predicates and axiomatic definitions involve the risk to introduce "false" in our assumptions, or to write imprecise definitions. Ghost code, if it does not verify some properties, opens the risk to silently modify the verified program … making us prove another program, which is not the one we want to prove.

6.1. Inductive definitions

Inductive predicates give a way to state properties whose verification requires to reason by induction, that is to say: for a property $p(\mathit{input})$, it is true for some base cases (for example, 0 is an even natural number), and knowing that $p(\mathit{input})$ is true, it is also true for a *bigger input*, provided that some conditions relating *input* and *bigger input* are verified (for example, knowing that n is even, we define that $n+2$ is also even). Thus, inductive predicates are generally useful to define properties recursively.

6.1.1. Syntax

For now, let us introduce the syntax of inductive predicates:

```
/*@
  inductive property{ Label0, ..., LabelN }(type0 a0, ..., typeN aN) {
  case c_1{Lq_0, ..., Lq_X}: p_1 ;
  ...
  case c_m{Lr_0, ..., Lr_Y}: p_km ;
  }
*/
```

Where all `c_i` are names and all `p_i` are logic formulas where `property` appears has a conclusion. Basically, `property` is true for some parameters and some labels, if it corresponds to one of the cases of the inductive definition.

We can have a look at the simple property we just talked about: how to define that a natural number is even using induction. We can translate the sentence: "0 is an even natural number, and if n is an even natural number, $n + 2$ is an even natural number":

```
/*@
  inductive even_natural{L}(integer n) {
  case even_nul{L}:
    even_natural(0);
  case even_not_nul_natural{L}:
    \forall integer n ;
      even_natural(n) ==> even_natural(n+2);
  }
*/
```

Which describes the two cases:

- 0 is even (base case),
- if a natural n is even, $n + 2$ is also even.

This can be used to prove the following assertions:

```c
void foo(){
  //@ assert even_natural(4);
  //@ assert even_natural(42);
}
```

Note that, since the solver has to recursively unfold the inductive definition, it might not work for any value. This capability depends on the heuristics of the solvers that decide or not to stop after N unfolds.

While this definition works, it is not completely satisfying. For example, we cannot deduce that an odd number is not even. If we try to prove that 1 is even, we will have to check if -1 is even, and then -3, -5, etc., infinitely. Leading to the fact that the following assertion cannot be proved:

```c
void bar(){
  int a = 1 ;
  //@ assert !even_natural(a);
}
```

Moreover, we generally prefer to define the induction cases the opposite way: defining the condition under which the wanted conclusion is true. For example, here, how can we verify that some n is natural and even? We first check whether it is 0, if not, we check if n is greater than 0, and then we verify that $n - 2$ is even:

```
/*@
  inductive even_natural{L}(integer n) {
  case even_nul{L}:
    even_natural(0) ;
  case even_not_nul_natural{L}:
    \forall integer n ; n > 0 ==> even_natural(n-2) ==>
      even_natural(n) ;
  }
*/
```

Here, we distinguish two cases:

- 0 is even,
- a natural n is even if it is greater than 0 and $n - 2$ is an even natural.

Taking the second case, we recursively decrease the number until we reach 0, and then the number is even, since 0 is even, or -1, and then there is no case in the inductive that corresponds to this value, so we can deduce that the property is false.

```
void bar(){
  int a = 1 ;
  //@ assert !even_natural(a);
}
```

Of course, defining that some natural number is even inductively is not a good idea, since we can simply define it using modulo. We generally use them to define complex recursive properties.

6.1.1.1. Consistency

Inductive definitions bring the risk to introduce inconsistencies. Indeed, the property specified in the different cases are considered to be always true, so if we introduce a property that allows to prove `false`, we will be able to prove everything. While we will give more details about axioms in the Section 6.2, let us give two hints to avoid building such a bad definition.

First, we can make sure that inductive predicates are well-founded. This can be done by syntactically restricting what we allow in an inductive definition, by making sure that each case has the form:

```
/*@
  \forall y1,...,ym ; h1 ==> ··· ==> hl ==> P(t1,...,tn) ;
*/
```

where the predicate `P` can only appear positively (so not negated with `!` - ¬) in the different hypotheses `hx`. Basically, it ensures that we cannot build both positive and negative occurrences of `P` for some parameters which would be incoherent.

This is for example verified by our previously defined predicate `even_natural`. While a definition like:

```
 1  /*@
 2    inductive even_natural{L}(integer n) {
 3    case even_nul{L}:
 4      even_natural(0) ;
 5    case even_not_nul_natural{L}:
 6      \forall integer n ; n > 0 ==> even_natural(n-2) ==>
 7      // negative occurrence of even_natural
 8      !even_natural(n-1) ==>
 9        even_natural(n) ;
10    }
11  */
```

does not respect this constraint as the property `even_natural` appears negatively on line 8.

Second, we can simply write a function that has a contract that needs the predicate `P`. For example, we can write a function:

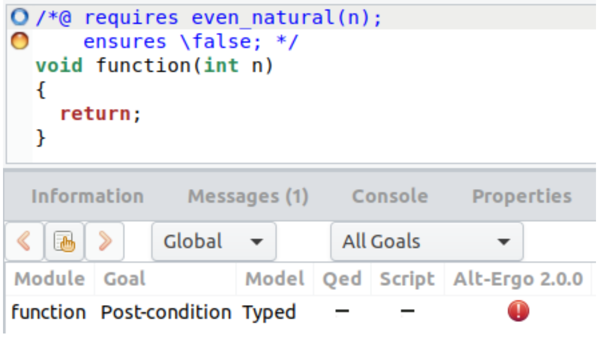
```
/*@
  requires P( params ... ) ;
  ensures  BAD: \false ;
*/ void function(params){

}
```

For example for our definition of `even_natural`, we would write:

```
/*@
  requires even_natural(n) ;
  ensures  \false ;
*/ void function(int n){

}
```

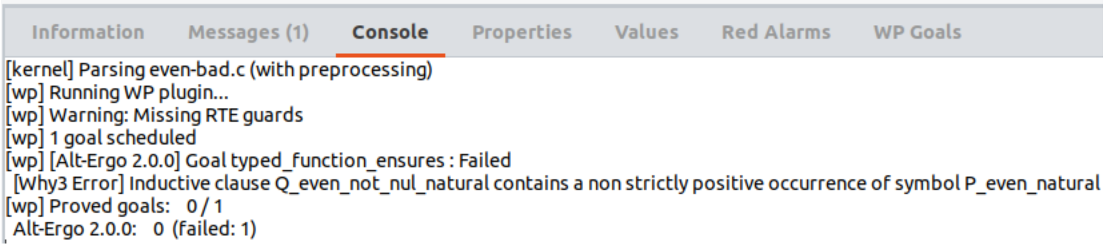
During the generation of the verification conditions, WP asks Why3 to create an inductive definition from the one written in ACSL. As Why3 is stricter than Frama-C and WP on this kind of definition, if the inductive predicate is too strange (if it is not well-founded), it will be rejected with an error. And indeed, with the bad definition of `even_natural` we just proposed, Why3 complains when we try to prove the `ensures \false` clause, because there exists a non-positive occurrence of `P_even_natural` that encodes the `even_natural` predicate we wrote in ACSL.

```
frama-c-gui -wp -wp-prop=BAD file.c
```



```
/*@ requires even_natural(n);
    ensures \false; */
void function(int n)
{
  return;
}
```

Module	Goal	Model	Qed	Script	Alt-Ergo 2.0.0
function	Post-condition	Typed	–	–	●



```
[kernel] Parsing even-bad.c (with preprocessing)
[wp] Running WP plugin...
[wp] Warning: Missing RTE guards
[wp] 1 goal scheduled
[wp] [Alt-Ergo 2.0.0] Goal typed_function_ensures : Failed
 [Why3 Error] Inductive clause Q_even_not_nul_natural contains a non strictly positive occurrence of symbol P_even_natural
[wp] Proved goals:   0 / 1
 Alt-Ergo 2.0.0:   0  (failed: 1)
```

However, that does not mean that we cannot write an inconsistent inductive definition. For example, the following definition is well-founded, while it allows us to prove false:

```
/*@ inductive P(int* p){
      case c: \forall int* p ; \valid(p) && p == (void*)0 ==> P(p);
    }
*/

/*@ requires P(p);
    ensures \false ; */
void foo(int *p){}
```

Here, we could detect the problem as `-wp-smoke-tests` can detect that the precondition is unsatisfiable. However, we must be careful when defining inductive definitions to not introduce a definition that can lead to a proof of false.

Before Frama-C 21 Scandium, the inductive definitions were translated, in Why3, as axioms. That means that this check was not performed. Thus, to get a similar behavior with older Frama-C versions, one has to use Coq and not a Why3 prover.

6.1.2. Recursive predicate definitions

Inductive predicates are often useful to express recursive properties since it prevents the provers to unroll the recursion when it is possible.

For example, if we want to define that an array only contains 0s, we could write it as follows:

```
/*@
  inductive zeroed{L}(int* a, integer b, integer e){
  case zeroed_empty{L}:
    \forall int* a, integer b, e; b >= e ==> zeroed{L}(a,b,e);
  case zeroed_range{L}:
    \forall int* a, integer b, e; b < e ==>
      zeroed{L}(a,b,e-1) && a[e-1] == 0 ==> zeroed{L}(a,b,e);
  }
*/
```

And we can again prove that our reset to 0 is correct with this new definition:

```
/*@
  requires \valid(array + (0 .. length-1));
  assigns  array[0 .. length-1];
  ensures  zeroed(array,0,length);
*/
void reset(int* array, size_t length){
  /*@
    loop invariant 0 <= i <= length;
    loop invariant zeroed(array,0,i);
    loop assigns i, array[0 .. length-1];
    loop variant length-i;
  */
  for(size_t i = 0; i < length; ++i)
    array[i] = 0;
}
```

Depending on the Frama-C or automatic solvers versions, the proof of the preservation of the invariant could fail. A reason for this fail is the fact that the prover forgets that cells preceding the one we are currently processing are actually still set to 0. We can add a lemma in our knowledge base, stating that if a set of values of an array did not change between two program points, the first one being a point where the property "zeroed" is verified, then the property is still verified at the second program point.

```
/*@
  predicate same_elems{L1,L2}(int* a, integer b, integer e) =
    \forall integer i; b <= i < e ==> \at(a[i],L1) == \at(a[i],L2);

  lemma no_changes{L1,L2}:
    \forall int* a, integer b, e;
      same_elems{L1,L2}(a,b,e) ==> zeroed{L1}(a,b,e) ==> zeroed{L2}(a,b,e);
*/
```

Then we can add an assertion to specify what did not change between the beginning of the loop block (pointed by the label `L` in the code) and the end (which is `Here` since we state the property at the end):

```
  for(size_t i = 0; i < length; ++i){
  L:
    array[i] = 0;
    //@ assert same_elems{L,Here}(array,0,i);
  }
```

Note that in this new version of the code, the property stated by our lemma is not proved using automatic solver, that cannot reason by induction. If the reader is curious, the (quite simple) Coq proof can be found in 6.4.

In this case study, using an inductive definition is not efficient: our property can be perfectly expressed using the basic constructs of the first order logic as we did before. Inductive definitions are meant to be used to write definitions that are not easy to express using the basic formalism provided by ACSL. It is here used to illustrate their use with a simple example.

6.1.3. Example: sort

Let us prove a simple selection sort:

```c
size_t min_idx_in(int* a, size_t beg, size_t end){
  size_t min_i = beg;
  for(size_t i = beg+1; i < end; ++i)
    if(a[i] < a[min_i]) min_i = i;
  return min_i;
}

void swap(int* p, int* q){
  int tmp = *p; *p = *q; *q = tmp;
}

void sort(int* a, size_t beg, size_t end){
  for(size_t i = beg ; i < end ; ++i){
    size_t imin = min_idx_in(a, i, end);
    swap(&a[i], &a[imin]);
  }
}
```

The reader can exercise by specifying and proving the search of the minimum and the swap function. We hide there a correct version of these specification and code (Answers 6.4), we will focus on the specification and the proof of the sort function that is an interesting use case for inductive definitions.

Indeed, a common error we could do, trying to prove a sort function, would be to write this specification (which is correct!):

```c
/*@
  predicate sorted(int* a, integer b, integer e) =
    \forall integer i, j; b <= i <= j < e ==> a[i] <= a[j];
*/

/*@
  requires \valid(a + (beg .. end-1));
  requires beg < end;
  assigns  a[beg .. end-1];
  ensures sorted(a, beg, end);
*/
void sort(int* a, size_t beg, size_t end){
  /* @ // add invariant */
  for(size_t i = beg ; i < end ; ++i){
    size_t imin = min_idx_in(a, i, end);
    swap(&a[i], &a[imin]);
  }
}
```

This specification is correct. However, if we recall correctly the part of the tutorial about specifications, we have to *precisely* express what we expect of the program. With this specification, we do not prove all required properties expected for a sort function. For example, this function correctly answers to the specification:

```c
/*@
  requires \valid(a + (beg .. end-1));
  requires beg < end;

  assigns  a[beg .. end-1];

  ensures sorted(a, beg, end);
*/
void fail_sort(int* a, size_t beg, size_t end){
  /*@
    loop invariant beg <= i <= end;
    loop invariant \forall integer j; beg <= j < i ==> a[j] == 0;
    loop assigns i, a[beg .. end-1];
    loop variant end-i;
  */
  for(size_t i = beg ; i < end ; ++i)
    a[i] = 0;
}
```

Our specification does not express the fact that all elements initially present in the array must still be in the array after executing the sort function. That is to say: the sort function produces a sorted permutation of the original array.

Defining the notion of permutation can be done using an inductive definition. While we will see later a version of this property that is more general, let us for now limit us to a notion of permutation that is more specific to our current need. We can limit us to a few cases. First, the array is a permutation of itself, then swapping two values of the array produces a new permutation if we do not change anything else. And finally, if we create the permutation p_2 of p_1, and then the permutation p_3 of p_2, then by transitivity p_3 is a permutation of p_1.

The corresponding inductive definition is the following:

```c
/*@
  predicate swap_in_array{L1,L2}(int* a, integer b, integer e, integer i, integer j) =
    b <= i < e && b <= j < e &&
    \at(a[i], L1) == \at(a[j], L2) &&
    \at(a[j], L1) == \at(a[i], L2) &&
    \forall integer k; b <= k < e && k != j && k != i ==>
      \at(a[k], L1) == \at(a[k], L2);

  inductive permutation{L1,L2}(int* a, integer b, integer e){
  case reflexive{L1}:
    \forall int* a, integer b,e ; permutation{L1,L1}(a, b, e);
  case swap{L1,L2}:
    \forall int* a, integer b,e,i,j ;
      swap_in_array{L1,L2}(a,b,e,i,j) ==> permutation{L1,L2}(a, b, e);
  case transitive{L1,L2,L3}:
    \forall int* a, integer b,e ;
      permutation{L1,L2}(a, b, e) && permutation{L2,L3}(a, b, e) ==>
        permutation{L1,L3}(a, b, e);
  }
*/
```

We can then specify that our sort function produces the sorted permutation of the original array, and we can then prove it by providing the invariant of the function:

```
 1  /*@
 2    requires beg < end && \valid(a + (beg .. end-1));
 3    assigns  a[beg .. end-1];
 4    ensures sorted(a, beg, end);
 5    ensures permutation{Pre, Post}(a,beg,end);
 6  */
 7  void sort(int* a, size_t beg, size_t end){
 8    /*@
 9      loop invariant beg <= i <= end;
10      loop invariant sorted(a, beg, i) && permutation{Pre, Here}(a, beg, end);
11      loop invariant \forall integer j,k; beg <= j < i ==> i <= k < end ==> a[j] <= a[k];
12      loop assigns i, a[beg .. end-1];
13      loop variant end-i;
14    */
15    for(size_t i = beg ; i < end ; ++i){
16      //@ ghost begin: ;
17      size_t imin = min_idx_in(a, i, end);
18      swap(&a[i], &a[imin]);
19      //@ assert swap_in_array{begin,Here}(a,beg,end,i,imin);
20    }
21  }
```

This time, our property is precisely defined, the proof is relatively easy to produce, only requiring to add an assertion in the block of the loop to state that it only performs a swap of values in the array (and then giving the transition to the next permutation). To define this swap notion, we use a particular annotation (at line 16), introduced using the `ghost` keyword. Here, the goal is to introduce a label in the code that in fact does not exist from the program point of view, and is only visible from a specification point of view. We present the “ghost” features in the final section of this chapter, for now let us focus on axiomatic definitions.

6.1.4. Exercises

6.1.4.1. Sum of the N first integers

Take back your solution to the exercise 5.4.5.3 about the sum of the N first integers. Rewrite the logic function using an inductive predicate that states that some integer equals the sum of the N first integers.

```
#include <limits.h>

/*@
  inductive is_sum_n(integer n, integer res) {
  case C: \true; // to complete
  }
*/

/*@
  requires n*(n+1) <= 2*INT_MAX ;
  assigns \nothing ;
  // ensures ... ;
*/
int sum_n(int n){
  if(n < 1) return 0 ;

  int res = 0 ;
  /*@
    loop invariant 1 <= i <= n+1 ;
    // loop invariant ... ;
    loop assigns i, res ;
    loop variant n - i ;
  */
  for(int i = 1 ; i <= n ; i++){
    res += i ;
  }
  return res ;
}
```

Adapt the contract of the function and the lemma(s). Note that lemma(s) could certainly not be proved by SMT solvers. We provide a solution and corresponding Coq proofs on the GitHub repository of this book.

6.1.4.2. Greatest Common Divisor

Write an inductive predicate that states that some integer is the greatest common divisor of two others. The goal of the exercise is to prove that the function `gcd` computes the greatest common divisor. Thus, we do not have to specify all the cases for the predicate. Indeed, a close look at the loop shows us that after the first iteration `a` is greater or equals to `b`, and it is maintained by the loop. Thus, we consider two cases for the inductive predicate:

- `b` is 0,
- if some `d` is the GCD of `b` and `a % b`, then it is the GCD of `a` and `b`

```
#include <limits.h>

/*@ inductive is_gcd(integer a, integer b, integer div) {
    case gcd_zero: \true ; // to complete
    case gcd_succ: \true ; // to complete
    }
*/

/*@
  requires a >= 0 && b >= 0 ;
  assigns \nothing ;
  // ensures ... ;
*/
int gcd(int a, int b){
  /*@
    loop invariant \forall integer t ; \true ; // to complete
  */
  while (b != 0){
    int t = b;
    b = a % b;
    a = t;
  }
  return a;
}
```

Express the postcondition of the function, and complete the invariant to prove that it is verified. Note that the invariant should make use of the inductive case `gcd_succ`.

6.1.4.3. Power function

In this exercise, we do not consider RTEs.

Write an inductive predicate that states that some integer `r` equals to x^n. Consider the two cases: either n is 0 or it is greater, and then it should be related to the value x^{n-1}.

```
/*@
  inductive is_power(integer x, integer n, integer r) {
  case zero: \true; // to complete
  case N: \true; // to complete
  }
*/
```

First prove the naive version of the power function:

```
/*@
  requires n >= 0 ;
  // assigns ...
  // ensures ...
*/
int power(int x, int n){
  int r = 1 ;
  /*@
    loop invariant 1 <= i <= n+1 ;
    // loop invariant ...
  */
  for(int i = 1 ; i <= n ; ++i){
    r *= x ;
  }
  return r ;
}
```

Now, let us prove a faster version of the power function:

```
/*@
  requires n >= 0 ;
  // assigns ...
  // ensures ...
*/
int fast_power(int x, int n){
  int r = 1 ;
  int p = x ;
  /*@
    loop invariant \forall integer v ; \true; // to complete
  */
  while(n > 0){
    if(n % 2 == 1) r = r * p ;
    p *= p ;
    n /= 2 ;
  }
  //@ assert is_power(p, n, 1) ;

  return r ;
}
```

In this version, we exploit two properties about the power operator:

- $(x^2)^n = x^{2n}$

- $x \times (x^2)^n = x^{2n+1}$

that allows us to divide n by 2 at each step of the loop instead of decreasing it by one (which makes the algorithm $O(\log n)$ instead of $O(n)$). Express the two previous properties in lemmas:

```
/*@
  lemma power_even: \true; // to complete
  lemma power_odd: \true; // to complete
*/
```

First express the lemma `power_even`, the SMT solver might be able to combine the use of this lemma and the inductive predicate to deduce `power_odd`. The Coq proof of the `power_even` lemma is provided on the GitHub repository of this book, as well as the proof of the `power_odd` in case the SMT solver does not prove it.

Finally, complete the contract and loop invariant of the `fast_power` function. For this notice that at the beginning of the loop $x^{old(n)} = p^n$, and that each iteration uses the previous properties to update r, in the sense that we have $x^{old(n)} = r \times p^n$ during all the loop, until we have $n = 0$ and thus $p^n = 1$.

6.1.4.4. Permutation

Take back the definitions of the `shifted` and `unchanged` predicates from the exercise 5.4.5.4. The `shited_cell` predicate can be inlined and removed. Use the shift predicate to express the `rotate` predicate that expresses that some elements of an array are rotated to the left in the sense that all elements are shifted of one element to the left except the last one that is put in the first cell of the range. Use this predicate to prove the rotate function:

```
/*@
  predicate rotate{L1, L2}(int* arr, integer fst, integer last) =
    \true ; // to complete
*/

/*@
  assigns arr[beg .. end-1] ;
  ensures rotate{Pre, Post}(arr, beg, end) ;
*/
void rotate(int* arr, size_t beg, size_t end){
  int last = arr[end-1] ;
  for(size_t i = end-1 ; i > beg ; --i){
    arr[i] = arr[i-1] ;
  }
  arr[beg] = last ;
}
```

Express a new version of the notion of permutation with an inductive predicate that considers four cases:

- permutation is reflexive,

- if the left part of the range (until an index of the range) is rotated between two labels, we still have a permutation,
- if the right part of the range (from an index of the range) is rotated between two labels, we still have a permutation,
- permutation is transitive.

```
/*@
  inductive permutation{L1, L2}(int* arr, integer fst, integer last){
  case reflexive{L1}: \true ; // to complete
  case rotate_left{L1,L2}: \true ; // to complete
  case rotate_right{L1,L2}: \true ; // to complete
  case transitive{L1,L2,L3}: \true ; // to complete
  }
*/
```

Complete the contract of `two_rotates` that successively rotates the first half of the array and then the second half and prove that it maintains the permutation.

```
void two_rotates(int* arr, size_t beg, size_t end){
  rotate(arr, beg, beg+(end-beg)/2) ;
  //@ assert permutation{Pre, Here}(arr, beg, end) ;
  rotate(arr, beg+(end-beg)/2, end) ;
}
```

6.2. Axiomatic definitions

Axioms are properties we state to be true no matter the situation. It is a good way to state complex properties that will allow the proof process to be more efficient by abstracting their complexity. Of course, as any property expressed as an axiom is assumed to be true, we have to be very careful when we use them to defined properties: if we introduce a false property in our assumptions, “false” becomes “true” and we can then prove anything.

6.2.1. Syntax

Axiomatic definitions are introduced using this syntax:

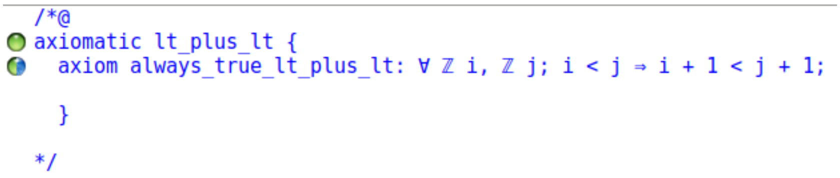
```
/*@
  axiomatic Name_of_the_axiomatic_definition {
    // here we can define or declare functions and predicates

    axiom axiom_name { Label0, ..., LabelN }:
      // property ;

    axiom other_axiom_name { Label0, ..., LabelM }:
      // property ;

    // ... we can put as many axioms we need
  }
```

```
*/
```

For example, we can write this axiomatic block:

```
/*@
  axiomatic lt_plus_lt{
    axiom always_true_lt_plus_lt:
      \forall integer i, j; i < j ==> i+1 < j+1 ;
  }
*/
```

And we can see that in Frama-C, this property is actually assumed to be true[1]:



```
/*@
axiomatic lt_plus_lt {
  axiom always_true_lt_plus_lt: ∀ ℤ i, ℤ j; i < j ⇒ i + 1 < j + 1;

  }

*/
```

6.2.1.1. Link with lemmas

Lemmas and axioms allows to express the same kinds of properties. Namely, properties expressed about quantified variables (and possibly global variables, but it is quite rare since it is often difficult to find a global property about such variables being both true and interesting). Apart this first common point, we can also notice that when we are not considering the definition of the lemma itself, lemmas are assumed to be true by WP exactly as axioms are.

The only difference between lemmas and axioms from a proof point of view is that we must provide a proof that each lemma is true, whereas an axiom is always assumed to be true.

6.2.2. Recursive function or predicate definitions

Axiomatic definitions of recursive functions and predicates are particularly useful since they will prevent provers from unrolling the recursion when it is possible.

The idea is then not to define directly the function or the predicate but to declare it and then to define axioms that specify its behavior. If we come back to the factorial function, we can define it axiomatically as follows:

```
/*@
  axiomatic Factorial{
    logic integer factorial(integer n);

    axiom factorial_0:
      \forall integer i; i <= 0 ==> 1 == factorial(i) ;

    axiom factorial_n:
      \forall integer i; i > 0 ==> i * factorial(i-1) == factorial(i) ;
```

[1] In section 6.4, we present an *extremely* useful axiom.

```
  }
*/
```

In this axiomatic definition, our function does not have a body. Its behavior is only defined by the axioms we have stated about it. Except this, nothing changes, in particular the logic function can be used in our specification just as before.

A small subtlety that we must take care of is the fact that if some axioms state properties about the content of some pointed memory cells, we have to specify considered memory blocks using the `reads` notation in the declaration. If we omit such a specification, the predicate or function will be considered to be stated about the received pointers and not about pointer memory blocks. So, if the code modifies the content of an array for which we had proven that the predicate or function gives some result, this result will not be considered to be potentially different.

For example, if we take the inductive property we stated for “zeroed” in the previous chapter, we can write it using an axiomatic definition, and it will be written like this:

```
/*@
  axiomatic A_all_zeros{
    predicate zeroed{L}(int* a, integer b, integer e) reads a[b .. e-1];

    axiom zeroed_empty{L}:
      \forall int* a, integer b, e; b >= e ==> zeroed{L}(a,b,e);

    axiom zeroed_range{L}:
      \forall int* a, integer b, e; b < e ==>
        zeroed{L}(a,b,e-1) && a[e-1] == 0 ==> zeroed{L}(a,b,e);
  }
*/
```

Notice the `reads[b .. e-1]` that specifies the memory location on which the predicate depends. While it is not necessary to specify what are the memory locations read in an inductive definition, we have to specify such an information for axiomatically defined properties.

6.2.3. Consistency

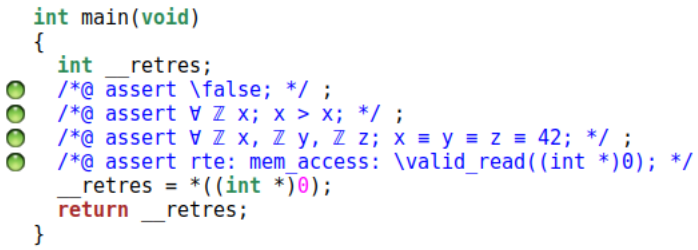
By adding axioms to our knowledge base, we can produce more complex proofs since some part of these proofs, expressed by axioms, do not need to be proved (they are already specified to be true) shortening the proof process. However, using axiomatic definitions, **we must be extremely careful**. Indeed, even a small error could introduce false in the knowledge base, making our whole reasoning futile. Our reasoning would still be correct, but relying on false knowledge, it would only learn incorrect things.

The simplest example is the following:

```
/*@
  axiomatic False{
    axiom false_is_true: \false;
  }
*/
```

```

int main(){
  // Examples of proved properties

  //@ assert \false;
  //@ assert \forall integer x; x > x;
  //@ assert \forall integer x,y,z ; x == y == z == 42;
  return *(int*) 0;
}
```

And everything is proved, comprising the fact that the dereferencing of 0 is valid:



```
int main(void)
{
  int __retres;
  /*@ assert \false; */ ;
  /*@ assert ∀ ℤ x; x > x; */ ;
  /*@ assert ∀ ℤ x, ℤ y, ℤ z; x ≡ y ≡ z ≡ 42; */ ;
  /*@ assert rte: mem_access: \valid_read((int *)0); */
  __retres = *((int *)0);
  return __retres;
}
```

Of course, this example is extreme, we would not write such an axiom. The problem is in fact that it is really easy to write an axiomatic definition that is subtly false when we express more complex properties, or adding assumptions about the global state of the system.

When we start to create axiomatic definitions, it is worth adding assertions or postconditions requiring a proof of false that we expect to fail to ensure that the definition is not inconsistent. However, it is often not enough! If the subtlety that creates the inconsistency is hard enough to find, provers could need a lot of information other than the axiomatic definition itself to be able to find and use the inconsistency, we then need to always be careful!

More specifically, specifying the values read by a function or a predicate is important for the consistency of an axiomatic definition. Indeed, as previously explained, if we do not specify what is read when a pointer is received, an update of a value in the array does not invalidate a property known about the content of the array. In such a case, the proof is performed but since the axiom does not talk about the content of the array, we do not prove anything.

For example, in the function that resets an array to 0, if we modify the axiomatic definition, removing the specification of the values that are read by the predicate (`reads a[b .. e-1]`), the proof will still be performed, but will not prove anything about the content of the arrays. For example, the following function:

```
/*@
  requires length > 10 ;
  requires \valid(array + (0 .. length-1));
  requires zeroed(array,0,length);
  assigns  array[0 .. length-1];
  ensures  zeroed(array,0,length);
*/
void bad_function(int* array, size_t length){
  array[5] = 42 ;
}
```

is proved to be correct, while we obviously changed a value in the array and the value is not 0 anymore.

Note that unlike inductive definitions, where Why3 provides us a way to control that what we write in ACSL is relatively well-defined, we do not have such a mechanism for axiomatic definitions. Basically, even with Why3 such a definition is translated into a list of axioms that are thus assumed.

6.2.4. Cluster of axiomatic blocks

Most global annotations (logic functions, predicates, lemmas, ...) can be defined at two different levels: either at top-level, the level of the functions, global variables, etc. (except for axioms and abstract functions and predicates) or in axiomatic blocks. While top-level global annotations (in particular lemmas) are always embedded in the context of verification conditions, it is not the case for the annotations in axiomatic blocks.

In the following example:

```c
/*@ axiomatic X {
      predicate P(int* p) reads *p;
      axiom x: \forall int *p ; *p == 0 ==> P(p) ;
    }
*/

/*@ axiomatic Y {
      predicate Q(int *p) reads *p ;
      axiom y: \forall int *p ; *p == 0 ==> P(p) && Q(p) ;
    }
*/

//@ ensures P(p) ;
void function(int* p){}
```

Since the `ensures` clause only uses `P` which is defined in the axiomatic block `X`, WP only loads the axiom `x`. On the opposite, if we replace `P(p)` with `Q(p)` in the `ensures` clause, WP loads the axiomatic block `Y`, thus the axiom `y` that uses `P`. Consequently, the axiomatic block `X` is loaded too. The transitive closure of the loaded axiomatic blocks forms a cluster of axiomatic definitions.

One can use this behavior to avoid providing too many lemmas and axioms to SMT solvers. This can improve proof performances in some situations. We will present more details about how to guide proof using lemmas in Section 7.2.

6.2.5. Example: counting occurrences of a value

In this example, we want to prove that an algorithm actually counts the occurrences of a value in an array. We start by axiomatically defining what is the number of occurrences of a value in an array:

```c
/*@
  axiomatic Occurrences_Axiomatic{
    logic integer l_occurrences_of{L}(int value, int* in, integer from, integer to)
      reads in[from .. to-1];

    axiom occurrences_empty_range{L}:
      \forall int v, int* in, integer from, to;
        from >= to ==> l_occurrences_of{L}(v, in, from, to) == 0;

    axiom occurrences_positive_range_with_element{L}:
      \forall int v, int* in, integer from, to;
        (from < to && in[to-1] == v) ==>
          l_occurrences_of(v,in,from,to) == 1+l_occurrences_of(v,in,from,to-1);

    axiom occurrences_positive_range_without_element{L}:
      \forall int v, int* in, integer from, to;
        (from < to && in[to-1] != v) ==>
          l_occurrences_of(v,in,from,to) == l_occurrences_of(v,in,from,to-1);
  }
*/
```

We have three different cases:

- the considered range of values is empty: the number of occurrences is 0,
- the considered range of values is not empty and the last element is the one we are searching for: the number of occurrences is the number of occurrences in the current range without the last element, plus 1,
- the considered range of values is not empty and the last element is not the one we are searching for: the number of occurrences is the number of occurrences in the current range without the last element.

Then, we can write the C function that computes the number of occurrences of a value in an array and prove it:

```
/*@
  requires \valid_read(in+(0 .. length));
  assigns  \nothing;
  ensures  \result == l_occurrences_of(value, in, 0, length);
*/
size_t occurrences_of(int value, int* in, size_t length){
  size_t result = 0;

  /*@
    loop invariant 0 <= result <= i <= length;
    loop invariant result == l_occurrences_of(value, in, 0, i);
    loop assigns i, result;
    loop variant length-i;
  */
  for(size_t i = 0; i < length; ++i)
    result += (in[i] == value)? 1 : 0;

  return result;
}
```

An alternative way to specify, in this code, that `result` is at most `i`, is to express a more general lemma about the number of occurrences of a value in an array, since we know that it is comprised between 0 and the size of maximum considered range of values:

```
/*@
lemma l_occurrences_of_range{L}:
  \forall int v, int* in, integer from, to:
    from <= to ==> 0 <= l_occurrences_of(v, in, from, to) <= to-from;
*/
```

An automatic solver cannot discharge this lemma. It would be necessary to prove it interactively using Coq, for example. By expressing generic manually proved lemmas, we can often add useful tools to provers to manipulate more efficiently our axiomatic definitions, without directly adding new axioms that would augment the chances to introduce errors. Here, we still have to realize the proof of the lemma to get a complete proof.

6.2.6. Example: The `strlen` function

In this section, let us prove the C `strlen` function:

```
#include <stddef.h>

size_t strlen(char const *s){
  size_t i = 0 ;
  while(s[i] != '\0'){
    ++i;
  }
  return i ;
}
```

First, we have to provide a suitable contract. Let us suppose that we have a logic function `strlen` that returns the length of a string, that is to say, what we expect of our C function:

```
/*@
  logic integer strlen(char const* s) = // let's see later
*/
```

Basically, we want to receive a valid string in input, and we want to compute a value that equals to the result of our logic function `strlen` applied to this string, of course this function does not assign anything. Defining what is a valid string is not that simple. Indeed, previously in this tutorial, we only worked with arrays, receiving in input both the array and the size of the array, however here, and as it is common in C, we suppose that the string ends with a character `'\0'`. That means that we basically need the `strlen` function to define what is a valid string. Let us first use this definition (note that we use the `\valid_read` variant of pointer validity since we do not expect the function to modify the string) and provide a contract for `strlen`:

```
/*@
  predicate valid_read_string(char * s) =
    \valid_read(s + (0 .. strlen(s))) ;
*/

/*@
  requires valid_read_string(s) ;
  assigns \nothing ;
  ensures \result == strlen(s) ;
*/
```

```
size_t strlen(char const *s)
```

Defining the logic function `strlen` is a bit tricky. Indeed, we want to compute the length of a string by finding the character `'\0'`, we expect to find it but in fact, we can easily imagine that we receive a string of infinite length. In this case, we would like to return an error value, but it is basically impossible to guarantee that the computation terminates, thus a logic function cannot be used to express this property.

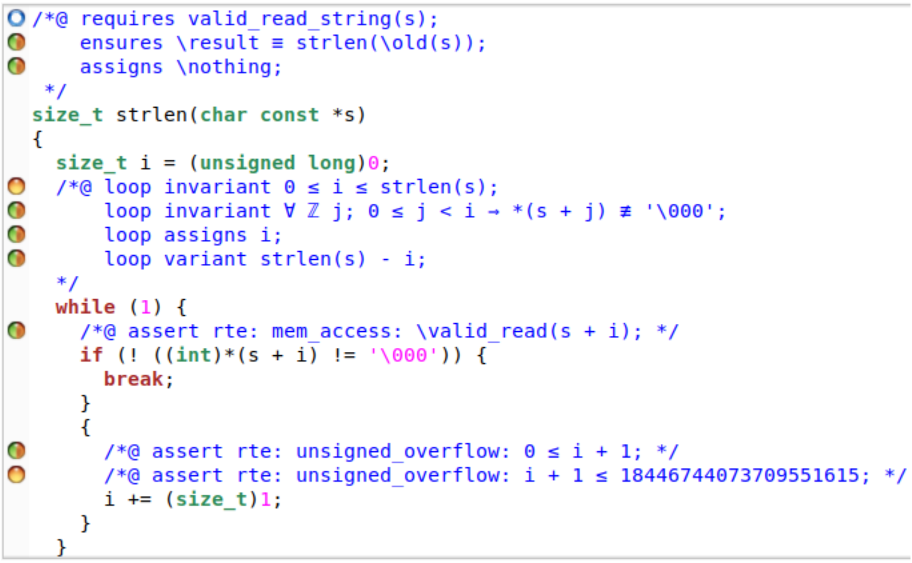
Thus, let us define this function axiomatically. First, let us define what is read by the function, which is: any memory cell from the pointer to an infinite range of address. Then we consider two cases: the string is finite, or it is not, that leads to two axioms: `strlen` returns a positive value that corresponds to the index of the first `'\0'` character, and returns a negative value if no such value exists.

```
/*@
  axiomatic StrLen {
    logic integer strlen(char * s) reads s[0 .. ] ;

    axiom pos_or_nul{L}:
      \forall char* s, integer i ;
        (0 <= i && (\forall integer j ; 0 <= j < i ==> s[j] != '\0') && s[i] == '\0') ==>
          strlen(s) == i ;

    axiom no_end{L}:
      \forall char* s ;
        (\forall integer i ; 0 <= i ==> s[i] != '\0') ==> strlen(s) < 0 ;
```

And now, we can be more precise for our definition of `\valid_read_string`, a valid string is a string such that it is valid from the first index to `strlen` of the string and, such that this value is greater than 0 (since an infinite string is not a valid string):

```
/*@
  predicate valid_read_string(char * s) =
    strlen(s) >= 0 && \valid_read(s + (0 .. strlen(s))) ;
*/
```

With this definition we can now go further and provide a suitable invariant to the loop of the `strlen` function. It is quite simple: `i` ranges between 0 and `strlen(s)`, for all values met before the iteration `i`, they are not `'\0'`. This loop assigns `i` and the variant corresponds to the distance between `i` and `strlen(s)`. However, if we try to produce the proof of correctness of the function, it fails. And to get more information we can try a verification asking RTE with the verification that unsigned integers do not overflow:



```
/*@ requires valid_read_string(s);
    ensures \result ≡ strlen(\old(s));
    assigns \nothing;
 */
size_t strlen(char const *s)
{
  size_t i = (unsigned long)0;
  /*@ loop invariant 0 ≤ i ≤ strlen(s);
      loop invariant ∀ ℤ j; 0 ≤ j < i → *(s + j) ≢ '\000';
      loop assigns i;
      loop variant strlen(s) - i;
  */
  while (1) {
    /*@ assert rte: mem_access: \valid_read(s + i); */
    if (! ((int)*(s + i) != '\000')) {
      break;
    }
    {
      /*@ assert rte: unsigned_overflow: 0 ≤ i + 1; */
      /*@ assert rte: unsigned_overflow: i + 1 ≤ 18446744073709551615; */
      i += (size_t)1;
    }
  }
```

We see that the prover fails to discharge the verification condition related to the range of `i`, and that `i` can exceed the maximum of an unsigned int. One could try to provide a limit to the value of `strlen(s)` in precondition:

```
requires valid_read_string(s) && strlen(s) <= SIZE_MAX ;
```

However, it is not enough, and the reason is that while we have defined that the value of `strlen(s)` is defined to be the index of the first `'\0'` in the array, the converse is not true: knowing that the value of `strlen(s)` is positive is not enough to deduce that the value at the corresponding index is `'\0'`. Thus, we extend the axiomatic definition with another proposition that gives us this fact (we also add one for the values that precede the `strlen(s)` index even if here, it is not necessary):

```
axiom index_of_strlen{L}:
  \forall char* s ;
    strlen(s) >= 0 ==> s[strlen(s)] == '\0' ;

axiom before_strlen{L}:
  \forall char* s ;
    strlen(s) >= 0 ==> (\forall integer i ; 0 <= i < strlen(s) ==> s[i] != '\0') ;
```

And this time the proof succeeds. Frama-C provides its own standard library headers, and they include an axiomatic definition for the `strlen` logic function. It can be found in the installation directory of Frama-C, under the directory `libc`, the file is named `__fc_string_axiomatic.h`. Note that this definition include more axioms in order to be able to deduce more properties about `strlen`.

6.2.7. Exercises

6.2.7.1. Occurrence counting

The following program cannot be proved with the axiomatic definition we previously defined about occurrences counting:

```
/*@
  requires \valid_read(in+(0 .. length));
  assigns  \nothing;
  ensures  \result == l_occurrences_of(value, in, 0, length);
*/
size_t occurrences_of(int value, int* in, size_t length){
  size_t result = 0;

  for(size_t i = length; i > 0 ; --i)
    result += (in[i-1] == value) ? 1 : 0;

  return result;
}
```

Re-express the axiomatic definition in a form that allows to prove the program.

6.2.7.2. Greatest Common Divisor

Express the logic function that allows to compute the greatest common divisor as an axiomatic definition, write the contract of the `gcd` function and prove it:

```
#include <limits.h>

/*@
  axiomatic GCD {
    // ...
  }
*/

/*@
  requires a >= 0 && b >= 0 ;
  // assigns ...
  // ensures ...
*/
int gcd(int a, int b){
  while (b != 0){
    int t = b;
    b = a % b;
    a = t;
  }
  return a;
}
```

6.2.7.3. Sum of the N first integers

Express the logic function that allows to compute the sum of the N first integers as an axiomatic definition. Write the contract of the following `sum_n` function and prove it:

```
#include <limits.h>

/*@ axiomatic Sum_n {
      // ...
    }
*/

/*@ lemma sum_n_value: \true; // to complete */

/*@
  requires n >= 0 ;
  // requires ...
  // assigns ...
  // ensures ...
*/
int sum_n(int n){
  if(n < 1) return 0 ;

  int res = 0 ;
  /*@ loop invariant 1 <= i <= n+1 ;
      // ...
  */
  for(int i = 1 ; i <= n ; i++){
    res += i ;
  }
  return res ;
}
```

6.2.7.4. Permutation

Take back the example about selection sort (section 6.1.3). Re-express the permutation predicate using an axiomatic definition. Take care of the `reads` clause (in particular, note that the predicate relates two memory labels).

```
#include <stddef.h>

/*@
  predicate
  swap_in_array{L1,L2}(int* a, integer b, integer e, integer i, integer j) =
    \true ; // to complete
*/

/*@
  axiomatic Permutation {
    // to complete
    predicate permutation{L, K}(int* a, integer b, integer e) ;
  }
*/

/*@
  predicate sorted(int* a, integer b, integer e) =
    \forall integer i, j; b <= i <= j < e ==> a[i] <= a[j];
*/

size_t min_idx_in(int* a, size_t beg, size_t end){
  return 0;
}

void swap(int* p, int* q){}

/*@
  requires beg < end && \valid(a + (beg .. end-1));
  assigns  a[beg .. end-1];
  ensures sorted(a, beg, end);
  ensures permutation{Pre, Post}(a,beg,end);
*/
void sort(int* a, size_t beg, size_t end){
  /*@
    loop invariant beg <= i <= end;
    loop invariant sorted(a, beg, i) && permutation{Pre, Here}(a, beg, end);
    loop invariant \forall integer j,k; beg <= j < i ==> i <= k < end ==> a[j] <= a[k];
    loop assigns i, a[beg .. end-1];
    loop variant end-i;
  */
  for(size_t i = beg ; i < end ; ++i){
    //@ ghost begin: ;
    size_t imin = min_idx_in(a, i, end);
    swap(&a[i], &a[imin]);
    //@ assert swap_in_array{begin,Here}(a,beg,end,i,imin);
  }
}
```

6.3. Ghost code

The previous techniques we have seen in this chapter are meant to make the specification more abstract. The role of ghost code is the opposite, here, we find in fact a way to enrich our specification with information expressed as concrete C code. The idea is to add variables and source code that is not part of the actual program to verify and is thus only visible for the verification tool. It is used to make explicit some logic properties that, else, would only be known implicitly.

6.3.1. Syntax

Ghost code is added using annotations that contain C code introduced using the `ghost` keyword:

```
/*@
  ghost
  // C code
*/
```

In ghost code, we write normal C code. Some subtleties will be explained later in this section. For now, let us have a look to the basic elements we can write in ghost code.

We can declare variables:

```
//@ ghost int ghost_glob_var = 0;

void foo(int a){
  //@ ghost int ghost_loc_var = a;
}
```

These variables can be modified via instructions and conditional structures:

```
//@ ghost int ghost_glob_var = 0;

void foo(int a){
  //@ ghost int ghost_loc_var = a;
  /*@ ghost
    for(int i = 0 ; i < 10 ; i++){
      ghost_glob_var += i ;
      if(i < 5) ghost_local_var += 2 ;
    }
  */
}
```

We can declare ghost labels that can be used into annotations (or to perform a `goto` from the ghost code itself, under some conditions that we will explain later):

```
void foo(int a){
  //@ ghost Ghost_label: ;
  a = 28 ;
  //@ assert ghost_loc_var == \at(a, Ghost_label) == \at(a, Pre);
}
```

A conditional structure `if` can be extended with a ghost `else` if it does not have one originally. For example:

```
void foo(int a){
  //@ ghost int a_was_ok = 0 ;
  if(a < 5){
    a = 5 ;
  } /*@ ghost else {
    a_was_ok = 1 ;
  } */
}
```

A function can be provided with ghost parameters that can be used to give more information for the verification of the function. For example, if we imagine the verification of a function that receives a linked list, we could provide a ghost parameter that represents the length of the list:

```
void function(struct list* l) /*@ ghost (int length) */ {
  // visit the list
  /*@ variant length ; */
  while(l){
    l = l->next ;
    //@ ghost length--;
  }
}
void another_function(struct list* l){
  //@ ghost int length ;

  // ... do something to compute the length

  function(l) /*@ ghost(n) */ ; // we call 'function' with the ghost argument
}
```

Note that when a function takes ghost parameters, all calls to this function must provide the corresponding ghost arguments.

An entire function can be ghost. For example, in the previous example, we could have used a ghost function to compute the length of the list:

```
/*@ ghost
  /@ ensures \result == logic_length_of_list(l) ; @/
  int compute_length(struct list* l){
    // does the right computation
  }
*/

void another_function(struct list* l){
  //@ ghost int length ;

  //@ ghost length = compute_length(l) ;
  function(l) /*@ ghost(n) */ ; // we call function with the ghost parameter
}
```

Here, we can see a specific syntax for the contract of the ghost function. Indeed, it is often useful to write some contracts or assertions in ghost code. As it must be specified in code that is already in C comments, we must use a specific syntax to provide ghost contracts or assertions. We open ghost annotations with the syntax `/@` and close them with `@/`. Of course, it applies to loops in ghost code for example:

```
void foo(unsigned n){
 /*@ ghost
   unsigned i ;

   /@
     loop invariant 0 <= i <= n ;
     loop assigns i ;
     loop variant n - i ;
   @/
   for(i = 0 ; i < n ; ++i){

   }
   /@ assert i == n ; @/
 */
}
```

6.3.2. Ghost code validity, what Frama-C checks

Frama-C verifies several properties about the ghost code we write:

- the ghost code cannot modify the control flow graph of the program,
- the normal code cannot access to the ghost memory,
- the ghost code can only write the ghost memory.

Verifying these properties, our goal is to guarantee that for any program, for any input, its observable behavior is the same with or without the ghost code.

> Before Frama-C 21 Scandium, most of these properties were not verified by the Frama-C kernel. Thus, if we work with a previous version, we have to ensure that they are verified ourselves.

If some of these properties are not verified, it would mean that the ghost code can change the behavior of the verified program. Let us have a closer look to each of these constraints.

6.3.2.1. Maintain the same control flow

The control flow of a program is the order in which the instructions are executed by the program. If the ghost code changes this order, or let the program ignore some instructions of the original program, then the behavior of the program is not the same, and thus, we are not verifying the same program.

For example, this function compute the sum of the n first integers :

```c
int sum(int n){
  int x = 0 ;
  for(int i = 0; i <= n; ++i){
    //@ ghost break;
    x += i ;
  }
  return x;
}
```

By introducing, in ghost code, the instruction `break` in the body of the loop, the program does not have the same behavior anymore: instead of visiting all is between 0 and $n+1$, we stop immediately in the first iteration of the loop. Consequently, this program is rejected by Frama-C:

```
[kernel:ghost:bad-use] file.c:4: Warning:
  Ghost code breaks CFG starting at:
  /*@ ghost break; */
  x += i;
```

It is important to note that when a ghost code changes the control flow, the instruction that is pointed by Frama-C is the starting point of the ghost code. For example, if we introduce a conditional around our `break` instruction:

```c
int sum(int n){
  int x = 0 ;
  for(int i = 0; i <= n; ++i){
    //@ ghost if(i < 3) break;
    x += i ;
  }
  return x;
}
```

The error message will point to the enclosing `if` :

```
[kernel:ghost:bad-use] file.c:4: Warning:
  Ghost code breaks CFG starting at:
  /*@ ghost if (i < 3) break; */
  x += i;
```

Note that verifying that the control flow is not changed by ghost code is done in a purely syntactic way. For example, if the `break` is unreachable in any execution of the program, an error will still be triggered:

```
int sum(int n){
  int x = 0 ;
  for(int i = 0; i <= n; ++i){
    //@ ghost if(i > n) break;
    x += i ;
  }
  return x;
}
```

```
[kernel:ghost:bad-use] file.c:4: Warning:
  Ghost code breaks CFG starting at:
  /*@ ghost if (i > n) break; */
  x += i;
```

Finally, we can remark that there exists two ways to change the control flow. The first one is to use jumps (thus `break`, `continue`, or `goto`), the second is to introduce a non-terminating code. For this last one, unless the non-termination is trivial, the kernel cannot verify that the control flow is changed (and thus, it never does, this is delegated to plugins). We will treat this question in section 6.3.3.

6.3.2.2. Access to the memory

The ghost code is an observer of the normal code. Consequently, normal code is not authorized to access ghost code, neither to its memory, nor to the functions. Ghost code can read the memory of the normal code, but cannot write it. Currently, ghost code cannot call normal functions, we will give more details about this restriction later in this section.

Normal code is not allowed to read ghost code for a very simple reason: if the normal code tries to access ghost variables, it would simply not compile: the compiler does not see the variables declared in annotations. For example:

```
int sum_42(int n){
  int x = 0 ;
  //@ ghost int r = 42 ;
  for(int i = 0; i < n; ++i){
    x += r;
  }
  return x;
}
```

cannot be compiled:

```
# gcc -c file.c
file.c: In function ‘sum_42’:
file.c:5:10: error: ‘r’ undeclared (first use in this function)
    5 |     x += r;
      |          ^
```

and is thus also refused by Frama-C:

```
[kernel] file.c:5: User Error:
Variable r is a ghost symbol. It cannot be used in non-ghost context. Did you forget a /*@
    ghost ... /?
  3       //@ ghost int r = 42 ;
  4       for(int i = 0; i < n; ++i){
  5         x += r;
                 ^
  6       }
  7       return x;
```

In ghost code, normal variables cannot be modified. Indeed, it would imply that we can change the result of a program by just adding ghost code. For example, in the following code:

```c
int sum(int n){
  int x = 0 ;
  for(int i = 0; i <= n; ++i){
    x += i ;
    //@ ghost x++;
  }
  return x;
}
```

The result would not be the same with or without the ghost code. Frama-C thus forbids such a code:

```
[kernel:ghost:bad-use] file.c:5: Warning:
  'x' is a non-ghost lvalue, it cannot be assigned in ghost code
```

Note that this verification is done thanks to the types of the different variables. A variable declared in normal code has a normal status, while a variable declared in ghost code has a ghost status. Consequently, even if the ghost code does not really change the behavior of the program, any write on a normal variable in ghost code is forbidden:

```c
int sum(int n){
  int x = 0 ;
  for(int i = 0; i <= n; ++i){
    x += i ;
    /*@ ghost
      if (x < INT_MAX){
        x++;
        x--; // assure that x remains coherent
      }
    */
  }
  return x;
}
```

```
[kernel:ghost:bad-use] file.c:9: Warning:
  'x' is a non-ghost lvalue, it cannot be assigned in ghost code
[kernel:ghost:bad-use] file.c:10: Warning:
  'x' is a non-ghost lvalue, it cannot be assigned in ghost code
```

This also applies to the `assigns` clause when it is in ghost code:

```
int x ;

/*@ ghost
  /@ assigns x ; @/
  void ghost_foo(void);
*/

void foo(void){
  /*@ ghost
    /@ loop assigns x ; @/
    for(int i = 0; i < 10; ++i);
  */
}
```

```
[kernel:ghost:bad-use] file.c:4: Warning:
  'x' is a non-ghost lvalue, it cannot be assigned in ghost code
[kernel:ghost:bad-use] file.c:11: Warning:
  'x' is a non-ghost lvalue, it cannot be assigned in ghost code
```

On the other hand, normal functions and loops contracts can (and must) specify assigned ghost memory locations. For example, if we fix the previous program by making `x` ghost, first, our previous `assigns` clauses are allowed, but we can also specify that the function `foo` writes the ghost global variable `x`:

```
//@ ghost int x ;

/*@ ghost
  /@ assigns x ; @/
  void ghost_foo(void);
*/

/*@ assigns x ; */
void foo(void){
  /*@ ghost
    /@ loop assigns x ; @/
    for(int i = 0; i < 10; ++i);
  */
}
```

6.3.2.3. Typing of ghost elements

We should now provide some more details about the typing of variables declared in ghost code. For example, it is sometimes useful to create a ghost array to store some information:

```c
void function(int a[5]){
  //@ ghost int even[5] = { 0 };

  for(int i = 0; i < 5; ++i){
    //@ ghost if(a[i] % 2) even[i] = 1;
  }
}
```

Here, we use indices to access our arrays, but we could for example access their content using pointers:

```c
void function(int a[5]){
  //@ ghost int even[5] = { 0 };
  //@ ghost int *pe = even ;

  for(int *p = a; p < a+5; ++p){
    //@ ghost if(*p % 2) *pe = 1;
    //@ ghost pe++;
  }
}
```

However, we immediately see that Frama-C does not like our way to do it:

```
[kernel:ghost:bad-use] file.c:3: Warning:
  Invalid cast of 'even' from 'int \ghost *' to 'int *'
[kernel:ghost:bad-use] file.c:6: Warning:
  '*pe' is a non-ghost lvalue, it cannot be assigned in ghost code
```

In particular, the first message indicates that we try to cast a pointer to `int \ghost` into a pointer to `int`. Indeed, when a variable is declared in ghost code, only the variable is considered ghost. Thus, for a pointer, the pointed memory is not considered ghost (and thus here, while `pe` is ghost, the memory pointed by `pe` is not). To solve this problem, Frama-C provides the `\ghost` qualifier that can be used to add a ghost status to a type:

```c
void function(int a[5]){
  //@ ghost int even[5] = { 0 };
  //@ ghost int \ghost * pe = even ;

  for(int *p = a; p < a+5; ++p){
    //@ ghost if(*p % 2) *pe = 1;
    //@ ghost pe++;
  }
}
```

On some aspects, the `\ghost` qualifier looks like the `const` keyword. However, it does not behave exactly the same way for two main reasons.

First, while the following `const` definition is allowed, it is not allowed to write something similar with the `\ghost` qualifier:

```
int const * * const p ;
//@ ghost int \ghost * * p ;
```

```
[kernel:ghost:bad-use] file.c:2: Warning:
  Invalid type for 'p': indirection from non-ghost to ghost
```

Declaring a const pointer to a mutable location that contains pointers to some constant locations can make sense. On the other hand, it does make sense for the `\ghost` qualifier. That would mean that some normal memory contains pointers to ghost memory locations, and we do not want to allow this.

Second, we can assign a pointer to mutable locations to a pointer to const locations:

```
int a[10] ;
int const * p = a ;
```

This code is valid as we only restrict our capacity to write memory locations when we initialize (or assign) `p` to `&a[0]`. On the other hand, both the two following pointer initialization (or equivalent assignments) are forbidden with the `\ghost` qualifier:

```
int non_ghost_int ;
//@ ghost int ghost_int ;

//@ ghost int \ghost * p = & non_ghost_int ;
//@ ghost int * q = & ghost_int ;
```

While the reason why we refuse the first initialization is quite direct: it would allow writing into the normal memory from ghost code through `p`, the reason why we refuse the second one is not that intuitive. And, indeed, we must use workarounds to create a problem with this conversion:

```
/*@ ghost
  /@ assigns *p ;
     ensures *p == \old(*q); @/
  void assign(int * \ghost * p, int * \ghost * q){
    *p = *q ;
  }
*/
void caller(void){
  int x ;

  //@ ghost int \ghost * p ;
  //@ ghost int * q = &x ;
  //@ ghost assign(&p, &q) ;
  //@ ghost *p = 42 ;
}
```

Here, we make a conversion that seems to be reasonable. Indeed, we give the address of a pointer to some ghost memory location to a function that takes a pointer to a normal memory, thus we only restrict our ability to access to the pointed memory. However, by this function call, the function `assign` assigns the current value of `q` (which is `&x`) to `p`. Thus, via the last instruction, we can write `x` in ghost code, which should be forbidden. Consequently, such a conversion is never allowed.

Finally, a ghost code cannot currently call a non ghost function for similar reasons. Some particular cases could be allowed, but this is not currently supported by Frama-C.

6.3.3. Ghost code validity, what remains to be verified

Except for the restrictions mentioned in the previous section, ghost code is just normal C code. That means that if we want to verify the original program, we must take care of at least two more aspects:

- absence of runtime errors,
- ghost code termination.

The first case does not require more attention than the rest of the code. Indeed, absence of runtime errors in ghost code will be treated by the RTE plugin as for normal code.

As we said in section 4.2.3, there are two kinds of correctness: partial and total correctness, the second one allowing to prove that a program terminates. For normal code, showing that the code terminates is not always something that we want. On the other hand, if we use some ghost code during the verification, it is absolutely necessary to prove total correctness of this ghost code. Indeed, if it is non-terminating (if it contains an infinite loop for example), it could allow us to prove anything about the program.

```
/*@ ensures \false ; */
void foo(void){
  /*@ ghost
    while(1){}
  */
}
```

6.3.4. Make a logical state explicit

The goal of ghost code is to make explicit some information that would be else implicit. For example, in the verification of the sort algorithm, we used it to create a label in the program that is not visible by the compiler but that we can use in the verification. The fact that the values were swapped between the two labels was implicitly provided by the contract of the function, adding this ghost label allows us to write an explicit assertion of this fact.

Let us take a more complex example where we more clearly create explicit knowledge about the program. Here, we want to prove that the following function returns the value of the maximal sum of subarrays of a given array. A subarray of an array `a` is a contiguous subset of values of `a`. For example, for an array `{ 0 , 3 , -1 , 4 }`, some subarrays can be `{}`,

`{ 0 }`, `{ 3 , -1 }`, `{ 0 , 3 , -1 , 4 }`, ... Note that as we allow empty arrays for subarrays, the sum is at least 0. In the previous array, the subarray with the maximal sum is `{ 3 , -1 , 4 }`, the function would then return 6.

```c
int max_subarray(int *a, size_t len) {
  int max = 0;
  int cur = 0;
  for(size_t i = 0; i < len; i++) {
    cur += a[i];
    if (cur < 0)   cur = 0;
    if (cur > max) max = cur;
  }
  return max;
}
```

In order to specify the previous function, we need an axiomatic definition about sum. This is not too complex, the careful reader can express the needed axioms as an exercise:

```c
/*@
  axiomatic Sum_array{
    logic integer sum(int* array, integer begin, integer end) reads array[begin .. (end-1)];
```

Some correct axioms are available at: 6.4

The specification of the function is the following:

```c
/*@
  requires \valid(a+(0..len-1));
  assigns \nothing;
  ensures \forall integer l, h;  0 <= l <= h <= len ==> sum(a,l,h) <= \result;
  ensures \exists integer l, h;  0 <= l <= h <= len &&  sum(a,l,h) == \result;
*/
```

For any bounds, the value returned by the function must be greater or equal to the sum of the elements between these bounds, and there must exist some bounds such that the returned value is exactly the sum of the elements between these bounds. About this specification, when we want to add the loop invariant, we realize that we miss some information. We want to express what are the values `max` and `cur`, and what are the relations between them, but we cannot do it!

Basically, in order to prove our postcondition, we need to know that there exists some bounds `low` and `high` such that the computed sum corresponds to these bounds. However, in our code, we do not have anything that expresses it from a logic point of view, and we cannot *a priori* make the link between this logic formalization. We then use ghost code to record these bounds and express the loop invariant.

We first need two variables to record the bounds of the maximum sum range, let us call them `low` and `high`. Every time we find a range where the sum is greater than the current one, we update our ghost variables. These bounds then correspond to the sum currently stored by `max`. That induces that we need other bounds: the ones that correspond to the sum stored by the variable `cur` from which we build the bounds corresponding to `max`. For these bounds, we only add a single ghost variable: the current low bound `cur_low`, the high bound being the variable `i` of the loop.

```
/*@
  requires \valid(a+(0..len-1));
  assigns \nothing;
  ensures \forall integer l, h;  0 <= l <= h <= len ==> sum(a,l,h) <= \result;
  ensures \exists integer l, h;  0 <= l <= h <= len &&  sum(a,l,h) == \result;
*/
int max_subarray(int *a, size_t len) {
  int max = 0;
  int cur = 0;
  //@ ghost size_t cur_low = 0;
  //@ ghost size_t low = 0;
  //@ ghost size_t high = 0;

  /*@
    loop invariant BOUNDS: low <= high <= i <= len && cur_low <= i;

    loop invariant REL :   cur == sum(a,cur_low,i) <= max == sum(a,low,high);
    loop invariant POST:   \forall integer l;    0 <= l <= i       ==> sum(a,l,i) <= cur;
    loop invariant POST:   \forall integer l, h; 0 <= l <= h <= i ==> sum(a,l,h) <= max;

    loop assigns i, cur, max, cur_low, low, high;
    loop variant len - i;
  */
  for(size_t i = 0; i < len; i++) {
    cur += a[i];
    if (cur < 0) {
      cur = 0;
      /*@ ghost cur_low = i+1; */
    }
    if (cur > max) {
      max = cur;
      /*@ ghost low = cur_low; */
      /*@ ghost high = i+1; */
    }
  }
  return max;
}
```

The invariant `BOUNDS` expresses how the different bounds are ordered during the computation. The invariant `REL` expresses what the variables `cur` and `max` mean depending on the bounds. Finally, the invariant `POST` allows us to create a link between the invariant and the postcondition of the function.

The reader can verify that this function is indeed correctly proved without RTE verification. If we add RTE verification, the overflow on the variable `cur`, that is the sum, seems to be possible (and it is indeed the case).

Here, we do not try to fix this because it is not the topic of this example. The way we can prove the absence of RTE here strongly depends on the context where we use this function. A possibility is to strongly restrict the contract, forcing some properties about values and the size of the array. For example, we could limit the maximal size of the array and bound each value of the different cells. Another possibility would be to add an error value in case of overflow (-1 for example), and to specify that when an overflow is produced, this value is returned.

6.3.5. Exercises

6.3.5.1. Ghost code validity

In these example functions, and without running Frama-C, explain what is wrong with the ghost code. When Frama-C should reject the code, explain why. Note that one can execute Frama-C without checking ghost code validity using the option `-kernel-warn-key ghost=inactive`.

```c
#include <stddef.h>
#include <limits.h>

/*@
  assigns \nothing;
  ensures \result == a || \result == b ;
  ensures \result >= a && \result >= b ;
*/
int max(int a, int b){
  int r = INT_MAX;
  //@ ghost r = (a > b) ? a : b ;
  return r ;
}

/*@
  requires \valid(a) && \valid(b);
  assigns *a, *b;
  ensures *a == \old(*b) && *b == \old(*a);
*/
void swap(int* a, int* b){
  int tmp = *a ;
  *a = *b ;
  //@ ghost int \ghost* ptr = b ;
  //@ ghost *ptr = tmp ;
}

/*@
  requires \valid(a+(0 .. len-1));
  assigns  \nothing ;
  ensures \result <==> (\forall integer i ; 0 <= i < len ==> a[i] == 0);
*/
int null_vector(int* a, size_t len){
  //@ ghost int value = len ;
  /*@ loop invariant 0 <= i <= len ;
    loop invariant \forall integer j ; 0 <= j < i ==> a[j] == 0 ;
    loop assigns i ;
    loop variant len-i ;
  */
  for(size_t i = 0 ; i < len ; ++i)
    if(a[i] != 0) return 0;
  /*@ ghost
    /@ loop assigns \nothing ; @/
    while(value >= len);
  */
  return 0;
}
```

6.3.5.2. Two times

This program computes `2 * x` using a loop. Use a ghost variable `i` to express as an invariant that the value of `r` is `i * 2` and complete the proof.

```
/*@
  requires x >= 0 ;
  assigns  \nothing ;
  ensures  \result == 2 * x ;
*/
int times_2(int x){
  int r = 0 ;
  /*@
    loop invariant 0 <= x ;
    loop invariant r == 0 ; // to complete
    loop invariant \true ; // to complete
  */
  while(x > 0){
    r += 2 ;
    x -- ;
  }
  return r;
}
```

6.3.5.3. Playing with arrays

In this function, we receive an array, and we have a loop where we do nothing except that we have indicated that it assigns the content of the array. However, we would like to prove in postcondition that the array has not been modified.

```
/*@
  requires \valid(a + (0 .. 9)) ;
  assigns  a[0 .. 9] ;
  ensures  \forall integer j ; 0 <= j < 10 ==> a[j] == \old(a[j]) ;
*/
void foo(int a[10]){
  //@ ghost int g[10] ;
  /*@ ghost
    ; // to complete
  */

  /*@
    loop invariant 0 <= i <= 10 ;
    loop invariant \true ; // to complete
    loop assigns i, a[0 .. 9] ;
    loop variant 10 - i ;
  */
  for(int i = 0; i < 10; i++);
}
```

Without modifying the `assigns` clause of the loop and without using the keyword `\at`, prove that the function does not modify the array. For this, complete the ghost code and the loop invariant, by assuring that the array `g` represents the old value of `a`.

Once it is done, create a ghost function that performs the copy and use it in `foo` to perform the same proof.

6.3.5.4. Search and replace

The following program performs a search and replace:

```
#include <stddef.h>

void replace(int *a, size_t length, int old, int new) {
  for (size_t i = 0; i < length; ++i) {
    if (a[i] == old)
      a[i] = new;
  }
}

/*@
  requires \valid(a + (0 .. length-1));
  assigns a[0 .. length-1];
  ensures \forall integer i ; 0 <= i < length ==> -100 <= a[i] <= 100 ;
*/
void initialize(int *a, size_t length);

void caller(void) {
  int a[40];

  initialize(a, 40);

  //@ ghost L: ;

  replace(a, 40, 0, 42);

  // here we want to obtain the updated locations via a ghost array
}
```

Assuming that the function `replace` requires `old` and `new` to be different, write a contract for `replace` and prove that the function satisfies it.

Now, we would like to know which cells have been updated by the function. Add a ghost parameter to `replace` so that it can receive a second array that will record the updated (or not) cells. Adding also the following code after the call to replace:

```
/*@ ghost
  /@ loop invariant 0 <= i <= 40 ;
     loop assigns i;
     loop variant 40 - i ;
  @/
  for(size_t i = 0 ; i < 40 ; ++i){
    if(updated[i]){
      /@ assert a[i] != \at(a[\at(i, Here)], L); @/
    } else {
      /@ assert a[i] == \at(a[\at(i, Here)], L); @/
    }
  }
*/
```

Everything should be proved.

6.4. Hidden content

6.4.1. Coq Proof of the `no_changes` lemma

```
Inductive P_zeroed: (addr -> Z) -> addr ->
  Z -> Z -> Prop :=
  | Q_zeroed_empty : forall (Mint:addr -> Z) (a:addr) (b:Z) (e:Z),
      (e <= b)%Z -> is_sint32_chunk Mint -> P_zeroed Mint a b e
  | Q_zeroed_range : forall (Mint:addr -> Z) (a:addr) (b:Z) (e:Z),
      let x := ((-1%Z)%Z + e)%Z in
      let x1 := Mint (shift a x) in
      (x1 = 0%Z) -> (b < e)%Z -> is_sint32_chunk Mint ->
      P_zeroed Mint a b x -> is_sint32 x1 -> P_zeroed Mint a b e.

Definition P_same_elems (Mint:addr -> Z)
    (Mint1:addr -> Z) (a:addr) (b:Z) (e:Z) : Prop :=
  forall (i:Z),
  let a1 := shift a i in (b <= i)%Z -> (i < e)%Z -> ((Mint1 a1) = (Mint a1)).

(* The property to prove *)
Theorem wp_goal :
  forall (t:addr -> Z) (t1:addr -> Z) (a:addr) (i:Z) (i1:Z),
  is_sint32_chunk t1 -> is_sint32_chunk t -> P_zeroed t1 a i i1 ->
  P_same_elems t t1 a i i1 -> P_zeroed t a i i1.
Proof.
  Require Import Psatz. (* Used for reasoning on integers *)

  (* We introduce our variable and the main hypothese *)
  intros Mi' Mi arr b e tMi tMi' H.
  (* We reason by induction on our first (inductive) hypothese *)
  induction H ; intros Same.
  + (* Base case, immediate by using the first case of the inductive predicate *)
    constructor 1 ; auto.
  + unfold x in * ; clear x.
    (* Induction case, by using the second case of the inductive predicate.
       Most premises are trivial or just simple integers relations. *)
      constructor 2 ; auto ; try lia.
    - (* First: the first cell in new memory must be zero, we replace 0 with
         the cell in old memory *)
      rewrite <- H ; symmetry.
      (* And show that the cells are the same *)
      apply Same ; lia.
    - (* Third we use our induction hypothesis to show that the property
         holds on the first part of the array *)

      apply IHP_zeroed ; auto.
      intros i' ; intros.
      apply Same ; lia.
Qed.
```

6.4.2. Specified sort functions

```
/*@
  requires \valid_read(a + (beg .. end-1));
  requires beg < end;

  assigns  \nothing;

  ensures  \forall integer i; beg <= i < end ==> a[\result] <= a[i];
  ensures  beg <= \result < end;
```

```
*/
size_t min_idx_in(int* a, size_t beg, size_t end){
  size_t min_i = beg;

  /*@
    loop invariant beg <= min_i < i <= end;
    loop invariant \forall integer j; beg <= j < i ==> a[min_i] <= a[j];
    loop assigns min_i, i;
    loop variant end-i;
  */
  for(size_t i = beg+1; i < end; ++i){
    if(a[i] < a[min_i]) min_i = i;
  }
  return min_i;
}

/*@
  requires \valid(p) && \valid(q);
  assigns  *p, *q;
  ensures  *p == \old(*q) && *q == \old(*p);
*/
void swap(int* p, int* q){
  int tmp = *p; *p = *q; *q = tmp;
}
```

6.4.3. An important axiom

Currently, our automatic solvers are not powerful enough to compute *the Answer to the Ultimate Question of Life, the Universe, and Everything*. We can help them by stating it as an axiom! Now, we just have to understand the question to determine in which case this result can be useful ...

```
/*@
  axiomatic Ax_answer_to_the_ultimate_question_of_life_the_universe_and_everything {
    logic integer the_ultimate_question_of_life_the_universe_and_everything{L} ;

    axiom answer{L}:
      the_ultimate_question_of_life_the_universe_and_everything{L} = 42;
  }
*/
```

6.4.4. Sum axioms

```
/*@
  axiomatic Sum_array{
    logic integer sum(int* array, integer begin, integer end) reads array[begin .. (end-1)];

    axiom empty:
      \forall int* a, integer b, e; b >= e ==> sum(a,b,e) == 0;
    axiom range:
      \forall int* a, integer b, e; b < e ==> sum(a,b,e) == sum(a,b,e-1)+a[e-1];
  }
*/
```

In this part, we have covered some advanced constructions of the ACSL language that allow to express and prove more complex properties about programs.

Badly used, these features can make our analyses incorrect, we then need to be careful manipulating them and not hesitate to check them again and again, or possibly express properties to verify about them to assure us that we are not introducing an incoherence in our program or our assumptions.

7. Proof methodologies

Now that we have presented most of the important features of ACSL for program proof, let us have a more general view of program proof with Frama-C and WP. We will present some approaches that can be used depending on the target of verification, the expected level of confidence and the kind of formalization (and features of ACSL) we use.

7.1. Absence of runtime errors: Minimal contracts

We have seen that program proof allows verifying two main aspects of program correctness: first that programs do not contain runtime errors, second that programs correctly implement their specification. However, it is sometimes hard to guarantee the latter, and the former is already an interesting step for program correctness.

Indeed, runtime errors often cause the presence of so-called “undefined behaviors” in C programs. These undefined behaviors are often vectors of security breaches and thus, guaranteeing their absence already protects us of a lot of attack vectors. The absence of runtime errors can be verified thanks to an approach called minimal contracts.

7.1.1. Principle

The minimal contracts approach is guided by the RTE plugin. Basically, the idea is to generate the assertions related to absence of runtime errors for all the functions of a module or a project and to write the minimal set of (correct) contracts that are sufficient to prove that the runtime errors cannot happen. Most of the time, we need far fewer lines of specification that what is usually required to prove the functional correctness of the program.

Let us take a simple example with the absolute value function.

```c
int abs(int x){
  return (x < 0) ? -x : x ;
}
```

Here, we can generate the assertions required to prove the absence of runtime errors, which generate this program:

```c
/* Generated by Frama-C */
int abs(int x)
{
  int tmp;
  if (x < 0)
    /*@ assert rte: signed_overflow: -2147483647 ≤ x; */
    tmp = - x;
  else tmp = x;
  return tmp;
}
```

Thus, we only need to specify as a requirement that the value of `x` must be greater than `INT_MIN`:

```
/*@
  requires x > INT_MIN ;
*/
int abs(int x){
  return (x < 0) ? -x : x ;
}
```

This condition is enough to prove that no runtime error can happen in the function.

As we will see later, the function is however generally used in a particular context. So this contract will likely not be enough. For example, we often have global variables in our program and here we do not specify what is assigned by the function. Most of the time the "assigns" clause cannot be ignored (which is expected in a language where everything is mutable by default). Moreover, if one take the absolute value of an integer, it is surely because they need a positive value. In reality, the minimal contract of the absolute value function is more likely the following:

```
/*@
  requires x > INT_MIN ;
  assigns  \nothing ;
  ensures  \result >= 0 ;
*/
int abs(int x){
  return (x < 0) ? -x : x ;
}
```

But this addition should only be guided by the verification of the context(s) where the function is used once we have proved the absence of runtime errors in the function itself.

7.1.2. Example: the search function

Now that we have the principle in mind, let us work on more complex examples, in particular with an example that involves a loop.

```
#include <stddef.h>

int* search(int* array, size_t length, int element){
  for(size_t i = 0; i < length; i++)
    if(array[i] == element) return &array[i];
  return NULL;
}
```

Once we have generated the assertions related to runtime errors, we have the following program:

```c
/* Generated by Frama-C */
#include <stddef.h>
int *search(int *array, size_t length, int element)
{
  int *__retres;
  {
    size_t i = (unsigned int)0;
    while (i < length) {
      /*@ assert rte: mem_access: \valid_read(array + i); */
      if (*(array + i) == element) {
        __retres = array + i;
        goto return_label;
      }
      i += (size_t)1;
    }
  }
  __retres = (int *)0;
  return_label: return __retres;
}
```

We have to prove that any cell visited by the program can be read, thus we need to express as a precondition that the array is `\valid_read` on the corresponding range of indices. However, this is not enough to complete the proof since we have a loop in this program, so we have to provide a suitable invariant. We also probably want to prove that the loop terminates.

Thus, we get the following minimally specified function:

```c
#include <stddef.h>

/*@
  requires \valid_read(array + (0 .. length-1)) ;
*/
int* search(int* array, size_t length, int element){
  /*@
    loop invariant 0 <= i <= length ;
    loop assigns i ;
    loop variant length - i ;
  */
  for(size_t i = 0; i < length; i++)
    if(array[i] == element) return &array[i];
  return NULL;
}
```

This contract can be compared with the contract provided for the search function in section 4.3.2, and we can see that it is much more simple.

Now let us imagine that the function is used in the following program:

```c
void foo(int* array, size_t length){
  int* p = search(array, length, 0) ;
  if(p){
    *p += 1 ;
  }
}
```

We again have to provide a suitable contract for the function, again by having a look at the assertion that RTE asks us to verify:

```
void foo(int *array, size_t length)
{
  int *p = search(array,length,0);
  if (p)
    /*@ assert rte: mem_access: \valid(p); */
    /*@ assert rte: mem_access: \valid_read(p); */
    /*@ assert rte: signed_overflow: *p + 1 ≤ 2147483647; */
    (*p) ++;
  return;
}
```

Thus, we have to verify that:

- the pointer we received from `search` is valid,
- `*p + 1` does not overflow,
- we respect the contract of the `search` function.

In addition to the contract of `foo`, we have to provide some more information in the contract of `search`. Indeed, we will not be able to prove that the pointer is valid when it is not null if the function does not guarantee that the pointer is in the range of our array in this case. Furthermore, we will not be able to prove that `*p` is less than `INT_MAX` if the function can modify it.

This leads us to this complete annotated program:

```
#include <stddef.h>
#include <limits.h>

/*@
  requires \valid_read(array + (0 .. length-1)) ;
  assigns \nothing ;
  ensures \result == NULL ||
          (\exists integer i ; 0 <= i < length && array+i == \result) ;
*/
int* search(int* array, size_t length, int element){
  /*@
    loop invariant 0 <= i <= length ;
    loop assigns i ;
    loop variant length - i ;
  */
  for(size_t i = 0; i < length; i++)
    if(array[i] == element) return &array[i];
  return NULL;
}

/*@
  requires \forall integer i ; 0 <= i < length ==> array[i] < INT_MAX ;
  requires \valid(array + (0 .. length-1)) ;
*/
void foo(int *array, size_t length){
  int *p = search(array,length,0);
  if (p){
    *p += 1 ;
  }
}
```

7.1.3. Advantages and limitations

The evident advantage of this approach is the fact that it can guarantee the absence of runtime errors in any function of a module or a program in (relative) isolation of the other functions. Furthermore, this absence of runtime errors is guaranteed for any use of the function as long as the precondition is verified when it is called. That allows to gain some confidence into a system with a relatively low-cost approach.

However, as we have seen, when we use a function it can change the knowledge we need about its behavior, requiring to make the contract richer and richer. Thus, we can progressively reach a state where we basically proved the functional correctness of the function.

Furthermore, even proving the absence of runtime errors is sometimes not trivial as we have for example seen with functions like the factorial or the sum of N integers, that require to give quite a lot of information to SMT solvers, in order to prove that we cannot meet an integer overflow.

Finally, sometimes the minimal contracts of a function or a module basically is the full functional specification, and thus formally verifying the absence of runtime errors requires a full functional verification. This is commonly the case when we have to deal with complex data structures where the properties that are required for the absence of runtime errors depend on the functional behavior of the function, maintaining some non-trivial invariant about the data structure.

7.1.4. Exercises

7.1.4.1. Some simple examples

Prove the absence of runtime errors in the following program using a minimal contracts approach:

```c
void max_ptr(int* a, int* b){
  if(*a < *b){
    int tmp = *b ;
    *b = *a ;
    *a = tmp ;
  }
}

void min_ptr(int* a, int* b){
  max_ptr(b, a);
}

void order_3_inc_min(int* a, int* b, int* c){
  min_ptr(a, b) ;
  min_ptr(a, c) ;
  min_ptr(b, c) ;
}

void incr_a_by_b(int* a, int const* b){
  *a += *b;
}
```

7.1.4.2. Reverse

Prove the absence of runtime errors for the following `reverse` function and its dependency using a minimal contracts approach. Note that the `swap` function should also be specified with minimal contracts only. Do not forget to add the options `-warn-unsigned-overflow` and `-warn-unsigned-downcast`.

```c
#include <stddef.h>

void swap(int* a, int* b){
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

void reverse(int* array, size_t len){
  for(size_t i = 0 ; i < len/2 ; ++i){
    swap(array+i, array+len-i-1) ;
  }
}
```

7.1.4.3. Binary search

Prove the absence of runtime errors for the following `bsearch` function using a minimal contracts approach. Do not forget to add the options `-warn-unsigned-overflow` and `-warn-unsigned-downcast`.

```c
#include <stddef.h>

size_t bsearch(int* arr, size_t len, int value){
  if(len == 0) return len ;

  size_t low = 0 ;
  size_t up = len ;

  while(low < up){
    size_t mid = low + (up - low)/2 ;
    if      (arr[mid] > value) up = mid ;
    else if(arr[mid] < value) low = mid+1 ;
    else return mid ;
  }
  return len ;
}
```

7.1.4.4. Sort

Prove the absence of runtime errors for the following `sort` function and its dependencies using a minimal contracts approach. Note that these dependencies should also be specified with minimal contracts only. Do not forget to add the options `-warn-unsigned-overflow` and `-warn-unsigned-downcast`.

```c
#include <stddef.h>

size_t min_idx_in(int* a, size_t beg, size_t end){
  size_t min_i = beg;
  for(size_t i = beg+1; i < end; ++i){
    if(a[i] < a[min_i]) min_i = i;
  }
  return min_i;
}

void swap(int* p, int* q){
  int tmp = *p; *p = *q; *q = tmp;
}

void sort(int* a, size_t beg, size_t end){
  for(size_t i = beg ; i < end ; ++i){
    size_t imin = min_idx_in(a, i, end);
    swap(&a[i], &a[imin]);
  }
}
```

7.2. Guiding assertions and triggering of lemmas

There are different levels of automation for the verification of programs. From tools that are completely automatic, like for example abstract interpreters that do not require any help from the human (or at least not so much), to interactive tools, like proof assistants where we write the proof mostly by hand and the tools is just there to check that we do it right.

Tools like WP (any many others like Why3, Spark, ...) tend to maximize automation. However, the more the properties we want to prove are complex, the harder it will be to get automatically the proof. Thus, we often need to help the tools in order to achieve the verification. This is done by providing more annotations to help the verification condition generation process. Adding a loop invariant is for example a way to be able to produce an inductive reasoning about a loop while automatic provers are generally bad at this kind of task.

This kind of technique has been called "auto-active" verification. This word is the contraction of "automatic" and "interactive". It is automatic in the sense that most of the proof is performed by automatic tools, but it is also somewhat interactive since as users, we manually provide information to the tools.

In this section, we will see in more details how we can use assertions to guide the proofs. By adding assertions, we create some base of knowledge (properties that are known to be true) that are collected by the verification condition generator during the WP computation process and that are then given to the automatic solvers that consequently have more information and thus can potentially prove more complex properties.

7.2.1. Proof context

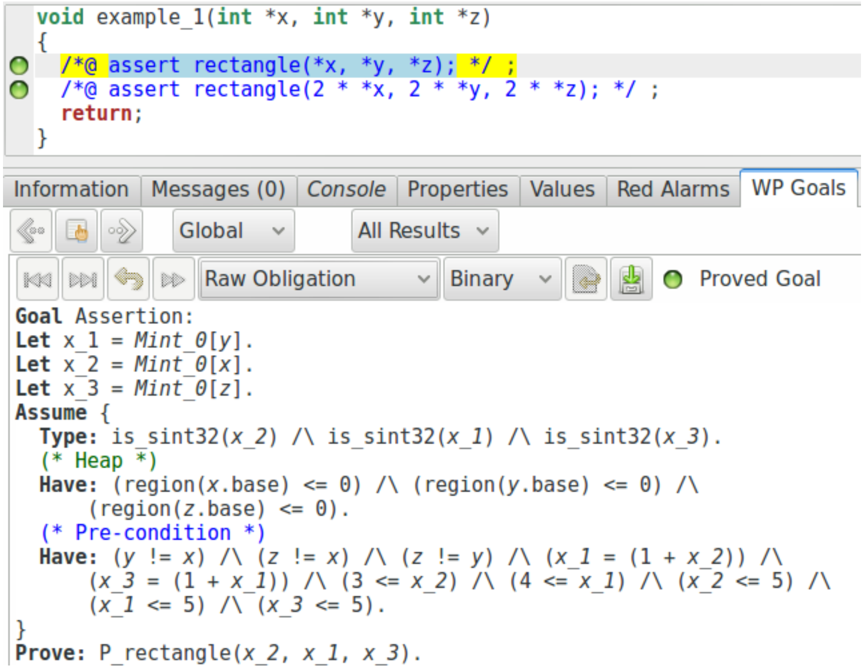
In order to understand what is exactly the benefit of adding assertions in annotations of a program, let us first have a closer look at the verification condition generated by WP from the annotated source code and how assertions are taken into account. For this, we consider the following predicate (one can recognize Pythagoras' theorem):

```
/*@
  predicate rectangle{L}(integer c1, integer c2, integer h) =
    c1 * c1 + c2 * c2 == h * h ;
*/
```

Let us first consider this example:

```
/*@
  requires \separated(x, y , z);
  requires 3 <= *x <= 5 ;
  requires 4 <= *y <= 5 ;
  requires *z <= 5 ;
  requires *x+2 == *y+1 == *z ;
*/
void example_1(int* x, int* y, int* z){
  //@ assert rectangle(*x, *y, *z);
  //@ assert rectangle(2* (*x), 2* (*y), 2* (*z));
}
```

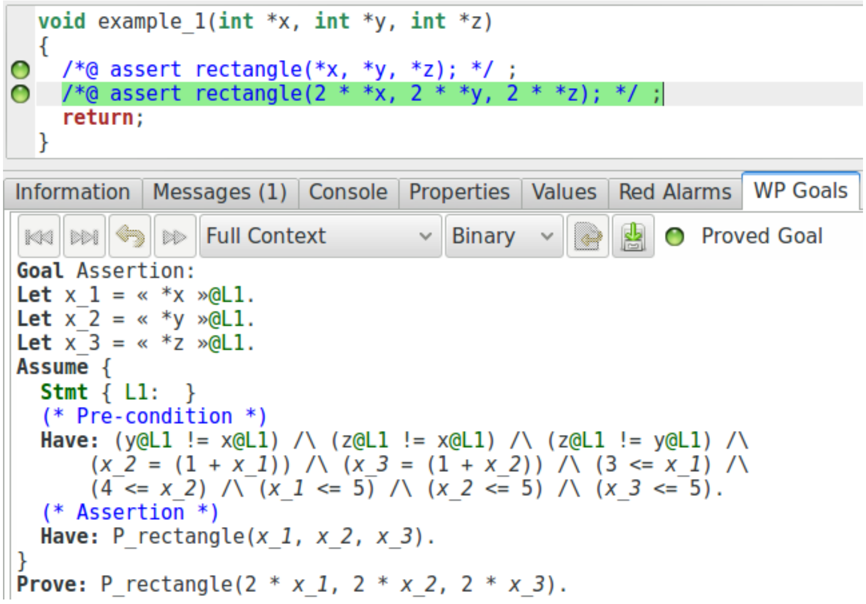
Here, we have specified a precondition that is complex enough so that WP cannot directly guess the values in input of the function. In fact, the values are exactly: `*x == 3`, `*y == 4` and `*z == 5`. Now, if we have a look at the verification condition generated for our first assertion, we can see this (be sure to select the view "Full Context" or "Raw Obligation" - they are not exactly the same but quite similar, the former is just slightly pretty printed):



```
void example_1(int *x, int *y, int *z)
{
  /*@ assert rectangle(*x, *y, *z); */ ;
  /*@ assert rectangle(2 * *x, 2 * *y, 2 * *z); */ ;
  return;
}
```

```
Goal Assertion:
Let x_1 = Mint_0[y].
Let x_2 = Mint_0[x].
Let x_3 = Mint_0[z].
Assume {
  Type: is_sint32(x_2) /\ is_sint32(x_1) /\ is_sint32(x_3).
  (* Heap *)
  Have: (region(x.base) <= 0) /\ (region(y.base) <= 0) /\
      (region(z.base) <= 0).
  (* Pre-condition *)
  Have: (y != x) /\ (z != x) /\ (z != y) /\ (x_1 = (1 + x_2)) /\
      (x_3 = (1 + x_1)) /\ (3 <= x_2) /\ (4 <= x_1) /\ (x_2 <= 5) /\
      (x_1 <= 5) /\ (x_3 <= 5).
}
Prove: P_rectangle(x_2, x_1, x_3).
```

That is to say the different constraints we have stated as a precondition to the function (note that the values are not exactly the same, and some more properties have been specified). What is more interesting is having a look at the verification condition generated for the second assertion (note that we have edited all the remaining screenshots in this section in order to focus on what is important, the other properties can just be ignored in our case):



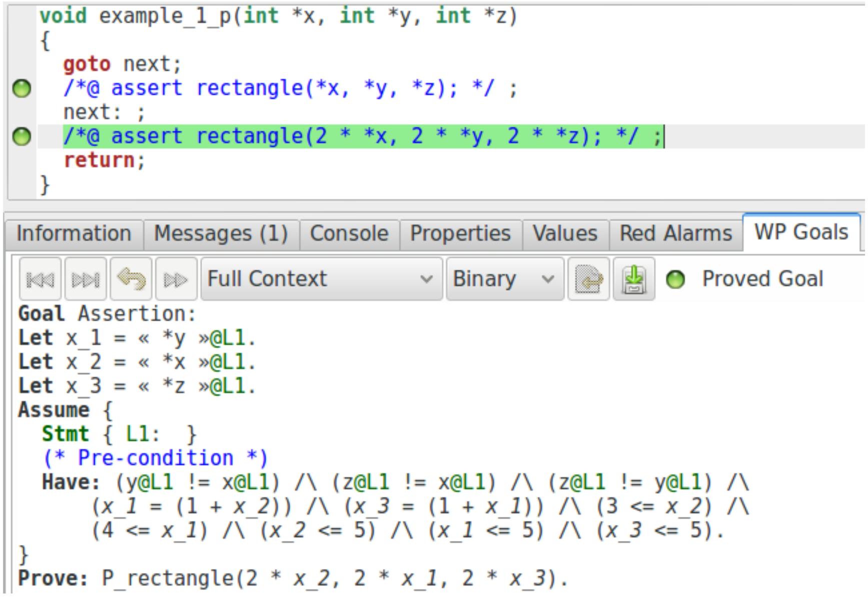
```
void example_1(int *x, int *y, int *z)
{
  /*@ assert rectangle(*x, *y, *z); */ ;
  /*@ assert rectangle(2 * *x, 2 * *y, 2 * *z); */ ;
  return;
}
```

```
Goal Assertion:
Let x_1 = « *x »@L1.
Let x_2 = « *y »@L1.
Let x_3 = « *z »@L1.
Assume {
  Stmt { L1:  }
  (* Pre-condition *)
  Have: (y@L1 != x@L1) /\ (z@L1 != x@L1) /\ (z@L1 != y@L1) /\
      (x_2 = (1 + x_1)) /\ (x_3 = (1 + x_2)) /\ (3 <= x_1) /\
      (4 <= x_2) /\ (x_1 <= 5) /\ (x_2 <= 5) /\ (x_3 <= 5).
  (* Assertion *)
  Have: P_rectangle(x_1, x_2, x_3).
}
Prove: P_rectangle(2 * x_1, 2 * x_2, 2 * x_3).
```

Here, we can see that for the proof of the second assertion, WP has collected and added the first assertion to the assumptions. Thus, WP considers that SMT solvers can assume this assertion. That means that they can rely on it, but also that it should be proved to be sure that the current verification condition is verified.

Note that WP only collects what we can find on the different paths that allows to reach the assertion. For example, if we modify the code such that the path to the assertion jumps over the first assertion, it does not appear in the verification condition.

```
void example_1_p(int* x, int* y, int* z){
  goto next;
  //@ assert rectangle(*x, *y, *z);
 next: ;
  //@ assert rectangle(2* (*x), 2* (*y), 2* (*z));
}
```

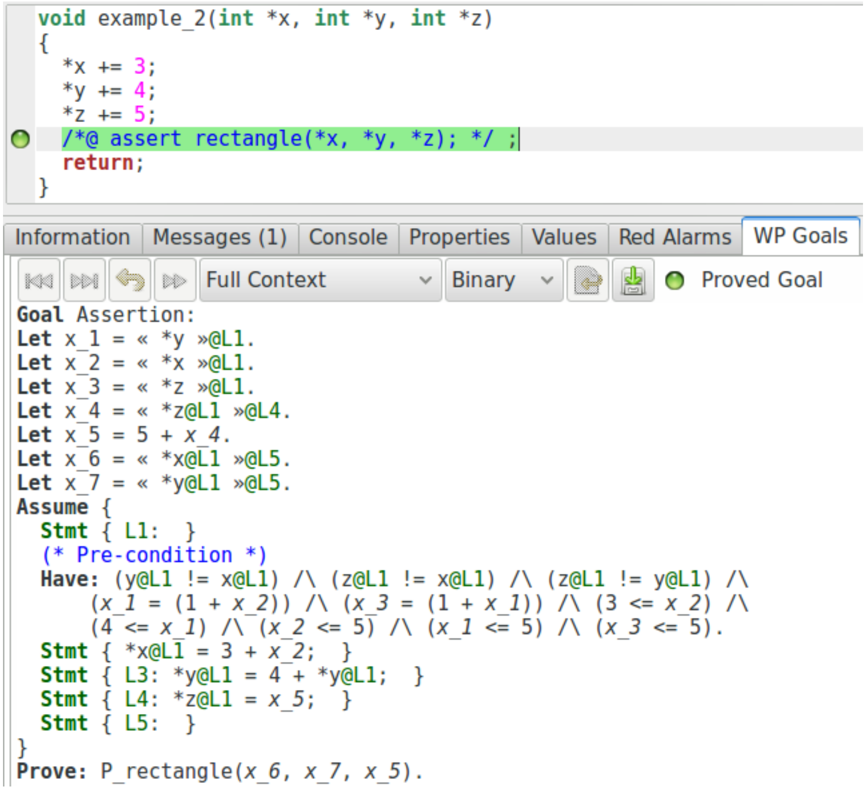


```
void example_1_p(int *x, int *y, int *z)
{
  goto next;
  /*@ assert rectangle(*x, *y, *z); */ ;
  next: ;
  /*@ assert rectangle(2 * *x, 2 * *y, 2 * *z); */ ;
  return;
}
```

```
Goal Assertion:
Let x_1 = « *y »@L1.
Let x_2 = « *x »@L1.
Let x_3 = « *z »@L1.
Assume {
  Stmt { L1:  }
  (* Pre-condition *)
  Have: (y@L1 != x@L1) /\ (z@L1 != x@L1) /\ (z@L1 != y@L1) /\
      (x_1 = (1 + x_2)) /\ (x_3 = (1 + x_1)) /\ (3 <= x_2) /\
      (4 <= x_1) /\ (x_2 <= 5) /\ (x_1 <= 5) /\ (x_3 <= 5).
}
Prove: P_rectangle(2 * x_2, 2 * x_1, 2 * x_3).
```

Now let us modify the example a bit in order to illustrate how assertions can change the way to prove a program. For example, let us first modify the different memory locations (doubling each value) and check that the resulting triangle is a right triangle.

```
/*@
  requires \separated(x, y , z);
  requires 3 <= *x <= 5 ;
  requires 4 <= *y <= 5 ;
  requires *z <= 5 ;
  requires *x+2 == *y+1 == *z ;
*/
void example_2(int* x, int* y, int* z){
  *x += 3 ;
  *y += 4 ;
  *z += 5 ;

  //@ assert rectangle(*x, *y, *z);
}
```



```
void example_2(int *x, int *y, int *z)
{
  *x += 3;
  *y += 4;
  *z += 5;
  /*@ assert rectangle(*x, *y, *z); */ ;
  return;
}
```

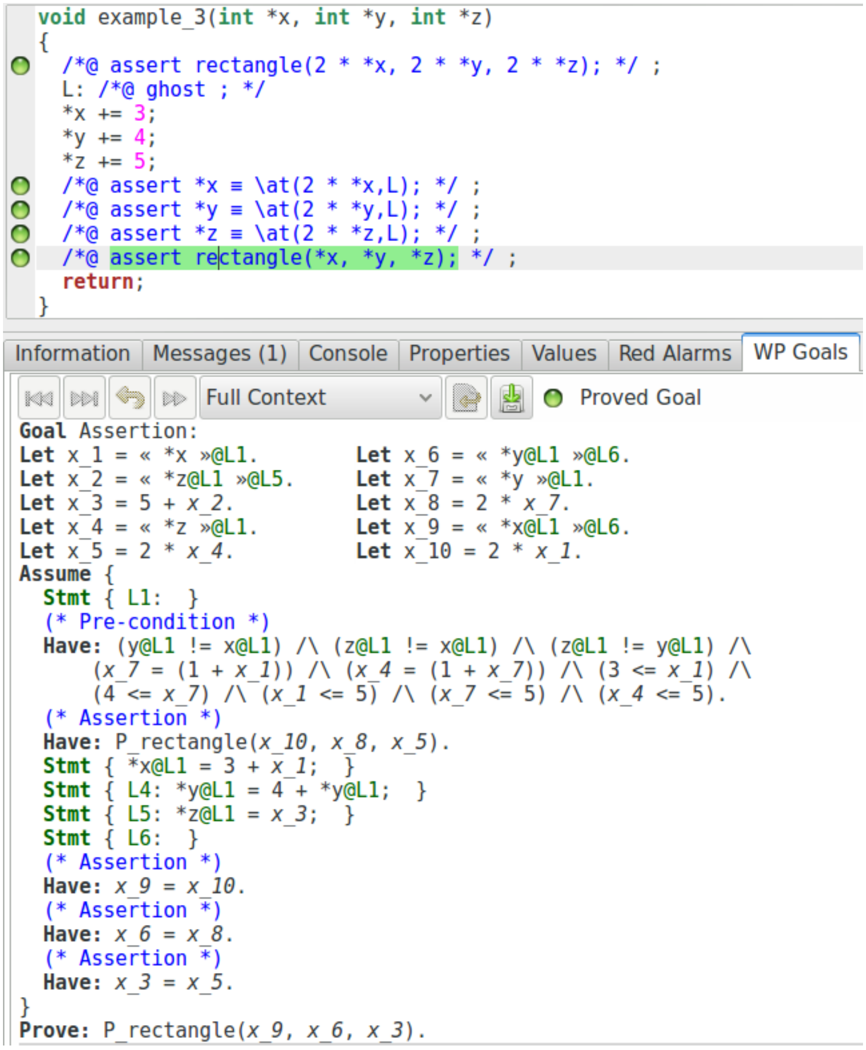
```
Goal Assertion:
Let x_1 = « *y »@L1.
Let x_2 = « *x »@L1.
Let x_3 = « *z »@L1.
Let x_4 = « *z@L1 »@L4.
Let x_5 = 5 + x_4.
Let x_6 = « *x@L1 »@L5.
Let x_7 = « *y@L1 »@L5.
Assume {
  Stmt { L1:  }
  (* Pre-condition *)
  Have: (y@L1 != x@L1) /\ (z@L1 != x@L1) /\ (z@L1 != y@L1) /\
      (x_1 = (1 + x_2)) /\ (x_3 = (1 + x_1)) /\ (3 <= x_2) /\
      (4 <= x_1) /\ (x_2 <= 5) /\ (x_1 <= 5) /\ (x_3 <= 5).
  Stmt { *x@L1 = 3 + x_2;  }
  Stmt { L3: *y@L1 = 4 + *y@L1;  }
  Stmt { L4: *z@L1 = x_5;  }
  Stmt { L5:  }
}
Prove: P_rectangle(x_6, x_7, x_5).
```

Here, the solver will likely unfold the predicate and directly check that the property is true, indeed from the information we have in this verification condition, we do not really have any other knowledge that would allow us to produce this proof in another way. Now, let us bring new information in annotation:

```
/*@
  requires \separated(x, y , z);
  requires 3 <= *x <= 5 ;
  requires 4 <= *y <= 5 ;
  requires *z <= 5 ;
  requires *x+2 == *y+1 == *z ;
```

```
*/
void example_3(int* x, int* y, int* z){
  //@ assert rectangle(2* (*x), 2* (*y), 2* (*z));
  //@ ghost L: ;

  *x += 3 ;
  *y += 4 ;
  *z += 5 ;

  //@ assert *x == \at(2* (*x), L) ;
  //@ assert *y == \at(2* (*y), L) ;
  //@ assert *z == \at(2* (*z), L);
  //@ assert rectangle(*x, *y, *z);
}
```

We first prove that if we multiply by 2 each of the values, the predicate is true for the new values. The solver in fact basically solves the same problem at first, but this is not what we want to point out now. Then, we re-express the values we have modified in another way: we show that all of them have been multiplied by 2. Now, we can have a new look at the generated verification condition for the last assertion:



```
void example_3(int *x, int *y, int *z)
{
  /*@ assert rectangle(2 * *x, 2 * *y, 2 * *z); */ ;
  L: /*@ ghost ; */
  *x += 3;
  *y += 4;
  *z += 5;
  /*@ assert *x ≡ \at(2 * *x,L); */ ;
  /*@ assert *y ≡ \at(2 * *y,L); */ ;
  /*@ assert *z ≡ \at(2 * *z,L); */ ;
  /*@ assert rectangle(*x, *y, *z); */ ;
  return;
}
```

```
Goal Assertion:
Let x_1 = « *x »@L1.           Let x_6 = « *y@L1 »@L6.
Let x_2 = « *z@L1 »@L5.        Let x_7 = « *y »@L1.
Let x_3 = 5 + x_2.             Let x_8 = 2 * x_7.
Let x_4 = « *z »@L1.           Let x_9 = « *x@L1 »@L6.
Let x_5 = 2 * x_4.             Let x_10 = 2 * x_1.
Assume {
  Stmt { L1:  }
  (* Pre-condition *)
  Have: (y@L1 != x@L1) /\ (z@L1 != x@L1) /\ (z@L1 != y@L1) /\
      (x_7 = (1 + x_1)) /\ (x_4 = (1 + x_7)) /\ (3 <= x_1) /\
      (4 <= x_7) /\ (x_1 <= 5) /\ (x_7 <= 5) /\ (x_4 <= 5).
  (* Assertion *)
  Have: P_rectangle(x_10, x_8, x_5).
  Stmt { *x@L1 = 3 + x_1;  }
  Stmt { L4: *y@L1 = 4 + *y@L1;  }
  Stmt { L5: *z@L1 = x_3;  }
  Stmt { L6:  }
  (* Assertion *)
  Have: x_9 = x_10.
  (* Assertion *)
  Have: x_6 = x_8.
  (* Assertion *)
  Have: x_3 = x_5.
}
Prove: P_rectangle(x_9, x_6, x_3).
```

While we have to prove exactly the same property as before (with a bit of renaming), we can see that we have here another way to prove it. Indeed, by just combining this set of properties:

```
(* Assertion *)
Have: P_rectangle(x_10, x_8, x_5).
(* Assertion *)
Have: x_9 = x_10.
(* Assertion *)
Have: x_6 = x_8.
(* Assertion *)
Have: x_3 = x_5.
```

It is easy to deduce:

```
Prove: P_rectangle(x_9, x_6, x_3).
```

By just replacing the values `x_9`, `x_6` and `x_3`. So the solver could use this to avoid unfolding the predicate. However, it will not necessarily do it: SMT solvers are based on heuristic methods, so we can just provide them properties and hope that they will use it.

Here, the property is simple to prove, so it was not really necessary to add assertions (and make more effort). In other cases, as we will see now, we have to give the right information so that they will find what they need to finish the proof.

7.2.2. Triggering lemmas

We often use assertions to express properties that precisely correspond to one or all of the premises of a lemma or to its conclusion. By doing this, we maximize the chances that the SMT solver "recognize" that what we have written corresponds to a particular lemma and that it should use it.

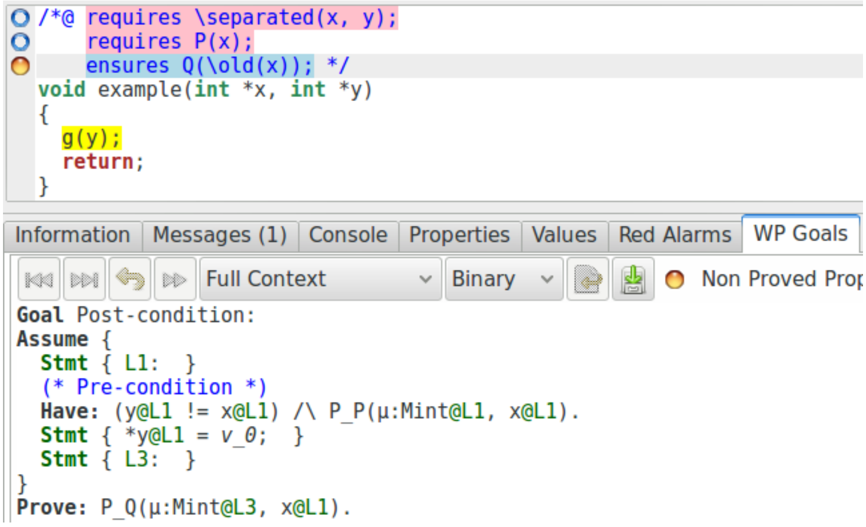
Let us illustrate this use with the following example. We use axioms and not lemmas as they are used exactly the same way by WP. First consider the following axiomatic definition. We define two predicates `P` and `Q` about a particular memory location `x`. We have two axioms: `ax_1` that states that if `P(x)` is true, then `Q(x)` is true, and a second axiom `ax_2` that state that if the pointed location does not change between two labels (modeled by the predicate `eq`) and `P(x)` holds for one of them, then it holds for the other one.

```
/*@
  predicate eq{L1, L2}(int* x) =
    \at(*x, L1) == \at(*x, L2) ;
*/

/*@
  axiomatic Ax {
    predicate P(int* x) reads *x ;
    predicate Q(int* x) reads *x ;

    axiom ax_1: \forall int* x ; P(x) ==> Q(x);
    axiom ax_2{L1, L2}:
      \forall int* x ; eq{L1, L2}(x) ==> P{L1}(x) ==> P{L2}(x);
  }
*/
```

And we want to prove the following program:

```
/*@ assigns *x ; */
void g(int* x);

/*@
  requires \separated(x, y);
  requires P(x) ;
  ensures  Q(x) ;
*/
void example(int* x, int* y){
  g(y);
}
```

However, we can see here that the proof fails on the following verification condition (again we have removed what is not useful for our explanations):

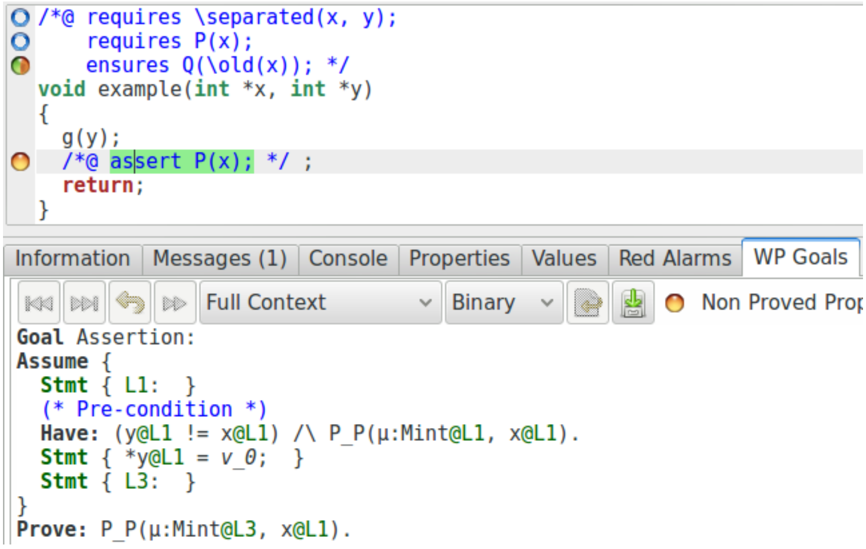


```
/*@ requires \separated(x, y);
    requires P(x);
    ensures Q(\old(x)); */
void example(int *x, int *y)
{
  g(y);
  return;
}
```

```
Goal Post-condition:
Assume {
  Stmt { L1:  }
  (* Pre-condition *)
  Have: (y@L1 != x@L1) /\ P_P(µ:Mint@L1, x@L1).
  Stmt { *y@L1 = v_0;  }
  Stmt { L3:  }
}
Prove: P_Q(µ:Mint@L3, x@L1).
```

Thus, the SMT solver seems to be unable to use one of the two axioms of our definition: either it cannot show that after the call to `g(y)`, `P(x)` is still true, or it can, and thus it cannot show that it implies that `Q(x)` is true. Let us try to add an assertion so that we verify that we can prove `P(x)` after the call:

```
/*@
  requires \separated(x, y);
  requires P(x) ;
  ensures  Q(x) ;
*/
void example(int* x, int* y){
  g(y);
  //@ assert P(x);
}
```



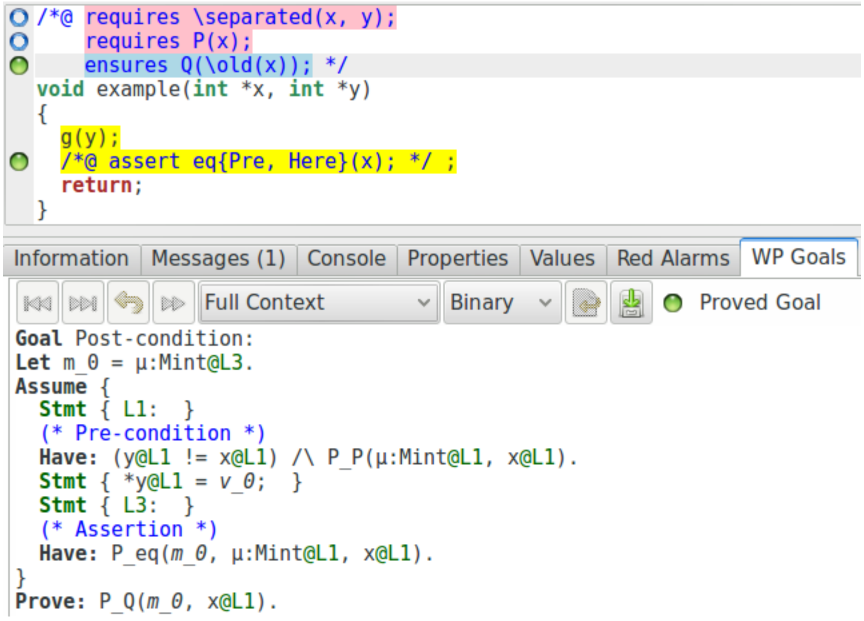
```
/*@ requires \separated(x, y);
    requires P(x);
    ensures Q(\old(x)); */
void example(int *x, int *y)
{
  g(y);
  /*@ assert P(x); */ ;
  return;
}
```

```
Goal Assertion:
Assume {
  Stmt { L1:  }
  (* Pre-condition *)
  Have: (y@L1 != x@L1) /\ P_P(µ:Mint@L1, x@L1).
  Stmt { *y@L1 = v_0;  }
  Stmt { L3:  }
}
Prove: P_P(µ:Mint@L3, x@L1).
```

It seems that despite the fact that it is clear that `*x` did not change during the call `g(y)`, and thus that `eq{Pre, Here}(x)` holds after the call, since this property is not directly provided in this verification condition, the SMT solver does not use the corresponding axiom `ax_2`. Thus, let us provide this information to the solver:

```
/*@
  requires \separated(x, y);
  requires P(x) ;
  ensures  Q(x) ;
*/
void example(int* x, int* y){
  g(y);
  //@ assert eq{Pre, Here}(x);
}
```

Now, everything is proved as, if we have a look at the verification condition, we can see that this important information is provided so that the SMT solver can use it:



```
/*@ requires \separated(x, y);
    requires P(x);
    ensures Q(\old(x)); */
void example(int *x, int *y)
{
  g(y);
  /*@ assert eq{Pre, Here}(x); */ ;
  return;
}
```

```
Goal Post-condition:
Let m_0 = µ:Mint@L3.
Assume {
  Stmt { L1:  }
  (* Pre-condition *)
  Have: (y@L1 != x@L1) /\ P_P(µ:Mint@L1, x@L1).
  Stmt { *y@L1 = v_0;  }
  Stmt { L3:  }
  (* Assertion *)
  Have: P_eq(m_0, µ:Mint@L1, x@L1).
}
Prove: P_Q(m_0, x@L1).
```

7.2.3. A more complex example: sort, again

Let us now consider a more complex example that involves some actual axiomatic definitions. This time, we will prove the insertion sort function:

```
#include <stddef.h>
#include <limits.h>

void insert(int* a, size_t beg, size_t last){
  size_t i = last ;
  int value = a[i] ;

  while(i > beg && a[i - 1] > value){
    a[i] = a[i - 1] ;
    --i ;
  }
  a[i] = value ;
}

void insertion_sort(int* a, size_t beg, size_t end){
  for(size_t i = beg+1; i < end; ++i)
    insert(a, beg, i);
}
```

The `insertion_sort` function visits each value, from the beginning of the array, to the end. For each value v, it is inserted (using the `insert` function) at the right place in the range of the already sorted values (at the beginning of the array), by shifting them until it meets a value that is smaller than v or the first cell of the array.

We want to prove the same postcondition as we already proved for the selection sort, that is: we want to create a sorted permutation of the original values. Again, each iteration of the loop must ensure that the new configuration is a permutation of the original values, and that the range from the beginning to the current visited cell is sorted. All these properties are ensured by the `insert` function. If we give a closer look at this function, we can see that it records the value to insert (which is at the end of the range) in the variable `value` and starting from this last position it shifts all value until it meets a value that is smaller than the one we want to insert or the first cell of the array, and finally inserts the value. For this proof, we enable the options `-warn-unsigned-overflow` and `-warn-unsigned-downcast`.

First, let us provide a suitable contract and loop invariant for the insertion sort function. The contract is equivalent to the one provided for the selection sort. Note however that the invariant is weaker: we do not need the values that are still not visited to be greater than the visited ones: we insert each value at the right place.

```
/*@
  requires beg < end && \valid(a + (beg .. end-1));
  assigns a[beg .. end-1];
  ensures sorted(a, beg, end);
  ensures permutation{Pre, Post}(a,beg,end);
*/
void insertion_sort(int* a, size_t beg, size_t end){
  /*@
    loop invariant beg+1 <= i <= end ;
    loop invariant sorted(a, beg, i) ;
    loop invariant permutation{Pre, Here}(a,beg,end);
    loop assigns a[beg .. end-1], i ;
    loop variant end-i ;
  */
  for(size_t i = beg+1; i < end; ++i) {
    insert(a, beg, i);
  }
}
```

Now, we can provide a contract for the insert function. The function requires that the first part of the range is already sorted from the beginning to the penultimate, and in exchange, it guarantees that the complete final range is sorted and is a permutation of the original one:

```
/*@
  requires beg < last < SIZE_MAX && \valid(a + (beg .. last));
  requires sorted(a, beg, last) ;

  assigns a[ beg .. last ] ;

  ensures permutation{Pre, Post}(a, beg, last+1);
  ensures sorted(a, beg, last+1) ;
*/
void insert(int* a, size_t beg, size_t last){
  size_t i = last ;
  int value = a[i] ;

  while(i > beg && a[i - 1] > value){
    a[i] = a[i - 1] ;
    --i ;
  }
  a[i] = value ;
}
```

Then, we need to provide a suitable invariant to the loop of the `insert` function. And this time, we can see that with our previously defined permutation predicate, we are in trouble. Indeed, our inductive definition of the permutation specifies three cases: a range is permutation of itself, or two (and only two) values have been swapped, or the permutation is transitive. But none of these cases can be applied to our `insert` function, since the shifted range is not obtained by successively exchanging values, and that the other cases obviously do not apply here.

Thus, we need to find a better definition to the notion of permutation. We can notice that what we really need to provide is a way to say “each value that was previously in the array is still in the array and if several values were equivalent, the number of occurrences of these values does not change”. And in fact, this last part of the expression is enough to express our permutation. A permutation of a range is a range such that for all value, the number of occurrences of this value in the array does not change from a program point to another:

```
/*@
  predicate permutation{L1, L2}(int* in, integer from, integer to) =
    \forall int v ; l_occurrences_of{L1}(v, in, from, to) ==
                    l_occurrences_of{L2}(v, in, from, to) ;
*/
```

Starting from this definition, we are able to provide lemmas that allow to reason efficiently about permutation, provided that some properties hold about the array between two program points. For example, it would be possible to define the case `Swap` in our previous inductive definition using a lemma. And it is of course also possible for our shifted range.

Let us determine what are the required lemmas by first considering the `insert_sort` function. The only property that is not proved is the invariant of the loop that expresses the fact that the array is a permutation of the original array. How can we deduce it? (We will consider the proofs of the lemmas later).

We can observe two facts: the first range in the array (from `beg` to `i+1`) is a permutation of the same range at the beginning of the iteration (by the contract of the `insert` function). The second part (from `i+1` to `end`) is unchanged, thus this is also a permutation. Let us use some assertions to see whether we can prove this or not. While the first property is easily deduced, we can see that the second one is not:

```
  /*@
    loop invariant beg+1 <= i <= end ;
    loop invariant sorted(a, beg, i) ;
    loop invariant permutation{Pre, Here}(a,beg,end);
    loop assigns a[beg .. end-1], i ;
    loop variant end-i ;
  */
  for(size_t i = beg+1; i < end; ++i) {
    //@ ghost L:
    insert(a, beg, i);
    //@ assert permutation{L, Here}(a, beg, i+1); // PROVED
    //@ assert permutation{L, Here}(a, i+1, end); // NOT PROVED
  }
```

So we need a first lemma for this property. Let us define two predicates `shifted` and `unchanged`, the latter being used to define the former (we will see why later) and express that an unchanged range is a permutation.

```
/*@
  predicate shifted{L1, L2}(integer s, int* a, integer beg, integer end) =
    \forall integer k ; beg <= k < end ==> \at(a[k], L1) == \at(a[s+k], L2) ;
*/
/*@
  predicate unchanged{L1, L2}(int* a, integer beg, integer end) =
    shifted{L1, L2}(0, a, beg, end);
*/
```

```
/*@ lemma unchanged_is_permutation{L1, L2}:
  \forall int* a, integer beg, integer end ;
    unchanged{L1, L2}(a, beg, end) ==> permutation{L1, L2}(a, beg, end) ;
*/
```

And now, we can verify that those two sub-arrays are permutations, this is done by adding an assertion that shows that the range `i+1` to `end` is unchanged, in order to trigger our lemma `unchanged_is_permutation`.

```
  /*@
    loop invariant beg+1 <= i <= end ;
    loop invariant sorted(a, beg, i) ;
    loop invariant permutation{Pre, Here}(a,beg,end);
    loop assigns a[beg .. end-1], i ;
    loop variant end-i ;
  */
  for(size_t i = beg+1; i < end; ++i) {
    //@ ghost L: ;
    insert(a, beg, i);
    //@ assert permutation{L, Here}(a, beg, i+1);
    //@ assert unchanged{L, Here}(a, i+1, end) ;
    //@ assert permutation{L, Here}(a, i+1, end) ;
  }
```

Thus, since those two parts are permutations, the global array is a permutation of the values present at the beginning of the iteration. However, this is not directly proved, so we also need a lemma for this:

```
/*@ lemma union_permutation{L1, L2}:
  \forall int* a, integer beg, split, end, int v ;
    beg <= split <= end ==>
    permutation{L1, L2}(a, beg, split) ==>
    permutation{L1, L2}(a, split, end) ==>
      permutation{L1, L2}(a, beg, end) ;
*/
```

And now we can deduce that a loop iteration produces a permutation by adding this conclusion as an assertion:

```
    //@ ghost L: ;
    insert(a, beg, i);
    //@ assert permutation{L, Here}(a, beg, i+1);
    //@ assert unchanged{L, Here}(a, i+1, end);
    //@ assert permutation{L, Here}(a, i+1, end);
    //@ assert permutation{L, Here}(a, beg, end); // PROVED
```

Finally, we need to add one more information, the permutation of a permutation is also a permutation. This time, we do not have to add more assertions, the context contains:

- `permutation{Pre, L}(a, beg, end)` (invariant)
- `permutation{L, Here}(a, beg, end)` (assertion)

which is enough to conclude `permutation{Pre, Here}(a, beg, end)` at the end of the loop block using the following lemma:

```
/*@ lemma transitive_permutation{L1, L2, L3}:
  \forall int* a, integer beg, integer end ;
    permutation{L1, L2}(a, beg, end) ==>
    permutation{L2, L3}(a, beg, end) ==>
      permutation{L1, L3}(a, beg, end) ;
*/
```

Now, we can have a closer look at the insertion function, by first considering how to maintain that the function produces a permutation.

It shifts the different elements to the left until it reaches the beginning of the array or an element that is smaller than the element to insert which is initially at the end of the range, and is inserted at the reached position. The cells from the beginning of the array to the location of insertion are left unchanged, so this is a permutation. We have a lemma for this, but we also have to state that the values are unchanged as an invariant of the loop. The second part of the array is a permutation because we rotate the elements, we need a lemma to express this, and to state at least that the element are shifted by the loop as an invariant. Finally, the union of the two permutations is a permutation, and we also have a lemma for this.

So first, we can give a suitable invariant for the permutation:

- we provide the bounds of `i`
- we state that the first part is left unchanged
- we state that the last part is shifted to the left

as well as some assertions that we want to be verified:

- first in order to trigger `unchanged_permutation`, we place a first assertion that state that the first part of the array is unchanged, which allows to prove
- the second one that asserts that the first part of the array is a permutation of the original one, and is used in combination with ...
- the third one that asserts that the second part of the array is a permutation of the original one (which allows triggering `union_permutation` and prove the postcondition).

```
  /*@
    loop invariant beg <= i <= last ;
    loop invariant \forall integer k ; beg <= k <= i    ==> a[k] == \at(a[k], Pre) ;
    loop invariant \forall integer k ; i+1 <= k <= last ==> a[k] == \at(a[k-1], Pre) ;

    loop assigns i, a[beg .. last] ;
    loop variant i ;
  */
  while(i > beg && a[i - 1] > value){
    a[i] = a[i - 1] ;
    --i ;
  }

  a[i] = value ;

  //@ assert unchanged{Pre, Here}(a, beg, i) ;   // PROVED
  //@ assert permutation{Pre, Here}(a, beg, i) ; // PROVED

  //@ assert rotate_left{Pre, Here}(a, i, last+1) ; //PROVED
  //@ assert permutation{Pre, Here}(a, i, last+1) ; // NOT PROVED
```

Then, for the last assertion, we need a lemma about the rotation of the elements:

```
/*@
  predicate rotate_left{L1, L2}(int* a, integer beg, integer end) =
    beg < end && \at(a[beg], L2) == \at(a[end-1], L1) &&
    shifted{L1, L2}(1, a, beg, end - 1) ;
*/
```

```
/*@ lemma rotate_left_is_permutation{L1, L2}:
  \forall int* a, integer beg, integer end ;
    rotate_left{L1, L2}(a, beg, end) ==> permutation{L1, L2}(a, beg, end) ;
*/
```

We also have to help a bit the provers to show that the range is sorted after the insertion. For this, we provide a new invariant to show that the shifted values are greater than the value to insert, and then we add some assertions to show that the array is sorted before the insertion, that all values before the cell where we insert are lower than the inserted value, and that the range is consequently sorted after the insertion. Leading us to the following annotated `insert` function:

```
/*@
  requires beg < last < SIZE_MAX && \valid(a + (beg .. last));
  requires sorted(a, beg, last) ;

  assigns a[ beg .. last ] ;

  ensures permutation{Pre, Post}(a, beg, last+1);
  ensures sorted(a, beg, last+1) ;
*/
void insert(int* a, size_t beg, size_t last){
  size_t i = last ;
  int value = a[i] ;

  /*@
    loop invariant beg <= i <= last ;
    loop invariant \forall integer k ; i <= k < last ==> a[k] > value ;
    loop invariant \forall integer k ; beg <= k <= i     ==> a[k] == \at(a[k], Pre) ;
    loop invariant \forall integer k ; i+1 <= k <= last ==> a[k] == \at(a[k-1], Pre) ;

    loop assigns i, a[beg .. last] ;
    loop variant i ;
  */
  while(i > beg && a[i - 1] > value){
    a[i] = a[i - 1] ;
    --i ;
  }
  //@ assert sorted(a, beg, last+1) ;
  //@ assert \forall integer k ; beg <= k < i ==> a[k] <= value ;
  a[i] = value ;
  //@ assert sorted(a, beg, last+1) ;

  //@ assert unchanged{Pre, Here}(a, beg, i) ;
  //@ assert permutation{Pre, Here}(a, beg, i) ;

  //@ assert rotate_left{Pre, Here}(a, i, last+1) ;
  //@ assert permutation{Pre, Here}(a, i, last+1) ;
}
```

Overall, we have 6 lemmas to prove:

- `l_occurrences_union`
- `shifted_maintains_occ`
- `unchanged_is_permutation`

- `rotate_left_is_permutation`
- `union_permutation`
- `transitive_permutation`

While the Coq proofs of these lemmas are beyond the scope of this tutorial (and we will see later that in this particular case we can get rid of them), let us give some ideas of how to prove them (nevertheless, the Coq scripts are available on the GitHub repository of this book).

To prove `l_occurrences_union`, we reason by induction on the size of the second part of the array, the basic case is trivial: if the size is 0, we immediately have the equality since `split == to`. So now, we have to prove that if it is true for a range of size i, it is true for a range of size $i+1$. As we know that it is true until i by our induction hypotheses, we simply analyze the different cases for the last element of the range: either this element is the one we count or not, and it adds the same value on both sides of the equality.

For `shifted_maintains_occ`, we reason by induction on the complete range, the first case is trivial (empty range), and for the induction case, we just have to show the value added to the range has been shifted and is thus the same.

The property `unchanged_is_permutation` is proved by SMT solvers thanks to the fact that we have expressed it using `shifted`, then the solver can immediately instantiate the previous lemma. If it is not the case (depending on the version of the prover), the proof is done by instantiating `shifted_maintains_occ` with a value of 0 for the shift property.

To prove `rotate_left_is_permutation` we split the range at `L1` into two subparts `beg .. beg+1` and `beg+1 .. end`, and the range at `L2` into two subparts `beg .. end-1` and `end-1 .. end`, using `l_occurrences_union`. We show that the number of occurrences in `beg+1 .. end` at `L1` and `beg .. end-1` at `L2` did not change thanks to `shifted_maintains_occ`, and that the number of occurrences in `beg .. beg+1` at `L1` and `end-1 .. end` at `L2` is the same by case analysis (and using the fact that the values equal).

For `union_permutation`, we instantiate the lemma `l_occurrences_union`. Finally, the lemma `transitive_permutation` is automatically proved by SMT solvers thanks to the transitivity of equality.

7.2.4. How to correctly use assertions?

There is no perfect guideline on when we should use assertions or not. Most of the time we use them to first understand why some proof fails at some point by expressing properties that we expect to be true at some program point. Moreover, most of the time, the verification conditions are too long and complex to be read directly, and we somewhat have to keep in mind the different lemmas we have already expressed and to determine whether some deduction requires the use of a lemma that we already expressed, or if it requires to reason by induction on some property or value such that SMT solvers cannot make this deduction, and thus we might need to add another lemma.

With a bit of experience, the use of assertions and lemmas become more and more natural, however one has to take care of the fact that it is easy to abuse of these. Basically, the more we add lemmas and assertions, the more the proof context is rich and might contain the required information to produce the proof that we need. However, there is also a risk to add too much information so that the proof context mainly contains garbage that is not useful for the proof but pollute the proof context and makes the job of SMT solvers harder. Thus, we have to find the good trade-off.

7.2.5. Exercises

7.2.5.1. Understanding the proof context

In the following function, the last assertion is automatically proved by SMT solvers, probably by unfolding the predicate and directly proving the corresponding property. Using assertions, provide another way to prove the last property. In the proof context, find the generated properties that could allow to prove the assertion and explain how.

```c
/*@
  predicate rectangle{L}(integer c1, integer c2, integer h) =
    c1 * c1 + c2 * c2 == h * h ;
*/

/*@
  requires \separated(x, y , z);
  requires 3 <= *x <= 5 ;
  requires 3 <= *y <= 5 ;
  requires 2 <= *z <= 5 ;
  requires *x+2 == *y+1 == *z ;
*/
void exercise(int* x, int* y, int* z){
  *x += 2 * (*x) ;
  *y += *y ;
  *y += (*y / 2);
  *z = 3 * (*z) ;
  //@ assert rectangle(*x, *y, *z);
}
```

7.2.5.2. Trigger lemmas

In the following program, WP fails to prove that the postcondition of the function `g` is verified. Add the right assertion (at the right place), such that the proof succeeds.

```c
/*@
  axiomatic Ax {
    predicate X{L1, L2}(int* p, integer l)
        reads \at(p[0 .. l-1], L1), \at(p[0 .. l-1], L2) ;
    predicate Y{L1, L2}(int* p, integer l)
        reads \at(p[0 .. l-1], L1), \at(p[0 .. l-1], L2) ;

    axiom Ax_axiom_XY {L1,L2}:
      \forall int* p, integer l, i ; 0 <= i <= l ==> X{L1, L2}(p, i) ==> Y{L1, L2}(p, l) ;
    axiom transitive{L1,L2,L3}:
      \forall int* p, integer l ; Y{L1,L2}(p, l) ==> Y{L2,L3}(p, l) ==> Y{L1,L3}(p, l);
  }
*/

/*@
  assigns p[0 .. l-1] ;
  ensures X{Pre, Post}(p, l) ;
*/
void f(int* p, unsigned l);

/*@
  ensures Y{Pre,Post}(p, l);
*/
void g(int* p, unsigned l){
  f(p, l) ;
  f(p, l) ;
}
```

7.2.5.3. Trigger lemmas, under condition

In the following program, WP fails to prove that the postcondition of the function `example` is verified. However, we can notice that the `g` function indirectly ensures that the pointed value is either increased or decreased. Add two assertions that shows which predicate holds depending on the value of `*x`.

```
/*@
  predicate dec{L1, L2}(int* x) =
    \at(*x, L1) > \at(*x, L2) ;
  predicate inc{L1, L2}(int* x) =
    \at(*x, L1) < \at(*x, L2) ;
*/

/*@
  axiomatic Ax {
    predicate P(int* x) reads *x ;
    predicate Q(int* x) reads *x ;

    axiom ax_1: \forall int* x ; P(x) ==> Q(x);
    axiom ax_2{L1, L2}:
      \forall int* x ; dec{L1, L2}(x) ==> P{L1}(x) ==> P{L2}(x);
    axiom ax_3{L1, L2}:
      \forall int* x ; inc{L1, L2}(x) ==> P{L1}(x) ==> P{L2}(x);
  }
*/

/*@
  assigns *x ;
  behavior b_1:
    assumes *x < 0 ;
    ensures *x >= 0 ;
  behavior b_2:
    assumes *x >= 0 ;
    ensures *x < 0 ;
  complete behaviors ;
  disjoint behaviors ;
*/
void g(int* x);

/*@
  requires P(x) ;
  ensures  Q(x) ;
*/
void example(int* x){
  g(x);
}
```

The assertions should look like:

```
//@ assert *x ... ==> ... ;
//@ assert *x ... ==> ... ;
```

Another way to bring some information in the context is to use ghost code. For example, the truth value of a condition appears in the context of a verification condition. Modify the annotation to make the code look like:

```
void example(int* x){
  g(x);
  /*@ ghost
   if ( ... ){
    /@ assert ... @/
   } else {
    /@ assert ... @/
   }
  */
}
```

Compare the verification condition of each assertion with the previous ones.

Finally, one can in fact notice that “the pointed value is either increased or decreased” can be expressed with a single simple assertion. Write the corresponding annotation.

7.2.5.4. An actual example with sum

The following function increases by 1 the value of a cell in an array, thus it also increases the value of the sum of the content of this array. Write a contract to the function that expresses this fact.

```
#include <stddef.h>

/*@
  axiomatic Sum_array{
    logic integer sum(int* array, integer begin, integer end) reads array[begin .. (end-1)];
    axiom empty:
      \forall int* a, integer b, e; b >= e ==> sum(a,b,e) == 0;
    axiom range:
      \forall int* a, integer b, e; b < e ==> sum(a,b,e) == sum(a,b,e-1)+a[e-1];
  }
*/

/*@
  predicate unchanged{L1, L2}(int* array, integer begin, integer end) =
    \forall integer i ; begin <= i < end ==> \at(array[i], L1) == \at(array[i], L2) ;
*/

/*@
  lemma sum_separable:
    \forall int* array, integer begin, split, end ;
      begin <= split <= end ==> sum(array, begin, end) == 0 ; // to complete
  lemma unchanged_sum{L1, L2}:
    \forall int* array, integer begin, end ;
      unchanged{L1, L2}(array, begin, end) ==> \true ; // to complete
*/

void inc_cell(int* array, size_t len, size_t i){
  array[i]++ ;
}
```

In order to prove that this function fulfills its contract, we need to provide some assertions that will guide the proof. More precisely, we have to show that since all values before the modified cell has not been modified, the sum has not been modified in this part of the array, and same for the cells that follow the modified cell.

Thus, we need two lemmas:

- `sum_separable` should express that we can split an array into two subparts, count in each of them and sum the results to get the sum of the entire array,
- `unchanged_sum` should express that if a range of an array has not changed between two labels, the sum of the content is the same.

Complete the code of the lemmas and use assertions to ensure that they will be used to complete the proof. We do not ask the proof of the lemmas, the Coq proofs are available on the GitHub repository of this book.

7.3. More on ghost code: lemma functions and lemma macros

Assertions provide a way to give clues to the verification condition generator so the SMT solvers will get enough information to make the proofs we need. However, it is sometimes hard to write the assertion that will create exactly the property needed by the SMT solver to trigger the right lemma (for example, since the generator makes some optimization on the verification condition that might slightly modify it and the context of the proof). Furthermore, we rely on lemmas that often need to be proved with the Coq proof assistant, and that means that we need to learn Coq.

In this section, we will see some techniques that we can use to make all of this more predictable and does not require from us to use the Coq proof assistant. While these techniques cannot be used in any case (and we will explain what are the cases when it is not applicable), they are quite efficient to get almost full automatic proof. This relies on ghost code.

7.3.1. Proof by induction

Previously, we mentioned that SMT solvers are bad at reasoning by induction (most of them), and this is the reason why we often need to express lemmas that we then prove using the Coq proof assistant that allows us to write our proof by induction. However, in the section 4.2 about loops, we find a subsection 4.2.1 named "Induction and invariant" where we explain how to prove that a loop does the right job ... by induction. What is this sorcery?!

In fact, it is quite simple. When we prove a loop invariant by induction using SMT solvers, they do not have to perform the reasoning by induction themselves. The job of splitting the proof into two sub-proofs, one for the establishment of the invariant (the base case of the proof), and one for the preservation (the induction case) is performed by the verification condition generator. So when the verification conditions are transmitted to the SMT solvers, this work is not needed anymore.

How can we exploit this idea? We explained before that ghost code can be used to provide more information than what is explicitly provided by the source code. For this, we add some code (and possibly annotations about this code) that allows to deduce more properties. Let us illustrate this with a simple example. In a previous exercise (5.4.5.2), we wanted to prove the following function call (we have excluded the postconditions to shorten the example):

```c
#include <stddef.h>
#include <limits.h>

/*@ predicate sorted(int* arr, integer end) =
      \forall integer i, j ; 0 <= i <= j < end ==> arr[i] <= arr[j] ;
    predicate element_level_sorted(int* array, integer end) =
      \forall integer i ; 0 <= i < end-1 ==> array[i] <= array[i+1] ;
*/

/*@ requires \valid_read(arr + (0 .. len-1));
    requires sorted(arr, len) ;
*/
size_t bsearch(int* arr, size_t len, int value);

/*@ requires \valid_read(arr + (0 .. len-1));
    requires element_level_sorted(arr, len) ;
*/
unsigned bsearch_callee(int* arr, size_t len, int value){
  return bsearch(arr, len, value);
}
```

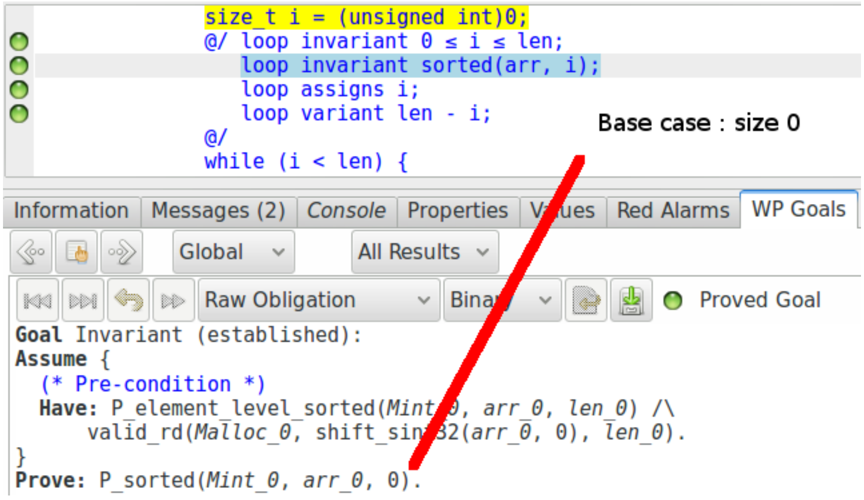
For this, the solution that was asked in the exercise was to provide a lemma that states that if a range is “locally sorted”, meaning that each element is greater or equals to the one that precedes it, then we can say that it is “globally sorted”, that is to say for each pair of indices i and j, if $i \leq j$ then the j^{th} of the array is greater or equals to the i^{th} element. Then, the precondition could be proved by SMT solvers, but not the lemma itself that requires a Coq proof. Can we do something using a loop?

The answer is yes. Before calling the function, we can build a proof that shows that because the array is locally sorted, we can deduce that it is globally sorted (which is basically a proof of the lemma we would need). We want to prove that the range is globally sorted. To write this proof by hand, we would proceed by induction on the size of the range. We have two cases. First, the range is empty and the property trivially true. Second, let us suppose that some range of size i with $i < length$ ($length$ being the size of the complete range), is globally sorted and show that if it is the case, then the range of size $i + 1$ is sorted. This is easy because, by our precondition, we know that the i^{th} element is greater than the $(i - 1)^{th}$ element, that is itself greater than all the preceding elements.

Now, how can we translate this into ghost code? We write a loop that goes from 0 (our base case), to the end of the range `len` and provide as an invariant that the array is globally sorted from 0 to the current step of the loop. We also add some assertions to help the provers (namely the fact that the current element is greater than the one that precedes it):

```
/*@ requires \valid_read(arr + (0 .. len-1));
    requires element_level_sorted(arr, len) ;
*/
unsigned bsearch_callee(int* arr, size_t len, int value){
  /*@ ghost
    /@
      loop invariant 0 <= i <= len ;
      loop invariant sorted(arr, i) ;
      loop assigns i ;
      loop variant len-i ;
    @/
    for(size_t i = 0 ; i < len ; ++i){
      /@ assert 0 < i ==> arr[i-1] <= arr[i] ; @/
    }
  */
  return bsearch(arr, len, value);
}
```

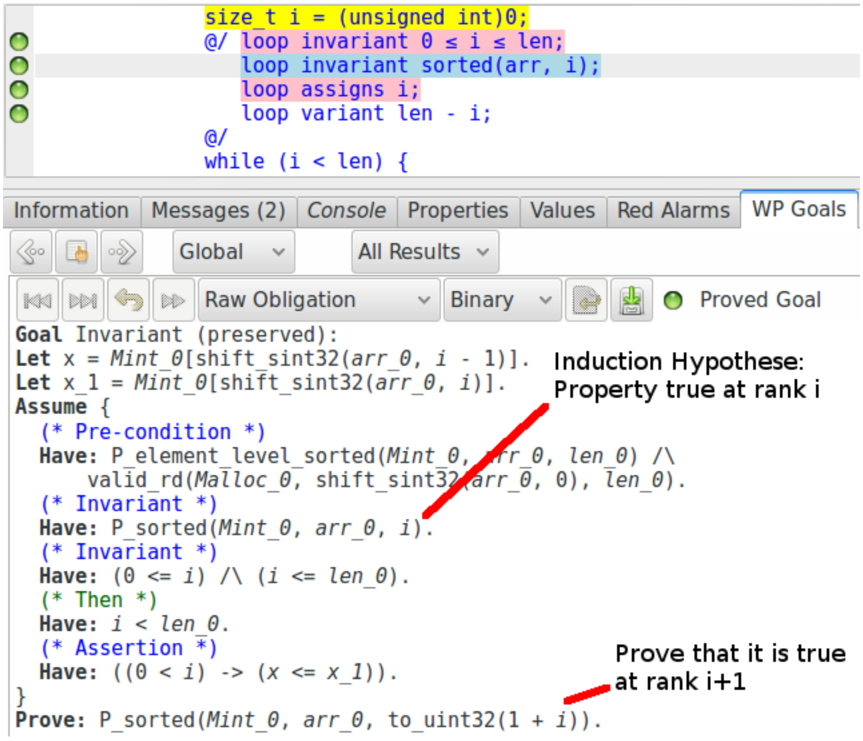
And we can see that all verification conditions are easily verified by SMT solvers, without requiring to write a Coq proof or a lemma. The verification conditions that are created respectively for the establishment and the preservation of the invariant correspond to the two cases we needed to prove in our proof by induction.



```
size_t i = (unsigned int)0;
@/ loop invariant 0 ≤ i ≤ len;
   loop invariant sorted(arr, i);
   loop assigns i;
   loop variant len - i;
@/
while (i < len) {
```

Base case : size 0

```
Goal Invariant (established):
Assume {
  (* Pre-condition *)
  Have: P_element_level_sorted(Mint_0, arr_0, len_0) /\
      valid_rd(Malloc_0, shift_sint32(arr_0, 0), len_0).
}
Prove: P_sorted(Mint_0, arr_0, 0).
```



```
size_t i = (unsigned int)0;
@/ loop invariant 0 ≤ i ≤ len;
   loop invariant sorted(arr, i);
   loop assigns i;
   loop variant len - i;
@/
while (i < len) {
```

```
Goal Invariant (preserved):
Let x = Mint_0[shift_sint32(arr_0, i - 1)].
Let x_1 = Mint_0[shift_sint32(arr_0, i)].
Assume {
  (* Pre-condition *)
  Have: P_element_level_sorted(Mint_0, arr_0, len_0) /\
      valid_rd(Malloc_0, shift_sint32(arr_0, 0), len_0).
  (* Invariant *)
  Have: P_sorted(Mint_0, arr_0, i).
  (* Invariant *)
  Have: (0 <= i) /\ (i <= len_0).
  (* Then *)
  Have: i < len_0.
  (* Assertion *)
  Have: ((0 < i) -> (x <= x_1)).
}
Prove: P_sorted(Mint_0, arr_0, to_uint32(1 + i)).
```

Induction Hypothese:
Property true at rank i

Prove that it is true
at rank i+1

This kind of code is called a proof carrying code: we have written some code and annotations that carries a proof that some property we want to verify holds.

Note that here, as we can write quite a lot of ghost code, we augment our risk to introduce a mistake that would change the properties of the verified code. We must thus verify that the ghost code we write terminates and that it does not contain runtime errors (through RTE plugin) to be confident about the verification.

In this example, we had to write the ghost code directly in annotation of the program, and that mean that if we have another call somewhere else in the code with some similar precondition, we would have to do it again. Let us make this easier by using lemma functions.

7.3.2. Lemma function

The principle of lemma functions is basically the same as lemmas: from some premises, we want to prove some conclusion. And once it has been done, we want to use it at some other place to directly deduce the conclusion from the premises without having to do the proof again, by instancing it with actual values.

The way to do this is to use a function, using the `requires` to express the premises of the lemma, and the `ensures` to express the conclusion of the lemma. The universally quantified variables can either still be quantified, or correspond to a parameter of the function. Namely, if a variable is only bounded to premises, or only to conclusions, it can be translated into a universally quantified variable, provided that it is not necessary to bind its value to a C variable in our proof carrying code (a quantified variable is not visible from the C code). If it is bounded to both premises and conclusion, it must be a parameter of the function (as we cannot quantify a variable for all a function contract in ACSL).

Let us first consider an example where we do not use (directly) universally quantified variables in the contract, with our previous example about sorted values. From the property `element_level_sorted(arr, len)`, we want to deduce `sorted(arr, len)`. The corresponding lemma could be:

```
/*@
  lemma element_level_sorted_is_sorted:
    \forall int* arr, integer len ;
       element_level_sorted(arr, len) ==> sorted(arr, len) ;
*/
```

So let us write a function that takes two parameters: `arr` and `len`, requires that the array is locally sorted and ensures that it is globally sorted:

```
/*@ ghost
  /@
    requires element_level_sorted(arr, len) ;
    assigns  \nothing ;
    ensures  sorted(arr, len);
  @/
  void element_level_sorted_implies_sorted(int* arr, size_t len);
*/
```

Note that this function must assign `\nothing`, indeed we use it to deduce some properties about the program, with ghost code, and thus it should not modify the content of the array, else the ghost code would modify the behavior of the program. Now let us provide a body to this function, the proof carrying code that guarantees that the conclusion is verified, provided that the precondition holds. It corresponds to the code we previously wrote to prove the precondition of the call to `bsearch`:

```
/*@ ghost
  /@
    requires element_level_sorted(arr, len) ;
    assigns  \nothing ;
    ensures  sorted(arr, len);
  @/
  void element_level_sorted_implies_sorted(int* arr, size_t len){
    /@
      loop invariant 0 <= i <= len ;
      loop invariant sorted(arr, i) ;
      loop assigns i ;
      loop variant len-i ;
    @/
    for(size_t i = 0 ; i < len ; ++i){
      /@ assert 0 < i ==> arr[i-1] <= arr[i] ; @/
    }
  }
*/
```

With this specified loop, we get an inductive proof that the lemma holds, now we can use this lemma function by simply calling it when we need to perform this deduction:

```
/*@ requires \valid_read(arr + (0 .. len-1));
    requires element_level_sorted(arr, len) ;
*/
unsigned bsearch_callee(int* arr, size_t len, int value){
  //@ ghost element_level_sorted_implies_sorted(arr, len) ;
  return bsearch(arr, len, value);
}
```

Which asks us to establish the premises thanks to the precondition of the lemma function (and which we trivially get from the precondition of the `bsearch_callee` function), and provides us the conclusion for free thanks to the postcondition of the lemma function (and we can use it as a precondition for the call to `bsearch`).

As we explained, when universally quantified variables are bounded to both conclusion and premises. They must be parameters, and it is the case here for the variables `arr` and `len`. Whereas the quantified variable that are used in the predicates:

```
/*@ predicate sorted(int* arr, integer end) =
      \forall integer i, j ; 0 <= i <= j < end ==> arr[i] <= arr[j] ;
    predicate element_level_sorted(int* array, integer end) =
      \forall integer i ; 0 <= i < end-1 ==> array[i] <= array[i+1] ;
*/
```

as they are only bound to respectively the premise and the conclusion remain universally quantified (even if it is hidden by the predicate). We could for example have written the contract like this:

```
/*@ ghost
  /@
    requires \forall integer i ; 0 <= i < len-1 ==> arr[i] <= arr[i+1] ;
    assigns  \nothing ;
    ensures  \forall integer i, j ; 0 <= i <= j < len ==> arr[i] <= arr[j] ;
  @/
  void element_level_sorted_implies_sorted(int* arr, size_t len){
    /@
      loop invariant 0 <= i <= len ;
      loop invariant sorted(arr, i) ;
      loop assigns i ;
      loop variant len-i ;
    @/
    for(size_t i = 0 ; i < len ; ++i){
      /@ assert 0 < i ==> arr[i-1] <= arr[i] ; @/
    }
  }
*/
```

where we perfectly see that variables are still universally quantified. However, we are not forced to maintain them universally quantified, and we could perfectly translate them into parameters (provided that the conclusion we want to get from the premises still makes sense). Let us for example translate the `i` and `j` of the conclusion into parameters:

```
/*@ ghost
  /@
    requires \forall integer i ; 0 <= i < len-1 ==> arr[i] <= arr[i+1] ;
    assigns  \nothing ;
    ensures  0 <= i <= j < len ==> arr[i] <= arr[j] ;
  @/
  void element_level_sorted_implies_greater(int* arr, size_t len, size_t i, size_t j){
    element_level_sorted_implies_sorted(arr, len);
  }
*/
```

Which is also perfectly fine, and we could for example use this function to deduce some properties about the content of the array. Note that here, we use a call to the previous lemma function to make the proof easier. We even can go further by transferring the “premise of our conclusion” as another premise of a new lemma:

```
/*@ ghost
  /@
    requires \forall integer i ; 0 <= i < len-1 ==> arr[i] <= arr[i+1] ;
    requires 0 <= i <= j < len ;
    assigns  \nothing ;
    ensures  arr[i] <= arr[j] ;
  @/
  void element_level_sorted_implies_greater_2(int* arr, size_t len, size_t i, size_t j){
    element_level_sorted_implies_sorted(arr, len);
  }
*/
```

All these lemmas state the same global relation, the difference is related to the amount of information that is required to instantiate them (and thus the precision of the property that we get in return).

Finally, let us present a last usage of lemma functions. On all previous examples, we have considered only universally quantified variable. In fact, what we have said before is applicable to existentially quantified variables: if they are bound to both premises and conclusions, they must be parameters, else they can either be parameters or remain quantified. However, about existentially quantified variables, we sometimes can go further by building a function that directly provide a witness for an existentially quantified formula.

For example, let us consider the axiomatic definition for occurrence counting, and imagine that at some point in a program, we want to prove the following assertion from the precondition:

```
/*@
  requires \valid(in + (0 .. len-1)) ;
  requires l_occurrences_of(v, in, 0, len) > 0 ;
*/
void foo(int v, int* in, size_t len){
  //@ assert \exists integer n ; 0 <= n < len && in[n] == v ;

  // ... code
}
```

Of course, there exists some index `n` such that `in[n]` is `v`, else the number of occurrences of this value would be 0. But, instead of just proving that such an index exists, let us directly find some index that satisfies the constraints on `n` by using a lemma function that returns it:

```
/*@ ghost
  /@
    requires \valid(in + (0 .. len-1)) ;
    requires l_occurrences_of(v, in, 0, len) > 0 ;
    assigns \nothing ;
    ensures 0 <= \result < len && in[\result] == v ;
  @/
  size_t occ_not_zero_some_is_v(int v, int* in, size_t len){
    /@
      loop invariant 0 <= i < len ;
      loop invariant l_occurrences_of(v, in, 0, i) == 0 ;
      loop assigns i ;
      loop variant len-i ;
    @/
    for(size_t i = 0 ; i < len ; ++i){
      if(in[i] == v) return i ;
    }
    /@ assert \false ; @/
    return SIZE_MAX ;
  }
*/
```

If we only look at the body of the function, it has two behaviors: either some cell of the array contains `v` and the function returns its index, or there is not, and then the function returns `SIZE_MAX`. The first behavior is easy to show, the return statement is performed in a branch where we know that the considered index corresponds to a cell that is in the range of the array and has a value `v`.

We prove that the second behavior ensures the postcondition by showing that it leads to a contradiction. If there is no cell of value `v`, then the number of occurrences of `v` is 0, this is expressed by the second invariant that shows that as we have not met any `v` from the beginning of the loop, the number of occurrences is 0. However, the precondition of the function states that the number of occurrences is more than 0 which leads to a contradiction that we model by an assertion of false (note that this is not necessary, we explicitly write it for our explanation) which means here that this path is infeasible.

Finally, we can call this function to show that there exists some index that allows our assertion to be validated:

```
/*@
  requires \valid(in + (0 .. len-1)) ;
  requires l_occurrences_of(v, in, 0, len) > 0 ;
*/
void foo(int v, int* in, size_t len){
  //@ ghost size_t witness = occ_not_zero_some_is_v(v, in, len);
  //@ assert \exists integer n ; 0 <= n < len && in[n] == v ;

  // ... code
}
```

The use of lemma functions makes reasoning by induction feasible for lemmas without the need of interactive proof. Furthermore, triggering lemmas becomes more predictable as we instantiate them by hand. However, while lemmas can consider multiple labels:

```
/*@
  lemma my_lemma{L1, L2}:  P{L1} ==> P{L2} ;
*/
```

Lemma functions do not provide an equivalent mechanism as they are basically normal C functions that cannot take labels in input. Let us show what we can do when we need such a construct.

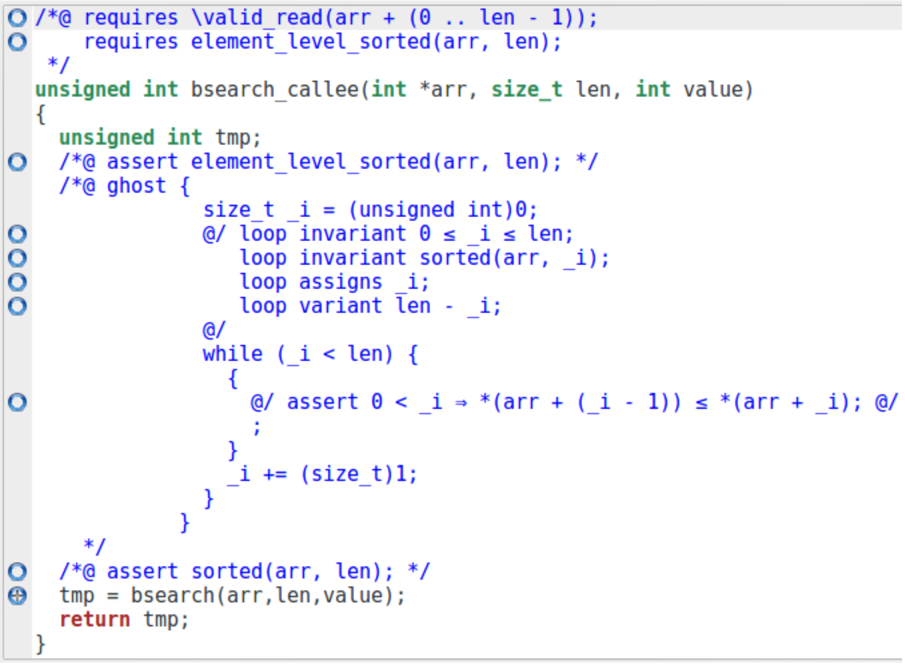
7.3.3. Lemma macro

When we have to deal with multiple labels, the idea is to directly "inject" the proof carrying code at the place where it is needed exactly as we did at the beginning of the section. However, we do not want to write this code by hand every time we need such a proof, so let us use macros to do it.

For now, let us translate our previous code into a macro instead of a function. As we use this macro in ghost code (thus, in annotation) we have to take care to use the ghost annotation syntax to write the invariant of the loop and the assertions:

```
#define element_level_sorted_implies_sorted(_arr, _len) \
  /@ assert element_level_sorted(_arr, _len) ; @/       \
  /@ loop invariant 0 <= _i <= _len ;                   \
     loop invariant sorted(_arr, _i) ;                  \
     loop assigns _i ;                                  \
     loop variant _len-_i ; @/                          \
  for(size_t _i = 0 ; _i < _len ; ++_i){                \
    /@ assert 0 < _i ==> _arr[_i-1] <= _arr[_i] ; @/    \
  }                                                     \
  /@ assert sorted(_arr, _len); @/

/*@ requires \valid_read(arr + (0 .. len-1));
    requires element_level_sorted(arr, len) ;
*/
unsigned bsearch_callee(int* arr, size_t len, int value){
  //@ ghost element_level_sorted_implies_sorted(arr, len) ;
  return bsearch(arr, len, value);
}
```

Instead of providing a pre and a postcondition, we state these properties using assertions before and after the proof carrying code. The proof carrying code itself is basically the same as before, and it is used exactly as it was used in the case of functions. However, we can see that it makes an important difference once it has been preprocessed by Frama-C as the block of code and annotations is directly injected in the function `bsearch_callee`.



```
/*@ requires \valid_read(arr + (0 .. len - 1));
    requires element_level_sorted(arr, len);
 */
unsigned int bsearch_callee(int *arr, size_t len, int value)
{
  unsigned int tmp;
  /*@ assert element_level_sorted(arr, len); */
  /*@ ghost {
            size_t _i = (unsigned int)0;
            @/ loop invariant 0 ≤ _i ≤ len;
               loop invariant sorted(arr, _i);
               loop assigns _i;
               loop variant len - _i;
            @/
            while (_i < len) {
              {
                @/ assert 0 < _i ⇒ *(arr + (_i - 1)) ≤ *(arr + _i); @/
                ;
              }
              _i += (size_t)1;
            }
          }
    */
  /*@ assert sorted(arr, len); */
  tmp = bsearch(arr,len,value);
  return tmp;
}
```

So in fact, we use the macro to generate the code we previously wrote. In this case, it is not interesting, since a function call allows us to make things more modular. So let us study a case where we do not have any other choice than using a macro.

We illustrate using the following lemma:

```
/*@
  predicate shifted{L1, L2}(int* arr, integer fst, integer last, integer shift) =
    \forall integer i ; fst <= i < last ==> \at(arr[i], L1) == \at(arr[i+shift], L2) ;

  lemma shift_ptr{L1, L2}:
    \forall int* arr, integer fst, integer last, integer s1, s2 ;
      shifted{L1, L2}(arr, fst+s1, last+s1, s2) ==> shifted{L1, L2}(arr+s1, fst, last, s2) ;
*/
```

In order to prove the following program:

```
/*@
  requires \valid(array+(beg .. end+shift-1)) ;
  requires shift + end <= SIZE_MAX ;
  assigns array[beg+shift .. end+shift-1];
  ensures shifted{Pre, Post}(array, beg, end, shift) ;
*/
void shift_array(int* array, size_t beg, size_t end, size_t shift);

/*@
  requires \valid(array+(0 .. len+s1+s2-1)) ;
  requires s1+s2 + len <= SIZE_MAX ;
  assigns array[s1 .. s1+s2+len-1];
  ensures shifted{Pre, Post}(array+s1, 0, len, s2) ;
*/
void callee(int* array, size_t len, size_t s1, size_t s2){
  shift_array(array, s1, s1+len, s2) ;
}
```

Where the lemma `shift_ptr` is necessary to prove the postcondition of `callee` from the postcondition of `shift_array` . Our goal is of course to get rid of the lemma, replacing it by a lemma macro.

There is no precise guideline for designing a macro used from the injection of proof carrying code. However, most lemmas stated about multiple labels are quite similar in the way they relate labels. So let us illustrate with this example, most of the time designing a macro in such a situation will more or less follow the same scheme.

In order to build the macro, we need a context where we can work on it. We build the context using a function, let us name this function `context_to_prove_shift_ptr` . The idea is to use the function to build the macro in isolation of the rest of the program to make the verification of the property easier. However, while lemma functions are then called to deduce some properties in some other function, this function will never be called, its only role is to provide us a “place” where we can build our proof. In particular, as we need multiple memory labels, our function **needs** to modify the content of the memory (else, there is a single memory state for all the function).

Let us illustrate with our current problem to make all of this clearer. First, we create a macro `shift_array` that will contain our proof carrying code, for now let us just indicate that it is an empty statement. In the parameters of this lemma, we take the labels that are considered. Note that the rules we previously mentioned about quantified variables still apply to macros.

```c
#define shift_ptr(_L1, _L2, _arr, _fst, _last, _s1, _s2) ;
```

Then we create our context function:

```c
/*@
  assigns arr[fst+s1+s2 .. last+s1+s2] ;
  ensures shifted{Pre, Post}(arr, fst+s1, last+s1, s2) ;
*/
void assign_array(int* arr, size_t fst, size_t last, size_t s1, size_t s2);

/*@
  requires fst <= last ;
  requires s1+s2+last <= SIZE_MAX ;
*/
void context_to_prove_shift_ptr(int* arr, size_t fst, size_t last, size_t s1, size_t s2){
 L1: ;
  assign_array(arr, fst, last, s1, s2);
 L2: ;
  //@ assert shifted{L1, L2}(arr, fst+s1, last+s1, s2) ;

  //@ ghost shift_ptr(L1, L2, arr, fst, last, s1, s2) ;

  //@ assert shifted{L1, L2}(arr+s1, fst, last, s2) ;
}
```

Let us decompose this code, starting from the context function. In input, we receive all the variables of the lemma. We also state some properties about the bounds of the integer values we consider, basically these should be requirements that are not related to memory states, or related to the first one. Then, we introduce the label `L1` , and we call the function `assign_array` that leads us to the label `L2` . The role of this call is to ensure that WP will create a new memory label (thus, it will not consider that the memory is the same), and to establish our premises. Indeed, if we have a look at the contract of `assign_array`, we see that it assigns the array (which guarantees the creation of a new memory label) and in postcondition, it ensures that the content of the array, between the pre and the postcondition (thus, when we call it: `L1` and `L2`) satisfies the premise of our lemma (which we repeat on line 15, by adding an assertion). Then we use our `shift_ptr` macro (that will later contain the proof carrying code), and we then expect to be able to prove the postcondition of our lemma (line 19).

By doing this, we ensure that we built a context that only contains the information we need to build the proof carrying code that allows us to deduce the conclusion (line 19) from the premise (line 15). Now let us write the macro.

```
1  #define shift_ptr(_L1, _L2, _arr, _fst, _last, _s1, _s2)\
2    /@ assert shifted{_L1, _L2}(_arr, _fst+_s1, _last+_s1, _s2) ; @/      \
3    /@ loop invariant _fst <= _i <= _last ;                               \
4       loop invariant shifted{_L1, _L2}(_arr+_s1, _fst, _i, _s2) ;        \
5       loop assigns _i ;                                                  \
6       loop variant _last-_i ; @/                                         \
7       for(size_t _i = _fst ; _i < _last ; ++_i){                         \
8         /@ assert \let _h_i = \at(_i, Here) ;                            \
9            \at(_arr[_h_i+_s1], _L1) == \at(_arr[_h_i+_s1+_s2], _L2) ; @/ \
10      }                                                                  \
11   /@ assert shifted{_L1, _L2}(_arr+_s1, _fst, _last, _s2) ; @/
```

We will not detail this code which is quite similar to what we have written in the beginning of this section. The only small subtlety is the assertion that helps the SMT solvers to relate the memory locations between `L1` and `L2` together on lines 8–9. With this macro, we can see that the assertion at the end of the function `context_to_prove_shift_ptr` is correctly, proved. Thus, we expect the macro to help the provers to get a similar conclusion in a similar context (that is to say, a context were we now that `shifted` holds for some array between two memory labels).

Finally, we can complete the proof of our function `callee` by using our lemma macro:

```
/*@
  requires \valid(array+(0 .. len+s1+s2-1)) ;
  requires s1+s2 + len <= SIZE_MAX ;
  assigns array[s1 .. s1+s2+len-1];
  ensures shifted{Pre, Post}(array+s1, 0, len, s2) ;
*/
void callee(int* array, size_t len, size_t s1, size_t s2){
  shift_array(array, s1, s1+len, s2) ;
  //@ ghost shift_ptr(Pre, Here, array, 0, len, s1, s2) ;
}
```

As one could notice, while this technique allows to inject the proof carrying code with a single line of code, it can inject quite a lot of code and annotations each time we use it. Furthermore, once we inject the code in the location where we expect it to be actually useful, the corresponding context can sometimes be already complex. Thus, we could need to slightly modify the code of the macro in order to add more information that is unnecessary in a clean context.

Thus, we can see that there is an important difference between lemma functions and lemma macros. A lemma function has a behavior which is quite similar to “classic” lemmas: when we use it we make an immediate deduction of some conclusion from some premises because the proof of this lemma has been done separately. A lemma macro has a different behavior: every time we use it, we have to do the proof again.

All of this can make the proof context bigger, and harder to use for SMT solvers. There are other limitations to this technique and the careful reader might have noticed them. Let us now talk about it.

7.3.4. Limitations

The main limitation of lemma functions and lemma macros is the fact that we are limited to C types. For example, if we compare our lemma `element_level_sorted_is_sorted` with its corresponding lemma function, the original type for the variable `len` is a mathematical integer while in the lemma function its type is `size_t`. It means that while the lemma is true for any integer, and so it could be used no matter if in the program the type of the variable that represents the size is an `int`, or an `unsigned` (or another integer type), on the opposite, our lemma function can be used only if this type can be safely converted to `size_t`. However, this limitation is often not a problem: we just have to express our specification for the biggest type we have to consider in our program and most of the time, it will be enough. And if it is not, we can for example duplicate the lemma for the types we are interested in. Most of the time this limitation is not a big deal since during a verification, we just tend to work with the same types as the program uses.

In some cases, however, it can constrain our modeling of some properties, and it is mainly related to the logic types we can use to model some actual data structures. For example, in order to model a linked list, one could use the ACSL logic type `\list<Type>`, and express an inductive or axiomatic definition in order to define how an actual linked list can be modeled by a logic list, thus we could have some lemmas about logic lists. For example:

```
/*@
  lemma in_list_in_sublist:
    \forall \list<int> l, l1, l2, int element ;
      l == (l1 ^ l2) ==>      // Here, ^ denotes lists concatenation
      (in_list(element, l) <==> (in_list(element, l1) || in_list(element, l2))) ;
*/
```

We cannot write lemma functions with proof carrying code for this property as we have no way to use this type in C code, and thus, no way to write a loop and an invariant that would allow us to prove this property.

The other limitation is related to lemma macros and what we already mentioned in the previous part about assertions. By adding too many assertions, the proof context can become too big and complex, thus hard to manipulate for SMT solvers. Using lemma macros that can generate quite a lot of code and annotations can lead to bigger proof contexts, thus it should be used with care.

Finally, depending on the property to prove, it can be hard to find a proof carrying code. Indeed, proof assistants like Coq have been designed to be able to express proofs even for complex properties, mainly relying on a high level view of our problems, while C has been designed to write programs, and with really detailed low level view of our problems. Thus, it can be sometimes difficult to write a C program to handle some properties and even more to find a suitable invariant for the loops it would involve.

7.3.5. Back to the insertion sort

Now let us go back to our proof of the insertion sort algorithm and see how we can get rid of all our interactive proofs for this function. Note however that in this proof, we often need to deal with macros since the program has not been particularly written with the idea to formally verify it later (for this, the reader can refer to the version proposed in the book ACSL by Example which can be adapted with a similar technique and is easier to prove). Thus, in this example, we push the solvers to their limit because of big proof contexts. With this example, depending on how powerful the machine is, we might need to increase the proof timeout to 60 seconds (which is already quite long for an SMT solver). We again use the options `-warn-unsigned-overflow` and `-warn-unsigned-downcast` . In this example, we will illustrate three actual usage of ghost code that we have seen so far:

- directly writing code to build a proof,
- writing (and using) lemma functions,
- writing (and using) lemma macros.

We also make use of assertions to make the proof context richer so SMT solvers succeed in proving the properties we are interested in. Some parts of the annotations are equivalent to what we have done previously. First, we use some assertions that were also useful in our previous proof. We will recall their purpose for each function. Second, we re-use the same axiomatic definition for occurrences counting. Furthermore, we keep the following predicate definitions:

```
/*@
  predicate sorted(int* a, integer b, integer e) =
    \forall integer i, j; b <= i <= j < e ==> a[i] <= a[j];

  predicate shifted{L1, L2}(integer s, int* a, integer beg, integer end) =
    \forall integer k ; beg <= k < end ==> \at(a[k], L1) == \at(a[s+k], L2) ;

  predicate unchanged{L1, L2}(int* a, integer beg, integer end) =
    shifted{L1, L2}(0, a, beg, end);

  predicate rotate_left{L1, L2}(int* a, integer beg, integer end) =
    beg < end && \at(a[beg], L2) == \at(a[end-1], L1) &&
    shifted{L1, L2}(1, a, beg, end - 1) ;

  predicate permutation{L1, L2}(int* in, integer from, integer to) =
    \forall int v ; l_occurrences_of{L1}(v, in, from, to) ==
                    l_occurrences_of{L2}(v, in, from, to) ;
*/
```

As we will need them, as well as the lemma about the transitivity of occurrences counting as it is automatically proved by SMT solvers (thus we can keep it since it does not require an interactive proof from us):

```
/*@ lemma transitive_permutation{L1, L2, L3}:
  \forall int* a, integer beg, integer end ;
    permutation{L1, L2}(a, beg, end) ==>
    permutation{L2, L3}(a, beg, end) ==>
      permutation{L1, L3}(a, beg, end) ;
*/
```

Let us start with the `insertion_sort` function itself. In this function, we made use of three assertions:

```
/*@
  requires beg < end && \valid(a + (beg .. end-1));
  assigns a[beg .. end-1];
  ensures sorted(a, beg, end);
  ensures permutation{Pre, Post}(a,beg,end);
*/
void insertion_sort(int* a, size_t beg, size_t end){
  /*@
    loop invariant beg+1 <= i <= end ;
    loop invariant sorted(a, beg, i) ;
    loop invariant permutation{Pre, Here}(a,beg,end);
    loop assigns a[beg .. end-1], i ;
    loop variant end-i ;
  */
  for(size_t i = beg+1; i < end; ++i) {
    //@ ghost L:;
    insert(a, beg, i);
    //@ assert permutation{L, Here}(a, beg, i+1);
    //@ assert unchanged{L, Here}(a, i+1, end) ;
    //@ assert permutation{L, Here}(a, beg, end) ;
  }
}
```

The first one makes sure that the part of the array where we just inserted a value is a permutation of the same range of values before the call to the `insert` function, as this is the postcondition of the function, it is not necessary but let us keep it for illustration. The last assertion is the property we want to prove in order to get enough knowledge to use the lemma that states that permutation is transitive (and show that after the block of the loop since our array is a permutation of the array at the beginning, which is itself a permutation of the original one, then after the body of the loop we have maintained that the array is a permutation of the original one).

The second assertion says that the second part of the array is unchanged, and we want to use this knowledge to show that the number of occurrences of the values is unchanged. Here we could use a combination of lemma functions and macros to prove that the complete range is a permutation (as we will do for the other function) however, directly writing the code is here a bit simpler (requires fewer proofs, as we will see later) so let us directly write the code that will create the proof of our property.

In order to show that the complete range is a permutation, we have to show that the number of occurrences did not change. We know that the first part of the array is a permutation of the same range at the beginning of the body of the loop. Thus, we already know that the number of occurrences of any v did not change for a part of our array. By using a loop with `j` from `i` to `end` with an invariant `permutationL,PI(a, beg, j)`, let us continue the occurrences counting for the rest of our array, knowing that the second part is unchanged (when `i+1` is lower than `end` as else, we do not have anything to count):

```
  for(size_t i = beg+1; i < end; ++i) {
    //@ ghost L: ;
    insert(a, beg, i);
    //@ ghost PI: ;
    //@ assert permutation{L, PI}(a, beg, i+1);
    //@ assert unchanged{L, PI}(a, i+1, end) ;
    /*@ ghost
      if(i+1 < end){
        /@ loop invariant i+1 <= j <= end ;
           loop invariant permutation{L, PI}(a, beg, j) ;
           loop assigns j ;
           loop variant end - j ;
        @/
        for(size_t j = i+1 ; j < end ; ++j);
      }
    */
  }
```

which is enough to ensure that the `insertion_sort` function conforms to its specification as long as we finish the proof of the `insert` function. This second function makes a more complex action. Let us depart from this annotated version:

```
/*@
  requires beg < last < SIZE_MAX && \valid(a + (beg .. last));
  requires sorted(a, beg, last) ;

  assigns a[ beg .. last ] ;

  ensures permutation{Pre, Post}(a, beg, last+1);
  ensures sorted(a, beg, last+1) ;
*/
void insert(int* a, size_t beg, size_t last){
  size_t i = last ;
  int value = a[i] ;

  /*@
    loop invariant beg <= i <= last ;
    loop invariant \forall integer k ; i <= k < last ==> a[k] > value ;
    loop invariant \forall integer k ; beg <= k <= i    ==> a[k] == \at(a[k], Pre) ;
    loop invariant \forall integer k ; i+1 <= k <= last ==> a[k] == \at(a[k-1], Pre) ;

    loop assigns i, a[beg .. last] ;
    loop variant i ;
  */
  while(i > beg && a[i - 1] > value){
    a[i] = a[i - 1] ;
    --i ;
  }
  a[i] = value ;
  //@ assert sorted(a, beg, last+1) ;

  //@ assert rotate_left{Pre, Here}(a, i, last+1) ;
  //@ assert permutation{Pre, Here}(a, i, last+1) ;

  //@ assert unchanged{Pre, Here}(a, beg, i) ;
  //@ assert permutation{Pre, Here}(a, beg, i) ;
}
```

Again, the proof that this function maintains a permutation of the array is the hardest part of the job. The fact that the function guarantees that the value are sorted is already easily established. Using the same technique as for `insertion_sort` is not so easy here. Indeed, the second part of the array has been rotated which makes the properties slightly more complex. So, let us show that we can split the array at the position where we insert into two parts, in which we respectively show that:

- for the first part, since it is unchanged, for any v, the number of occurrences did not change either,
- for the second part, since it is rotated, for any v, the number of occurrences did not change.

First let us define a lemma function that allows to explicitly split a range of values into two subparts in which we can count separately.

```
/*@ ghost
  /@
    requires beg <= split <= end ;

    assigns \nothing ;

    ensures \forall int v ;
      l_occurrences_of(v, a, beg, end) ==
      l_occurrences_of(v, a, beg, split) + l_occurrences_of(v, a, split, end) ;
  @/
  void l_occurrences_of_explicit_split(int* a, size_t beg, size_t split, size_t end){
    /@
      loop invariant split <= i <= end ;
      loop invariant \forall int v ; l_occurrences_of(v, a, beg, i) ==
        l_occurrences_of(v, a, beg, split) + l_occurrences_of(v, a, split, i) ;
      loop assigns i ;
      loop variant end - i ;
    @/
    for(size_t i = split ; i < end ; ++i);
  }
*/
```

We can note that this property is proved in a way that is quite similar to what we wrote for the body of our loop in `insertion_sort`, we start from the point where we want to count and show that the property remains true until the end of the array.

We can use our function to split the array at the right place after the loop. However, we can only do it for the new content of the array, indeed, in order to establish it for the original array, we have to call the function on the original array, when we still do not know the value of i. Thus, let us write another version of the "split" property that shows that we can split the array at any index, thus make the `split` variable a universally quantified variable, and use the previous function to prove that it is true:

```
/*@ ghost
  /@
    requires beg <= end ;

    assigns \nothing ;

    ensures \forall int v, size_t split ; beg <= split <= end ==>
      l_occurrences_of(v, a, beg, end) ==
      l_occurrences_of(v, a, beg, split) + l_occurrences_of(v, a, split, end) ;
  @/
  void l_occurrences_of_split(int* a, size_t beg, size_t end){
    /@
```

```
      loop invariant beg <= i <= end ;
      loop invariant \forall int v, size_t split ; beg <= split < i ==>
        l_occurrences_of(v, a, beg, end) ==
        l_occurrences_of(v, a, beg, split) + l_occurrences_of(v, a, split, end) ;
      loop assigns i ;
      loop variant end - i ;
    @/
    for(size_t i = beg ; i < end ; ++i){
      l_occurrences_of_explicit_split(a, beg, i, end);
    }
  }
*/
```

And we can split our original array and the new one:

```
void insert(int* a, size_t beg, size_t last){
  size_t i = last ;
  int value = a[i] ;

  // split before modifying
  //@ ghost l_occurrences_of_split(a, beg, last+1);

  /*@ LOOP ANNOT */
  while(i > beg && a[i - 1] > value){
    a[i] = a[i - 1] ;
    --i ;
  }
  a[i] = value ;
  // Assertions ...

  // split after modifying, now we know "i"
  //@ ghost l_occurrences_of_explicit_split(a, beg, i, last+1);
}
```

Now, the only remaining parts of the proof are first to show that an unchanged array is a permutation and second that the rotate operation also maintains a permutation. Here, we need macros. Let us start with the easiest: the unchanged property that we already almost exactly proved in the `insertion_sort` function. We start by building the context for our proof:

```
/*@
  assigns arr[fst .. last-1] ;
  ensures unchanged{Pre, Post}(arr, fst, last);
*/
void unchanged_permutation_premise(int* arr, size_t fst, size_t last);

/*@
  requires fst <= last ;
*/
void context_to_prove_unchanged_permutation(int* arr, size_t fst, size_t last){
 L1: ;
  unchanged_permutation_premise(arr, fst, last);
 L2: ;
  //@ ghost unchanged_permutation(L1, L2, arr, fst, last) ;

  //@ assert permutation{L1, L2}(arr, fst, last) ;
}
```

The function `unchanged_permutation_premise` ensures that we have modified the array (thus created a new memory state) and that the array is unchanged from the precondition to the postcondition. We can build our lemma macro:

```
#define unchanged_permutation(_L1, _L2, _arr, _fst, _last)      \
  /@ assert unchanged{_L1, _L2}(_arr, _fst, _last) ; @/         \
  /@ loop invariant _fst <= _i <= _last ;                       \
     loop invariant permutation{_L1, _L2}(_arr, _fst, _i) ;     \
     loop assigns _i ;                                          \
     loop variant _last - _i ;                                  \
   @/                                                           \
   for(size_t _i = _fst ; _i < _last ; ++_i) ;                  \
   /@ assert permutation{_L1, _L2}(_arr, _fst, _last); @/
```

Which almost corresponds to what we have written previously in the `insert_sort` function, and we can use the macro where it is needed in the `insert` function.

```
  //@ assert unchanged{Pre, Here}(a, beg, i) ;
  //@ ghost unchanged_permutation(Pre, Here, a, beg, i) ;
  //@ assert permutation{Pre, Here}(a, beg, i) ;
```

The only remaining property to prove is the hardest one and is about the `rotate_left` predicate. Let us first write our context to prepare the macro.

```
 1  /*@
 2    assigns arr[fst .. last-1] ;
 3    ensures rotate_left{Pre, Post}(arr, fst, last);
 4  */
 5  void rotate_left_permutation_premise(int* arr, size_t fst, size_t last);
 6
 7  /*@
 8    requires fst < last ;
 9  */
10  void context_to_prove_rotate_left_permutation(int* arr, size_t fst, size_t last){
11   L1: ;
12    //@ ghost l_occurrences_of_explicit_split(arr, fst, last-1, last) ;
13    rotate_left_permutation_premise(arr, fst, last);
14   L2: ;
15    //@ ghost rotate_left_permutation(L1, L2, arr, fst, last) ;
16
17    //@ assert permutation{L1, L2}(arr, fst, last) ;
18  }
```

How can we prove this property? Basically, one has to notice that since all the elements from the beginning to the penultimate have been shifted of one index to the right the number of occurrences in the shifted part did not change. Then one has to show that the number of occurrences of any v in respectively the last cell in the original array, and the first cell in the new array is the same (since the corresponding element is the same). Again, we rely on the split function to count separately the elements that are shifted and the one which is moved from the end to the beginning. However, the call corresponding to the original array has again to be put before the code that modifies the memory (see line 12) in the previous code, and we will have to take that in account when we will insert our use of the macro in the `insert` function.

Let us now present the macro that we use to prove that the lemma holds:

```
 1 #define rotate_left_permutation(_L1, _L2, _arr, _fst, _last)            \
 2   /@ assert rotate_left{_L1, _L2}(_arr, _fst, _last) ; @/               \
 3   /@ loop invariant _fst+1 <= _i <= _last ;                             \
 4      loop invariant \forall int _v ;                                    \
 5        l_occurrences_of{_L1}(_v, _arr, _fst, \at(_i-1, Here)) ==        \
 6        l_occurrences_of{_L2}(_v, _arr, _fst+1, \at(_i, Here)) ;         \
 7      loop assigns _i ;                                                  \
 8      loop variant _last - _i ;                                          \
 9    @/                                                                   \
10    for(size_t _i = _fst+1 ; _i < _last ; ++_i) {                        \
11      /@ assert \at(_arr[\at(_i-1, Here)], _L1) ==                       \
12                \at(_arr[\at(_i, Here)], _L2) ;                          \
13      @/                                                                 \
14    }                                                                    \
15    l_occurrences_of_explicit_split(_arr, _fst, _fst+1, _last) ;         \
16    /@ assert \forall int _v ;                                           \
17      l_occurrences_of{_L1}(_v, _arr, _fst, _last) ==                    \
18        l_occurrences_of{_L1}(_v, _arr, _fst, _last-1) +                 \
19        l_occurrences_of{_L1}(_v, _arr, _last-1, _last) ;                \
20    @/                                                                   \
21    /@ assert \at(_arr[_fst], _L2) == \at(_arr[_last-1], _L1) ==>        \
22      (\forall int _v ;                                                  \
23        l_occurrences_of{_L2}(_v, _arr, _fst, _fst+1) ==                 \
24        l_occurrences_of{_L1}(_v, _arr, _last-1, _last)) ;               \
25    @/                                                                   \
26    /@ assert permutation{_L1, _L2}(_arr, _fst, _last); @/
```

The loop invariant is pretty similar to what we have written so far, the only difference is that it takes in account the shift of the elements. Furthermore, to prove the invariant we had to add an assertion to help the solvers notice that the last element of both ranges is the same (note however that depending on the versions of the solvers or how powerful is the machine, this might be unneeded sometimes). A more important difference compared to our previous examples is that here, we need to provide more information to SMT solvers by adding other ghost functions calls (line 15, in order to split the first element of the array), as well as assertions to guide the last steps of the proof:

- 16–20: we recall that in the original array we can split the last element,
- 21–25: we show that as the first element of the array is the last element of the original array (21), the number of occurrences for any value in these ranges is the same (22–24).

We can use the macro in our program:

```
  //@ assert rotate_left{Pre, Here}(a, i, last+1) ;
  //@ ghost rotate_left_permutation(Pre, Here, a, i, last+1) ;
  //@ assert permutation{Pre, Here}(a, i, last+1) ;
```

However, we have to show that the considered range at label `Pre` can be split at `last`. For this, we use another variant of the split function, that shows that any sub-range can be split before the last element (if it is non-empty):

```
/*@ ghost
  /@
    requires beg < end ;

    assigns \nothing ;

    ensures \forall int v, size_t any ; beg <= any < end ==>
      l_occurrences_of(v, a, any, end) ==
      l_occurrences_of(v, a, any, end-1) + l_occurrences_of(v, a, end-1, end) ;
  @/
  void l_occurrences_of_from_any_split_last(int* a, size_t beg, size_t end){
    /@
      loop invariant beg <= i <= end-1 ;
      loop invariant \forall int v, size_t j ;
        beg <= j < i ==>
        l_occurrences_of(v, a, j, end) ==
        l_occurrences_of(v, a, j, end-1) + l_occurrences_of(v, a, end-1, end) ;
      loop assigns i ;
      loop variant (end - 1) - i ;
    @/
    for(size_t i = beg ; i < end-1 ; ++i){
      l_occurrences_of_explicit_split(a, i, end-1, end);
    }
  }
*/
```

That we have to call before the loop in the `insert` function:

```
void insert(int* a, size_t beg, size_t last){
  size_t i = last ;
  int value = a[i] ;

  //@ ghost l_occurrences_of_split(a, beg, last+1);
  //@ ghost l_occurrences_of_from_any_split_last(a, beg, last+1);
```

Note that depending on the version of the solvers, the assertion on lines 21 to 25 of the macro, about the element at the beginning/the end of the array might fail due to the complexity of the proof context. Let us help the solvers a last time by adding a last lemma (automatically proved by SMT solvers) that states this relation for any array and position in the array:

```
/*@ lemma one_same_element_same_count{L1, L2}:
  \forall int* a, int* b, int v, integer pos_a, pos_b ;
    \at(a[pos_a], L1) == \at(b[pos_b], L2) ==>
    l_occurrences_of{L1}(v, a, pos_a, pos_a+1) ==
    l_occurrences_of{L2}(v, b, pos_b, pos_b+1) ;
```

which guarantees that our resulting annotated insertion function is entirely proved:

```
/*@
  requires beg < last < SIZE_MAX && \valid(a + (beg .. last));
  requires sorted(a, beg, last) ;

  assigns a[ beg .. last ] ;

  ensures permutation{Pre, Post}(a, beg, last+1);
  ensures sorted(a, beg, last+1) ;
*/
void insert(int* a, size_t beg, size_t last){
  size_t i = last ;
  int value = a[i] ;

  //@ ghost l_occurrences_of_split(a, beg, last+1);
  //@ ghost l_occurrences_of_from_any_split_last(a, beg, last+1);

  /*@
    loop invariant beg <= i <= last ;
    loop invariant \forall integer k ; i <= k < last ==> a[k] > value ;
    loop invariant \forall integer k ; beg <= k <= i     ==> a[k] == \at(a[k], Pre) ;
    loop invariant \forall integer k ; i+1 <= k <= last ==> a[k] == \at(a[k-1], Pre) ;

    loop assigns i, a[beg .. last] ;
    loop variant i ;
  */
  while(i > beg && a[i - 1] > value){
    a[i] = a[i - 1] ;
    --i ;
  }
  a[i] = value ;
  //@ assert sorted(a, beg, last+1) ;

  /*@ assert
      \forall int v ;
      l_occurrences_of{Pre}(v, a, \at(i, Here), last+1) ==
        l_occurrences_of{Pre}(v, a, \at(i, Here), last) +
        l_occurrences_of{Pre}(v, a, last, last +1);
  */

  //@ assert rotate_left{Pre, Here}(a, i, last+1) ;
  //@ ghost rotate_left_permutation(Pre, Here, a, i, last+1) ;
  //@ assert permutation{Pre, Here}(a, i, last+1) ;

  //@ assert unchanged{Pre, Here}(a, beg, i) ;
  //@ ghost unchanged_permutation(Pre, Here, a, beg, i) ;
  //@ assert permutation{Pre, Here}(a, beg, i) ;

  //@ ghost l_occurrences_of_explicit_split(a, beg, i, last+1);
}
```

We finally highlight how the proof context can make the proof harder for SMT solvers. Basically, if we swap the proofs for each part of the array, that is, starting with the "unchanged" part and after the "rotate" part, the proof has more chance to fail, since it would make the proof context bigger for the hardest proof.

7.3.6. Exercises

7.3.6.1. Sum of N integers

Using lemma functions, we can prove the lemma about the sum of the N first integers that we previously expressed. You might have done this proof when you were in high school, now it is time to do it in C and ACSL. Write a contract for the following function that expresses in postcondition that the sum of the N first integers is `N(N+1)/2`. Complete the body of the function with a loop in order to prove this property. We advise to slightly transform the invariant in order to be sure that the property does not contain any division (division on integers have some properties that can make them hard to deal with for SMT solvers depending on the constraint that exists on the used values).

```
/*@
  logic integer sum_of_n_integers(integer n) =
    (n <= 0) ? 0 : sum_of_n_integers(n-1) + n ;
*/

/*@ ghost
```

```
  /@
    assigns \nothing ;
    ensures \true ; // to complete
  @/
  void lemma_value_of_sum_of_n_integers_2(unsigned n){
    // ...
  }
*/
```

Now, let us generalize to any bounds with the sum of all the integers between `fst` and `lst`. We provide the logic function and the contract. Write a body for the function such that the postcondition is proved. Note that again, we advise to express the invariant without division.

```
/*@
  logic integer sum_of_range_of_integers(integer fst, integer lst) =
    ((lst <= fst) ? lst : lst + sum_of_range_of_integers(fst, lst-1)) ;
*/

/*@ ghost
  /@
    requires fst <= lst ;
    assigns \nothing ;
    ensures ((lst-fst+1)*(fst+lst))/2 == sum_of_range_of_integers(fst, lst) ;
  @/
  void lemma_value_of_sum_of_range_of_integers(int fst, int lst){
    // ...
  }
*/
```

Finally, let us prove this function:

```
/*@
  requires n*(n+1) <= UINT_MAX ;
  assigns \nothing ;
  ensures \result == sum_of_n_integers(n);
*/
unsigned sum_n(unsigned n){
  return (n*(n+1))/2 ;
}
```

This should not be too hard, and what we obtain is a proof that we have written a correct optimization for the function that computes the sum of the N first integers.

7.3.6.2. Properties about occurrences counting

In this exercise, we want to prove some interesting properties about the axiomatically defined logic function `l_occurrences_of`:

```
#include <stddef.h>

/*@ ghost
  void occ_bounds(int v, int* arr, size_t len){
    // ...
  }

  void not_in_occ_0(int v, int* arr, size_t len){
    // ...
  }

  void occ_monotonic(int v, int* arr, size_t pos, size_t more){
    // ...
  }

  void occ_0_not_in(int v, int* arr, size_t len){
    // ...
    // needs occ_monotonic
  }

  size_t occ_pos_find(int v, int* arr, size_t len){
    // ...
    // needs occ_monotonic
  }

  void occ_pos_exists(int v, int* arr, size_t len){
    // ...
    // should use occ_pos_find
  }
*/
```

The function `occ_bounds` should state that the number of occurrences of `v` in the array is comprised between 0 and `len`.

The function `not_in_occ_0` should state that if `v` is not in the array then the number of occurrences of `v` in the array is 0.

The function `occ_monotonic` should state that the number of occurrences of `v` in the array from 0 to `pos` is lower or equals to the number of occurrences of `v` in the array from 0 to `more`, if `more` is greater or equals to `pos`.

The function `occ_0_not_in` should state that if the number of occurrences of `v` in the array is 0 then `v` is not in the array. Note that you will probably need to use `occ_monotonic`.

The function `occ_pos_find` should find an index `i` such that the value `arr[i]` is `v`, provided that the number of occurrences of `v` is positive. Note that you will probably need to use `occ_monotonic`.

Finally, the function `occ_pos_exists` should translate the contract of the previous function using an existentially quantified variable, and use the previous function to obtain the proof for free.

For all these functions, WP should be parameterized with the control of the absence of runtime errors as well as the options `-warn-unsigned-overflow` and `-warn-unsigned-downcast`.

7.3.6.3. An actual example with sum

Take back the proof performed in the previous chapter for the exercise 7.2.5.4. Modify the annotations in order to ensure that no more classic lemmas are necessary. The skeleton of the file follows:

```
#include <limits.h>
#include <stddef.h>

/*@
  axiomatic Sum_array{
    logic integer sum(int* array, integer begin, integer end) reads array[begin .. (end-1)];

    axiom empty:
      \forall int* a, integer b, e; b >= e ==> sum(a,b,e) == 0;
    axiom range:
      \forall int* a, integer b, e; b < e ==> sum(a,b,e) == sum(a,b,e-1)+a[e-1];
  }
*/

/*@
  predicate unchanged{L1, L2}(int* array, integer begin, integer end) =
    \forall integer i ; begin <= i < end ==> \at(array[i], L1) == \at(array[i], L2) ;
*/

/*@ ghost
  void sum_separable(int* array, size_t begin, size_t split, size_t end){
    // ...
  }
*/

#define unchanged_sum(_L1, _L2, _arr, _beg, _end) ;


/*@
  requires i < len ;
  requires array[i] < INT_MAX ;
  requires \valid(array + (0 .. len-1));
  assigns array[i];
  ensures sum(array, 0, len) == sum{Pre}(array, 0, len)+1;
*/
void inc_cell(int* array, size_t len, size_t i){
  // ...
  array[i]++ ;
  // ...
}
```

As we try to prove more complex properties, particularly when programs involve loops, there is a part of “trial and error” in order to understand what the provers miss establishing the proof.

It can miss hypotheses. In this case, we can try to add assertions to guide the prover, or write ghost code with the right invariant that allows to make a part of the reasoning by ourselves when it is too hard for SMT solvers.

With some experience, we can read the content of the verification condition or try to perform the proof with the Coq interactive prover to see whether the proof seems to be possible. Sometimes, the prover just needs more time, in such a case, we can (sometimes a lot) augment the timeout value. Of course, the property can be too hard for the prover and ghost code might be sometimes unsuitable, and in this case, we have to write the proof ourselves with an interactive prover.

Finally, the implementation can be indeed incorrect, and in this case we have to fix it. Here, we use test and not proof, because a test allows us to prove the presence of a bug and to analyze this bug.

8. Conclusion

> Voilà, c'est fini …
>
> *Jean-Louis Aubert, Bleu Blanc Vert, 1989*

… for this introduction to the proof of C programs using Frama-C and WP.

Along this tutorial, we have seen how we can use these tools to specify what we expect of our programs and verify that the source code we have produced indeed corresponds to the specification we have provided. This specification is provided using annotations of our functions that includes the contract they must respect. These contracts are properties required about the input to ensure that the function will correctly work, which is specified by properties about the output of the function and enforced by the tool that allow us to check specific problems related to the use of C (namely, the absence of runtime errors).

Starting from specified programs, WP is able to produce the weakest precondition of our functions, provided what we want in postcondition, and to ask some provers if the specified precondition is compatible with the computed one. The reasoning is completely modular, which allows proving functions in isolation from each other and to compose the results.

WP cannot currently work with dynamic allocation. A function that would use it could not be proved. However, even without dynamic allocation, a lot of function can be proved since they work with data-structures that are already allocated. And these functions can then be called with the certainty that they perform a correct job. If we cannot or do not want to prove the client code of a function, we can still write something like this:

```c
/*@
  requires some_properties_on(a);
  requires some_other_on(b);

  assigns ...
  ensures ...
*/
void my_function(int* a, int b){
  //this is indeed the  "assert" defined in "assert.h"
  assert(/*properties on a*/ && "must respect properties on a");
  assert(/*properties on b*/ && "must respect properties on b");
}
```

Which allows us to benefit from the robustness of our function having the possibility to debug an incorrect call in a source code that we cannot or do not want to prove.

Writing specifications is sometimes long or tedious. Higher-level features of ACSL (predicates, logic functions, axioms) allow us to lighten this work, as well as our programming languages allow us to define types containing other types, functions calling functions, bringing us to the final program. But, despite this, write specification in a formal language, no matter which one, is generally a hard task.

However, this **formalization** of our need is **crucial**. Concretely, such a formalization is a work every developer should do. And not only in order to prove a program. Even the definition of tests for a function requires to correctly understand its goal if we want to test what is necessary and only what is necessary. And writing specification in a formal language is incredibly useful (even if it can be sometimes frustrating) to get a clear specification.

Formal languages, that are close to mathematics, are precise. Mathematics have this: they are precise. What is more terrible than reading a specification written in a natural language, with complex sentences, using conditional forms, imprecise terms, ambiguities, compiled in administrative documents composed of hundreds of pages, and where we need to determine, "so, what this function is supposed to do? And what do I have to validate about it?".

Formal methods are probably not used enough currently. Sometimes because of mistrust, sometimes because of ignorance, sometimes because of prejudice based on ideas born at the beginning of the tools, 30 years ago. Our tools evolve, the ways we develop change, probably faster than in any other technical domain. Saying that these tools could never be used for real life programs would certainly be a too big shortcut. After all, we see every day how much development is different from what it were 10 years, 20 years, 40 years ago and can barely imagine how much it will be different in 10 years, 20 years, 40 years.

During the past few years, safety and security questions have become more and more visible and crucial. Formal methods also progressed a lot, and the improvement they bring for these questions are greatly appreciated. For example, this link brings to the report of a conference about security that brought together people from academic and industrial world, in which we can read:

> Adoption of formal methods in various areas (including verification of hardware and embedded systems, and analysis and testing of software) has dramatically improved the quality of computer systems. We anticipate that formal methods can provide similar improvement in the security of computer systems.
> ...
> **Without broad use of formal methods, security will always remain fragile.**
>
> *Formal Methods for Security, 2016*

8.1. Going further

8.1.1. With Frama-C

Frama-C provides different ways to analyze programs. Among these tools, the most commonly used and interesting to know from a static and dynamic verification point of view are certainly those:

- abstract interpretation analysis using EVA ,
- the transformation of annotation into runtime verification using E-ACSL .

The goal of the first one is to compute the domain of the different variables at each program point. When we precisely know these domains, we can determine if these variables can produce errors when they are used. However, this analysis is executed on the whole program and not modularly. It is also strongly dependent of the type of domain we use (we will not enter into details here) and it is not so good at keeping the relations between variables. On the other side, it is really completely automatic, we do not even need to give loop invariant! The most manual part of the work is to determine whether an alarm is a true error or a false positive.

The second analysis allows generating from an original program, a new program where the assertions are transformed into runtime verifications. If these assertions fail, the program fails. If they are valid, the program has the same behavior it would have without the assertions. An example of use is to generate the verification of absence of runtime errors as assertions using `-rte` and then to use E-ACSL to generate the program containing the runtime verification that these assertions do not fail.

There exist a lot of different plugins for very different tasks in Frama-C.

Finally, a last possibility that will motivate the use of Frama-C is the ability to develop their own analysis plugins using the API provided by the Frama-C kernel. A lot of tasks can be realized by the analysis of the source code and Frama-C allows to build them as easily as possible.

8.1.2. With deductive proof

Along this tutorial we used WP to generate verification conditions starting from programs with their specification. Next we have used automatic solvers to assure that these properties were indeed verified.

When we use other solvers than Alt-Ergo and Coq, the communication with this solver in provided by a translation to the Why3 language that will next be used to bridge the gap to automatic solvers. But this is not the only way to use Why3. It can also be used itself to write programs and prove them. It especially provides a set of theories for some common data structures.

There are some proofs that cannot be discharged by automatic solvers. In such a case, we have to provide these proofs interactively. WP, like Why3, can extract its verification conditions to Coq, and it is very interesting to study this language. In the context of Frama-C, we can produce reusable lemmas libraries proved with Coq. But Coq can also be used for many tasks, including programming. Note that Why3 can also extract its verification conditions to Isabelle or PVS that are also proof assistants.

Finally, there exists other program logics, for example separation logic or concurrent program logics. Again these notions are interesting to know in the context of Frama-C: if we cannot directly use them, they can inspire the way we specify our program in Frama-C for the proof with WP. They could also be implemented into new plugins to Frama-C.

A whole new world of methods to explore.